CISC 322/326 Assignment 2
Concrete Architecture of GNUstep

March 14, 2025

Group #30: Big Hero 6

Terence Jiang 21twxj@queensu.ca
Kevin Chen 21kzc2@queensu.ca
Alexander Zhao 21atxz@queensu.ca
Carter Gillam 21cng3@queensu.ca
Kavin Arasu 21kla5@queensu.ca
Daniel Gao 21dg59@queensu.ca

Table of Contents

1. Abstract

The GNUstep system was looked at in our last report to determine its conceptual architecture. This report will examine the concrete architecture of the GNUstep, by analyzing its structure through the use of the Understand tool and comparing it to the conceptual architecture that we've updated from our previous report based on group 20's conceptual architecture. After investigating and recovering the actual system dependencies, we've identified the interactions between each subsystem and their interactions, finding many differences from our conceptual model. Furthermore we will touch upon a detailed study of the GORM subsystem, exploring its internal architecture, interactions and alignment with conventional conceptual expectations. Finally, we will provide a rationale for every identified divergence, examining constraints and trade offs. Finally, in order to illustrate real-world cases, we have provided two sequence diagrams for non-trivial use cases.

2. Introduction and Overview

This report explores the concrete architecture of the GNUstep software system, providing an in-depth analysis of its as-built structure based on extracted dependency data. Our goal is to systematically derive the concrete architecture by grouping top-level entities into subsystems, identifying their interactions using the "Understand" tool, and investigating how they align with or diverge from our conceptual architecture established in Assignment 1.

The first section of this report details the derivation process, explaining the steps taken to analyze dependencies, structure subsystems, and refine our understanding of GNUstep's architecture. The next section presents a comparative analysis between the conceptual and concrete architectures, identifying key discrepancies and their underlying rationales. Any necessary modifications to the conceptual architecture are documented, including influences from other group's work where applicable.

A deeper investigation into our selected top-level subsystem 'Gorm' is then conducted, analyzing its internal structure and dependencies. We take a look at this module in detail exploring its conceptual and concrete views. We will then present two sequence diagrams for non-trivial use cases, showing how each part interacts. We take a look at using gorm to load and parse an objective-c header file and connect a button to an action.

Finally, we reflect on lessons learned throughout the process, discussing challenges encountered, insights gained, and potential improvements. The report concludes with a final summary of findings and references supporting our analysis.

3. Derivation Process

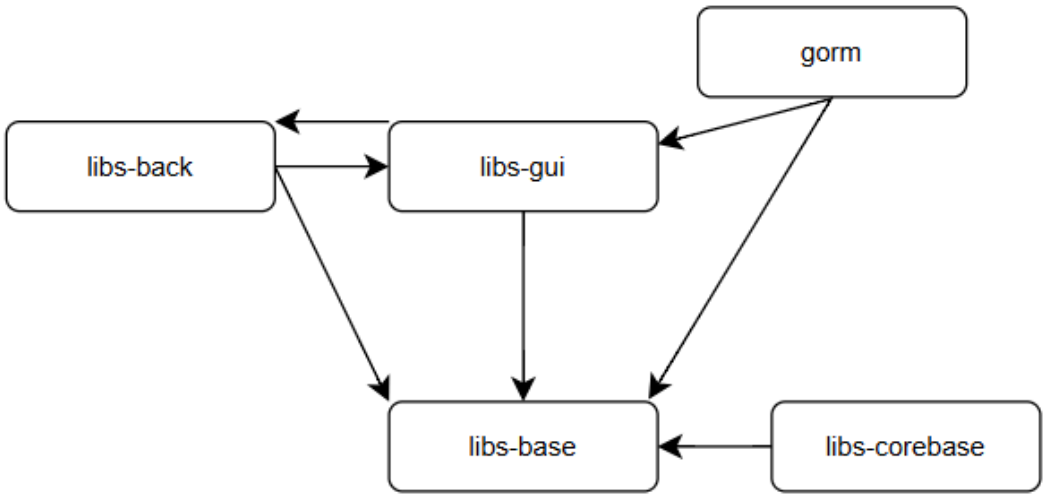
The key to beginning the derivation process was reexamining our conceptual architecture from the previous report. Upon closer inspection and analysis, we collectively determined it was impossible for GNUstep to represent the actual architecture of the program. We came to the

realization that our conceptual architecture contained many irrelevant subsystems that were not part of the core GNUstep architecture. In accordance with our own analysis and the professors recommendation, we've decided to base our conceptual architecture off Group 20's interpretation. We found that their interpretation was much simpler and honed in on the important subsystems used in the GNUstep application. Furthermore, the interactions between subsystems in Group 20's architecture was much simpler to follow and understand.

In order to properly derive the concrete architecture of GNUstep we leveraged an architectural analysis tool called *Understand*. *Understand* is a powerful tool which helps us visualize the relationships between subsystems of a particular program. After importing the source code for GNUstep, Understand allowed us to see the interactions and dependencies between subsystems of the GNUstep program without combing through the source code. This was a huge breakthrough in our process to definitely understand the concrete architecture of GNUstep.

There are many differences between our conceptual architecture and the concrete architecture of GNUstep. The concrete architecture has an extra subsystem called 'libsobjc2' that was not accounted for. Furthermore, the system of interactions between each subsystem was much more intricate than we would have previously thought. These ideas will further be explored in the coming sections of the report. Despite this, the concrete architecture of GNUstep proves our initial conjecture of the architecture style being object oriented.

4. Conceptual Architecture (High level)

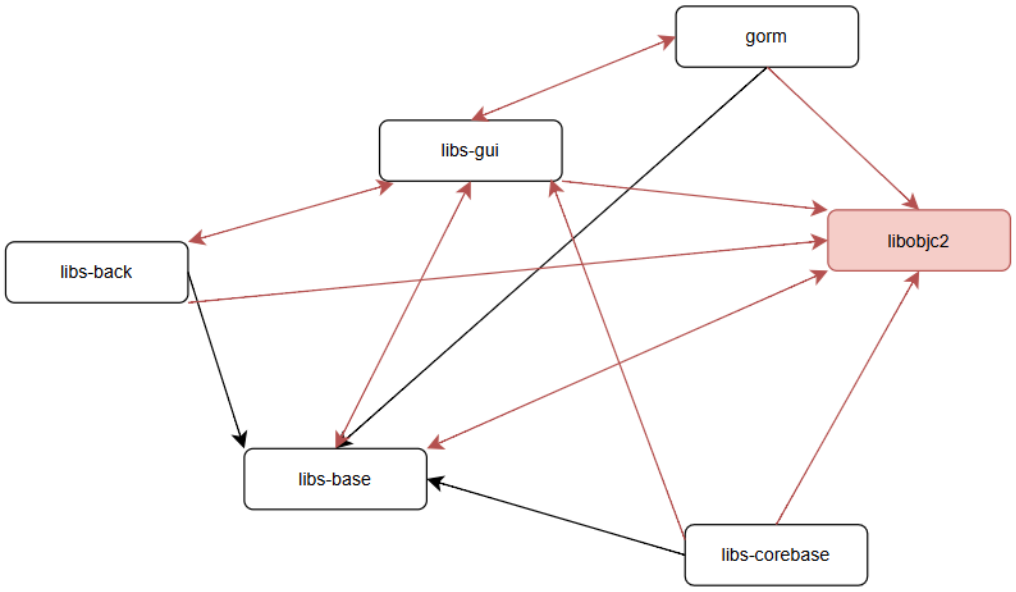


Conceptual Architecture of GNUstep Based on Group 20's A1

After examining our original conceptual architecture and realizing that we had irrelevant subsystems and dependencies, our updated architecture is based off of group 20's conceptual architecture.

As mentioned in our previous report, GNUstep follows an object oriented architecture style. This new conceptual architecture has five interconnected subsystems, each playing a crucial role in its overall functionality.

5. Concrete Architecture (High Level)



Concrete Architecture of GNUstep

This is a redrawn version of the graph generated by *Understand* with red arrows and boxes representing added dependencies or subsystems. Note that many of the red arrows are double dependencies replacing single dependencies from the previously noted conceptual architecture. We can conclude from *Understand* that the main architectural style of GNUstep is object oriented.

5.1 Concrete Subsystems and Interactions (High level)

In our derived concrete architecture, one new subsystem is introduced:

libobjc2: This is the Objective-C runtime used by GNUstep, providing the foundation for the system's object oriented behaviour. It manages dynamic features such as message passing, method resolution, and class/object data. Designed to be modern and compatible with Apple's runtime, **libobjc2** supports key features like Automatic reference counting, blocks, and optimized method dispatching for better performance. It enables dynamic loading, reflection, and runtime manipulation of objects, making it essential for GNUstep's modular and flexible architecture.

Subsystem Interactions:

libs-base: A core component that has strong dependencies on libobjc2, as it relies on Objective-C runtime features for class management, method dispatching, and object interactions. It also has dependencies on libs-corebase, which helps with CoreFoundation compatibility.
libs-gui: Builds on top of libs-base, inheriting its core functionalities while providing graphical user interface capabilities. It has bidirectional dependency with libs-back, as it relies on it for rendering, while libs-back ensures graphical compatibility across platforms.
libs-back: Interacts with both libs-gui and libobjc2 as it provides the rendering backend and ensures that GUI applications can run on different windowing systems.
libs-corebase: Has a relatively minor but important interaction with libs-base, extending its functionality to offer CoreFoundation compatibility.
apps-gorm: Represents Gorm, depends heavily on libs-base and libs-gui, as it provides a drag and drop UI design tool that generates objective-C code based on GNUstep's framework. It also interacts with libs-back for rendering and libobjc2 for object oriented functionality

Overall, the system exhibits strong interconnections through libs-base, which acts as a central foundation, and libobjc2, which ensures runtime object oriented behaviour. The GUI components (libs-gui, libs-back) and apps-gorm build on top of these core subsystems to provide a cohesive development environment.

5.2 Reflexion Analysis of High-Level Architecture

5.2.1 New Dependencies

apps-gorm → libs-base (7665 dependencies)

The conceptual model expected apps-gorm to rely primarily on libs-gui for UI-related functions, with no direct dependency on libs-base. However, analysis reveals a strong connection between apps-gorm and libs-base, suggesting that apps-gorm bypasses libs-gui for system-level operations such as file management and event processing. This likely stems from performance optimizations or historical design choices, leading to tighter coupling between the UI builder and core system logic, making future modifications more complex.

libs-gui → libs-base (24,009 dependencies, 13 primary)

libs-gui was expected to depend mainly on libs-back for rendering while remaining independent of system-level functions in libs-base. However, a significant dependency exists, indicating that GUI operations rely heavily on core utilities like window management and event handling. This reduces modularity, making GUI functionality more dependent on system-level changes, potentially complicating future updates.

libs-back ↔ libs-corebase (29/3 dependencies)

libs-back was intended to function independently as a rendering engine, but the analysis reveals an unexpected bidirectional dependency with libs-corebase. This suggests that certain rendering operations require CoreFoundation functionalities, and libs-corebase, in turn, depends on libs-back. This interdependency likely arises from platform compatibility concerns but increases architectural complexity, reducing modularity and making optimizations more difficult.

libs-base ↔ libs-corebase (475/294 dependencies)

libs-corebase was expected to enhance CoreFoundation compatibility without altering libs-base's behavior. However, the two now share a bidirectional dependency, indicating that libs-base has come to rely on libs-corebase for certain functionalities. This shift suggests that system responsibilities have been redistributed over time, leading to tighter coupling between these components, reducing flexibility, and increasing maintenance complexity.

5.2.2 Removed Dependencies

apps-gorm → libs-gui (Previously Expected Strong Connection, Now Weaker – 5235/3 dependencies)

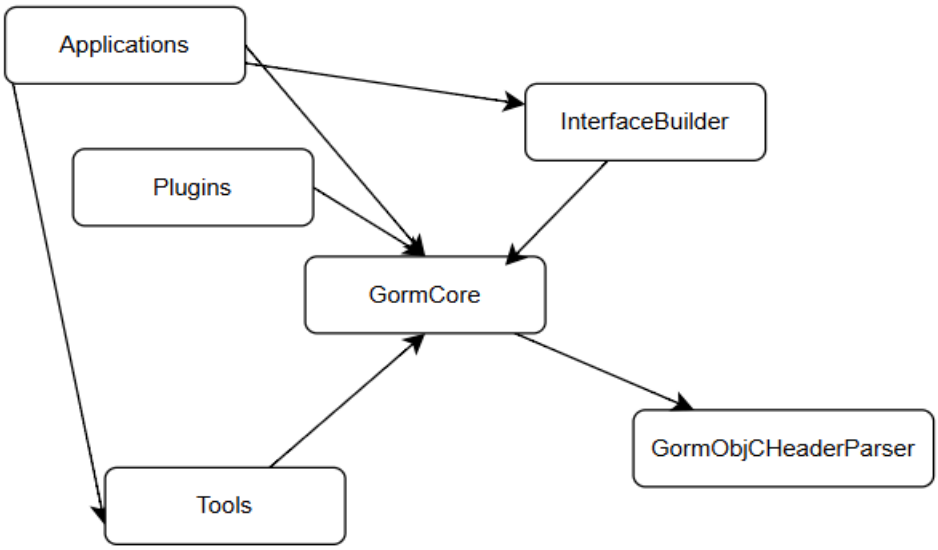
apps-gorm was expected to have a strong dependency on libs-gui for UI construction, but the connection is significantly weaker than anticipated. Instead, apps-gorm interacts more heavily with libs-base, suggesting that some UI-related operations are handled at the system level rather than being abstracted within libs-gui. This deviation makes GUI behavior harder to modify and maintain, as UI logic is now spread across multiple layers.

libs-gui ↔ libs-corebase (Previously Expected to Be Stronger, Now Weaker – 10/35 dependencies)

libs-gui was expected to rely more on CoreFoundation functions provided by libs-corebase, but the actual dependency is weaker than anticipated. This suggests that libs-gui has been designed to function more autonomously, reducing the need for CoreFoundation extensions. While this improves modularity, it also means that the conceptual model overestimated libs-gui's reliance on CoreFoundation utilities.

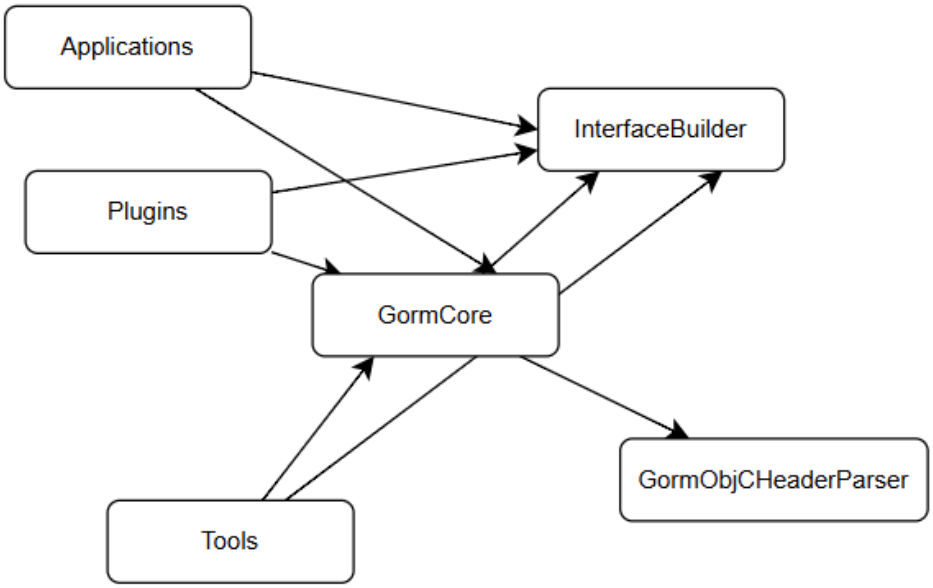
6. Conceptual Architecture (Gorm)

In this section, we will provide our conceptual architecture of Gorm, given that we already know the key submodules based on *Understand*



Conceptual Architecture of Gorm

7. Concrete Architecture (Gorm)



Redrawing of Gorm's Concrete Architecture from Understand

This is a redrawing of Gorm's concrete architecture provided by *Understand*. We can see that there are six key subsystems that all seem to run through a core system. Therefore we can conclude that Gorm has a **'Repository style'** architecture.

7.1 Concrete Subsystems and Interactions (Gorm)

Gorm Core is the central system for database management, while InterfaceBuilder handles the UI components and GormObjCHeaderParser helps with parsing Objective-C objects. Applications, Plugins, and Tools all use these to interact with the system and extend functionality.

We can also see that this is a **repository-style architecture.** The core system (GormCore) is central to managing the application's business logic and data. Other components extend or interact with this core system while remaining loosely coupled.

Applications: Relies on GormCore for database interaction and InterfaceBuilder for UI design
Plugins: Extend GormCore's functionality and use InterfaceBuilder to add custom UI elements
Tools: Interact with GormCore for database utilities and use Interface Builder for UI components
GormCore: Depends on InterfaceBuilder for UI generation and GormObjCHeaderParser for parsing Objective-C headers to integrate objects
GormObjCHeaderParser: Stands alone, parsing Objective-C headers for GormCore
InterfaceBuilder: Stands alone, is a tool used by other components to design UIs.

7.2 Reflexion Analysis of Gorm

We have already analyzed the high-level architecture of GNUStep, this time we will be analyzing the differences between the conceptual architecture and concrete architecture of a subsystem (Gorm) we introduced within Section 6 and 7.

7.2.1 New Dependencies

Plugins → InterfaceBuilder
Rationale: The Plugins subsystem would only be dependent on GormCore within the conceptual architecture. We notice that Plugins require direct dependency and access to Interface Builder to allow efficient UI component management and handle unique external GUI design standards. This also ensures that plugins are able to integrate with the broader GNU ecosystem.
GormCore → InterfaceBuilder
Rationale: To effectively handle nib files and manage UI layouts, GormCore requires the essential utilities that the InterfaceBuilder offers. This dependency also enhances the maintainability through centralizing operations related to UI within InterfaceBuilder.

Tools → InterfaceBuilder
Rationale: In the conceptual architecture, Tools only rely on GormCore. This is ineffective because Tools need direct access to InterfaceBuilder like Plugins and GormCore. InterfaceBuilder is essential for allowing Tools to perform advanced inspection and debugging of InterfaceBuilder-compatible designs along with providing developers with other crucial utilities used for UI development.

7.2.2 Removed Dependencies

Application → Tools
Rationale: In the Conceptual architecture Applications depended on Tools for debugging along with inspection functions. We can access all the necessary functions directly from GormCore without Tools. This helps simplify the system and reduces the possibilities of conflicts which improves the overall efficiency of the system.
InterfaceBuilder → GormCore
Rationale: Within the original design, InterfaceBuilder depended on GormCore for UI management but it deems unnecessary within the real implementation. Removing this dependency allows InterfaceBuilder to be separately maintained and extended and removes any sort of unwanted side effects that could arise from changes in GormCore.

8. Use Cases

8.1 Use Case 1

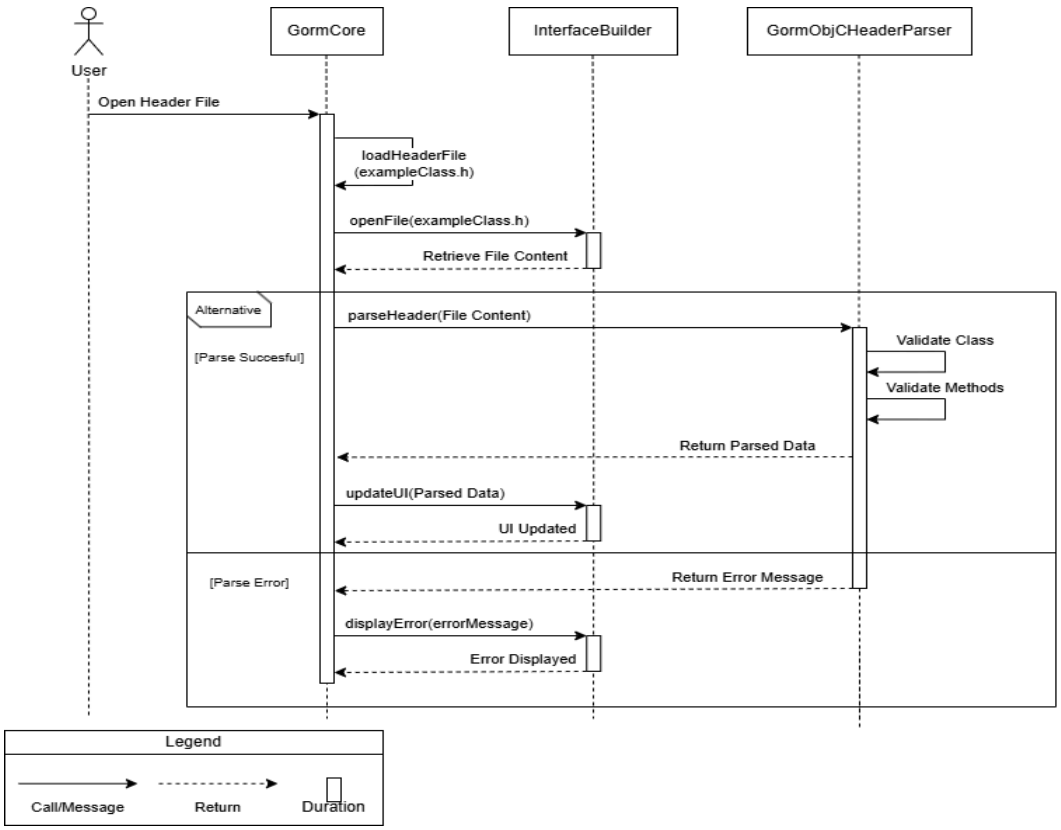


Figure 1. Use Case 1 Loading and Parsing an Objective-C Header File

This sequence diagram illustrates a use case that covers the process of loading and parsing an Objective-C header file, identifying its available classes, methods and properties. The user starts by requesting to load a header file through a command represented as loadHeaderFile() on GormCore. GormCore then calls the function openFile() so the contents can be evaluated in the InterfaceBuilder which eventually retrieves the file contents and returns it back to GormCore. Once the file data is obtained, GormCore sends the content to GormObjCHeaderParcer where it reads the file, extracting and validating the class and method definitions. If parsing is successful, GormObjCHeaderParcer returns the parsed classes and methods back to GormCore where it is forwarded to InterfaceBuilder through a method so it can be displayed in the interface. If any parsing error occurs, GormObjCHeaderParcer will instead send an error message back to GormCore, triggering an error message in InterfaceBuilder to notify the user. This process ensures that the available Objective-C classes and methods are properly reflected in the Gorm interface.

8.2 Use Case 2

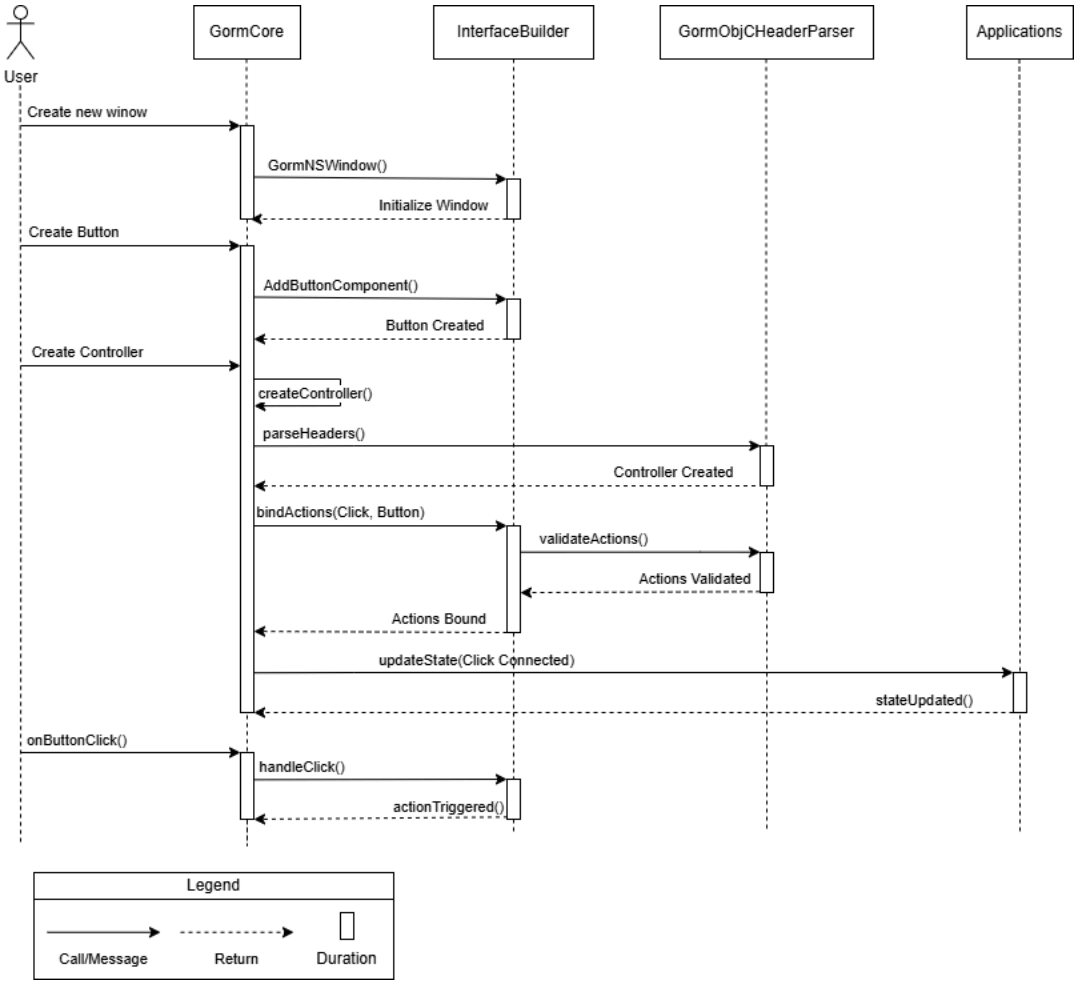


Figure 2.

Use Case 2 Connecting Button Component to an Action

In this use case, the user creates a button and binds it to an action. The user begins by creating a new window through GormNSWindow in GormCore which delegates the task to InterfaceBuilder, effectively initializing the window and returning control back to GormCore. The user then creates a button by sending a command from GormCore to InterfaceBuilder using a function call, which returns a confirmation once the button is initialized. Next, the user initializes a controller in GormCore and calls GormObjCHeaderParser to parse headers and identify the available actions provided by the controller. Once the controller is created, the user selects the controller's Click method to bind a clicking action to the button. The user invokes the bindActions(Click, Button) command on InterfaceBuilder, which connects the click action to the button. InterfaceBuilder then sends a request to GormObjCHeaderParser to validate the method definition, which returns a confirmation. After the action connection is established, GormCore updates the state in Applications to confirm the changes. When the user clicks the button, handleClick() is triggered in InterfaceBuilder, which processes the event and returns a confirmation to GormCore. This process ensures that the button's actions are properly configured and persist across sessions.

9. Lessons Learned

During our analysis of GNUstep's concrete architecture, we gained valuable insights into the discrepancies between conceptual and concrete architecture. The conceptual model depicted a simplified architecture with clear, high-level subsystems, and we expected those subsystems to only interact with the components specifically designed to interact with them. For example, apps-gorm was expected to rely only on libs-gui for UI-related tasks. However, the concrete model showed significantly more dependencies that were not accounted for in the conceptual design. Our biggest lesson learned was how to use Understand to analyze the dependencies between modules, as the analysis results allowed for us to clearly understand the interactions and roles of each module through the architecture. Without using Understand, it would have been difficult to create a concrete architecture through only analyzing the conceptual architecture ourselves, as the concrete architecture was so much more complex than the conceptual one.

Another lesson we learnt that might be considered trivial was the importance of team dynamics. Every member of our team has their own strengths and shortcomings, and through this assignment we were able to help each other out when we needed help. For example, after we decided what each member was going to complete, our members still worked on other parts of the assignment that were considered "harder" than other parts. For example, designing the concrete architecture was a pain point, but with multiple members helping each other out we were able to effectively complete it. Lastly, the team meeting up in person to work on assignments proved effective in allowing better workflow, as any questions and challenges we had were handled with as a team.

10. Data Dictionary

GNUstep: An open-source implementation of Apple's Cocoa frameworks, providing a foundation for developing macOS-compatible applications.

Gorm (GNUstep Object Relationship Modeler): A GUI builder for GNUstep that allows developers to create and manage user interfaces through a drag-and-drop environment.

GormCore: The central subsystem within Gorm responsible for managing database interactions and core business logic.

GormObjCHeaderParser: A subsystem within Gorm that parses Objective-C headers, allowing integration of Objective-C objects into Gorm.

InterfaceBuilder: A subsystem within Gorm used for UI component management, layout generation, and widget creation. Functions independently but is used by other components like GormCore, Plugins, and Tools.

libs-base: A core component of GNUstep responsible for essential system utilities, including file management, threading, and object-oriented programming support.

libs-back: The rendering engine of GNUstep that manages graphical rendering and ensures cross-platform compatibility.

libs-corebase: A subsystem that extends GNUstep's compatibility with macOS CoreFoundation, providing additional system utilities.

libs-gui: The primary user interface toolkit within GNUstep, responsible for managing UI elements and event handling.

Objective-C Runtime (libobjc2): The runtime environment for Objective-C applications in GNUstep, supporting dynamic message passing, method resolution, and memory management.

11. Conclusion

Conceptual architecture provides a high-level blueprint of a system, while concrete architecture reflects how the system is actually implemented. In this report, we explored the concrete object oriented style architecture of GNUstep, uncovering key differences from our initial conceptual model. Using the *Understand* tool, we identified new dependencies, such as the role of libobjc2, and examined unexpected subsystem interactions. Our detailed analysis of the Gorm subsystem further revealed its repository-style structure and its critical role within GNUstep. Through this process, we gained valuable insights into how conceptual models evolve when mapped to real-world implementations, highlighting the importance of architectural analysis in understanding complex software systems.

12. References

[1] *"apps-gorm Repository."* GitHub, https://github.com/gnustep/apps-gorm.

[2] GNUtered Team. "CISC 322 Group 20 - A1 Report." Queen's University, 2025.

[3] SciTools. Understand [Computer software]. Version 7.0, SciTools, 2025. Available at: https://scitools.com.