CISC 322 – Assignment 2

Concrete Architecture Report

March 14, 2025

Group 10:
Chuka Uwefoh – 22khl2@queensu.ca
Lore Chiepe – 21lpc2@queensu.ca
Henry Chen – 19hc38@queensu.ca
James Wang – 22jw16@queensu.ca
Ryan Jokhu – 21raaj@queensu.ca
Zachary Stephens – 20zs46@queensu.ca

Table of Contents

Abstract

This report details the concrete architecture of the GNUstep framework, focusing on the implementation of its main subsystems and their interactions between each other. We examine the derivation of the GNUstep architecture from the OpenStep specification, with note on decisions made to ensure modularity and compatibility. Followed by an analysis of the concrete architecture and its major subsystems, with a focus on lib-back; our chosen 2nd layer subsystem. Using reflexion analysis, discrepancies between the conceptual and concrete architectures are identified with rationales. Furthermore, sequence diagrams are provided for illustrating key uses cases within the framework to demonstrate concrete method calls and interactions. Finally, we discuss lessons learned during the analysis process.

Introduction and overview

Purpose

This report aims to provide a detailed analysis of GNUstep's concrete architecture, focusing on the actual implementation and interactions between its subsystems with the assistance of the Understand analysis tool. On top of that, by comparing the concrete architecture with the conceptual design, we aim to identify and rationalize discrepancies within the implementation.

Report Organization

This report is structured to systematically analyze GNUstep's concrete architecture and its relationship to the conceptual design. Beginning with a overview of the derivative process, followed by architectural analysis alongside auxiliary information that assists in understanding the structure.

Derivation Process

To identify and analyze the concrete architecture of GNUstep, the Understand software by SciTools was used along with publicly available system documentation and diagrams. Through Understand, we extracted detailed dependency graphs, such as "depends on", "depended on by", and butterfly graphs. These could be used to visualize and investigate the internal structure and relationships between GNUstep's subsystems. These visualizations allowed us to identify concrete subsystems and their interactions, as well as compare them against our conceptual architecture of GNUstep, which we previously based on official documentation and community resources.

To relate the conceptual and concrete architectures, the organization of source code files and their functional roles could be analyzed. If the role of a file or component was unclear, its purpose could be uncovered through its naming conventions, position within dependency graphs, or by opening the file and reading more about its functions. This comparison helped us explore concepts like architectural drift and erosion, which are components that interact beyond the conceptual boundaries or unexpected dependencies.

In addition to high-level architecture analysis, we performed a deep dive into the specific subsystem of libs-back, analyzing its internal structure and dependencies. Through this, we assessed how well the subsystem's concrete implementation aligns with its conceptual design.

Overall, Understand was an important tool for identifying dependencies to generate the concrete architecture, enabling us to refine our conceptual model to account for discrepancies and insights gained through the analysis. This structured process allowed us to evaluate GNUstep's architecture at both a high level and within individual subsystems.

Conceptual Architecture

The **conceptual architecture** of GNUstep outlines its core components and their roles at a high level. It includes **core libraries** (**libs-base** and **libs-corebase**) for fundamental data structures, **graphical frameworks** (**libs-gui** and **libs-back**) for UI development, and **development tools** like **Gorm** and **GNUstep-make** for building applications. Additionally, it highlights **cross-platform compatibility** as a distinct layer, ensuring applications run smoothly on different operating systems. This model provides a broad understanding of how GNUstep supports both graphical and non-graphical application development.

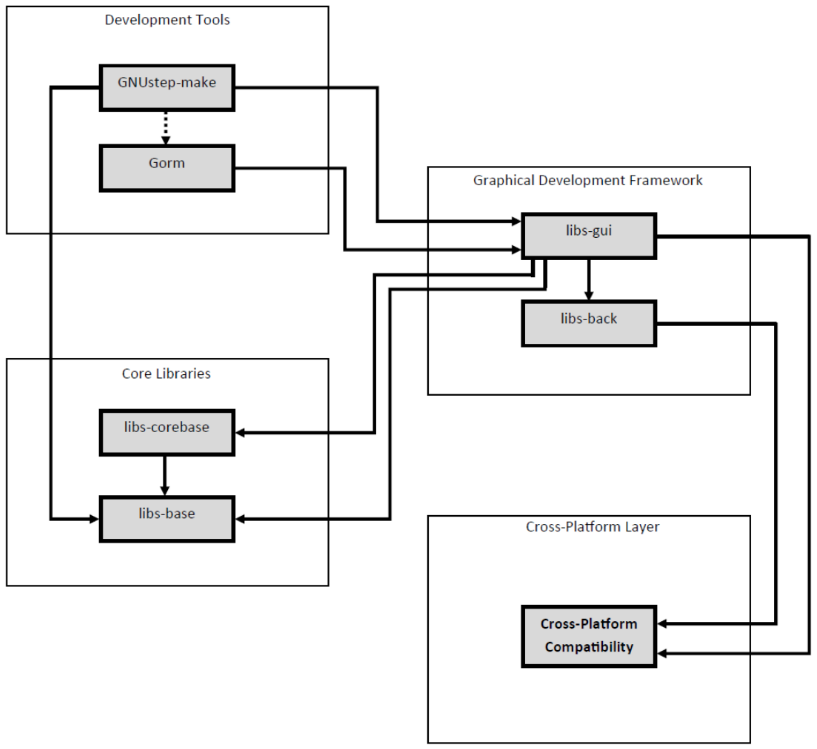


Figure 1 - Conceptual Architecture

Concrete Architecture

Overview

The concrete architecture of GNUstep builds upon the conceptual architecture by detailing the actual implementation, dependencies, and structural interactions between components. This section focuses on how the major subsystems interact at the implementation level, including specific modules, libraries, and frameworks. The layered structure remains intact but includes additional concrete details regarding method calls, dependencies, and platform-specific adaptations. We were able to get a graph of the system using Understand by Scitools.

High-Level Structure

GNUstep follows a layered architecture with clear distinctions between different functional layers. The primary layers include:

1. **Application Layer** – User-developed applications using GNUstep libraries.
2. **GUI Layer (libs-gui)** – Manages graphical elements, user interaction, and event handling.
3. **Foundation Layer (libs-base & libs-corebase)** – Provides core Objective-C classes for data structures, I/O, and threading.

4. **Rendering and Backend Layer (libs-back)** – Interfaces with system graphics backends (Cairo, X11, Windows GDI, etc.).
5. **Build System and Tools** – GNUstep-make, Gorm, and ProjectCenter facilitate development and compilation.
6. **Runtime Layer (libobjc2)** – Provides the Objective-C runtime, including message passing, dynamic method resolution, and introspection.

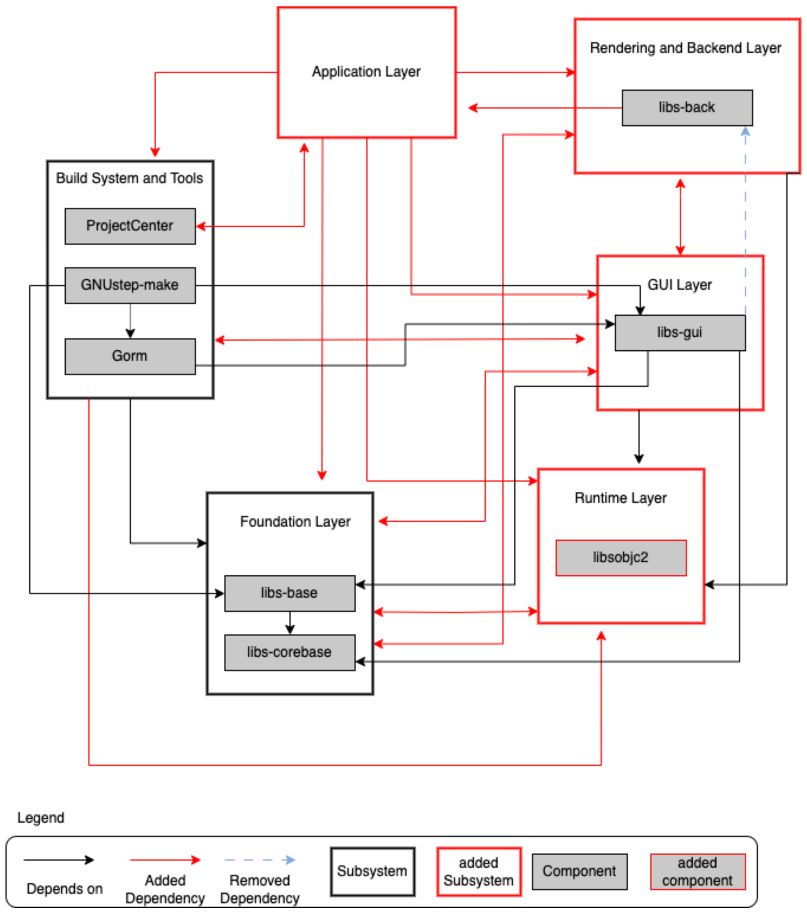


Figure 2 - Concrete Architecture built from Understand

Subsystem Breakdown

1. Application Layer

This layer consists of user-developed applications that interact with the underlying GNUstep framework. These applications make calls to GUI and foundation layers to render content and handle user input.

2. GUI Layer (libs-gui)

- Implements Cocoa-compatible graphical elements.

- Handles event propagation via NSApplication and NSResponder.
- Interfaces with libs-back for rendering.

3. Foundation Layer (libs-base & libs-corebase)

- libs-base provides essential classes like NSString, NSArray, NSDictionary, and NSFileManager.
- libs-corebase extends libs-base with additional utilities, memory management, and threading support.

4. Rendering and Backend Layer (libs-back)

- Abstracts platform-specific rendering via backend modules.
- Supports rendering on X11, Windows GDI, and Cairo.
- Bridges graphical calls from libs-gui to native OS rendering.

5. Build System and Tools

- **GNUstep-make**: Automates compilation and linking.
- **Gorm**: GUI builder that generates Objective-C code.
- **ProjectCenter**: IDE for managing GNUstep projects.

6. Runtime Layer (libobjc2)

- **Method Dispatch** – Handles Objective-C message passing (objc_msgSend).
- **Dynamic Class and Method Loading** – Supports runtime creation and modification of classes and methods.
- **Introspection** – Provides metadata about classes, methods, and properties.
- **Memory Management** – Implements reference counting (ARC & manual retain/release).
- **Categories and Protocols** – Enables method extensions and interface abstraction.
- **Exception Handling** – Supports Objective-C exceptions (@try, @catch, @finally).

The **conceptual architecture** of GNUstep provides a high-level overview of its components, focusing on their intended roles and relationships rather than their specific implementations. It categorizes GNUstep into **core libraries**, **graphical frameworks**, **development tools**, and **cross-platform compatibility**, presenting a broad understanding of how the system supports application development. This abstraction helps developers see how different parts interact, such as how **libs-base** and **libs-corebase** provide foundational functionality, while **libs-gui** and **libs-back** handle graphical elements. The conceptual model also highlights GNUstep's **cross-platform capabilities** as a distinct layer, reinforcing the idea that applications should run on multiple operating systems with minimal changes.

In contrast, the **concrete architecture** translates this abstract model into a structured breakdown with clearly defined layers, each assigned specific libraries and responsibilities. It introduces technical distinctions such as the **Runtime Layer (libobjc2)**, which is crucial for Objective-C execution but not explicitly emphasized in the conceptual model. Additionally, instead of treating cross-platform compatibility as a separate layer, the concrete architecture embeds it within the **Rendering and Backend Layer (libs-back)** and the **Foundation Layer (libs-base & libs-corebase)**, reflecting how platform independence is implemented through backend abstractions. The concrete view is more detailed and structured, focusing on how the system is built and operates, making it essential for developers implementing and optimizing GNUstep applications.

Key Interactions (Use Cases)

Application Lifecycle

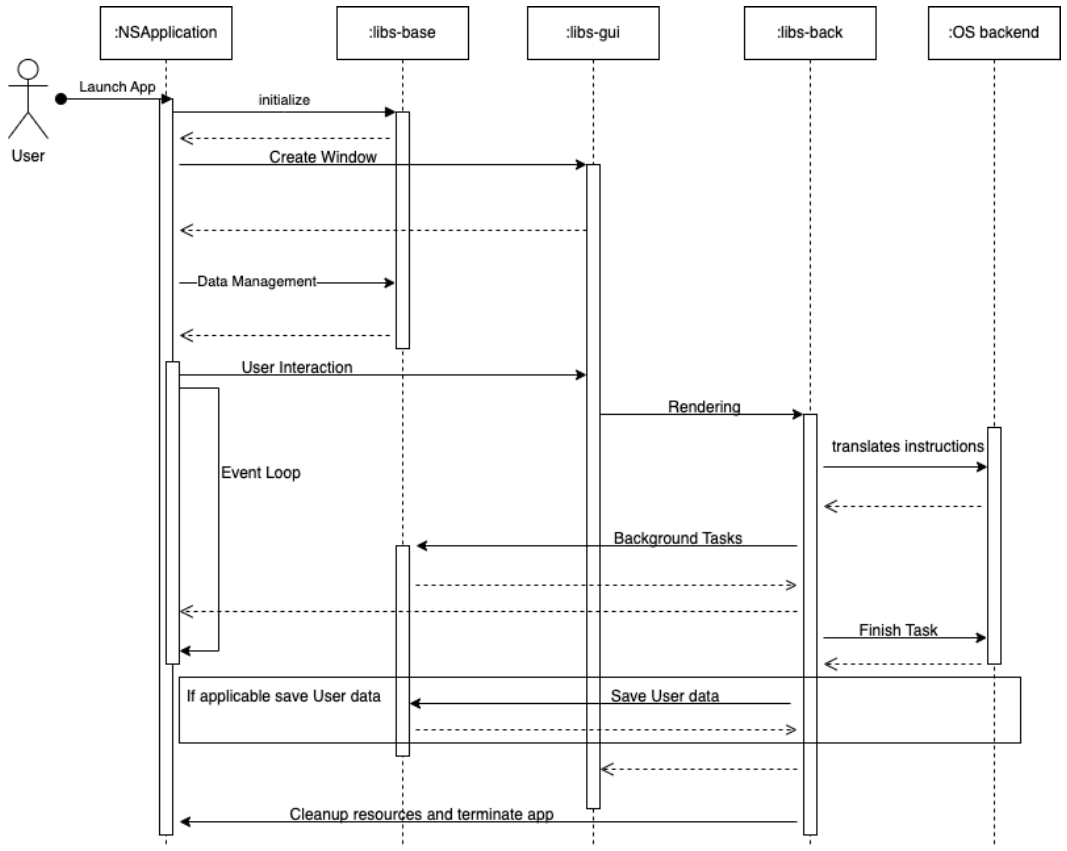


Figure 3 - Application lifecycle use case

1. The user launches a GNUstep application.
2. NSApplication initializes, setting up the event loop.
3. The application interacts with libs-base for data management.
4. libs-gui handles user interactions and passes rendering tasks to libs-back.
5. libs-back translates rendering instructions to the underlying OS backend.

Event Handling Sequence

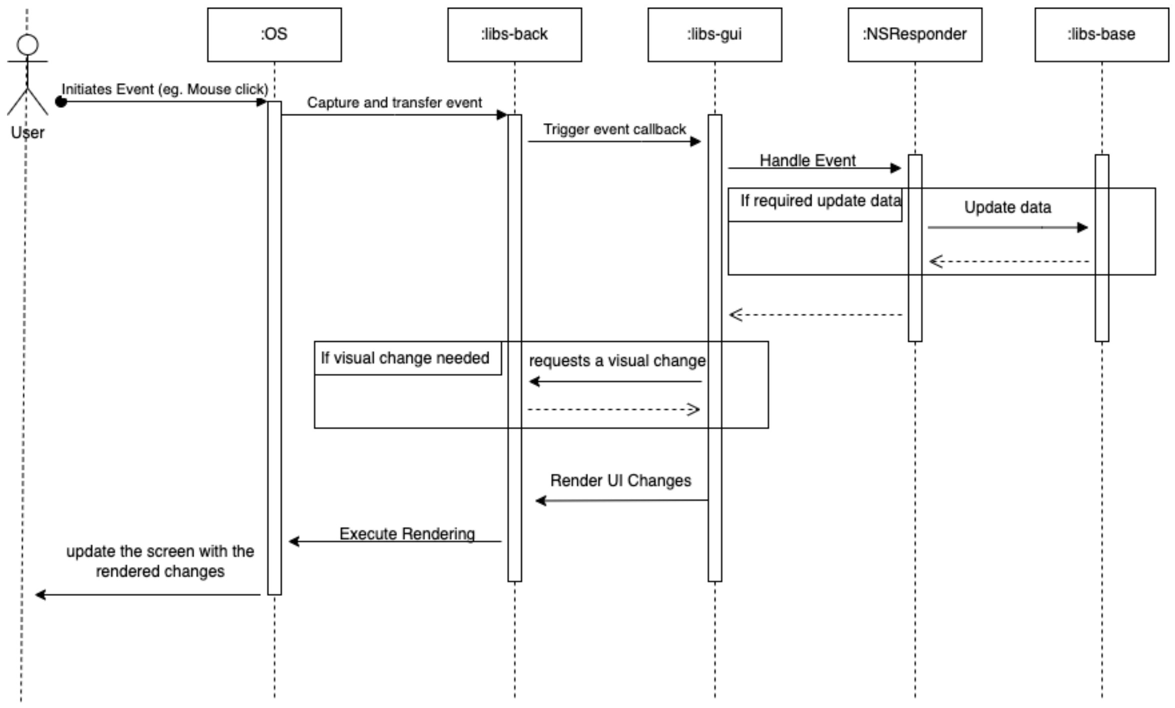
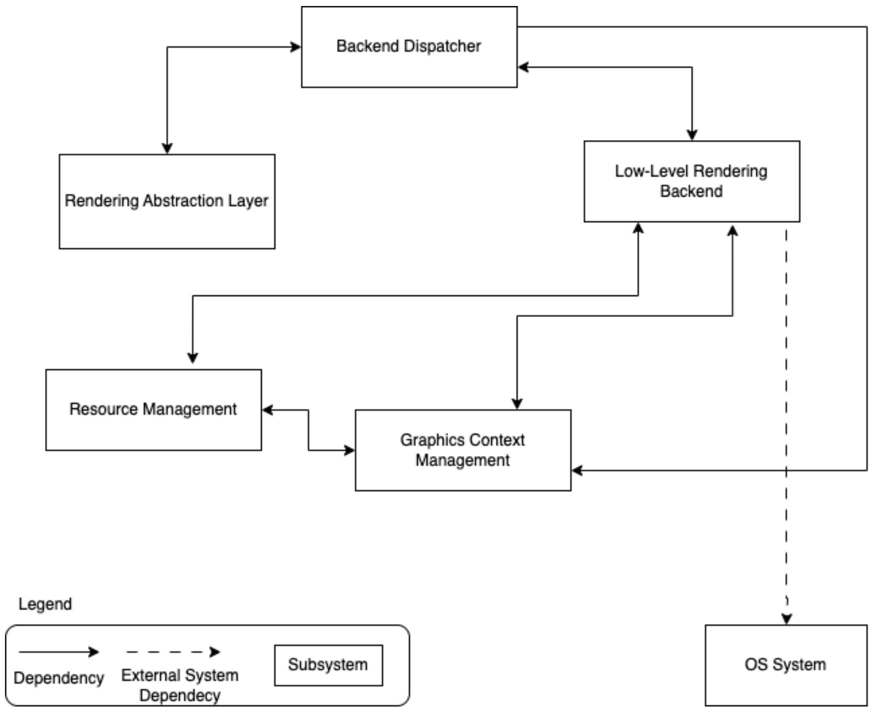


Figure 4 - Event handling sequence use case

1. User input is captured by the OS and forwarded to libs-back.
2. libs-back notifies libs-gui components via event callbacks.
3. NSResponder processes the event and updates the interface.
4. If necessary, the application logic updates state via libs-base.
5. The UI is refreshed via libs-gui, using libs-back to render changes.

Chosen 2nd-Level – libs-back



Backend Dispatcher
Rendering Abstraction Layer
Low-Level Rendering Backend
Resource Management
Graphics Context Management
OS System
Legend
Dependency
External System Dependecy
Subsystem

Figure 5 - Conceptual architecture of libs-back

This section reflects on the **libs-back** subsystem, analyzing its architecture from both conceptual and concrete perspectives. The conceptual architecture provides an overview of how the system components interact at a high level, abstracting platform-specific details to ensure efficient rendering. The concrete architecture delves deeper into the specific flow of processes, detailing the step-by-step execution and communication between components, such as the **Rendering Abstraction Layer**, **Backend Dispatcher**, and **Low-Level Rendering Backend**. Together, these views offer a comprehensive understanding of the subsystem's design and operation.

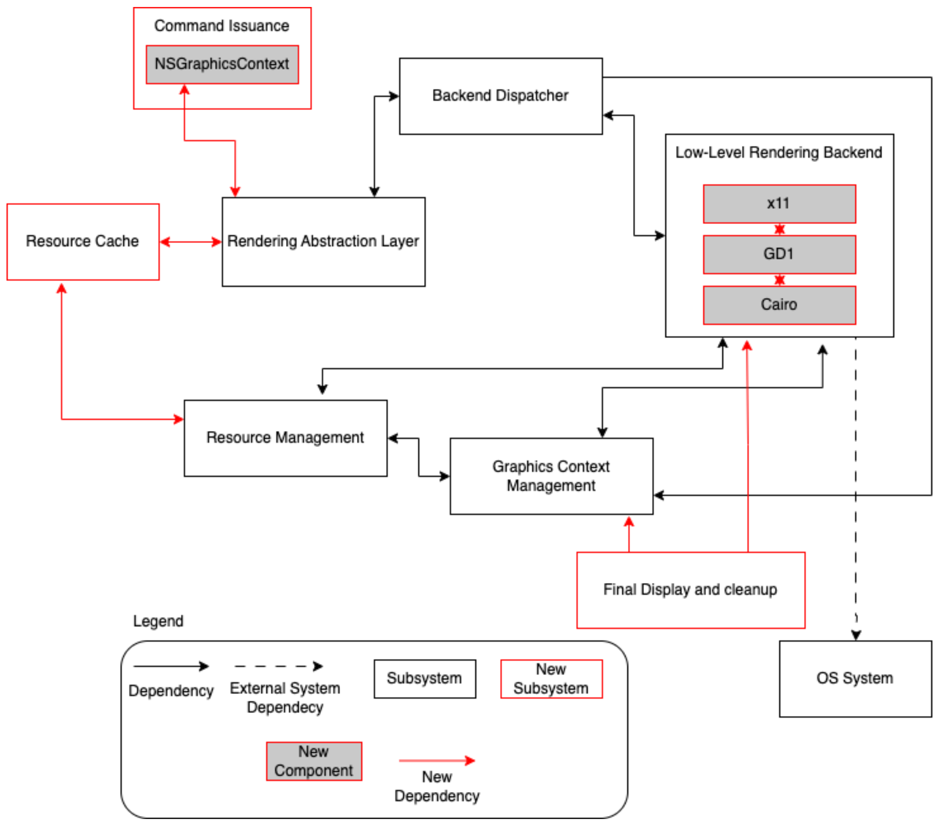


Figure 6 - Concrete architecture of libs-back

NSGraphicsContext: Initiates rendering requests, such as drawing shapes or text. Passes commands to the **Rendering Abstraction Layer (RAL)** for standardization.

Rendering Abstraction Layer (RAL): Standardizes rendering commands and abstracts platform-specific details. Forwards requests to the **Backend Dispatcher** and checks the **Resource Cache** for optimization.

Backend Dispatcher: Determines the appropriate backend (e.g., X11, GDI) based on the system. Routes rendering commands to the **Low-Level Rendering Backend** for execution.

Low-Level Rendering Backend: Executes backend-specific rendering via system APIs (X11, GDI, Cairo). Interacts with **Graphics Context Management** and **Resource Management** to handle drawing.

Graphics Context Management: Maintains rendering state, including color, transformations, and clipping. Ensures the correct context is applied before execution.

Resource Management: Handles loading, allocation, and caching of textures, images, and other resources. Works with the **Resource Cache** to optimize performance.

Resource Cache: Stores preloaded assets to avoid redundant computations. Ensures efficient reuse of frequently accessed resources.

Final Display & Cleanup: Transfers rendered content to the screen if double buffering is enabled. Resets the **Graphics Context** to prepare for the next render cycle.

Dependencies

NSGraphicsContext → Rendering Abstraction Layer

> NSGraphicsContext sends rendering requests (e.g., drawRectangle) to the **Rendering Abstraction Layer**, ensuring a standardized interface.

Rendering Abstraction Layer → Backend Dispatcher

> The **RAL** forwards standardized commands to the **Backend Dispatcher**, which determines the appropriate backend based on the system.

Backend Dispatcher → Low-Level Rendering Backend

> The dispatcher routes rendering commands to the selected **Low-Level Rendering Backend** (e.g., X11, GDI, Cairo) for execution.

Low-Level Rendering Backend → System API Interface

> The **backend** calls OS-specific APIs via the **System API Interface**, ensuring compatibility with the underlying platform.

Rendering Abstraction Layer → Resource Cache

> Before processing a rendering request, the **RAL** checks the **Resource Cache** to reuse previously loaded assets, improving efficiency.

Resource Cache → Resource Management

> If an asset isn't cached, the **Resource Management** component loads or rasterizes it before passing it back to the **Resource Cache**.

Graphics Context Management → Low-Level Rendering Backend

> The **Graphics Context Management** ensures the correct state (color, transform, clipping) is applied before the backend executes rendering commands.

Low-Level Rendering Backend → Double Buffering System

> If double buffering is enabled, the backend first renders off-screen in the **Double Buffering System** before copying it to the display.

Double Buffering System → Final Display & Cleanup

Once rendering is complete, the **Double Buffering System** transfers the final image to the screen and resets the rendering state.

Reflexion Analysis and Architectural Discrepancies

Through a comparison of our conceptual architecture report against the concrete implementation as analyzed by the Understand tool, we can identify a few notable discrepancies between the two.

High-Level Architecture

1. Bidirectional Component Dependencies

The conceptual architecture depicted a unidirectional dependency from libs-gui to libs-back, suggesting a clean layered architecture with one-way relationships. However, the concrete implementation features bidirectional dependencies between these components. This difference likely emerged from a practical need for two-way communication for event handling and rendering callbacks, making strict and unidirectional layering impractical.

Chosen 2nd Subsystem – libs-back

The conceptual architecture of libs-back relatively simple and has some less important components that are not shown. An example of this is the *Final Display & Cleanup* component depicted in the concrete architecture which was likely added during development when double buffering was added. Another example would be the *Resource Cache* which is an optimization to avoid having to re-render the same assets that were already rendered in the rendering abstraction layer as well as providing assets to it.

The final new subsystem is the *NSGraphicsContext* that handles passing of the request for rendering to the *Rendering Abstraction Layer (RAL)*. This component/subsystem does not appear on the conceptual architecture due to it being only slightly different from an input.

Discrepancies and Rationales for libs-back

1. Bidirectional Dependencies with libs-gui
   - *What we expected*: libs-gui would send rendering requests one way to libs-back, keeping things clean and separate
   - *What happened*: Instead, libs-back talks back to libs-gui, providing updates on rendering, managing state, and handling platform-specific quirks.
   - This back and forth is necessary for real-time updates, event handling, and making sure everything runs smoothly on different platforms. If there was no communication between the two, things like screen resolution changes wouldn't work as efficiently
2. Direct Interaction with System APIs
   - *What we expected*: libs-back would act as a simple middleman, just forwarding rendering commands without directly interacting with system APIs.

- *What happened*: instead of staying abstracted, libs-back talks directly to X11, Windows GDI, and Cairo, bypassing come expected abstraction layers
- Cutting out extra layers helps with performance, avoiding unnecessary overhead and allowing libs-back to use platform specific optimizations that wouldn't be possible with a strict separation

3. Platform-Specific Conditional Code
   - *What we expected*: A unified API that automatically handles platform differences behind the scenes.
   - *What happened*: Instead, the libs-back codebase is filled with hardcoded platform checks like #ifdef _WIND32 and #ifdef __APPLE__, meaning different code runs depending on the OS
   - This makes platform-specific maintenance easier and reduces duplicate code, but it also increases system coupling, making it harder to keep things completely modular.
4. Decentralized Resource Management
   - *What we expected*: A single shared resource cache that would store assets efficiently and prevent unnecessary reloading
   - *What happened*: different parts of libs-back handle their own caching instead of one centralized system
   - A fully centralized cache could cause sync issues and slow things down overall, especially with multi-threaded rendering. By decentralizing caching, each subsystem can manage resources more flexibly without bottlenecks.

Data Dictionary

Bidirectional Dependencies: A relationship where two subsystems or components depend on each other for functionality. For example, `libs-gui` and `libs-back` have bidirectional dependencies for event handling and rendering updates.

Cross-Platform Compatibility: A design goal of GNUstep to ensure that applications can run on multiple operating systems with minimal changes. This is achieved through abstraction layers like `libs-back`.

Extensibility: The ability of a system to be extended with new features or functionalities without requiring significant changes to the existing architecture.

System API Interface: An interface within `libs-back` that allows the Low-Level Rendering Backend to interact with OS-specific APIs, ensuring compatibility with the underlying platform.

Naming Convention

Application Lifecycle: The sequence of events that occur when a GNUstep application is launched, initialized, and interacts with the framework's layers to manage data, handle user input, and render content.

Double Buffering System: A mechanism within libs-back that renders content off-screen before transferring it to the display. This reduces flickering and improves rendering performance. Here's the requested format for the remaining terms and concepts from the report:

Event Handling: A process where user input is captured by the OS, forwarded to `libs-back`, and processed by `libs-gui` components. It involves updating the interface and refreshing the UI via rendering.

Conclusion

Over the course of the analysis of the concrete architecture of GNUstep it emerged that some of the subsystems in the conceptual architecture are broken down into multiple interdependent subsystems. The concrete architecture is in a layered style same as the conceptual architecture but some of the layers are split up and the application layer is shown and is very connected with the other subsystems. This pattern continues in the further analysis of libs-back, with the concrete architecture being more interconnected often through adding smaller subsystems. This interconnectivity in the layered style allows for easier and less computationally heavy use of the GNUstep application.

Lessons Learned

One of the key lessons was experiencing the challenge of maintaining a consistent shared understanding within the group while analyzing a complex software architecture. Although members contributed to analyzing different subsystems, we believe that a separate more detailed collaboration document could have improved our efficiency. Such a document would ideally contain a plentiful number of detailed screenshots, explicit file paths, or direct references to the Understand tool outputs to enable better cross-verification of findings. This would have ensured that everyone had a unified understanding of how particular components were identified and interpreted.

References

GNUstep Project, "GNUstep Official Website," GitHub Pages. [Online]. Available: https://gnustep.github.io/. [Accessed: Mar. 12, 2025].

GNUstep Project, "Developer Documentation," GNUstep. [Online]. Available: https://www.gnustep.org/developers/documentation.html. [Accessed: Mar. 14, 2025].

GNUstep Project, "Developer Tools," GNUstep. [Online]. Available: https://www.gnustep.org/experience/DeveloperTools.html. [Accessed: Mar. 14, 2025].

[9] NeXTSTEP/OpenStep, OpenStep Development Tools, PDF document. [Online]. Available: https://cdn.preterhuman.net/texts/computing/nextstep-openstep/802-2110%20-%20OpenStep%20Development%20Tools.pdf. [Accessed: Mar. 14, 2025].