GNUstep: Concrete Architecture Overview

March 14th, 2025

Group 17: Architecture Addicted Aficionados

Group Members:

Sungmoon Choi	22sc@queensu.ca
Dayna Corman	21drc13@queensu.ca
Michael Cox	21mwc3@queensu.ca
Aydan Macgregor	21ajm32@queensu.ca
Samantha Mak	21yssm@queensu.ca
Kylie Wong	21kycw@queensu.ca

Abstract

This report will analyze the concrete architecture of GNUstep, an open-source software framework used to develop desktop applications. The report will be a continuation of our previous assessment of the conceptual architecture of GNUstep and will delve further into the architecture that was mentioned in the previous report. The main purpose of this report is to showcase the concrete architecture of the main components in GNUstep and provide an in-depth view of one of those main components. We will explain our derivation process for sourcing and analyzing information about GNUstep. The report will include a look into the top-level concrete architecture, showcasing the main components, their interactions, and dependency diagrams. We have also chosen Gorm as the main component to focus on to research further. This section of our report will include an analysis of the components, inner component interactions, a sequence diagram with a notable use case, and design patterns within the subsystem. A reflexion analysis for the high-level architecture of Gorm and the lower subsystem level of Gorm will be performed to identify discrepancies between our previous report and our current report. Sequence diagrams will also showcase a lower-level architecture of GNUstep and Gorm with two common user use cases shown. Finally, we will present our conclusion and lessons learned from the research and creation of this report.

1. Introduction

In our report, we will examine the concrete architecture of the open-source framework for desktop development, GNUstep. We have learned much from our previous report on the conceptual architecture of GNUstep, much of which is due to our ability to view the architecture at a much more granular level. Moving from the overview of documentation to viewing the source code has provided powerful insight into the concrete architecture of the framework. To parse and make sense of the source code we used a tool called Understand, which provides a wealth of ways to view the code base, giving us a more comprehensive perspective on GNUsteps architecture. From this, we were able to attain much of the information in the report and improve our knowledge of GNUstep.

Our report consists of five main sections. The derivation process will demonstrate how we came to understand GNUstep's concrete architecture with our use of the software tool Understand. The second section delves into the components and their concrete architecture and interactions with other components. Diagrams will also be provided to showcase concrete interactions with different components. The third section of the report explores the inner architecture of Gorm, our chosen component to analyze in more detail. We will showcase the components and interactions within Gorm and provide sequence diagrams for a use case of Gorm. Furthermore, we will also present notable design patterns of the architecture used in Gorm. The next section will focus on a high-level reflexion analysis of GNUstep as a whole, using Understand to perform the analysis and presenting a diagram of the layered architecture from our conceptual report and our concrete layered architecture to compare the two. Later in this section, we will also provide a reflexion analysis for Gorm specifically by comparing the conceptual and concrete designs. This analysis also focuses on interactions and dependencies of Gorm. The final section of the report includes two sequence diagrams that demonstrate simple use cases of GNUstep, and Gorm. These sequence diagrams will also show more of the concrete architecture for each use case.

1.1 Derivation Process

The main tool used for the derivation of our architecture was the software tool Understand, specifically the dependency graphs provided by Understand. The Dependency graphs were integral in the derivation of the concrete architecture. This tool allowed us to not only see the overall view and interactions between the major components, but we could also explore interactions inside of our chosen component, Gorm. These graphs provided valuable insight into what may not have been mentioned in the documentation we viewed for our conceptual architecture, allowing us to refine our interpretation of GNUstep and confirm some of our beliefs. For example, the layered architecture we proposed in our conceptual architecture was close to the concrete architecture found in the dependency graphs. But with the dependency graphs, we also found interactions with libs-corebase which we added to our diagram, allowing us to improve our proposed architecture.

Further information was sourced from GNUstep documentation, public forums such as the Wikipedia page for GNUstep and GNUstep's repository to find additional information for various sections of the report, such as sequence diagrams and reflexion analysis. Tools such as git blame were used to find divergences and filter the commit history of GNUstep. Additional information was discussed and decided upon in group meetings from a combination of the various sources mentioned above and cited at the end of the report.

2. Top-Level Concrete Architecture

GNUstep's concrete architecture from the top level is organized mainly in a layered architectural style. It maintains a layered structure with clear hierarchical relationships between layers, but it deviates from the standard layered architecture in a few ways. Firstly, the libs-base module contains standard code that is used by all of the other modules, regardless of layer, and the libs-corebase module contains standard code that is sometimes used by other modules but is also exposed for user programs to use. However, GNUstep's architecture also contains several unexpected dependencies that complicate the overall architecture. [1]

2.1 Components

GNUstep is organized into five main components within the scope provided in the assignment. These components are as follows:

apps-gorm

Gorm is described as a clone of the "Cocoa (OpenStep/NeXTSTEP) 'Interface Builder'" [2] and allows a user to design and program the functionality of a Graphical User Interface which is given functionality and form by the rest of the GNUstep library. Gorm resides on the highest layer of the architecture, and aside from a very small amount of unexpected dependencies, it only depends on apps–gui, libs-corebase, and libs-base, and is not depended on by any other module. Gorm also depends on libs-base, fulfilling the role of a "shared code library", and libs-gui, which provides the UI elements that Gorm allows a user to construct with. [1]

Most of this functionality is contained within the GormCore subdirectory, which includes the core functionality and methods for Gorm to use, and the InterfaceBuilder subdirectory, which contains the methods and classes to allow the user to create an interface. [1]

libs-gui
Libs-gui is described as a “library of graphical user interface classes” [2]. It contains the user-facing functionality of all of GNUstep’s user interface elements. It resides in the second layer of the architecture, where it is depended on by user applications, like Gorm, and contains hooks to operating-system-specific code that is implemented in libs-back. libs-gui also depends on code in libs-corebase and heavily depends on code from libs-base. [1]

Libs-gui is divided further into subdirectories that contain; utilities like speech synthesis, text converters, color pickers, and then the Source subdirectory that contains definitions for the majority of the widgets. libs-gui also contains a couple of definitions that are referenced by libs-base, which creates some unexpected dependencies. [1]

libs-back
Libs-back is described as a “back-end component for the GNUstep GUI Library.” [2] It implements all of the operating-system-specific calls to the windowing system to actually make the UI elements work on the computer GNUstep is run on, allowing for programs created with Gorm to be platform-independent from the perspective of the developer. libs-back falls on a layer under libs-gui, and is only dependent on the class structures defined in libs-gui, and heavily uses code from libs-base. [1]

Libs-back contains subfolders for headers and subfolders for source code, each subdivided further by windowing system, however, the “art” subdirectory is not sorted properly and has the majority of its header files inside the source directory. Every other subdirectory properly follows this organizational rule. [1]

libs-corebase
Libs-corebase is described as a “library of general-purpose, non-graphical C objects.” [2] It implements the non-graphical functionality of Apple’s CoreFoundation framework, something that is plain to see in the naming convention of the files and folders that make up the module. A few of these methods are used in libs-gui, but the majority of them are used in libs-base. These methods are also exposed so that user programs can implement them. [1]

The dependency relationship between corebase and base is such that they both depend on each other a relatively equal amount, and as such are very bidirectional.[1]

The folder structure of libs-corebase is such that the headers are stored under a Headers/CoreFoundation subfolder, again drawing attention to the inspiration for the methods, in which the source folder just contains the source code to all the aforementioned headers. [1]

libs-base
Libs-base is the major core library that every other module references. This module is firmly set aside from the standard layered architecture and is the main reason why GNUstep cannot be described solely as being a layered architecture with no more distinction. [1]

Libs-base contains subdirectories for mac os x related code, which is referenced by libs-back, configs, and then an absolute mass of code in the source and headers directories that form the

backbone of most of the other functionality of GNUstep. It also cross-depends on libs-corebase as well.[1]

2.2 Component Interactions

Here are some example interactions between the aforementioned components:

Implementation of Features between libs-gui and libs-back:
In the *w32_create.m* file, libs-back depends on the NSWindow object framework defined in libs-gui, but uses that to initialize the NSWindow object in a way that depends on win32, like creating the window icon or just asking the windowing system to create the window in the first place. This is an example of how GNUstep is able to handle many different windowing systems to allow for cross-platform deployment. [1]

Use of libs-gui in apps-gorm:
Inside of GormCore, the critical functionality of Gorm is defined. Inside the *GormBoxEditor.m* file, we can see that defining these User Interface elements requires referencing other, smaller parts of code from all over, including the definitions of a "Box" element from libs-gui and many other methods from lib-base that encode simple data types, as seen in the following example. [1]

Use of Lib-Base everywhere:
With the paradigm of being a shared library, almost any time a "string" is used, which is a data type allowing for the storage of text, NSString must be imported from libs-base. NSString is such an essential class in an environment where text is being displayed to the end-user, so NSString is referenced almost 5.7 thousand times across the rest of GNUStep.[1]

2.3 Dependency and Architecture Diagram

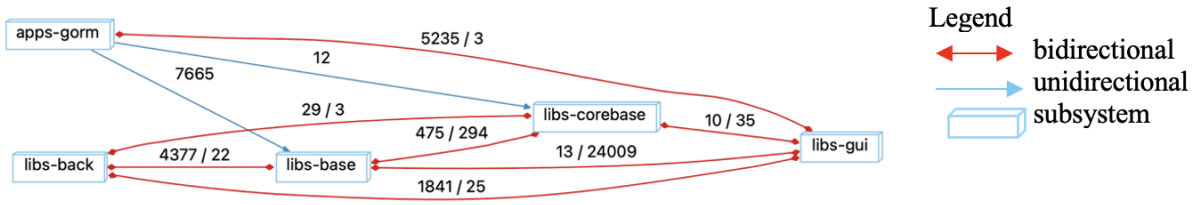


Figure 1.1: Top-Level dependency graph of GNUStep created using Understand [1]

In Figure 1.1, each module is represented by a box, and blue arrows represent dependance on, as an example, libs-back depends on libs-gui, with 1841 references that Understand detected. Red bidirectional arrows highlight mutual dependencies.

3. Inner Concrete Architecture of Subsystem apps-gorm

Gorm, a subsystem for GNUstep, is an interface builder that allows developers to design graphical user interfaces for GNUstep applications. Gorm comprises different components to function consisting of GormCore, InterfaceBuilder, and GormObjCHeaderParser frameworks with supporting plugins and tools. Gorm employs a mix of object-oriented and publish-subscribe architectural styles:

Object-oriented: Components are structured like objects with their responsibilities where they interact through method calls.

Publish-subscribe: Components interactions are event-based and react according to the current state.

3.1 Components and Dependencies

GormCore: The central component of Gorm, responsible for managing the application state and handling .gorm files (loading, saving and parsing files).

Below in Figure 2.1 is the dependency graph of GormCore. GormCore depends heavily on libs-base, especially in GormClassManager, GormInspectorsManager, and GormPalettesManager. [1] This is because libs-base is the foundation for all other libraries, and GormCore requires this foundation to interact with libraries, like libs-gui. libs-base provides the fundamental building blocks including system interaction, object-oriented design, data management, as well as file system tasks that are required for GormCore. [3] GormCore also depends on libs-corebase for handling object management, string encoding, and file handling, particularly for managing XML files. [2]

GormCore and libs-gui have a bidirectional dependency. libs-gui creates, modifies, and manages UI components, without libs-gui, GormCore wouldn't be able to manipulate or render UI elements. Therefore, GormCore depends heavily on libs-core. libs-gui also has 3 dependencies on GormCore, specifically overriding three functions in NSTabViewController.m. This creates a circular dependency, where GormCore relies on libs-gui for UI functionality, while libs-gui requires GormCore for certain behaviors in NSTabViewController. [1]

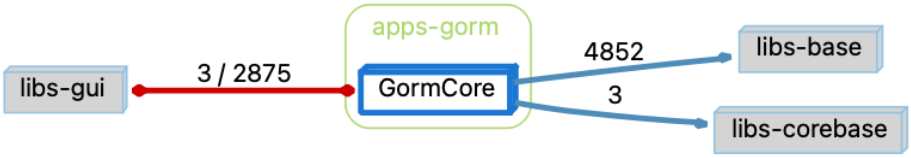


Figure 2.1: Dependency graph of GormCore in apps-gorm, generated using Understood [1]

GormObjCHeaderParser: Header file parser to read Objective-C.h files to extract class and method definitions used to understand the structure of custom classes and integrate them into Gorm's UI design process. GormObjCHeaderParser depends on libs-base because it requires fundamental Objective-C utilities such as string processing and file management to parse Objective-C header files. These functionalities enable Gorm to analyze, extract, and store class properties. Without this dependency, Gorm would not be able to import custom classes, limiting its ability to integrate user-defined components into the interface builder. [4]

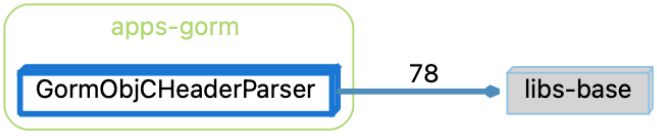


Figure 2.2: Dependency graph of GormObjCHeaderParser in apps-gorm, generated using Understood [1].

InterfaceBuilder: The graphical user interface for designing and modifying .gorm files, allow users to drag and group UI components and property inspectors to modify attributes. In Figure 2.3, InterfaceBuilder relies on libs-base in supporting InterfaceBuilding in UI component management, customization, and extensibility. libs-base provides data storage,

serialization, and resource management for Gorm to manage UI components effectively. InterfaceBuilder also depends on libs-gui to handle window management, rendering user interactions, and event handling.

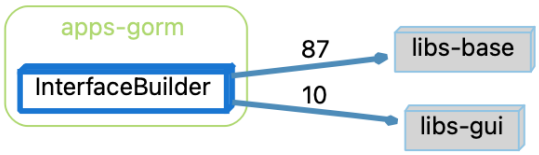


Figure 2.3: Dependency graph of InterfaceBuilder in apps-gorm, generated using Understood [1].

Applications: Standalone applications that complement Gorm's functionality. As shown in Figure 2.4, Applications in Gorm rely on libs-base and libs-gui because they are essential for building and running GNUstep-based graphical applications.

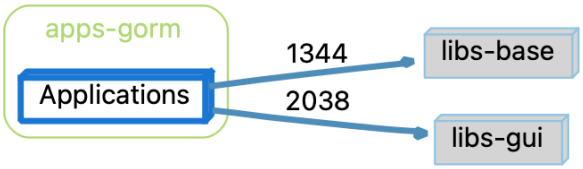


Figure 2.4: Dependency graph of Applications in apps-gorm, generated using Understood [1].

Plugins: Plugins are modular extensions that enhance their functionality by adding support to different UI formats, data models, and additional interface features. They allow Gorm to be flexible and extensible without modifying its core architecture. Therefore, as shown in Figure 2.5, it depends heavily on libs-base for its core Objective-C utilities for data handling and system operations, with key features used in file operations for Nib/Xib processing. [1] libs-corebase provides the string encoding for file format in parsing and exporting Nib/Xib files. Plugins also depend on libs-gui for UI components and rendering support for handling graphical elements.

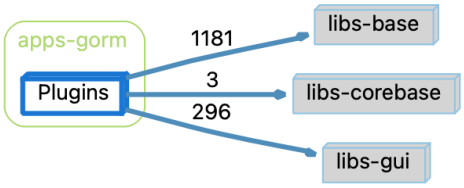


Figure 2.5: Dependency graph of Plugins in apps-gorm, generated using Understood [1].

Tools: This component provides various utilities and tools that assist in the design and development process. In Figure 2.6, Tools relies on libs-base for object creation, memory management (alloc/init), and user settings (NSUserDefaults). Tools also depend on libs-corebase for its string encoding, which is crucial for saving and loading interface files. libs-gui provides the UI components and event handling for interacting with the graphical environment in Gorm.

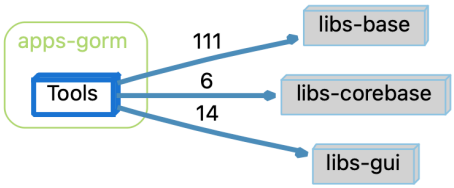


Figure 2.6: Dependency graph of Tools in apps-gorm, generated using Understood [1].

3.2 Component Interactions

GormCore ↔ InterfaceBuilder:
GormCore provides the foundational functionality for the InterfaceBuilder to create and edit UI elements and InterfaceBuilder depends on GormCore for managing the document. For example, when a user drags a button from the palette onto the canvas from InterfaceBuilder, GormDocument.addObject in GormCore is called to add the button.

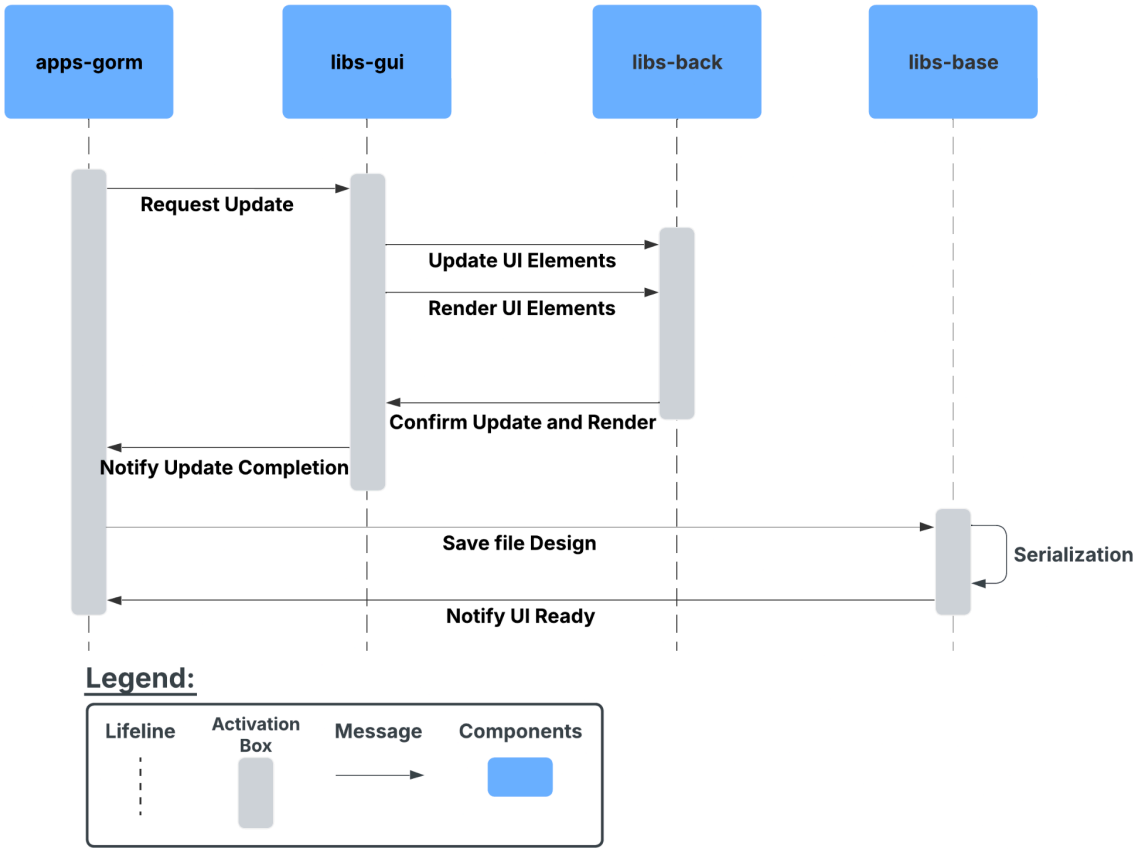
GormCore ↔ GormObjCHeaderParser:
GormObjCHeaderParser parses header files and provide method information to GormCore, such as in GormObjCHeaderParser.m, it parses an Objective-C header file and extracts class and method information which is then registered in GormClassManager.m in GormCore, being used for UI.

GormCore ↔ Plugins:
Plugins interact with GormCore to extend its functionality where in GormPluginPalette.m, it manages and loads the available UI elements into the palette where it is added to the document's hierarchy in GormDocument.m.

GormCore ↔ Tools:
Tools interact with GormCore to manipulate the UI or generate code. For example, in GromClassManager.m, tools provided the generated Objective-C code based on the UI design to GormDocument.m where it retrieved the UI hierarchy from the document.

3.3 Sequence Diagram of Gorm

Gorm sequence diagram of the user making a UI change and saving the file.



3.4 Notable Design Patterns

- **Model View Controller:** Gorm acts as a view builder that shows the user the UI while we keep the model and controller separate as it works in Objective-C. By separating the different UI aspects, Gorm can promote flexibility, maintainability, and ease of testing in its subsystem.
- **Decoder Pattern:** Gorm files store the UI information in a serialized format that must be decoded at runtime to reconstruct the UI components and connections without hardcoding. This allows for UI state preservation, making saving and loading Gorm files easier.

4.1 Reflexion Analysis for High-Level Architecture

4.1.1 Concert Implementation – Static Reflexion Analysis by SciTools Understand

At high level, the dependency diagram above (Figure 1.1) can be represented by a layered architecture as shown in Figure 3.1 at left side below. For comparison purposes later, we include the layer diagram of conceptual design (control flow) from Assignment 1 in Figure 3.2 at right side below.

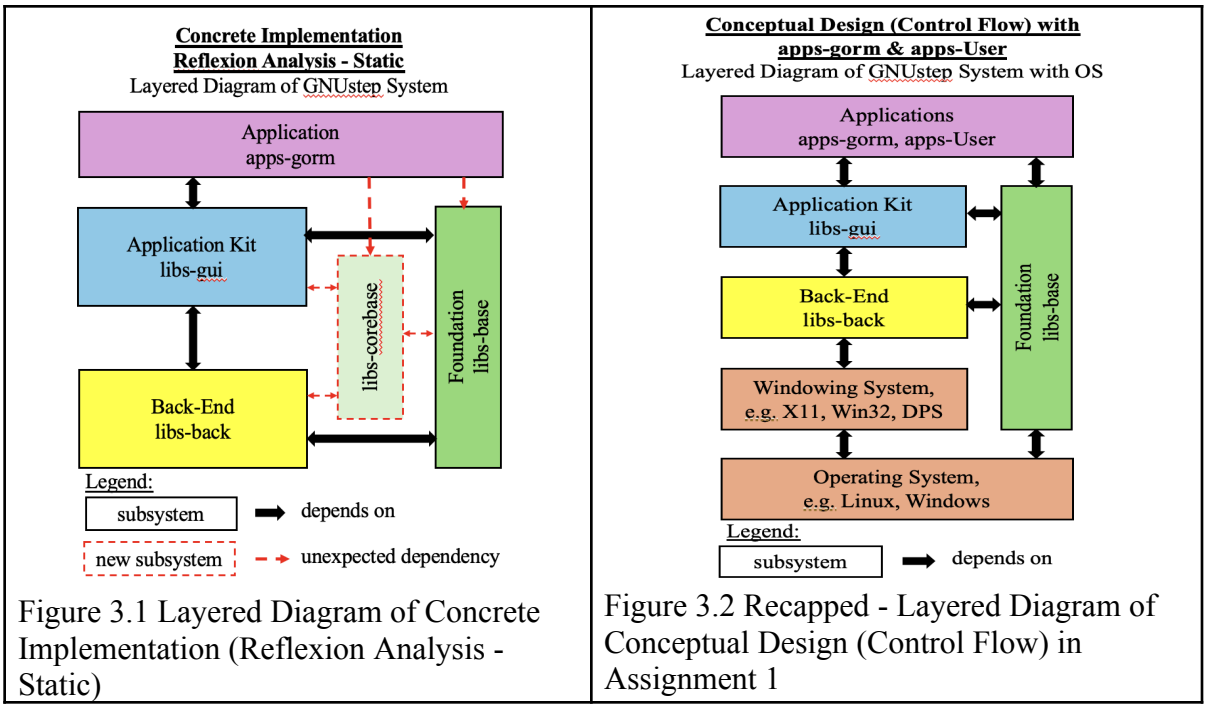


Figure 3.1 Layered Diagram of Concrete Implementation (Reflexion Analysis - Static)

Figure 3.2 Recapped - Layered Diagram of Conceptual Design (Control Flow) in Assignment 1

From the static reflexion analysis above, we have verified that the concert implementation achieves at least one of GNUstep architecture objectives, i.e. to provide platform-independent GUI application development. This is shown in Figure 3.1 that the application apps-gorm at the top does not interact with the back-end libs-back.

4.1.2 Discrepancies and Rationale

With intensive studies at GNUstep documentation in preparing Assignment 1, there are not many discrepancies between conceptual design as in Figure 3.2, and concrete implementation as in Figure 3.1. There are only two main discrepancies found as described below,

Discrepancy #1: presence of libs-corebase
The conceptual design did not include libs-corebase before while the concrete implementation does.

Rationale: We have missed the relations, and entity, of libs-corebase.

There is minimal public documentation about libs-corebase. The key description is only "*The GNUstep CoreBase Library is a library of general-purpose, non-graphical C objects*". On the other hand, its functions look greatly overlapping with libs-base which is described in many documentations about its usages by other libraries.

So instead of roughly guessing how libs-corebase interacts with other libraries or putting it by itself without any relations with others in conceptual design, we have not included it in the conceptual design of our Assignment 1.

Discrepancy #2: directions of flow between apps-gorm and libs-base
In the conceptual design (control flow) of Figure 3.2, the flow between apps-gorm and libs-base is bidirectional as indicated in black. But the concrete implementation from the static reflexion analysis indicates the flow is unidirectional as indicated in red from apps-gorm to libs-base.

Rationale: We used difference mechanisms in creating conceptual design vs. concrete implementation. The conceptual design was based on GNUstep's dynamic runtime behaviours while the concrete implementation is generated by the reflexion analysis on static source codes.

SciTools Understand is very powerful in performing static analysis over a program's source codes but has limited knowledge on runtime behaviours of the program, e.g. advanced messages to an object via its object pointer with the object's class and methods determined at runtime.

The conceptual design in Assignment 1 shows runtime behaviour between apps-gorm and libs-base. This is described in the Use Case 3 of Section 4 Control and Data Flow of our Assignment 1 report. The libs-base's notification center invokes an event handler of a user defined object in any GNUstep's applications including apps-gorm and apps-User (a custom user application).

At runtime, a user-defined object in a GNUstep application (apps-gorm or apps-User) can register its interest in a notification with its selector method by names as string parameters to the notification center. Its selector method will be invoked by libs-base's notification center when the notification occurs in runtime. More details are described in Section 5 Advanced Messaging of "*Objective-C Language and GNUstep Base Library Programming Manual*".[3]

As shown below, line 1288 of libs-base's source file *NSNotificationCenter.m* contains the related codes, which the *NSNotificationCenter* invokes the receiver object's selector method with the notification as a parameter via the receiver's object pointer.

```
[o->receiver performSelector: o->selector
                  withObject: notification];
```

The conceptual design in Figure 3.2 includes such runtime behaviours to invoke methods in any GNUstep applications incl. apps-gorm or apps-User, and so results a bidirectional flow between apps-gorm & libs-base.

The reflexion analysis is static only and not aware of runtime behaviors, particularly with object pointers. Its result can only show a unidirectional flow from apps-gorm to libs-base.

So overall the reason behind this can be classified as using different mechanisms for conceptual design and concrete implementation analysis.

4.2 Reflexion Analysis on Subsystem apps-gorm

4.2.1 Conceptual vs. Concrete Architecture

The conceptual architecture of Gorm follows a modular, centralized system. GormCore acts as a central hub for UI management, and InterfaceBuilder, Plugins, Tools, and Applications interact with it. According to our dependency graph in Figure 3.2, our conceptual architecture suggests that there is a bidirectional dependency between Gorm and libs-gui for visual components, as well as Gorm and libs-base for core logic.

However, the concrete analysis of GNUstep suggests that Gorm only depends on libs-base, while the bidirectional dependency between Gorm and libs-gui remains unchanged, as seen

in Figure 3.1. Additionally, the introduction of libs-corebase in the concrete implementation has shifted some responsibilities from libs-base to libs-corebase, where Gorm now depends on libs-corebase, particularly for string encoding, which we originally expected to be handled by libs-base in our conceptual architecture. The dependencies on libs-base, libs-corebase, and libs-gui ensure data processing, encoding support, and UI rendering, making Gorm an essential part of GNUstep's interface builder. We will further discuss the divergence and absence of dependencies in the following sections.

4.2.2 Unexpected Dependencies

In our concrete architecture, we identified three unexpected dependencies on libs-corebase, which were not anticipated in our conceptual design. These dependencies are found in GormCore, Tools, and Plugins, suggesting that libs-corebase handles string encoding that was expected in libs-base. The following section analyzes how these dependencies were introduced, their historical context, and their impact on the system's architecture.

1. GormCore —(3)→ libs-corebase

There are two dependencies on libs-corebase in GormClassManager in using NSASCIIStringEncoding. This dependency was introduced by the lead developer of GNUstep Gregory Casamento on June 5, 2006. This is due to a merge from the NibCompatibility branch using NSUTF8StringEncoding and NSASCIIStringEncoding for handling the serialization and deserialization of class information. GormCore's GormXLIFFDocument also depends on libs-corebase, using NSUTF8StringEncoding, to handle XML-based data related to translation. [2]

2. Tools —(6)→ libs-corebase

Tools depend on libs-corebase for using NSUTF8StringEncoding, which is used in AppDelegate.m for processing tasks such as text encoding and conversion. The Git Blame Analysis in Figure 4.1 shows that this dependency was introduced by Gregory John on July 8, 2023, as part of a change that modified the output by incorporating NSUTF8StringEncoding. The switch to NSUTF8StringEncoding ensures explicit UTF-8 encoding, avoids extra metadata from NSLog, and provides more control over output formatting. [2][6]

```
(base) Kylie@Kylies-MacBook-Pro apps-gorm % git log --reverse -S "NSUTF8StringEncoding" --
Tools/gormtool/AppDelegate.m
commit f75d5d151be643c735e4aca794adcaadd4939f93
Author: Gregory John Casamento <greg.casamento@gmail.com>
Date:   Sat Jul 8 16:22:30 2023 -0400
    Remove warning from object editor, change output of plists so that they can be more easily parsed in
AppDelegate
```

Figure 4.1: Git history showing the first commit where NSUTF8StringEncoding was added to Tools/Gormtool/AppDelegate.m [2].

3. Plugins —(3)→ libs-corebase

Plugins depend on libs-corebase for using NSUTF8StringEncoding and NSASCIIStringEncoding, specifically in Xib and GModel. This dependency was introduced by Gregory Casamento on October 22, 2023, with the commit message: *"Move Plugins so that they link after GormCore is built since this is required on Windows, WIN32 does not*

support late linking" [2]. Originally, Plugins was part of GormCore, but it was later isolated into its module as part of a refactoring effort for better organization and build compatibility.

4.2.3 Reflexion Analysis on Divergence: apps-gorm → libs-corebase

In the concrete architecture analysis, we have seen that Gorm depends on libs-corebase for string encoding functionality. In the conceptual architecture we have proposed in A1, libs-corebase was not one of the components required, as we expected string encoding to be handled in libs-base, which has been the standard GNUstep Foundation library since its creation in 1994 by Andrew McCallum. However, the concrete analysis reveals that Gorm instead relies on libs-corebase, a library first introduced in 2010 by Eric Wasylishen. [2]

This divergence likely occurred due to historical code evolution and repository restructuring since libs-corebase was created much later than libs-base. libs-corebase handles core foundation functionality, such as managing memory, string processing, collection handling, and data storage, which has led to the dependencies in Gorm for string encoding. [5]

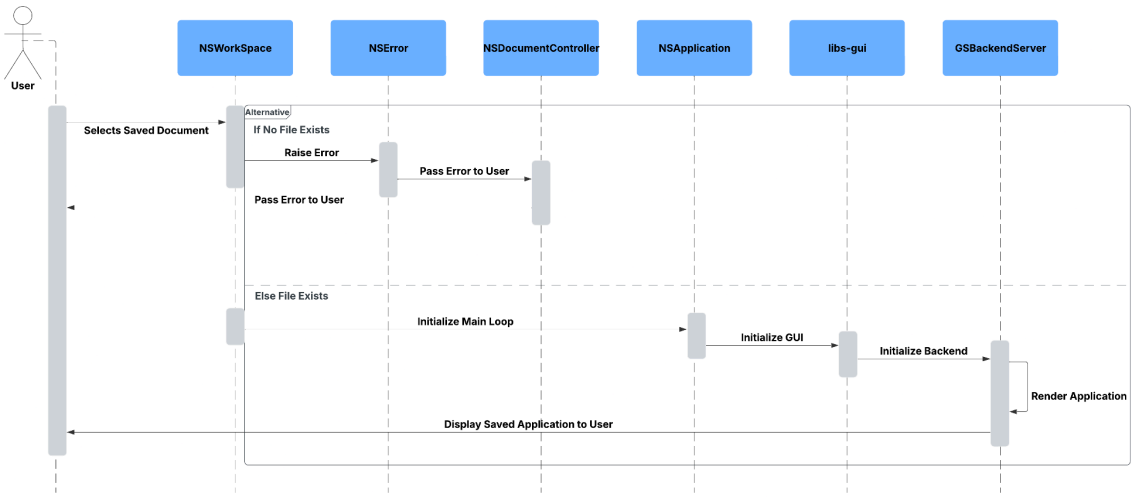
4.2.4 Reflexion Analysis on Absences: libs-base → apps-gorm

In our conceptual architecture, we expected a bidirectional dependency between libs-base and Gorm, where Gorm would depend on libs-base for fundamental functionalities, and in return, libs-base would incorporate elements from Gorm. However, our concrete analysis reveals that only Gorm depends on libs-base, while lib-base has no dependency on Gorm.

This absence suggests that libs-base does not integrate any functionality from Gorm. The conceptual assumption of libs-base → Gorm is not realistic, as libs-base serves as the foundation layer responsible for managing data and system-level operation, whereas, Gorm is a UI design tool that handles application-specific components. Since libs-base operates as a low-level library, it is designed to be independent of application-specific frameworks like Gorm. Therefore, the absence of a libs-base → Gorm dependency aligns with its purpose of providing a generalized, reusable foundation, rather than relying on higher-level components. To resolve this absence, the conceptual model should be adjusted to reflect the one-way dependency, where libs-base does not depend on Gorm.

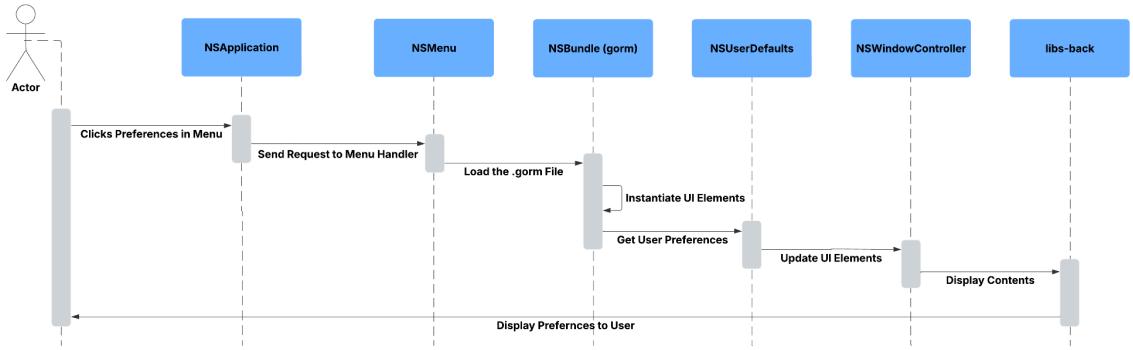
5. Use Case Sequence Diagrams

5.1 Use Case: The user selects a saved application to load



This use case sees a user going to the menu to load a saved application they were working on. This request will be sent to NSWorkSpace to locate the application. If there is no application then an error is raised, else it will continue to NSApplication to initialize the application, libs-gui will then handle the initialization of the GUI. GSBackendServer will connect to the windowing backend for the OS and render the application. Finally, it will be displayed to the user.

5.2 Use Case: The user loads preferences built with Gorm



This use case follows a user who tries to open their preferences with Gorm. NSApplication will take the request and send it to NSMenu to handle the rest of the sequence. NSMenu will pass a request to NSBundle to load the .Gorm file, NSBundle will also instantiate the UI elements and then send information to NSUserDefaults for the user preferences. NSWindowController will then update the UI elements and libs-back will work to render and display the contents to the user.

Legend:

6. Conclusions & Lessons Learned

Our report has sought to discuss and showcase the concrete architecture of GNUstep, with a specific focus on the five main components, their interactions, and dependencies. Specific focus was also put on Gorm, a tool within GNUstep used for creating and editing GUI's. This report studied the concrete architecture of these components and compared our findings to our results from the previous report on conceptual architecture. Through this comparison and reflection, we found some discrepancies in our understanding of GNUstep and our original proposed architecture, making amendments to better highlight all the interactions between the components. However, we have found that our proposed layered architecture from our previous report was similar to the concrete architecture of GNUstep, giving us greater confidence in our understanding of GNUstep.

Our understanding of GNUstep was aided greatly by the software tool Understand. This software provided valuable insight into the interactions and dependencies of the components in GNUstep through its powerful tools. The dependency graphs made studying and learning the concrete architecture painless compared to relying on the source code and repository for GNUstep. Writing this report taught us about the true scale and effort put into GNUstep and gave us a greater appreciation for projects of this size. Without tools such as Understand, learning about the concrete architecture for this report would have been a much more tedious and herculean task. Much like the team that worked on GNUstep, we learned quickly that to understand this software as a whole we need to split ourselves into sections and learn about individual sections, and come together to get a complete picture. Because of this, we have a greater admiration for GNUstep, and the work that went into creating it by all the various teams involved over the years.

References

[1] SciTools. 2025. Understand Version 7.0. Available at https://scitools.com/.

[2] GitHub Repository for GNUstep: "GNUstep GitHub Repository". GNUstep, https://github.com/gnustep.

[3] Botto, Francis, et al. Objective-C Language and GNUstep Base Library Programming Manual. http://andrewd.ces.clemson.edu/courses/cpsc102/notes/GNUStep-manual.pdf.

[4] GNUstep Documentation: "GNUstep Gorm Experience". GNUstep, https://www.gnustep.org/experience/Gorm.html.

[5] “GNUstep.” Wikipedia. https://en.wikipedia.org/wiki/GNUstep.

[6] Hillegass, A. 2014. Cocoa Programming for Mac OS X: Working with Text and Strings. https://www.informit.com/articles/article.aspx?p=2274038&seqNum=7.