CISC 322 – Software Architecture

Queen's University

School of Computing

GNUstep Concrete Architecture

Assignment 2

March 15, 2025

Spencer Hum
21seh19@queensu.ca

Aidan Flint
21akf3@queensu.ca

Tony Luo
tony.luo@queensu.ca

Brant Xiong
21zx38@queensu.ca

Anson Jia
21aj85@queensu.ca

Zhangzhengyang Song
20zs57@queensu.ca

Table of Contents

Abstract

This report explores the concrete architecture of the GNUstep software project and one of its top-level subsystems—Gorm. For both the top-level architecture and Gorm, we start by refining the conceptual architecture obtained from our first report. Next, we used the Understand software developed by Scitools. Inc to conduct an in-depth static analysis of the source code of various components of GNUstep and extract its concrete architecture. We report on the divergence and absences between the two architectures and compare the architectural style of the respective system based on the concrete architecture. To conclude, we develop two use cases exemplifying data flow for the identified architectures and show the concrete method calls involved in those use cases in the form of sequence diagrams.

1 Introduction

This report examines the concrete architecture of GNUstep by analyzing the dependencies among all subsystems and identifying any differences from the conceptual architecture. By using dependency analysis tools and reflection analysis, we will construct an accurate representation of the GNUstep's complete structure and explain the new subsystem and dependencies not depicted in the conceptual architecture.

We first revisit our conceptual architecture from A1 to re-examine our findings and update it based on our enhanced knowledge of the system and its components. We then analyze the concrete architecture by examining the dependency relationships among all subsystems to identify key components and their interactions, followed by a review of the architecture style based on this new information. We then perform a reflection analysis, comparing the conceptual architecture to the concrete architecture to discover any structural differences, determine whether new dependencies have appeared, and analyze the reasons behind these differences.

After establishing the top-level architecture, we focus on the second-level subsystem Gorm, to conduct a more specific analysis. By examining its structure and dependencies within GNUstep, we will explain its functionality, how it interacts with the system, and perform a reflection analysis to show the differences between its conceptual and concrete architecture.

Finally, we illustrate potential use cases using sequence diagrams, providing a comprehensive understanding of GNUStep's architecture and the interaction between its subsystems.

1.1 Derivation process

The derivation process to construct the concrete architecture involved analyzing the GNUstep's source code and dependency graph using Scitools Understand. We started by examining the overall organization of files and components and mapping them to the conceptual architecture in A1. By analyzing the source code, dependency graph and their relationships we concluded that our original conceptual architecture could not be properly used as a base because certain files and dependencies did not belong to a single module or were not mapped in our A1.

To extract the concrete architecture of GNUstep, we examined the source code of the five main GNUstep components in conceptual architecture and an additional component libobjc2, which appeared during the Understand analysis. While libobjc2 was not considered in our original conceptual architecture, its existence in the dependency graph proves that libobjc2 plays a fundamental role in the system.

By analyzing source code, function calls and module dependencies, we categorized different subsystems while ensuring that dependencies between components were accurately represented. Through this process, we reconstructed a refined concrete architecture that more accurately reflects the system's true structure.

2 Updated Conceptual architecture

We use the proposed conceptual architecture in our first report, which is shown below, as the base conceptual architecture that will be improved upon.

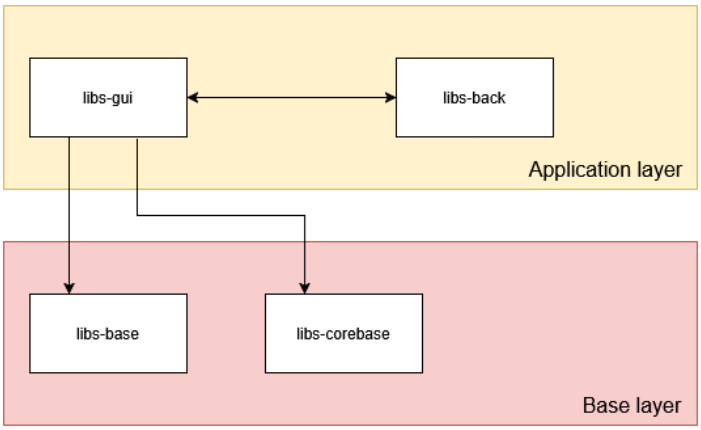

Figure 1: Conceptual top-level architecture from Assignment 1

Two important modifications were made to the architecture using information from the provided GNUstep project source and feedback to our first report. The first modification we made was adding the GNUstep runtime system—libobjc2 as a component in the base layer of the architecture. We had previously neglected to consider the runtime system in our first report as it was not explicitly stated to be in the scope of the project. Libobjc2 is the runtime system for GNUstep and is designed to be a drop-in replacement for the GCC runtime. It is responsible for setting up the execution environment and providing raw primitives supporting internal functions such as stack management, memory management and function closures that are invisible at the programming language level. Since every other component of GNUstep indirectly depends on libobjc2 when executing, we decided to place libobjc2 in the base layer. The second modification we made was adding Gorm as a component to the architecture. In our first report, we divided the analysis of GNUstep into two disjoint parts: libraries and development tools. As a consequence, we analyzed Gorm as an application built on top of the core libraries independently from the libraries themselves. This meant that it was not incorporated into the static top-level architecture. Our previous approach to interpreting GNUstep in terms of static parts is incompatible with the project source we are given to analyze. Thus we change our perspective in this report to view GNUstep at execution time. This means we start with an application and consider the transitive closure of dependencies within the GNUstep framework that arises during run time as components. In our case, we added Gorm as a component in the architecture in a new layer called userspace that sits above the application layer. The userspace layer represents applications that link against the GNUstep framework.

Apart from component addition, we also introduced new dependencies. We added libs-base and libs-corebase as dependencies for Gorm since both libraries are mostly interchangeable and provide abstractions for system facilities used in Gorm. Libobjc2 was added as a dependency for Gorm since the application runs on the GNUstep runtime system. Likewise, Gorm also depends on libs-gui for all of its GUI. The updated conceptual architecture consisting of all the aforementioned additions is shown below. This is the conceptual architecture we will use as a guide in our derivation of the concrete architecture and to compare against in the reflexion analysis.

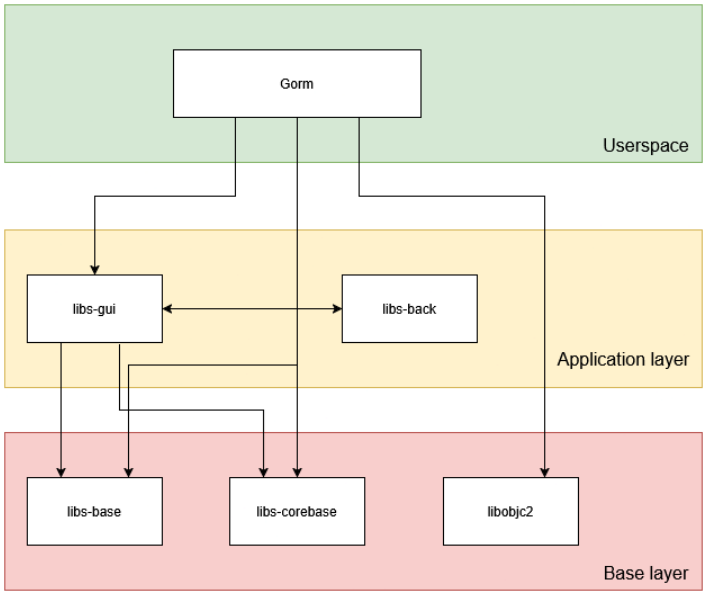

Figure 2: Updated top-level conceptual architecture

3 Concrete Architecture

We analyzed the supplied project source using the Understand tool to generate the following dependency diagram for the identified top-level components.

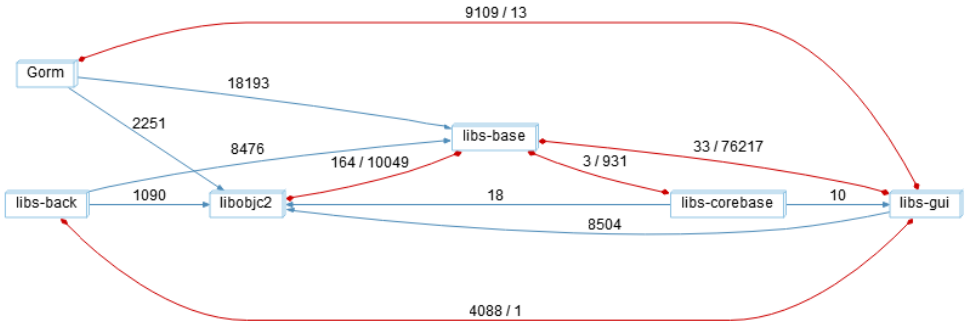

Figure 3: Top-level dependency diagram generated by Understand

The generated dependency diagram contains several spurious dependencies because Understand does not properly characterize the dependency relation between the implementation of an interface and the interface itself. For instance, the 13 function calls recorded from libs-gui to apps-gorm arise from Gorm implementing concrete subclasses of an abstract class in libs-gui or creating instances of a class in libs-gui that carries an interface type. Understand incorrectly determines that the latter component depends on the former since the object created eventually resolves to a call in the former with the callsite located in the latter. Other instances of false positives include the dependency from libs-back and libs-corebase on libobjc2. The function calls here come exclusively from the usage of the BOOL type which is defined in libobjc2. While this is a strict dependency at a technical level since Objective-C does not have a built-in boolean type at the language level, we felt that this dependency is nominal and does not present any real functionality. Hence we decided against its inclusion. After omitting the extraneous dependency and redrawing the dependency diagram with orthogonal edges, we obtain the following box-and-line diagram for the top-level concrete architecture.

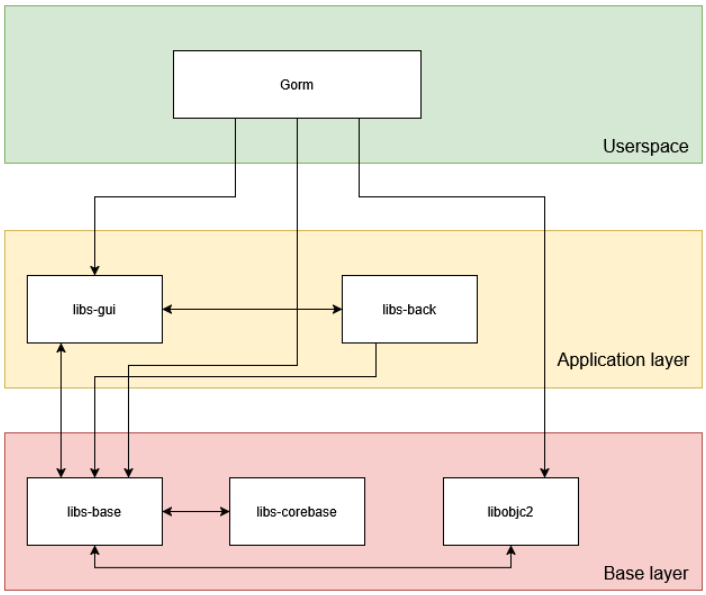

Figure 4: Top-level concrete architecture

The concrete architecture contains the same components as in our updated conceptual architecture but many dependency relations we did not anticipate.

4 Reflexion Analysis

In the previous sections, we have completed the first two stages. To understand the divergences between the conceptual and concrete architecture, we examined the source code involved in the dependencies and the relevant Git history.

Base Layer

libs-base $\longleftrightarrow$ libs-corebase

This bidirectional dependency was completely unexpected by us as we thought that libs-base and libs-corebase were largely independent implementations of FoundationKit in the OpenStep specification in different languages. In retrospect, the forward direction of this dependency is clear due to the fact libs-corebase supports the toll-free bridging, a mechanism that allows for the interop of types defined in libs-corebase to be used as if they were the corresponding type in libs-base [1]. Naturally, this requires knowing the types in libs-base. The reverse direction appears for expediency. The three function calls as seen in the dependency diagram correspond to importing and using implementation of select interfaces in libs-corebase that are not implemented in libs-base. This is likely a remnant of refactoring legacy code. In addition, we discovered upon further reading on the design of both libraries and documentation that, unlike Apple's Cocoa API which is built on top of Core Foundation, the counterpart to Cocoa (libs-base) is not built on top of libs-corebase and progress seems to have stagnated since 2020 due to lack of interest. Furthermore, the developer FAQ section for GNUstep suggests avoiding libs-corebase in favour of libs-base's object-oriented APIs [2].

libs-base $\longleftrightarrow$ libobjc2

We missed this bidirectional dependency as we thought libs-base only contained static data structures. While this is true to a large extent, we forgot about internal functionalities such as memory management, RPC and connections liberally used in libs-base which are implemented by the runtime system. This results in the forward dependency from libs-base to libobjc2. The reverse direction is interesting because a nontrivial portion of the runtime system support for objects specific to libs-base resides in the Objective-C subdirectory within libs-base itself. We could not find much information on why the source code is organized this way.

Application Layer

libs-gui $\longleftrightarrow$ libs-base

The forward direction of this dependency was included in the proposed conceptual architecture and well understood. The reverse direction surprised us as it breaks an invariant of the layered architecture that dependencies should propagate downwards from the topmost layer. We examine the call sites identified by Understand to figure out why this direction of the dependency relation materialized. All call sites reported by Understand turned out to be macro variable definitions that configure the runtime system features. For instance, in GSNibLoading.m, the variable `EXPOSE_NSKeyedUnarchiver_IVARS` is defined via a `#define` directive. This variable is checked in NSKeyedArchiver.h to trigger the conditional compilation

of parts of the NSKeyedArchiver object through the preprocessor. For this particular case, the variable `EXPOSE\_NSKeyedUnarchiver_IVARS` causes private instance variables of the NSKeyedArchiver object to be exposed [3]. Other instances include the definitions of `GSI_ARRAY_NO_RETAIN` and `GSI_ARRAY_NO_RELEASE` which are optimization flags that alter the automatic memory management of array objects in an autorelease pool. Thus, we conclude that the dependency from libs-base to libs-gui is spurious.

libs-gui $\longrightarrow$ libs-corebase

This dependency is included in the proposed conceptual architecture but absent in the extracted concrete architecture. Like the dependency between libs-base and libs-corebase, this discrepancy stems from our incorrect assumption about the role of libs-base and libs-corebase. As discovered in our prior investigation, libs-base appears to be the de facto main standard library in the GNUstep framework, with libs-corebase being a legacy artifact. Based on the history of the libs-gui Git repository, libs-corebase never became a dependency of libs-gui since its inception. Information regarding design decisions was sparse, and due to the wide timespan of the project, it was infeasible to conduct a meticulous reading of the source as it evolved. Nonetheless, we hypothesize that this dependency is absent because the GNUstep developer wanted to push for an object-oriented ecosystem, which is difficult to retrofit onto libs-corebase.

Userspace

apps-gorm $\longrightarrow$ libs-corebase

This dependency is included in the proposed conceptual architecture but absent in the extracted concrete architecture. As with our investigation into the missing dependency from libs-gui to libs-corebase, the history of the libs-gui Git repository revealed that libs-corebase was never a dependency of Gorm. We hypothesize the same rationale for the dependency from libs-gui to libs-corebase.

Overall, we found several notable divergences between our proposed conceptual and concrete architecture. Despite these differences, the extracted concrete architecture reaffirms our hypothesis that GNUstep mainly follows a layered architecture style. We note that a key invariant for layered architecture is the unidirectional flow of dependencies which is satisfied in our case. However, this does not mean that our proposed boundaries for the layers are completely accurate. It provides strong evidence that there is a hierarchy among the components. Another supporting evidence is the abundance of reuse of lower-layer functions in higher layers. The concrete architecture of the graphical components follows an implicit invocation architecture. AppKit as prescribed in the OpenStep specification strongly emphasizes delegation and message exchange. This is apparent in the implementation of GNUstep as almost all controls in libs-gui work by registering and responding to events. In terms of the implicit invocation model, the central notification center initiated on startup corresponds to the broadcast system, and the controls are the modules whose procedures are invoked. These observations provide evidence that the graphical parts of GNUstep follow the implicit invocation architecture.

5 Second Level Subsystem

5.1 Conceptual Architecture

The second level subsystem we chose to analyze for this report is Gorm. We use the proposed conceptual architecture for Gorm in our first report shown below as the base conceptual architecture to build on.

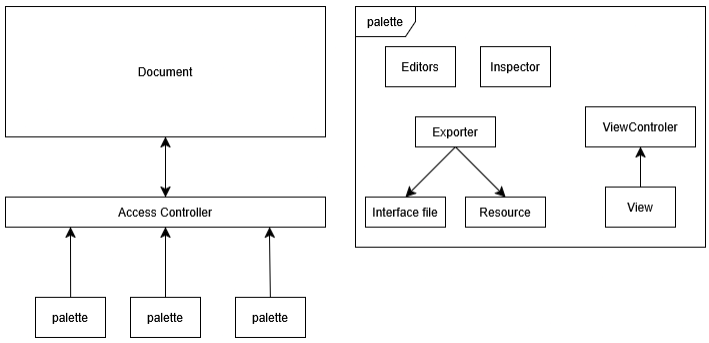

Figure 5: Conceptual architecture for Gorm from Assignment 1

In our original report, we had a very rudimentary understanding of the organization of Gorm. Before conducting a thorough analysis of Gorm's source code, we wanted to obtain a good idea of how GUI applications are developed using the OpenStep model. Due to the lack of documentation pertaining specifically to GNUstep, we used Apple's Cocoa documentation as a proxy. We found that Gorm appears to follow a standard document-based app architecture which is an instance of the model-view-controller design pattern. We give a summary of the main components of this architecture below.

A document in OpenStep/Cocoa terminology is a body of information that is/can be stored somewhere. This can be some lines of text or arbitrary binary data. A document is represented by a custom subclass of the `NSDocument` abstract class. The `NSDocument` class provides functions to read/write data that you implement. Each `NSDocument` has an associated `NSWindowController` which knows how to create a `NSWindow` object to render the document data. Access to `NSDocument` is handled with an `NSDocumentController` class [4].

The internals of Gorm contain many interacting components that would be infeasible to analyze in detail, so we decided to restrict our scope by omitting many auxiliary components inside the Palette component. We also group all plugins as a plugin rather than including them as separate components. In addition, we renamed the Document component to Application Document and the Access Controller to Document Controller. We also added a new component called Window Controller that oversees the rendering of the main document. Moreover, we added forward dependencies from the Document Controller to the Application Document, and from the Application Document to the Window Controller. Lastly, the importer and exporter modules originally placed as a sub-component in palettes are lifted to top-level components that depend on the Application Document as we realized import/export functionality is handled centrally.

The updated conceptual architecture containing all the aforementioned modifications is shown below. This is the conceptual architecture we will use as a guide in our derivation of the concrete architecture and compare against in the reflexion analysis.

5.2 Concrete Architecture

Using the Understand tool, we analyzed the supplied project source code to generate the following dependency diagram for the identified components of Gorm.

By redrawing the dependency diagram with orthogonal edges, we obtain the following box-and-line diagram for the top-level concrete architecture.

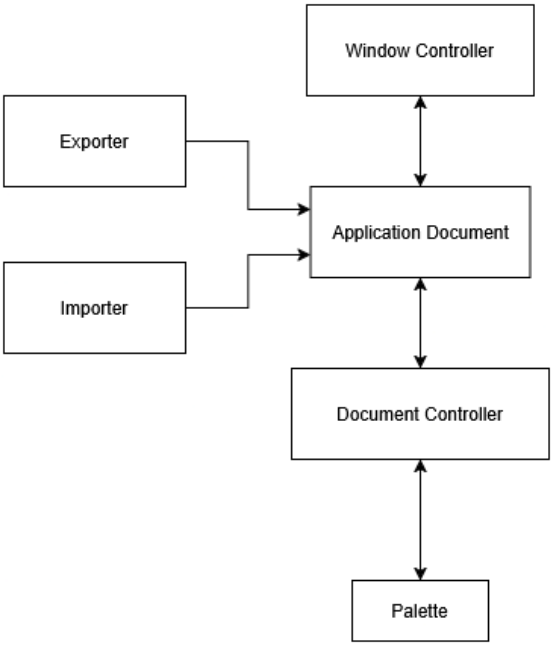

Figure 6: Updated conceptual architecture for Gorm

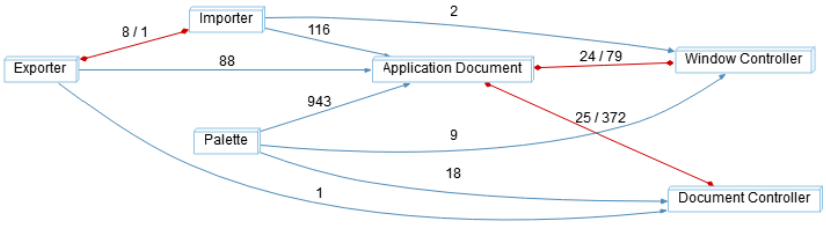

Figure 7: Dependency diagram of Gorm generated by Understand

The extracted concrete architecture was more nuanced than we had envisioned. In particular, many discrepancies are the result of a fundamental misunderstanding of how Gorm works. There were no absences observed in the concrete architecture but significant divergences. We outline the new dependencies below.

From Importer, there is a bidirectional dependency to Exporter and an unidirectional dependency from Importer to Window Controller. From Palette, there is a dependency to the Application Document and Window Controller. Lastly, there is a bidirectional dependency between the Window Controller and Document Controller.

5.3 Reflection Analysis

In the previous sections, we have completed the first two stages of reflexion analysis. To understand the divergences identified between the conceptual and concrete architecture of Gorm, we examine the source code involved in the dependencies and relevant documentation.

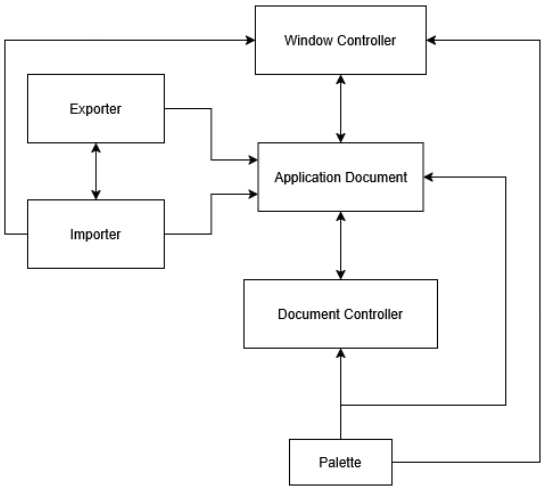

Figure 8: Concrete architecture for Gorm

Exporter ⟷ Importer

We missed this bidirectional dependency as we only viewed the components superficially and neglected to consider how serialization would work. It is evident that to import/export a project, we need to access the underlying document model. However, it is also necessary for the importer and exporter to cooperate such that their formats and conventions should agree. This forms the bidirectional linkage between these two components.

Importer ⟶ Window Controller

This dependency was unexpected as it bypasses the central knowledge source and is incompatible with our understanding of rendering the main document as a layered process. The dependency is realized by the setup of a window object in `GormXibWrapperLoader.m` and `GormNibWrapperLoader.m`.This suggests the loaders themselves are responsible for creating the window object rather than going through the Application Document or Document Controller. This aligns more closely with the object-oriented architecture style, thus our assumption that Gorm strictly follows a repository architecture may be incorrect.

Palette ⟶ Window Controller

This dependency was also unanticipated based on our assumption that Gorm strictly follows a repository architecture style. Following the source code revealed that the dependency stems from window inspect palettes that modify the properties of the windows. From this, we infer that the flow of data goes through the Window Controller first before appearing in the Application Document.

Palette ⟶ Application Document

This dependency consists of many function calls by various inspectors to access information about the main document. This is interesting because we assumed that inspectors go through the Document Controller. This dependency contradicts that assumption and suggests that the components Window Controller and Document Controller may be part of the Application Document.

Overall, we found significant divergences between our proposed conceptual and concrete architecture. Some

of these divergences arose from an incorrect working model of Gorm while others correspond to dependencies that were not obvious at a conceptual level. The extracted concrete architecture only partially agrees with our hypothesis that Gorm follows the repository architecture style. Although the Application Document is the shared blackboard data structure in this context, some computations and updates completely bypass it. Notable examples are the bypass of the Application Document seen in both Palette and Importer. These examples show that Gorm may partially follow an object-oriented architecture as these components directly identify each other to invoke methods. Another supporting evidence is that the three components, Application Document, Window Controller and Document Controller, hide their data representation from other components and only present an abstract interface for manipulating said data which is an important invariant of the object-oriented architecture style. These observations provide evidence that Gorm partially follows an object-oriented architecture. Thus, we conclude Gorm follows a hybrid architecture style composed of elements of both a repository architecture and an object-oriented architecture.

6 Use Cases

In this section, we present two use cases and their corresponding sequence diagrams that illustrate the derived concrete architecture for GNUstep as a whole and Gorm.

Before we delve into the use cases, we establish a common convention for the graphics used in the sequence diagrams.

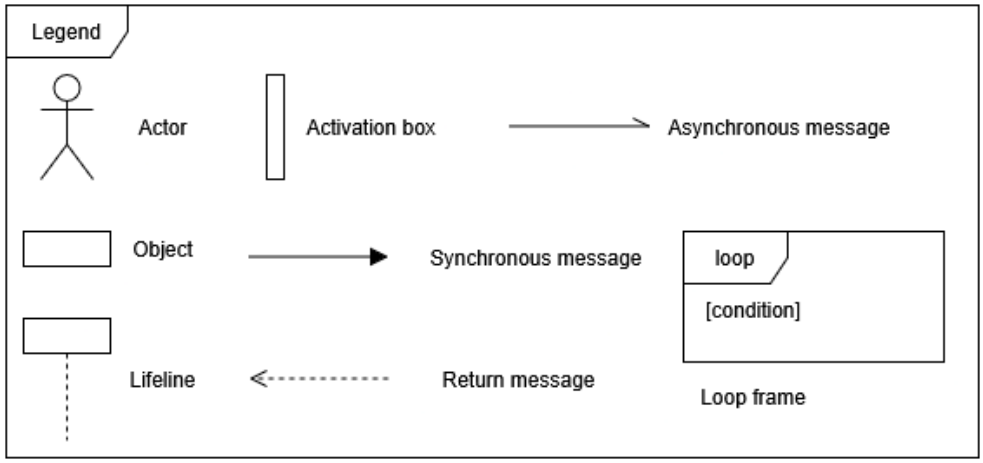

Figure 9: Sequence diagram legend

Our first use case concerns the loading of an interface (.nib) file through Gorm. This use case illustrates the activation of the Importer component and how deserialization occurs. Starting off, the document controller needs to create a new document housing the object and connectors to be loaded. The document then calls the loadFileWrapper method which eventually resolves to a dedicated loader. In our case, the loader being called is GormNibWrapperLoader.m. The loader calls NSKeyedUnarchiver to start unpacking objects from the saved file. Next, it loops over all the control objects returned by the unarchiver, parses their properties and configures them accordingly before attaching them to the main document. The same procedure occurs

for connectors as well, with the source and destination being linked to objects constructed in the earlier phase. Finally, when all classes in the interface file have been loaded, the main document issues a call to its attached window controller to retrieve its document window. This implicitly loads the window and displays it on the screen. The user then sees the workspace being populated by the loaded interface objects and connectors. Here is the sequence diagram that shows the specific data and control flows.

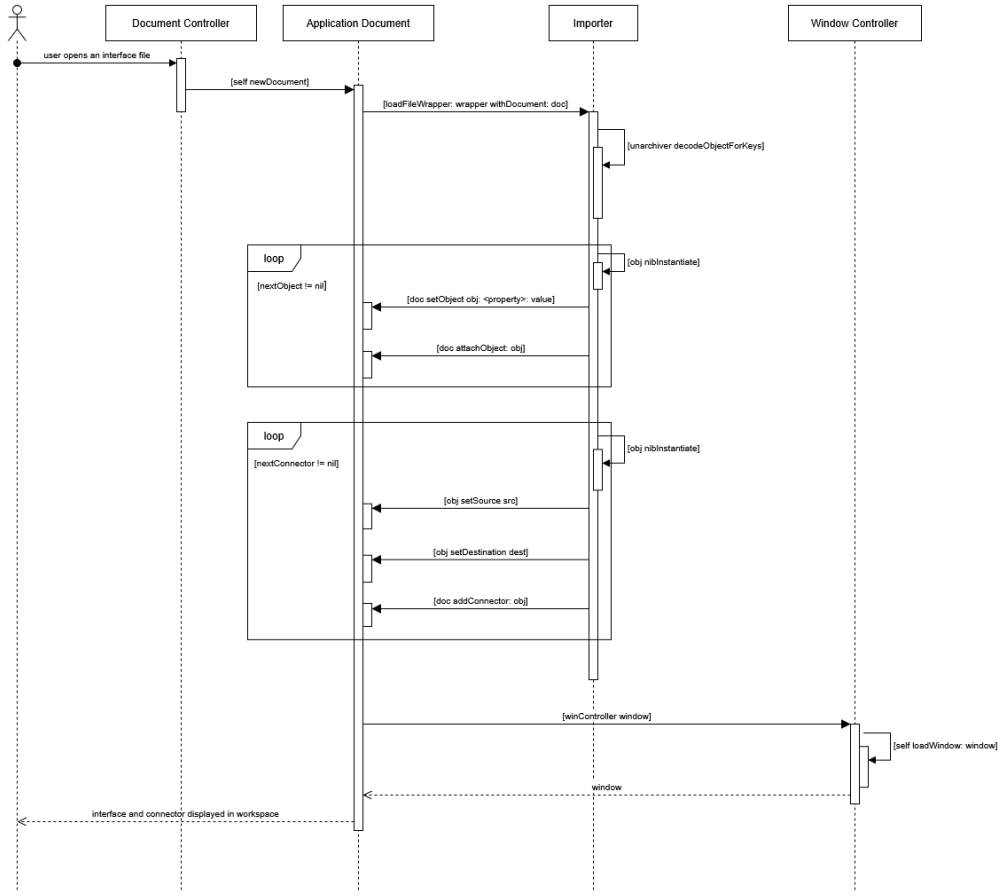

Figure 10: Sequence diagram for use case 1

Our second use case illustrates the data flow between top-level components. For demonstration purposes, consider a simple GNUstep application called "ButtonDemo". This application presents a button to the user. Upon clicking the button, the application performs a remote procedure call to a remote server that generates a random number. The button's title is then updated to show the returned random number. This use case illustrates how message passing between objects activates components in GNUstep. Here is the sequence diagram that shows the specific data and control flows.

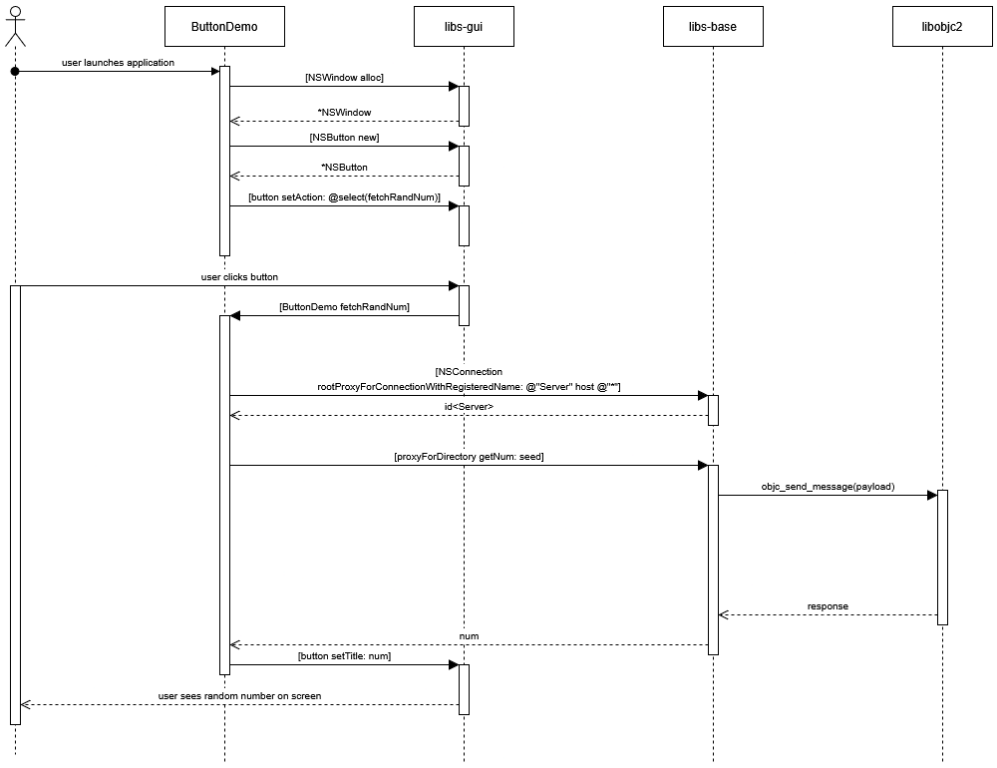

Figure 11: Sequence diagram for use case 2

7 Lessons Learned

While exploring the concrete architecture of GNUstep, our team faced several issues that hindered our progress and placed us behind schedule. Understanding the concrete architecture of GNUstep required us to delve into the source code written in Objective-C, a programming language we were all unfamiliar with. Using the Understand software developed by SciTools also caused some complications. Although Understand is incredibly useful for analyzing a code base as complex as GNUStep, its interface and options were overwhelming. Despite the tutorial providing a useful guide to get started, a few group members still struggled to use and navigate the program effectively. Fortunately, our team would actively help to sort things out and aid team members by guiding them through the process step by step in an easy-to-understand manner.

Another issue we ran into was confusion regarding what roles each member should do since they were not clearly assigned. A lack of team meetings due to conflicts and schedules also exacerbated this problem since no one was sure who was supposed to do what which led to disorganization and slow progress. We decided to work around this issue by making sure that the few team meetings we had were productive and worked on the sections that needed team cooperation during these meetings while assigning tasks that could be accomplished asynchronously to be worked on outside of meeting times.

8 Conclusion

By analyzing the source code of GNUstep with the Understand tool, we were able to derive its concrete architecture. This approach allowed us to discover hidden dependencies and examine in detail the different subsystems of the concrete architecture. We discovered dependencies in the concrete architecture that were not present in the original conceptual architecture, such as the bidirectional dependencies between libs-base and libs-corebase. These findings reveal that the system is more coupled than we originally thought. Through reflexion analysis of the top-level subsystems, we confirmed that GNUstep employs a layered architecture with elements of the implicit invocation architecture for its graphical subsystems. Furthermore, our use cases illustrate the derived concrete architecture for GNUstep and Gorm, demonstrating interface loading and data flow through object deserialization and message passing. These use cases showcase how GNUstep manages object interactions, updating the UI through structured control and data flows.

In preparing this report, we chose the second-level subsystem Gorm and examined its role, structure and dependencies within GNUstep. In contrast to our expectations, the dependency on libs-corebase was missing. However, Gorm still depends on libs-gui for its graphical interface and libs-base for its basic functions. Moving forward, we will bring our newfound knowledge of GNUstep's concrete architecture to propose potential enhancements in the next report.

9 Resources

9.1 Data Dictionary

GNUstep: A set of cross-platform object-oriented frameworks written in Objective-C for developing applications.
Libs-base: The fundamental library of GNUstep, providing basic functions and routines for data manipulation, network and file operations, datetime handling and other primitive system functionalities.
Libs-corebase: An alternative library, following Apple's Core Foundation framework, implemented in C.
Libs-gui: The graphical user interface library, following Apple's Core APIs, serves as the frontend part of the GNUstep GUI library.
Libs-back: The graphical user interface backend component, serves to draw graphical elements as well as managing windows.
Gorm: A graphical interface builder for GNUstep.
Libobjc2: The runtime system for GNUstep, designed as a drop-in replacement for the GCC runtime, sets up the execution environment and provides raw primitives for stack management, memory management and function closures, which operate behind the scenes at the language level.

9.2 Name Conventions

API - Application Programming Interface
GUI - Graphical User Interface
IDE - Integrated Development Environment
UI - User Interface

References

[1] Apple Inc. “Coca Encyclopedia: Toll Free Bridging,” [Online]. Available: https://developer.apple.com/library/archive/documentation/General/Conceptual/CocoaEncyclopedia/Toll-FreeBridgin/Toll-FreeBridgin.html Accessed: 03/07/2025.

[2] GNUstep maintainers. “GNUstep FAQ,” [Online]. Available: https://www.gnustep.org/resources/documentation/User/GNUstep/faq.pdf Accessed: 03/07/2025.

[3] GNUstep maintainers. “GSVersionMacros,” [Online]. Available: https://github.com/gnustep/libs-base/blob/9c6bd9ed97c835a14d53abe2a92fe5443b513def/Headers/GNUstepBase/GSVersionMacros.h#L276 Accessed: 03/07/2025.

[4] Apple Inc. “Devloping a Document-Based App,” [Online]. Available: https://developer.apple.com/documentation/appkit/developing-a-document-based-app?language=objc Accessed: 03/07/2025.