

# Concrete Architecture Report: GNUstep

Queen's University

CISC 322

Instructor: Prof. Bram Adams

Winter 2025

Cameron Jenkins 20358084 21cj32@queensu.ca

Christine Ye 20354325 21cy27@queensu.ca

Evelyn Lee 20334769 21esl5@queensu.ca

Yiting Ma 20232544 20ym2@queensu.ca

Liam Beenken 20364179 22ltb1@queensu.ca

# Table of contents

1. Abstract

2. Introduction

Architecture

3. Top-Level Concrete Architecture

4. Architecture Derivation Process

5. Subsystem Analysis

6. Divergence Analysis

7. Sequence Diagrams

8. Conclusion & Lesson learned

9. References

## **1. Abstract**

This report analyzes the concrete architecture of GNUstep, building on the conceptual architecture from Deliverable A1. Using SciTools Understand, we identify the system's architectural style as a layered, object-oriented design, composed of distinct subsystems that separate core functionality, user interface components, and platform-specific dependencies. Key subcomponents include libs-base, libs-gui, libs-corebase, and Gorm, with libs-gui selected for deeper investigation into its internal structure and interactions. A comparative analysis reveals key discrepancies between the conceptual and concrete architects including implicit dependencies, dynamic linking differences, and performance-driven modifications. These are examined in the context of architectural decisions, system constraints, and implementation trade-offs. Sequence diagrams illustrate real-world system interactions, while the methodology section details the process of loading, analyzing, and refining the architecture. The findings offer deeper insight into GNUstep's architectural structure and inform enhancements in future project phases.

## **2. Introduction**

GNUstep is a free and open-source implementation of the Cocoa framework, providing a cross-platform, object-oriented development environment. Based on a modular architecture, it makes it easier to build applications for Windows and Unix-like operating systems while still being compatible with macOS development techniques. Concrete architecture, which is influenced by implementation choices and real-world limitations, often differs significantly from its conceptual architecture, which depicts an idealized structural design.

In this report, the concrete architecture of GNUstep is examined, along with the practical interactions between its main subsystems. In order to determine how closely the actual implementation resembles the desired design, we examine extracted dependencies and structural elements. We also perform multiple reflection analysis to investigate architectural discrepancies, identifying the underlying reasons for divergence. In addition, we investigate the technical architecture of a selected second-level subsystem, getting insights into its internal organization and role within the larger system. Finally, we discuss the obstacles and lessons learned while examining the physical architecture, gaining insight into how software systems grow from their initial ideas.

## **Architecture**

### **3. Top-Level Concrete Architecture**

GNUstep is divided into several different subsystems, each responsible for a specific function inside the framework. Higher-level components use and expand the fundamental characteristics provided by lower-level subsystems in hierarchical interactions between these subsystems.

#### **Apps-gorm**

The apps-gorm subsystem includes GNUstep-based programs, with Gorm playing an important role. Gorm is an interface builder that allows developers to graphically create application layouts. This application layer largely relies on libs-gui to generate user interface components and libs-corebase to handle core logic and data processing. Gorm contains a variety of components that help with its functions. The major functionality is provided by GormCore, the visual editor is managed by InterfaceBuilder, the Objective-C code is read by GormObjCHeaderParser, and developers may automate processes using the gormtool. By organizing its features into separate libraries and frameworks, Gorm ensures its capabilities can be reused in other applications (*Gnustep/apps-gorm*).

## **Libobjc2**

The libobjc2 subsystem offers an Objective-C runtime implementation, which is essential for dynamic method resolution, message passing, and object lifecycle management. This Objective-C Runtime Layer is the backbone of GNUstep's object-oriented capabilities, allowing for dynamic behaviour and supporting the higher-level frameworks that rely on it. Libs-base, libs-corebase, libs-back and libs-gui all use libobjc2 to implement object-oriented programming structures (*Gnustep/Libobjc2*).

## **libs-back**

The libs-back subsystem is a critical component of GNUstep's graphics system. This component works with libs-gui to handle different rendering backends and ensure that programs appear consistent across platforms. It handles low-level drawing and interacts with systems like X11, Wayland, and Windows. This subsystem includes Headers, which define interfaces for various backends like cairo, fontconfig, and win32; Source, which contains key implementation files like GSBackend.m; and Tools, which includes utilities such as font\_cacher.m for optimizing rendering. By managing different graphics engines, libs-back allows applications to run smoothly on multiple operating systems while keeping their appearance consistent (*Gnustep/libs-back*).

## **libs-base**

Acting as the core foundation layer, libs-base offers essential data structures, collections, and utility classes, akin to Apple's Foundation framework. It contains documentation, examples, headers, source code, and tools that provide the basic building blocks for other subsystems. This Core Foundation Layer depends on libobjc2 for object management and supports higher-level layers such as libs-corebase and libs-gui (*Gnustep/libs-base*).

## **libs-corebase**

The libs-corebase subsystem extends libs-base by adding further tools and features that enhance data handling and system interactions. It serves as a link between the graphical user interface subsystem, libs-gui, and the foundational libs-base components. By doing so, libs-corebase allows complicated application logic and improves data flow (*Gnustep/libs-corebase*).

## **libs-gui**

The libs-gui subsystem is in charge of managing user interactions and producing graphical user interface components within GNUstep applications. This component offers crucial subsystems for UI development, utilizing libs-back for smooth cross-platform rendering and libs-corebase for effective data management. With crucial headers like Cocoa.h, Cocoa guarantees compatibility with Cocoa-based apps, while AppKit, which is based on Apple's AppKit, provides the framework for essential GUI components. These components work together to form libs-gui and ensure a flexible and reliable user interface layer for application development (*Gnustep/libs-GUI*).

## **Interactions**

Through encapsulation, inheritance, and polymorphism, GNUstep's object-oriented design encourages modularity and extensibility while also following a layered architectural style. The system is made up of interconnected subsystems, each managing specific functionalities, with lower layers providing foundational services for higher layers. The libobjc2 subsystem serves as the basis for higher-level components by offering dynamic object-oriented characteristics. The libs-base subsystem builds on libobjc2 to define core classes, which are further extended by libs-corebase and libs-gui. GUI components are implemented via the libs-gui subsystem, allowing for UI reusability and inheritance. Because the system is organized around reusable classes and objects, components may communicate with one another through clear interfaces. In addition, higher-level subsystems also depend on and extend the functionality of lower layers, maintaining a clear separation of responsibilities. Furthermore, the use of polymorphism ensures flexibility in component integration by enabling smooth interactions between various subsystems. By using this approach, developers are able to create new classes or extend existing ones without changing the framework's core components.

## **4. Architecture Derivation Process**

### **How the Concrete Architecture Was Derived from A1**

The derivation of the concrete architecture from the conceptual architecture in Assignment 1 followed a structured methodology using SciTools Understand to analyze the source code and actual implementation of GNUstep. The goal was to validate and refine our conceptual understanding by examining real dependencies and interactions within the system. We first downloaded and prepared the pre-built Understand project, loaded the .und project file into SciTools Understand, and systematically scanned and indexed the source code. This analysis provided a detailed overview of the system's actual subsystems and studied key components such as gorm, libs-base, libs-gui, and libs-corebase (Figure 2). Their dependencies and module interactions were broken down to understand how various components communicate, and a dependency graph was generated to visually represent their connections and identify structural patterns.

To assess how closely the theoretical design aligned with the real-world implementation, we compared dependency diagrams with the conceptual architecture outlined in A1. This comparison revealed unexpected connections, missing modules, and additional dependencies, which were analyzed based on implementation constraints, performance optimizations, and overlooked dependencies. To ensure an accurate representation of the system, the conceptual architecture was updated to reflect indirect dependencies, adjusting module interactions, and additional subsystems. Sequence diagrams were also refined to better depict real-world execution flows. To further validate the concrete architecture, `libs-gui` was chosen as a second-level subsystem for deeper analysis (Section 5). Its internal structures, method calls, and interdependencies were mapped against the original A1 expectations and a reflexion analysis was conducted to formally assess deviations caused by unexpected dependencies, system optimizations, or implementation limitations.

### **Justification for Modifications**

During the transition from conceptual to concrete architecture, several adjustments were necessary. One key discovery was the presence of implicit dependencies that were not identified in A1. The Understand tool revealed hidden dependencies between libraries and utility modules, contradicting the assumption of clear-cut interactions. For example, while `libs-gui` was expected to depend solely on `libs-base`, the analysis revealed unexpected interactions with `libs-corebase`, requiring updates to the model.

Another difference was in subsystem granularity. The conceptual architecture grouped certain components together based on assumed logical coherence, while the concrete architecture consisted of smaller, modular components that were functionally distinct yet interdependent. As a result, some subsystems were divided or reclassified to better reflect their true structure and interactions.

Additionally, the actual implementation optimized certain interactions for performance, altering the expected dependency structure. In some cases, direct function calls replaced abstract interfaces, reducing overhead and improving execution speed, but deviating from the original architectural assumptions regarding modularity and separation of concerns.

### **Implementation Challenges**

Several unexpected challenges led to modifications in the concrete architecture. One major issue was the discovery of circular dependencies between specific subsystems. Some components contained unexpected circular references, resulting in unintentional coupling that was not visible in the conceptual model. Addressing this required additional research and modification to accurately represent the system's structure.

Another challenge arose from the reliance on low-level system calls. Certain architectural components relied on built-in C functions, which were essential for implementation but outside the scope of the conceptual architecture. Since these system-level dependencies were not initially considered, modifications were needed to properly depict interactions with operating system functions.

Additionally, the use of dynamic linking in the concrete architecture differed significantly from the conceptual model, which assumed static dependencies. Analysis using Understand revealed that many components relied on runtime dynamic linking, affecting module relationships and interactions. This required adjustments to align the conceptual model with the actual implementation.

## 5. Subsystem Analysis

### Overview

The libs-gui subsystem acts as GNUstep's graphical user interface layer, comparable to Apple's AppKit. It provides classes and APIs to create windows, views, and controls and handles events triggered by users (mouse, keyboard). Libs-gui interacts closely with lower-level subsystems like libs-back for graphics rendering and libs-corebase for enhanced data management. This makes libs-gui crucial to GNUstep's goal of cross-platform GUI compatibility.

### Internal Responsibilities and Execution Flow

#### Event Handling:

Conceptually (from Assignment 1), we viewed libs-gui as a straightforward handler of user input events. However, concrete implementation analysis reveals that native OS events (like mouse clicks) first reach the backend subsystem (libs-back), converting them into platform-independent NSEvent objects. Libs-gui then receives these events, delegates them to appropriate responders (e.g., NSWindow or NSView), and handles user interactions. UI updates (like redrawing views) are marked as "dirty," with rendering deferred until the backend subsystem processes and flushes these updates through methods like `_processUpdateRegion` and `drawRect`.

#### Graphics Rendering:

Rendering within libs-gui is similarly layered. When an element requires redrawing, libs-gui uses methods like `setNeedsDisplay` and `lockFocus` to initiate drawing, delegating actual rendering to libs-back. Backend-specific operations then produce draw commands (via backends like Cairo or X11), ensuring consistent graphical output across different systems.

#### Comparison of Conceptual vs. Concrete Views of libs-gui

Aspect	Conceptual (A1)	Concrete (Actual)
Dependencies	Strict hierarchy (libs-gui → libs-base, libs-back only)	Additional dependencies (e.g., libs-corebase), dynamic backend integrations, and direct calls for performance.
Layered Structure	Clean, simple hierarchy; GUI atop libs-base	Complex layered structure; dependencies with

		libs-corebase, dynamic linking
Event Handling	Direct event handling within libs-gui	Two-step event handling (libs-back → libs-gui); event translation overhead
Internal Organization	Unified module with clear public APIs	Modular, historically evolved structure; multiple conditional paths and legacy compatibility code

### Key Observations:

**Unexpected Dependencies:** The conceptual architecture anticipated libs-gui relying solely on libs-base. However, it also frequently depends on libs-corebase for data transformations and utilities, a factor not accounted for initially.

**Performance & Optimizations:** Certain direct references and shortcuts are implemented for efficiency, creating tighter coupling than expected. Real-world demands like performance tuning and compatibility have significantly shaped the libs-gui structure.

**Complex Layering:** Concrete analysis revealed a more entangled architecture than originally designed, including circular dependencies and dynamic loading of rendering backends. Over GNUstep's long history, the GUI subsystem has naturally accumulated complexity, deviating from the original simple layered concept.

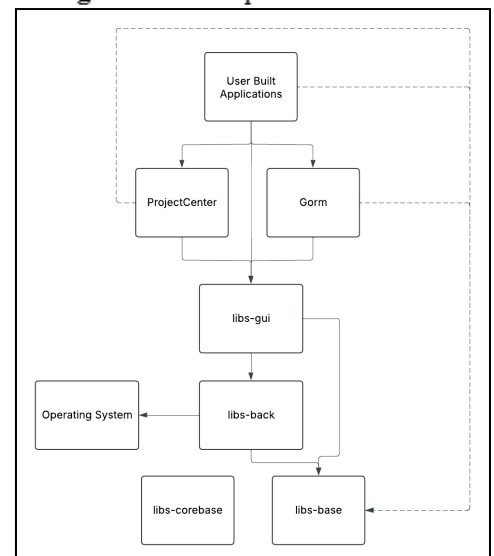
### Conclusion on libs-gui Subsystem

Libs-gui largely aligns with its intended conceptual design in providing high-level GUI classes and APIs. However, concrete analysis has uncovered complexities, such as frequent interactions with libs-corebase and libs-back, runtime plugin loading, and direct calls for performance reasons, that deviate significantly from the initial layered model. These differences, largely stemming from practical needs for implementation, a build-up of historical factors, and goals for optimization, lead to more interdependence and a more complex architecture. It's important to navigate these complexities with care so that we can ensure the GNUstep GUI subsystem remains maintainable, adaptable, and sustainable in the future.

## 6. Divergence Analysis

Figure 1, shown on the right, outlines our conceptual architecture of the GNUstep system and displays logical dependencies between key components. For instance, based on our understanding every component was dependent on libs-base, because the higher level

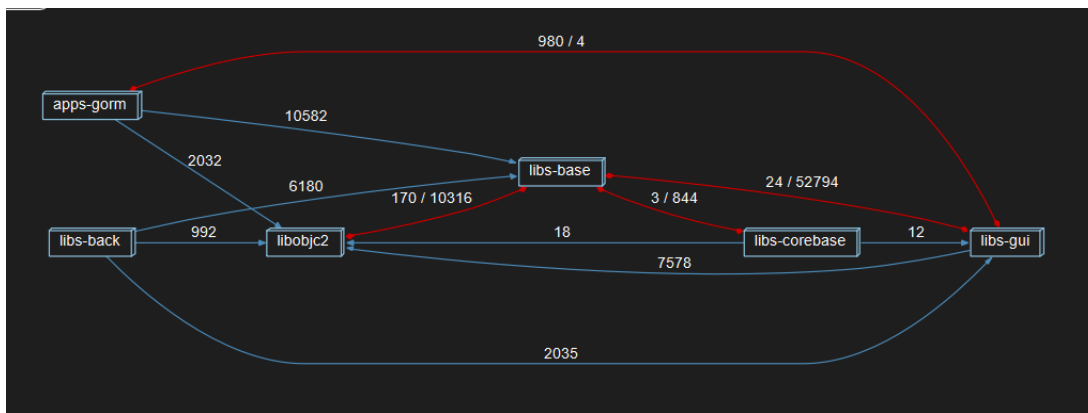
**Figure 1: Conceptual Architecture**





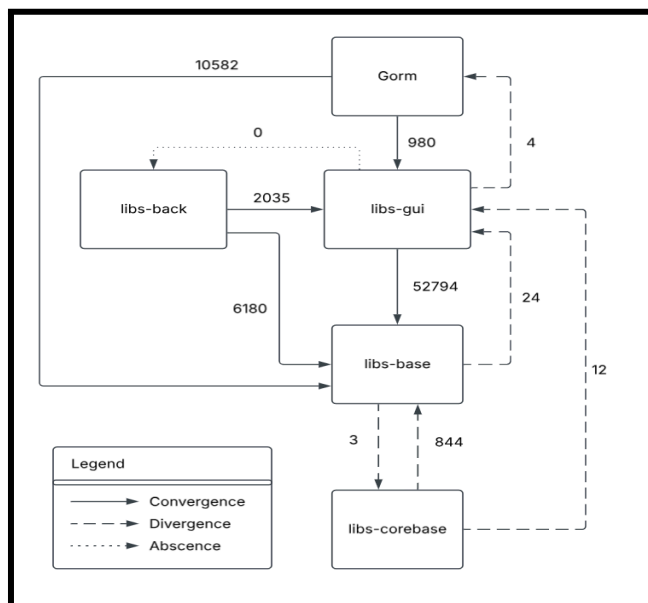
data structures that it provides would be key in building more useful components. Our conceptual architecture outlined an object oriented, layered style, where components encapsulate their data and services, generally for use by higher level components in the system. It also included a few breaks in the hierarchy, such as GORM's reliance on libs-base, which makes sense for development purposes. Having direct access to those core Objective-C data structures would make the process much easier, and avoid duplication of a component's responsibility.

This conceptual architecture provides a broad understanding of how core components are intended to interact in the system. In practice however, following an architectural design like this might be impossible or impractical. For many reasons, such as unreasonable deadlines or requirement changes, corners may be cut resulting in divergences from this conceptual design. In order to gain a better understanding of the system's implementation in reality, we can observe the concrete architecture, displaying the actual dependency edges between components in the source code. GNUStep's concrete architecture generated by a software observability tool "Understand", is shown below.



**Figure 2:** GNUStep concrete architecture generated by Understand

We can immediately notice a series of discrepancies between the conceptual and concrete architectures. There are many cases in which we would want to find the reasoning behind these discrepancies. Understanding why these choices were made gives us a better understanding of how to work with, or improve the system. It may also highlight shortcuts taken in development, potentially introducing technical debt in the future. In order to explain these discrepancies, we can perform a reflexion analysis.



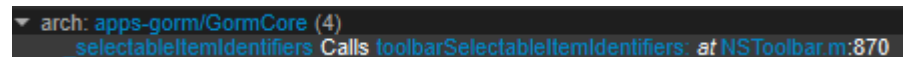
**Figure 3:** Reflexion Model

By comparing our conceptual and concrete diagrams, we arrive at this reflexion model of the system, shown in Figure 3. The diagram displays convergences, divergences, and absences between the two architectures, shown via different dependency arrow styles. It also shows the amount of calls to the dependent component above the arrow. Convergences are expected – they match our

conceptual overview of the architecture. Divergences are where we start to see differences in the real world implementation. We will examine each of these divergences and use the Sticky Note method for determining the reasoning behind these unexpected dependencies.

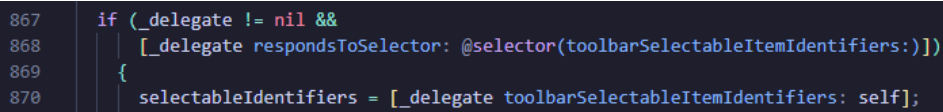
### Divergence 1. (libs-gui → GORM)

We can first observe a divergence from libs-gui to GORM through this model. In principle, this dependency relationship does not make sense. GORM is a UI interface builder, built on top of the functionality in libs-gui. It is not intended as a development library, so it seems improper for libs-gui to rely on any of its functionality. If we examine this dependency closer, we find the following:



▼ arch: apps-gorm/GormCore (4)  
\_selectableItemIdentifiers Calls toolbarSelectableItemIdentifiers at NSToolbar.m:870

Above is a screenshot from Understand, highlighting one of the calls to apps-gorm from libs-gui. It states that in NSToolbar.m, the \_selectableItemIdentifiers method calls the toolbarSelectableItemIdentifiers method from apps-gorm. \_selectableItemIdentifiers is a short method in the NSToolbar class, which is currently undocumented.

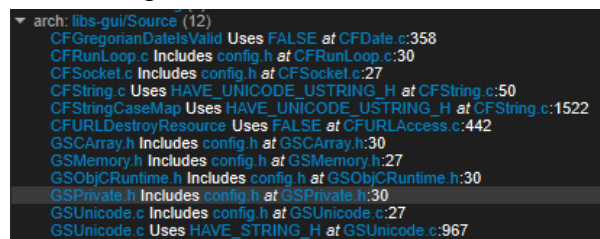


```
867 if (_delegate != nil &&
868     [_delegate respondsToSelector: @selector(toolbarSelectableItemIdentifiers)])
869 {
870     selectableIdentifiers = [_delegate toolbarSelectableItemIdentifiers: self];
```

Without documentation, we can still make a guess as to what this method is designed to do. It appears to be a private function to return a list of selectableIdentifiers. It checks if a delegate object exists and whether that delegate implements toolbarSelectableItemIdentifiers. If so, it simply returns the result of that function. What is important to note here is that toolbarSelectableItemIdentifiers is not a direct reference to the apps-gorm component. The calling method is instead using a delegate object, which is meant to implement its own toolbarSelectableItemIdentifiers method. This gives app developers using GNUStep freedom to design the selectable items in their toolbars. Understand is listing this as a dependency because apps-gorm creates a delegate that implements that method, much like any application built on GNUStep might. Libs-gui is not dependent on GORM, GORM just uses the library as it is intended, to build their toolbars. The other 3 “calls” to gorm arise from the same delegate pattern.

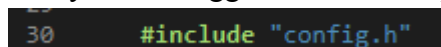
### Divergence 2. (libs-corebase → libs-gui)

This is another dependency that does not make much sense initially. If we examine the relationship in Understand, we see the following 12 calls to libs-gui from libs-corebase:



▼ arch: libs-gui/Source (12)  
CFGregorianDatelsValid Uses FALSE at CFDate.c:358  
CFRunLoop.c Includes config.h at CFRunLoop.c:30  
CFSocket.c Includes config.h at CFSocket.c:27  
CFString.c Uses HAVE\_UNICODE\_USTRING\_H at CFString.c:50  
CFStringCaseMap Uses HAVE\_UNICODE\_USTRING\_H at CFString.c:1522  
CFURLDestroyResource Uses FALSE at CFURLAccess.c:442  
GSCArray.h Includes config.h at GSCArray.h:30  
GSMemory.h Includes config.h at GSMemory.h:27  
GSOBJCRuntime.h Includes config.h at GSOBJCRuntime.h:30  
GSPrivate.h Includes config.h at GSPrivate.h:30  
GSUnicode.c Includes config.h at GSUnicode.c:27  
GSUnicode.c Uses HAVE\_STRING\_H at GSUnicode.c:967

Many of the flagged lines are simple include statements that look like this:



```
30 #include "config.h"
```

This is not directly referencing libs-gui’s config.h file, and it would be strange for users to have to specify that libs-gui’s config file should be used for the build. Since config.h is not present in

the libs-corebase folder of the Understand project, it is possible that Understand looks elsewhere for a matching file name, and incorrectly attributes these include statements to the config.h file present in libs-gui. It is likely that in reality, the referenced config.h file is meant to be auto generated at build time, due to the presence of a config.h.in file in the libs-corebase github repo. This does not denote an actual dependency, it looks to be a false flag by Understand. The other calls, like HAVE\_STRING\_H are typically defined in config.h files.

### Divergence 3. (libs-corebase → libs-base)

Libs-corebase and libs-base provide similar utility, in C and Objective-C respectively. In our conceptual architecture, these components had no interaction, because conceptually they are meant to provide bases for higher level components, not build on top of each other. In practice, this did not turn out to be the case. We'll look deeper into the NSCFArray.m file in the corebase library to see why this is. Understand shows us that this file references libs-base classes 4 times, as shown below:

```
NSCFArray.m Includes NSArray.h at NSCFArray.m:27
NSCFArray Bases NSMutableArray at NSCFArray.m:33
NSCFArray.m Extends NSArray at NSCFArray.m:37
NSCFArray.m Implement Extends NSArray at NSCFArray.m:114
```

This shows multiple uses of inheritance, borrowing functionality from the NSArray and NSMutableArray classes. If we look at the [commit history](#) of this file, we see that much of it was written by a single person, and left unchanged. While the commit message does not give much information about the reasoning behind this dependency, what we do gather is that this is a relatively simple file, made for a compatibility bridge between the NSArray and CFArray classes. It defines an Objective-C subclass of NSMutableArray, and designs an interface for calling libs-corebase array functionality on it. This means arrays created by NSArray in Objective-C can use the higher performance C functionality without direct casting or data copying by the app developer. This compatibility between base and corebase structures provides a ton of functionality to both libraries, and outweighs the negatives that come with the extra dependency.

### Divergence 4. (libs-base → libs-corebase)

As mentioned above, these two libraries have similar functionality, and all of the compatibility additions are made in the corebase library. With only 3 references in this direction, they are less likely to be a core feature, and more likely a quick fix or corner cut. If we examine the calls to corebase from base, we see the following:

```
▼ arch: libs-corebase/Headers/CoreFoundation (1)
  NSURL+GNUstepBase.m Includes CFURL.h at NSURL+GNUstepBase.m:131
▼ arch: libs-corebase/Source (2)
  pathWithEscapes Calls CFURLCopyPath at NSURL+GNUstepBase.m:162
```

We can observe that the file NSURL+GNUstepBase.m makes use of functionality defined in CFURL.h, specifically the function CFURLCopyPath. The line where this call occurs is shown below:

```
160 - (NSString*) pathWithEscapes
161 {
162     return CFURLCopyPath(self);
163 }
```

This method is very short, and based on the name, it is intended to return a URL path with escapes included. We can look further into the commit history to give us a better idea. This method was written by Richard Frith-MacDonald in 2012, and the [commit message](#) tells us this:

```
rfm committed on Mar 7, 2012
Apply patch by Jens Alfke with minor changes
```

We don't get much information here, but if we look into the changelog entry by Jens Alfke, we see the reasoning for including the `pathWithEscapes` method.

```
+      New -pathWithEscapes method to enable differentiation between '/'
+      characters in the original path and '%2F' escapes in it.
```

This leads us to believe that there was an error with the way URL strings were being processed when / and the escape characters were used. It is likely that the already implemented `CFURLCopyPath` function returned the URL string in the desired format, so rather than rewrite it, it was just imported and used as is.

### Divergence 5. (libs-base → libs-gui)

Many of the dependencies in this relationship don't seem intentional, similar to the previous falsely flagged dependencies. We can see from Understand that in most cases, this component is flagged as dependent because it references `GSI_ARRAY` related macros:

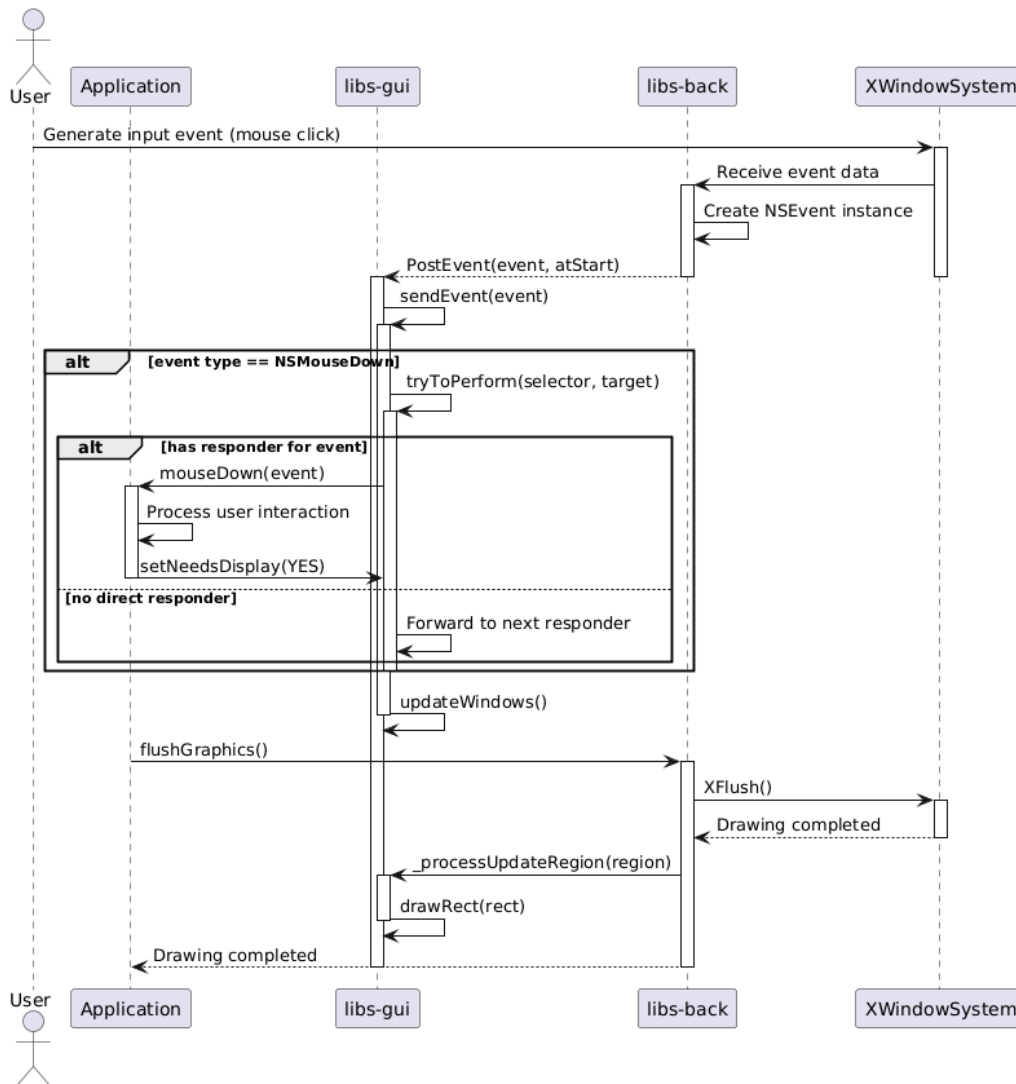
```
GSIArray.h Uses GSI_ARRAY_NO_RETAIN at GSIArray.h:95
GSIArray.h Uses GSI_ARRAY_NO_RELEASE at GSIArray.h:106
GSIArrayRemoveItemAtIndex Uses GSI_ARRAY_NO_RELEASE at GSIArray.h:450
GSIArrayRemoveLastItem Uses GSI_ARRAY_NO_RELEASE at GSIArray.h:477
GSIArraySetItemAtIndex Uses GSI_ARRAY_NO_RELEASE at GSIArray.h:496
GSIArrayRemoveItemsFromIndex Uses GSI_ARRAY_NO_RELEASE at GSIArray.h:564
GSIArrayRemoveAllItems Uses GSI_ARRAY_NO_RELEASE at GSIArray.h:577
```

It is true that all of these macros are defined somewhere in a `libs-gui` file, but the intention was for any file that uses `GSIArray` functionality to define these macros to set their desired behaviour. This intention is shown in a comment by Richard Frith-MacDonald in the `GSIArray.h` file:

```
/*
 * NB. This file is intended for internal use by the GNUstep Libraries
 * and may change significantly between releases.
 * While it is unlikely to be removed from the distribution any time
 * soon, its use by other software is not officially supported.
 *
 * This file should be INCLUDED in files wanting to use the GSIArray
 * functions - these are all declared inline for maximum performance.
 *
 * The file including this one may predefine some macros to alter
 * the behaviour (default macros assume the items are NSObject)
```

Understand flags this as `libs-base` depending on `libs-gui` because macros like `GSI_ARRAY_NO_RELEASE` are defined there, but the real relationship is `libs-gui` depending on functionality defined in `libs-base`. The macros just allow more customization of that `libs-base` functionality. Again, Understand is likely just looking for a location in the software where a matching macro name is defined, but this is not a dependency.

## 7. Sequence Diagrams



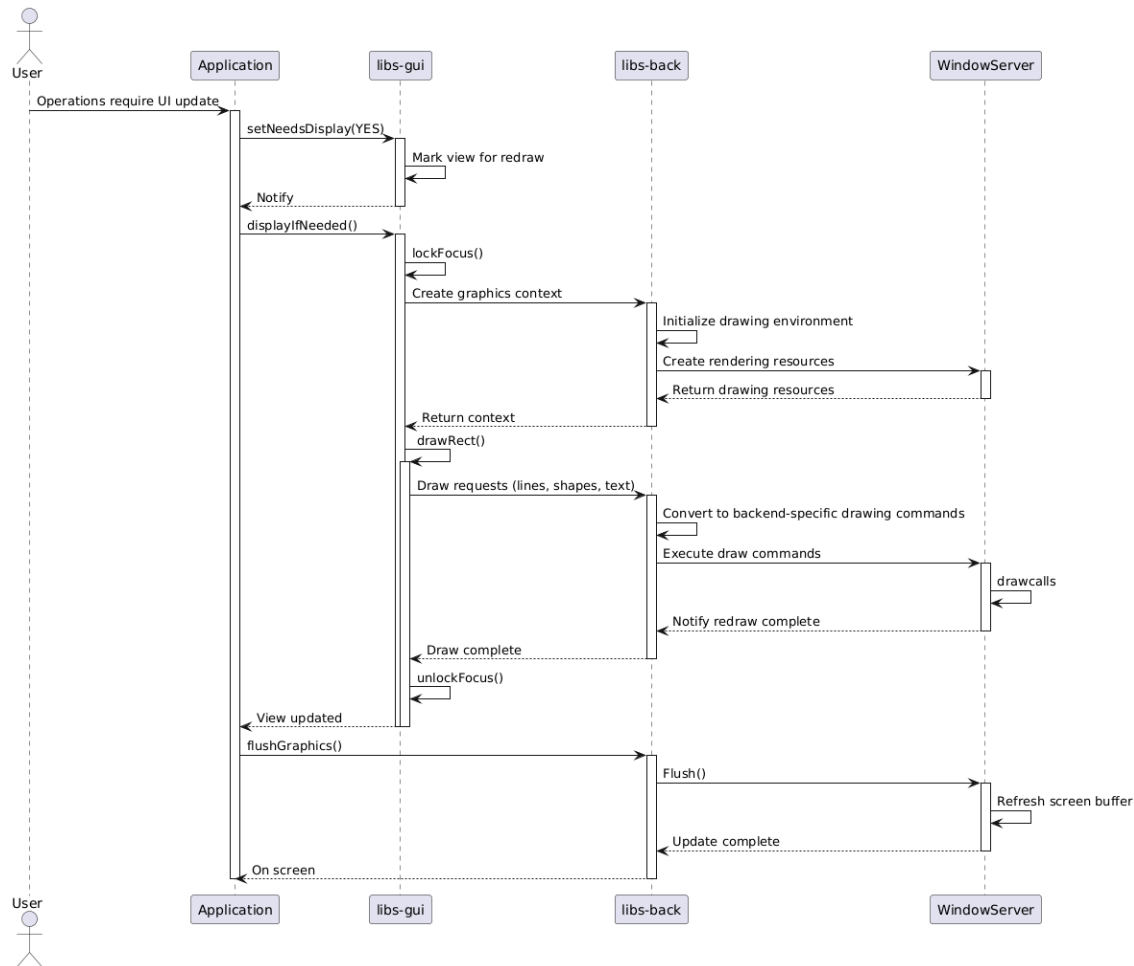
**Figure 4.**  
GNUstep  
Application Event  
Handling

Figure 4 shows a use case of the mouse event handling process of an application made with the GNUstep framework.

First, the user interacts with the window system through some input actions like mouse clicks. The corresponding input event is captured by the WindowSystem and passed to GSBackend, which is the backend implementation in libs-back. libs-back handles converting window system events into NSEvent objects that GNUstep can process.

After a NSEvent is created, this event is passed to libs-gui, which is the core controller of the application. libs-gui will direct the event to the appropriate responder. In this example the system will find a responder object that handles the event and callbacks of mouseDown event.

Mouse clicking usually results in UI refreshing. After the responder processes the event, it marks the view dirty. `libs-gui` then calls `updateWindows` to refresh the window content. The application will request the backend to flush graphics, which triggers `XWindowSystem` to perform a flush command to process all required drawcalls. At the end, the backend notifies the responder by using `_processUpdateRegion` and `drawRect` methods to complete the redraw.



**Figure 5.** GNUstep Graphics Rendering Update

Figure 5 shows the graphics rendering process for an application using GNUstep framework.

When an application needs to update its UI, a view is marked as needing update by using `setNeedsDisplay` method. This will flag the area as dirty and require redraw without immediately proceeding with the rendering process. When the application loop reaches an appropriate point, `displayIfNeeded` is called for the actual drawing.

AppKit (`libs-gui`) manages the rendering process by calling `lockFocus` on the view that is marked dirty. This will set the view as the current drawing destination and initializes the graphics context. This context creation process is done by `libs-gui` and `libs-back`.

libs-back here is a bridge that abstracts low-level rendering pipeline to high-level drawing APIs. It communicates with the platform-specific WindowServer to allocate necessary resources. The WindowServer represents the system's display server, for example, X11 and Win32.

Once all preparations are done, libs-gui will execute the drawRect method with specific drawing code. These operations are translated by libs-back into backend-specific draw commands. These draw commands will be used by WindowServer and further translated to draw calls for the GPU to execute. After everything is finished, libs-gui unlocks the focus to finalize the drawing process. By this time the updated content might not directly show, until the application flushes the graphics and instructs libs-back to communicate with the WindowServer, which then refreshes the screen buffer and changes will be on screen.

## **8. Conclusion & Lessons Learned**

From our further exploration of GNustep and its source code, we have noticed some significant differences from the conceptual architecture we had for the previous submission with the concrete architecture we observed in this report. After we analyzed the relationship and dependencies by using the Understand tool, we saw some unexpected dependencies between different components and subsystems in GNustep. For example, in our conceptual architecture, we thought that the libs-gui depends on libs-base, but this is not true in the actual implementation.

At the same time, our team met some difficulties during the analysis process. GNustep is a relatively old framework that was made in 1995. Although the major components are still under active development, there is very limited documentation and information about each subsystem. This causes significant delay to our analysis process and we have to look into the source code for deeper analysis.

Finally, we also noticed that team communication needs some improvement. At the beginning of A2, we have assigned tasks to everyone. However, as the analysis continues, we found out that some parts of this deliverable depend on one of our group member's work, for example, divergence analysis needs the concrete architecture analysis to finish first. We will take extra caution on task distribution for the upcoming group works.

## **9. References**

Gnustep. (n.d.-a). *Gnustep/apps-gorm*. GitHub. <https://github.com/gnustep/apps-gorm>

Gnustep. (n.d.-b). *Gnustep/Libobjc2*. GitHub. <https://github.com/gnustep/libobjc2>

Gnustep. (n.d.-c). *Gnustep/libs-back*. GitHub. <https://github.com/gnustep/libs-back>

Gnustep. (n.d.-d). *Gnustep/libs-base*. GitHub. <https://github.com/gnustep/libs-base>

Gnustep. (n.d.-e). *Gnustep/libs-corebase*. GitHub. <https://github.com/gnustep/libs-corebase>

Gnustep. (n.d.-f). *Gnustep/libs-GUI*. GitHub. <https://github.com/gnustep/libs-gui>