CISC322 A2

Concrete Architecture of GNUstep.

13, March, 2025

Quantum Loop

Mohamed Hirsi	22xlb@queensu.ca
Mirwais Morrady	22jl75@queensu.ca
Musdaf Hirsi	22pmw@queensu.ca
Daniel Bajenaru	21dsb12@queensu.ca
Mo Yafeai	21my32@queensu.ca
David Fabian	21dgf4@queensu.ca

Table of Contents

Abstract	3
Introduction	3
Derivation Process	4
Top-level Concrete Architecture	5
Conceptual vs Concrete Architecture	9
Second-level Subsystem	11
Sequence Diagrams	12
Conclusions	14
Lessons learned	14
References	15

1 Abstract

This report provides an analysis of our GNUstep system's architectural by showing important differs between conceptual and concrete structures. The used conceptual architecture is based on a three-layer conceptual model consisting of a GUI Layer, Core/Utility Layer, and System Abstraction Layer. The concrete architecture adopted a hybrid layered and object-oriented style, incorporating bidirectional dependencies and cross-layer interactions. Using SciTools Understand, we discovered unanticipated bidirectional dependencies, particularly between libs-gui and libs-back, suggesting a more integrated structure than previously thought. Furthermore, libobjc2, which was previously thought to be an abstracted dependence, surfaced as an explicit component in the system's structure, proving its crucial role in managing dependencies.

In our analysis, we also looked at the practical implications of these deviations, such as optimizations that improve performance, modularity, and system adaptability. We discovered that event-driven communication methods like NSNotification substituted direct function calls, which had a substantial influence on the Model-View-Controller (MVC) sub -systems in libs-gui.

Two use cases—user interaction with GUI elements and application initialization—show how components such as libs-gui and libs-back work together dynamically. The report indicates that real-world restrictions, such as flexibility and platform compatibility, need architectural changes that differ from idealized models. These findings emphasize the value of iterative design in software architecture.

2 Introduction

This report analyses the concrete architecture of GNUstep, an open-source framework for graphical applications by comparing it to its conceptual architecture to discover differences, causes, and practical implications. The goal is to understand how software design assumptions align or deviate from real-world applications. Our conceptual architecture took an object-oriented style. However, as our analysis developed, we discovered unexpected architectural differences. To address these differences, we modified our initial conceptual architecture. As we combined insights from Group 5's and Group 20's methodologies, we evolved into a hybrid model that combines layered and object-oriented approaches. This change better reflects GNUstep's real-world architecture.

This report is structured into several important sections, each concentrating on a distinct aspect of our analysis and results. The first section, Conceptual vs. Concrete Architecture, analyzes our initial design assumptions and their relationship to the real-world dependencies discovered in the concrete architecture. This comparison highlights the differences between conceptual models and concrete applications. The Architectural Discrepancies and Findings illustrates important differences in concrete architecture. These include unexpected bidirectional dependencies, the use of libobjc2, and changes to the interface between the GUI and backend components. The Use Case Analysis section includes actual examples of system interactions, such as user-driven events and rendering processes, that either support or contradict our initial assumptions about how the system should work.

GNUstep's concrete architecture adopts a hybrid layered-object-oriented style that prioritizes flexibility over rigid modularity, with key components evolving beyond their initial roles: libs-back moved from rendering to event handling and low-level system interactions, while libobjc2 evolved from an abstracted dependency to a critical system-layer component. Communication transitioned to indirect event-driven systems like NSNotification, which reduced coupling while complicating dependency management. While this hybrid architecture increased performance and cross-platform flexibility, it also presented dependency management issues, emphasizing the trade-off between architectural efficiency and complexity.

Finally, this report is intended for software architects, developers, and project managers who create and maintain complex systems. It offers tangible insights into balancing conceptual and concrete constraints, making it a valuable resource for both technical and managerial stakeholders.

3 Derivation Process

The derivation of the concrete architecture is split into three distinct stages. From the onset, we decided to switch our group's conceptual architecture to a hybrid between group 5's and group 20's. From group 5 we took their layered style and the components along with their respective subcomponents. From group 20 we took the dependencies between the components. The reason for changing our initial conceptual architecture was to adopt the layered architecture style which we now believe to better represent GNUstep's design. This new architecture clearly presents the relation between components and layers which makes the reflection analysis process much simpler.

Our method for deriving the concrete architecture was a collaborative process. One person presented their computer's screen to the group and was in control of placing components and directories together. The whole group contributed to deciding what components to be created and to which component files and directories are to be ascribed to. There are two iterations of our architecture which highlight our decision-making process. The first iteration featured the layers and main components of group 5's conceptual architecture, but omitted the libs-gui component's model, controller, and view subcomponents. We believed that these complicated the architecture, and we had difficulty partitioning the libs-gui directory into the three components. We opted for ascribing each directory to its shared name component, and for libobjc2 to be placed under an additional libobjc2 component in the System Abstraction Layer. After its completion were satisfied, but only shortly which led to the creation of a second concrete architecture

The problem with our initial design was that it did not faithfully follow the conceptual architecture. We initially did not mind this, but further research showed that the concrete architecture must include the same components found in the conceptual architecture. This meant that our architecture needed to be reworked. Initially, the new architecture did not include layers. This choice was made following the example architecture provided by Ivan T. Bowman and al. for the Linux operating system. Their work gave us the belief that a software architecture should include only the components and subcomponents. As a result, the model, controller, and view libs-gui components were added and the libobjc2 component was removed. The libs-gui directory was partitioned into the three subcomponents. The files and directories were closely studied to determine which functionality the best implement. A combination of the file names and code documentation were used to complete this goal. The libobjc2 directory was entirely placed under the libs-corebase component as it implements the low-level functionality of the component. As mentioned, this architecture excluded mention of the layers found in group 5's conceptual architecture and was presented to our TA for review. The result of consulting with our TA was the final addition of the layers to our concrete architecture finalizing its design.

4 Top-level Concrete Architecture

4.1 Main Architectural Style

Our concrete architecture for GNUstep follows Group 5's hybrid approach, which combines layered and object-oriented architectural styles. Initially, in our conceptual architecture, we only considered an object-oriented style. However, after conducting further research on GNUstep's actual architecture and analyzing Group 5's and Group 20's reports, we realized that GNUstep is best represented by a hybrid layered and object-oriented approach. Consequently, we adopted Group 5's layered structure and divided our concrete architecture into three distinct layers:

- Core and Utility Layer
- GUI Layer
- System Abstraction Layer

Within these layers, our system consists of five major components as can be seen in figure 1 dependency graph extracted from Understand tool:

- Gorm (Graphical Object Relationship Modeller)
- libs-back (Display Backend)
- libs-base (Foundation & Core Functionality)
- libs-corebase (CoreFoundation Compatibility)
- libs-gui (Graphical User Interface)

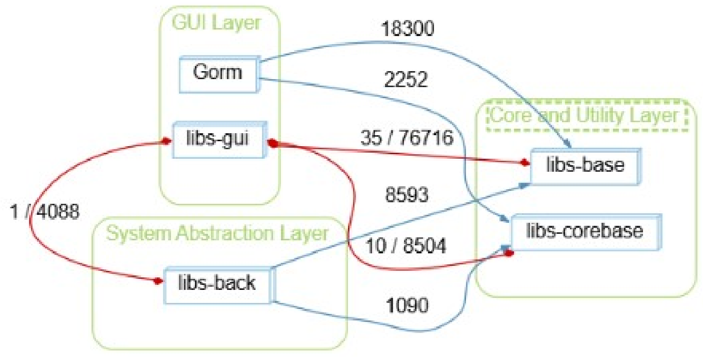

Figure 1: Concrete architecture of GNUstep generated by Understand

In our initial conceptual architecture, we had overlooked the Gorm component, which is a critical part of GNUstep's GUI design process. After reviewing Group 5's report, we recognized its importance and integrated it into our concrete architecture. Additionally, while our conceptual model outlined interactions between subsystems, it lacked a detailed breakdown of function calls and dependencies. We adopted these dependencies from Group 20, ensuring that our architecture correctly represents real-world interactions between components.

Furthermore, we identified several missing details in our A1 report, which we have now incorporated based on insights from Group 5 and Group 20 some of which are:

- Event handling in libs-back (from Group 5).
- libs-back calling libs-corebase for low-level system interactions (from Group 20).
- libs-base calling libs-gui for UI-related operations (from Group 20).
- Interactions between libs-corebase, libs-gui, and libs-back for CoreFoundation support (from Group 20).
- libs-gui (Graphical User Interface)

With these refinements, we hope that our concrete architecture provides a more accurate representation of GNUstep, integrating both structural correctness and precise dependencies.

4.2 Top-Level Subsystems and Their Roles

4.2.1 Gorm (Graphical Object Relationship Modeller)

Gorm is an essential graphical interface builder that allows developers to create UI elements through a drag-and-drop approach instead of manually writing interface code.

Interactions:

- Calls libs-base to initialize UI objects (18300 function calls).
- Calls libs-corebase for compatibility support (2252 function calls).

4.2.2 libs-back (Display Backend)

This subsystem is responsible for abstracting platform-specific rendering and adapting applications to different environments such as X11, Wayland, or Windows. It serves as the bridge between GUI components (libs-gui) and the system to ensure that UI elements are displayed correctly across different operating systems.

Interactions:

- libs-back depends on libs-base for core system utilities such as: file system management (loading UI resources, saving graphical configurations), memory management (handling graphical buffers and textures), Process handling (ensuring smooth rendering execution)

- libs-back interacts with libs-corebase to support: low-level system interactions that extend Apple's CoreFoundation functionalities, advanced rendering APIs that help with compatibility across different platforms, enhanced system calls for efficient communication between UI and system graphics

- libs-back receives rendering requests from libs-gui, ensuring Ui components are properly displayed, handles platform-specific drawing operations, so libs-gui remains platform-independent, event handling: It captures low-level user input events (mouse, keyboard, touch gestures) and sends them back to libs-gui for processing

4.2.3 libs-base (Foundation and Core Functionality)

The fundamental subsystem handling data structures, file operations, networking, concurrency, and memory management. It acts as the backbone for both GUI and non-GUI applications, providing essential utilities for object-oriented programming in GNUstep.

Interactions:

- Called by Gorm for UI object creation (18300 function calls)
- Called by libs-back for handling file access and system utilities (8593 function calls)
- Calls libs-gui (76716 times) for core functionalities like string handling, data storage, and memory management, while libs-gui calls libs-base (35 times) for system utilities such as file access and user input processing

4.2.4 libs-corebase (CoreFoundation Compatibility)

This subsystem is important for extending GNUstep's interoperability with Apple's CoreFoundation framework. It enables GNUstep applications to reuse macOS-compatible APIs, reducing the effort needed for cross-platform development.

Interactions:

- Receives calls from Gorm for compatibility with Apple-based UI components (2252 function calls)
- Communicates with libs-gui to integrate interface logic with CoreFoundation (8504 function calls), and is also called by libs-gui (10 function calls) for accessing CoreFoundation utilities related to UI state management and event processing
- Called by libs-back (1090 function calls) for low-level system interactions and extended CoreFoundation functionalities, ensuring platform-specific rendering compatibility

4.2.5 libs-gui (Graphical User Interface)

The primary UI framework in GNUstep, responsible for managing UI components, user interaction, and rendering elements. It follows the Model-View-Controller (MVC) paradigm, breaking UI logic into distinct modules.

Interactions:

- Communicates with libs-back through calling NSApplication to ensure platform-specific display support

- Calls libs-corebase to utilize CoreFoundation features (10 function calls)
- Depends on libs-base for event handling, user input, and UI control (35 function calls)

5 Conceptual vs Concrete Architecture

5.1 High-level Architecture Differences

For the conceptual architecture, we adopted Group 5's layered model, consisting of a strict three-layer separation: GUI, Core and Utility, and System Abstraction, with a one-directional dependency flow where each layer interacted only with the layer directly below it. We also incorporated elements from Group 20's object-oriented style to improve modularity and flexibility.

The concrete architecture, as revealed by SciTools Understand, showed unexpected dependencies. Unlike the initial assumption of strict layering, bidirectional dependencies emerged between components. A significant deviation is that libs-gui now has a direct dependency on libs-back, contradicting our expectation that it would remain isolated within the GUI layer. Additionally, libobjc2 is explicitly present in the dependency graph, confirming it plays an active system-level role rather than being abstracted away as we originally thought.

Figure 1 highlights the unexpected dependencies between libs-gui, libs-back, and libs-base, along with the internal breakdown of libs-base and the weaker role of libs-corebase. The conceptual architecture followed Group 5's layered model, structuring the system into three layers:

- GUI Layer (libs-gui, Gorm)
- Core and Utility Layer (libs-base, corebase)
- System Abstraction Layer (libobjc2, libs-back)

Initially, a one-directional dependency flow was assumed, to maintain modularity and separation of concerns. However, our analysis identified several key discrepancies:

- libs-gui now directly depends on libs-back, contradicting our assumption of strict modular separation
- libobjc2 is explicitly present in the System Abstraction Layer, confirming its role in dependency management and system interactions

5.2 Reasons for Architectural Discrepancies

Several factors contributed to these deviations from the original conceptual model:

Bidirectional Dependencies Instead of Strict Layering

- the conceptual architecture Expectation: A one-directional flow of dependencies
- our concrete architecture: Some dependencies go both ways, such as libs-gui now directly interacting with libs-back instead of only through libs-base

Direct interaction between libs-gui and libs-back

- the conceptual architecture Expectation: libs-gui would only communicate through libs-base
- our concrete architecture: libs-gui now interacts directly with libs-back, likely for performance optimizations and handling low-level UI functions

libs-back has a more significant role than expected

- the conceptual architecture Expectation: libs-back was assumed to handle only rendering
- our concrete architecture: It interacts with multiple components, suggesting it has a broader role beyond rendering

libobjc2 is explicitly represented

- the conceptual architecture Expectation: We assumed libobjc2 was either abstracted or merged into other components
- our concrete architecture: libobjc2 is clearly present in the System Abstraction Layer, meaning it still plays a crucial role in managing dependencies

5.3 Conclusion

Our analysis confirms that the conceptual model was more rigid than the actual implementation. Instead of a strictly layered system, we found bidirectional dependencies and a more interconnected structure. These findings highlight how real-world concerns—performance, modularity, and flexibility—shape architecture beyond initial design expectations. Moving forward, our conceptual models should account for these practical constraints to better reflect how GNUstep is actually structured.

6 Second-level Subsystem

6.1 Conceptual View

The chosen subsystem that we will be taking a closer look at is the libs-gui component. libs-gui is the main library in GNUstep that handles the creation of graphical applications. It contains important classes that provide the building blocks for common GUI elements like buttons, text fields and windows. Its conceptual architecture shows that it follows the Model-View-Controller design pattern which is commonly used when building GUIs. As talked about in our A1 report, the model component directly manages the data, logic and rules, the view component represents visualizations of information (charts, diagrams, tables, etc.), and the controller takes input to create commands for either the model or the view.

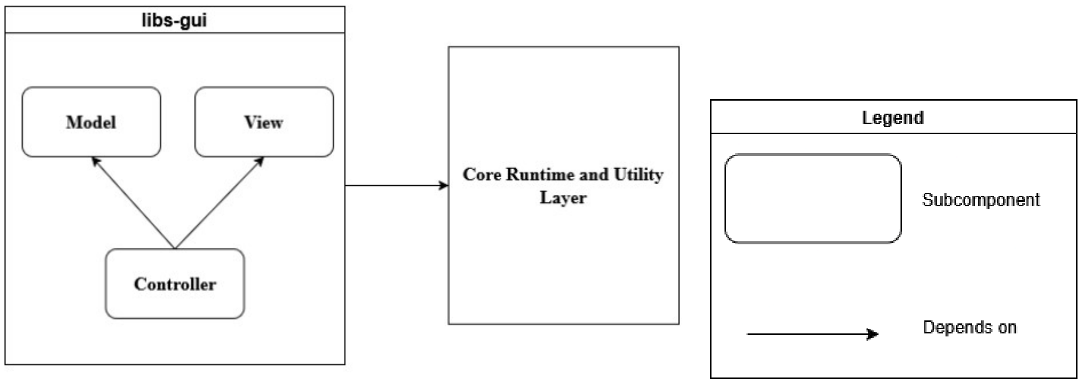

Figure 2: Conceputal architecture of libs-gui

In the conceptual architecture, the libs-gui component is part of the GUI layer, which depends on the Core Runtime and Utility Layer.

6.2 Concrete View

Using SciTools Understand, many differences are revealed. A simplified concrete architecture based on the one generated by Understand is shown below.

6.3 Reflexion Analysis

System Abstraction Layer ↔ View

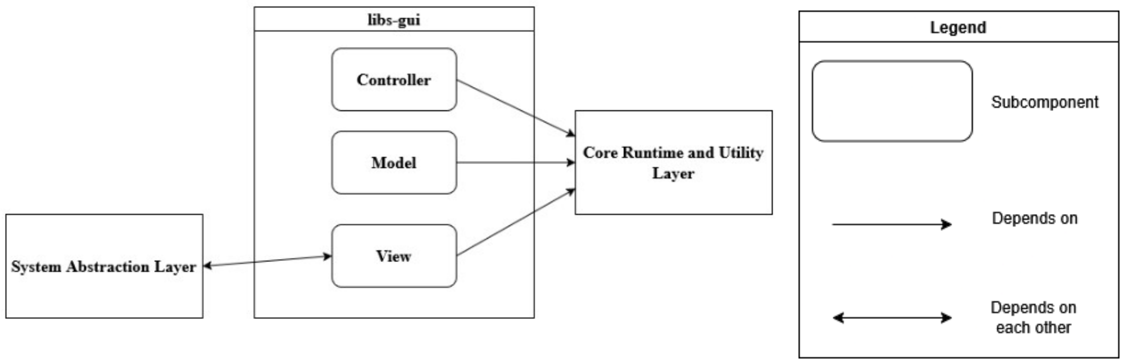

Figure 3: Concrete architecture of libs-gui

The system abstraction layer contains the libs-back subsystem. Libs-back is responsible for turning GUI commands (like drawing buttons, windows or text) into system operations. The cause of the dependency is that libs-back is calling some files that are in the View component. For example, there are calls to class files like NSView, NSColor, NSImage and NSTextView. This is likely a design oversight since libs-back should be sending instructions to the frontend, instead of calling the classes directly.

Indirect MVC communication

The main discrepancy between the conceptual and concrete architectures is the fact that the concrete architecture shows zero direct dependencies between model, view or controller, while in the conceptual architecture, the controller depended on the model and view. This is because GNUstep uses patterns like delegates or notifications to connect the model, view and controller indirectly. For example, if data in the Model changes, the Model component would post an NSNotification (which is in libs-base) [5] to broadcast updates; the Controller or View would then observe the notification and update themselves. This indirect communication avoids direct dependencies between the MVC components, everything is sent through the GNUstep framework.

7 Sequence Diagrams

Use case 1: The user interacts with the graphical user interface by clicking a button (e.g., "Refresh data"). Upon clicking the button, the application triggers event handling through GNUstep's internal event system. This involves the application's GUI components (libs-gui) sending requests to the controller, which retrieves updated data from the model and updates the GUI accordingly, clearly demonstrating GNUstep's event-driven MVC (Model-View-Controller) structure.

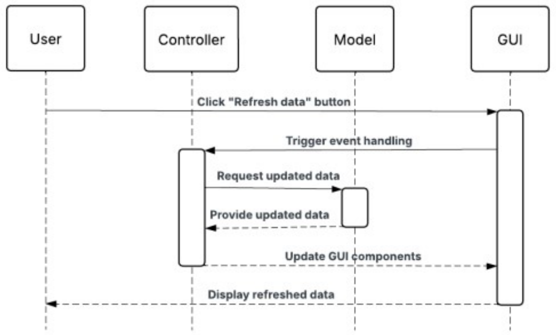

Figure 4: Sequence diagram for use case 1

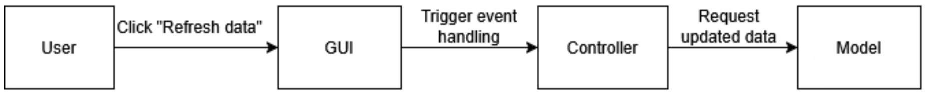

Figure 5: Box and arrow diagram for use case 1

Use case 2: When the user launches an application built with GNUstep, various GUI elements (e.g., windows, buttons, menus) are initialized and displayed. The rendering request flows from GNUstep's GUI subsystem (libs-gui) to the backend subsystem (libs-back), which performs the actual drawing operations. This use case demonstrates the essential interaction between the GUI layer and backend rendering components during applications initialization.

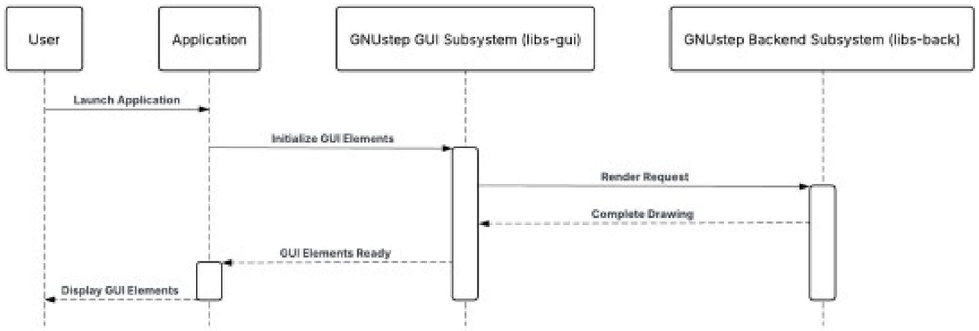

Figure 6: Sequence diagram for use case 2

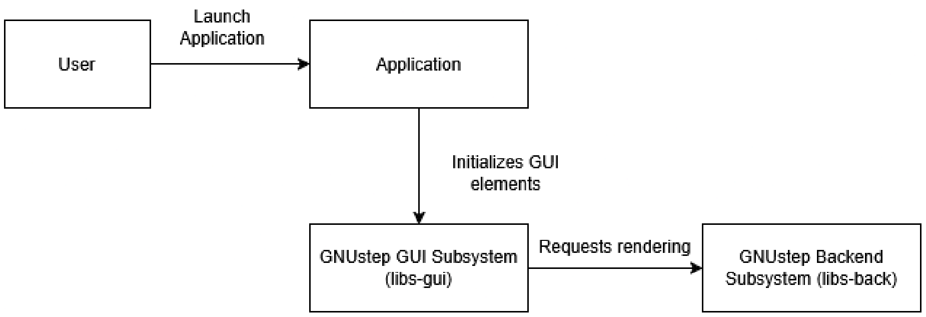

Figure 7: Box and arrow diagram for use case 2

8 Conclusions

Our analysis found some differences between GNUstep's conceptual and concrete structures. For instance, libs-gui directly interacts with libs-back, bypassing the Core and Utility layer. Furthermore, libobjc2, which we had previously regarded an abstraction component, played a more direct role in dependency management than expected. The Model-View-Controller (MVC) sub -system was also not strongly followed, as event-driven communication mechanisms like NSNotification replaced direct function calls, enhancing modularity but adding complexity to system interactions.

These findings highlight the differences between the conceptual and concrete architectures. While deviations from the conceptual model resulted in unexpected relationships, they also increased system performance and adaptability. Understanding these differences allows for better architectural planning, providing a balance of modular separation and efficiency. The insights gained from this study will guide future improvements in dependency management strategies and architectural modifications for GNUstep and similar systems.

9 Lessons Learned

The second stage of the assignment was full of learning opportunities. This assignment has students walk through the process of architecting a software system giving them hands on experience with applying the courses core concepts. There are three main lessons that this stage taught us. The first is that the design process is iterative. There were many times in the design process that we believed we completed the work, but a great design requires review, and sometimes major changes. Building something often is not done in one attempt, and this project highlights this lesson. To complete this stage, our knowledge of concrete and conceptual architectures required solidifying which is the second lesson learned. The best way to gauge understanding of a concept is to have it tested which exactly what building a concrete architecture did. This led to a much-needed learning opportunity. The final lesson learned through A1 was how to read filter through source code with a specific objective. To partition libs-gui into the model, controller, and view subcomponents, selective examination of GNUstep's source code was required. The skill of extracting the functionality of code from the file titles, documentation, and function names is critical to software developers, and A1 gave us a chance to practice this skill.

References

1. Our Own Group's A1 Report – Conceptual Architecture of GNUstep, Quantum Loop, February 2025.
2. Group 5's A1 Report – A Report on the Architectural Design of GNUstep, February 2025.
3. Group 20's A1 Report – Conceptual Architecture of GNUstep, Team GNUtered, February 2025.
4. Dependency Graph Extracted from Understand Tool – Function Call and File Dependency Analysis, March 2025.
5. Gnustep. "Gnustep/Libs-Base: The Gnustep Base Library Is a Library of General-Purpose, Non-Graphical Objective C Objects." GitHub, `github.com/gnustep/libs-base`. Accessed 13 Mar. 2025.