GNUSTEP CONCEPTUAL ARCHITECTURE REPORT

Group 12 – CISC/CMPE 322/326 – Queen's University

Daniel Tian
21dt41@queensu.ca
Samuel Tian
21st114@queensu.ca
James Choi
19jc132@queensu.ca
Christian Pierobon
christian.pierobon@queensu.ca
Luca Spermezan
22ls18@queensu.ca
Andrew Bissada
21ajb37@queensu.ca

Abstract

GNUstep is an open-source, cross-platform framework that provides a development environment for designing GUI applications. Users do not interact with GNUstep directly, but do so through either the Gorm interface builder, or the applications built on the GNUstep API. GNUstep adopts a layered architectural style and consists of two primary layers: the Foundation layer and the Application layer. Other alternative architectural styles that were considered are object-oriented and publish-subscribe style, which have relevant aspects present in GNUstep. The Foundation layer holds components of GNUstep responsible for low-level functionalities unrelated to GUIs. The Application layer holds components responsible for GUI-related functionalities for both the front and back-end. Gorm can also be considered as an additional layer above the Application layer, and consists of the main Gorm application, and the Gorm GUI. In GNUstep's conceptual architecture, Gorm is built using the GNUstep framework, meaning it uses the services provided by GNUstep to build itself. Some design patterns used in GNUstep include MVC for application development and Chain of Responsibility for handling user events. The evolution of GNUstep is heavily dependent on the evolution Apple's Cocoa, as GNUstep aims to implement a one-to-one equivalent API. The external interfaces that are relevant to GNUstep are the Window Manager, Display Server, and the OS, with which it communicates using the necessary API. Two major use cases of the GNUstep system involve a developer creating a button with audio in Gorm, and a user pressing such a button through a GNUstep application, which then opens a URL.

Introduction

Purpose

The purpose of this report is to derive the conceptual architecture of GNUstep, a cross-platform, open-source framework that provides a development environment for designing GUI applications. This includes understanding the various structural components in GNUstep, the dependencies and interactions between them, the architectural styles employed in GNUstep, and the design patterns that are present. This also includes understanding how the Gorm interface builder and other external interfaces fit into the rest of the conceptual architecture. This report should be read by a developer who wants to develop GUI applications using GNUstep, and wants to understand the characteristics of the underlying architecture for the application they are building. This report should also be read by anyone who wants to contribute to GNUstep, as understanding the conceptual architecture is important for ensuring that any changes and evolutions to the API are compliant with the framework's original intent and design.

Organization

This report is organized into five main sections. The first section is the introduction, which includes an abstract, information regarding the purpose of this report, and the derivation process for the conceptual architecture. The second section discusses the conceptual architecture of GNUstep, and is the largest section of the report. In this section, the structural components, component dependencies, and component interactions in the conceptual architecture are discussed. This section also includes discussion about evolvability, testability, and potential bottlenecks in the system. The third section discussed external interfaces that are outside of the scope of GNUstep, yet are still relevant to its conceptual architecture. The fourth section discusses two use cases that illustrate the flow of control in GNUstep, along with their associated sequence diagrams. Finally, the fifth section discusses key conclusions and lessons from this report.

Conclusions

The conceptual architecture of GNUstep follows a layered style, which provides multiple levels of abstraction to both users and developers who interact with the system. This makes GNUstep a solid cross-platform framework for user-friendly GUI development. However, a key bottleneck in the GNUstep framework involves the *Main Application Environment*, which could easily be overwhelmed by the multiple roles that it needs to fulfill. GNUstep should focus on continuing to evolve with Apple Cocoa so that it can remain relevant as a strong framework for GUI application development.

Derivation Process

The derivation process for the conceptual architecture of GNUstep consisted of three main stages. The first stage involved consulting the official documentation for GNUstep, such as the GNUstep website[1], Wiki[2], and official system overview page[3]. In this stage, we derived the initial design of the conceptual architecture for GNUstep, including its components and styles. The second stage involved consulting the official documentation for the OpenStep specification[4]. In this stage, we compared OpenStep and GNUstep to make adjustments to our conceptual architecture. The third stage involved consulting the official documentation for Apple Cocoa, which mainly consisted of official documentation[5]. In this stage, we further confirmed our conceptual architecture from the first two stages and derived some extra design patterns. Finally, we consulted the official Gorm application guide[6] to derive Gorm's placement in the conceptual architecture.

Conceptual Architecture

Architectural Styles

Layered Style

The primary architectural style in GNUstep is layered style with two main layers: Foundation and Application. The Application layer can be further partitioned into two sub-sections, which are the main Application Kit and the Back-end. Gorm and any other applications built on GNUstep form the Applications layer at the top.

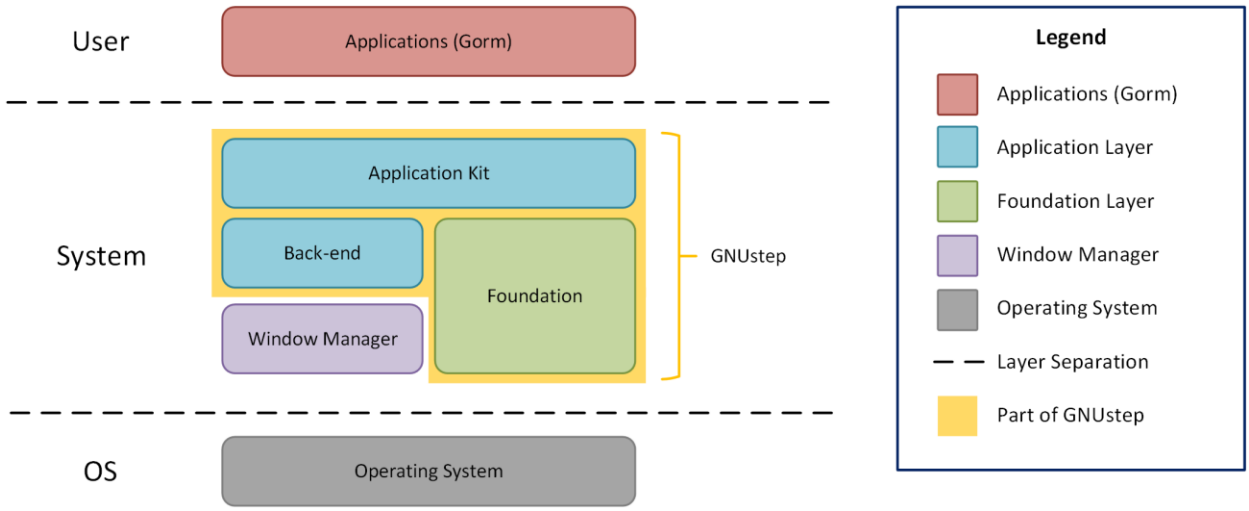


Figure 1. GNUstep layered architectural style

These three layers can be organized hierarchically, with each layer interacting only with adjacent layers. The layer ordering from bottom to top is: *Foundation*, *Application*, *Gorm*

The Foundation layer is responsible for all non-GUI-related components of GNUstep. The components in this layer provide functionalities that do not directly support the graphical interface displayed to the user. This layer also interacts directly with the OS, and has services that can perform low-level operations. All of the operations done in this layer are hidden from the user, who can only interact with the topmost level.

The Application layer is responsible for all GUI-related components of GNUstep (Application Kit), as well as components related to the back-end of the GUI (Back-end). The components in this layer provide functionalities that directly support the application running the graphical interface displayed to the user. This layer connects Gorm to the Foundation layer through the services that it provides. Gorm never directly interacts with the Foundation layer in the conceptual architecture, but only interacts with the Foundation layer indirectly through operations in the Application layer.

The topmost layer of the conceptual architecture consists of both Gorm, and other Applications that are built on GNUstep. The components of this layer are what interact directly with a developer using Gorm, or a user using a GNUstep application. This layer then interacts with the Application layer beneath it to access the services provided by GNUstep for running Gorm or any other application.

Structural Components

Foundation

Value Data Module:
This component provides services for managing primitive data, such as strings, numeric types, dates, and collections. Its primary roles are to create new instances of these data, which can be used by other objects in the system, and to perform operations related to these data, such as type conversion, arithmetic operations, string manipulation, and so on. It provides the foundation of basic data types to build more complex GUI objects.

OS Service Modules:
These components are responsible for handling OS-level operations in a GNUstep application. One of the main features of GNUstep is that its applications can be run on any operating system, so these components are crucial to support this portability requirement, as they ensure that the application can utilize the operating system services across multiple platforms. These components can be tested by running GNUstep applications on multiple different operating systems to see if they are compatible and operate smoothly.

- **Task Concurrency Module:** Manages multithreading across processes, synchronization through resource locks, and concurrent operations among higher-level components. Also manages low-level IPC through sockets, ports, and pipes.
- **File Management Module:** Manages connections to files and filesystems within the OS. Responsibilities include reading and saving files, managing directory structure, and loading file data for use during development and at runtime.
- **Networking Module:** Manages network connections, searches for and connects to available network services, and handles URL-related operations, such as opening and managing data in URLs, and supporting different URL protocols.

Notification Module:
This component provides the publish-subscribe style services in GNUstep. It handles operations related to broadcasting notifications to objects, communicating with the notifying objects about the status of their notifications, and handling notification queues. A major non-functional requirement for this component is performance, specifically response time, as other objects depend on it for real-time interaction with users.

Serialization and Archiving Module:
This component provides services for archiving objects into storable data when being saved into a file, as well as unarchiving those stored objects into usable data structures when they need to be loaded into the system for use.

Application

Main Application Environment Module:
This component provides services for managing the environment of the application during runtime. This includes managing the main event run loop, dispatching events to windows, and connecting with external interfaces such as the window manager and display server. Major non-functional requirements for this component are scalability and throughput, as it handles many responsibilities related to the main functionality of an application. This makes this component a potential bottleneck in the system if it is unable to keep up with high request load, thus causing delays in communication with the external interfaces.

Windowing Module:
This component provides services for managing an application's active windows. Its main functions include configuring window appearances and dimensions, managing element hierarchies in windows, and dispatching events to elements in the window.

GUI Element Modules:
These components provide services for managing the various GUI elements in an application's windows, and they define how those GUI elements appear and function in the application. These components are a good target for testability, as GUI elements must

work in conjunction with the main application and the windows in which they exist, thus allowing a developer to test multiple component interactions at the same time.

- **Control Element Module:** Manages control elements in the GUI, such as buttons. Creates control element objects and encapsulates all related functionalities within, such as those involving appearance, properties, and event-driven actions.
- **Graphics Module:** Manages graphical images in the GUI through objects that hold and determine image data, image display, storage, and integration into the GUI.
- **Text Module:** Manages text in the GUI through objects that hold and determine string data, text formatting, text display, and integration into the GUI.
- **Document Module:** Manages document files in the GUI through objects that hold and determine document display, document editing, and document state.

Input Event Handling Module:
This component provides services related to handling and dispatching input events. When an event is received by an object, such as a GUI element, this component determines the courses of action that can be taken in response. Similar to the notification module, a key requirement for this component is response time, as it is important for allowing the system to respond interactively to users in real-time. Therefore, this component is performance-critical, and should be designed carefully to prevent delays in user experience.

Drawing Module:
This component provides services for drawing the GUI application on the screen. When objects need to be displayed, this component determines how the display server renders those objects. It also manages functionalities related to colours and graphic contexts. Some major non-functional requirements for this component are availability and response time, as this component must ensure that the GUI is always visible with minimal delay.

Audio Module:
This component manages audio-related functionalities in the application through objects that control and determine audio data storage, audio playback, audio volume, and audio operations in response to events or notifications.

Gorm

Gorm GUI:
This component represents the graphical interface of Gorm that the user can interact with to develop GNUstep applications. In the conceptual architecture, this GUI is built using the services provided in the Application layer of GNUstep. This component provides users with an interactive environment for GUI development, and abstracts the lower-level code in the GNUstep framework. This makes GUI development in GNUstep more efficient and robust, as developers do not need to write any code themselves when designing the GUI.

Main Gorm Application:
This component represents the main Gorm application that is running when Gorm is open. In the conceptual architecture, this application is built on the GNUstep Application layer, and uses its services to run the Gorm application. It is also responsible for accessing the services provided by the GNUstep Application layer when building new GUI applications.

System Evolvability

As an open-source implementation of Apple's Cocoa, evolvability is a key issue for GNUstep. If GNUstep does not evolve with Apple, its functionalities will eventually become outdated, and it will no longer be a proper equivalent to Apple's Cocoa API. This will defeat the whole purpose and goal of GNUstep, which is to make Apple Cocoa's API for GUI development accessible in systems not running MacOS. Thus, GNUstep must prioritize evolvability as a non-functional requirement for its components, to ensure that it can keep up with new additions and changes to Apple's development tools. For example, in June 2023, Apple introduced a new feature that allowed for automatic hyphenation in text fields for non-English text. In GNUstep, the *Text Module* component would need to evolve accordingly to accommodate for the new feature. Thus, when designing this component, developers would need to ensure that it is easily modifiable, which would allow it to evolve correctly and efficiently without incurring extensive costs in the process.

Component Dependencies

The box-and-lines diagram in Figure 2 shows the GNUstep conceptual architecture dependencies. Only the major GNUstep dependencies will be discussed in further detail.

The *Main Application Environment* is the main component responsible for managing a running GUI application. Thus, it depends on the OS services to be able to manage the various tasks and operations in the application. It also depends on the *Windowing* component, as the application itself does not know how any GUI windows are organized, how they should be drawn, or how their events are to be handled. Thus, the *Windowing* component provides the *Main Application Environment* with instructions for window display. The *Main Application Environment* also dispatches events that it cannot handle to the *Windowing* component for specialized handling. The *Windowing* component then depends on the various GUI element components, as it must further pass events down the responder chain to reach the correct GUI element component.

The *Input Event Handling* component provides services to other components for handling user events. The *Main Application Environment* depends on this component for determining how events should be dispatched to the correct window, while the GUI element components depend on this component for determining how to actually handle those events. The *Windowing* component depends on this component for both reasons, as

it must both handle window-related events by itself, while also dispatching GUI element-related events to the correct GUI elements.

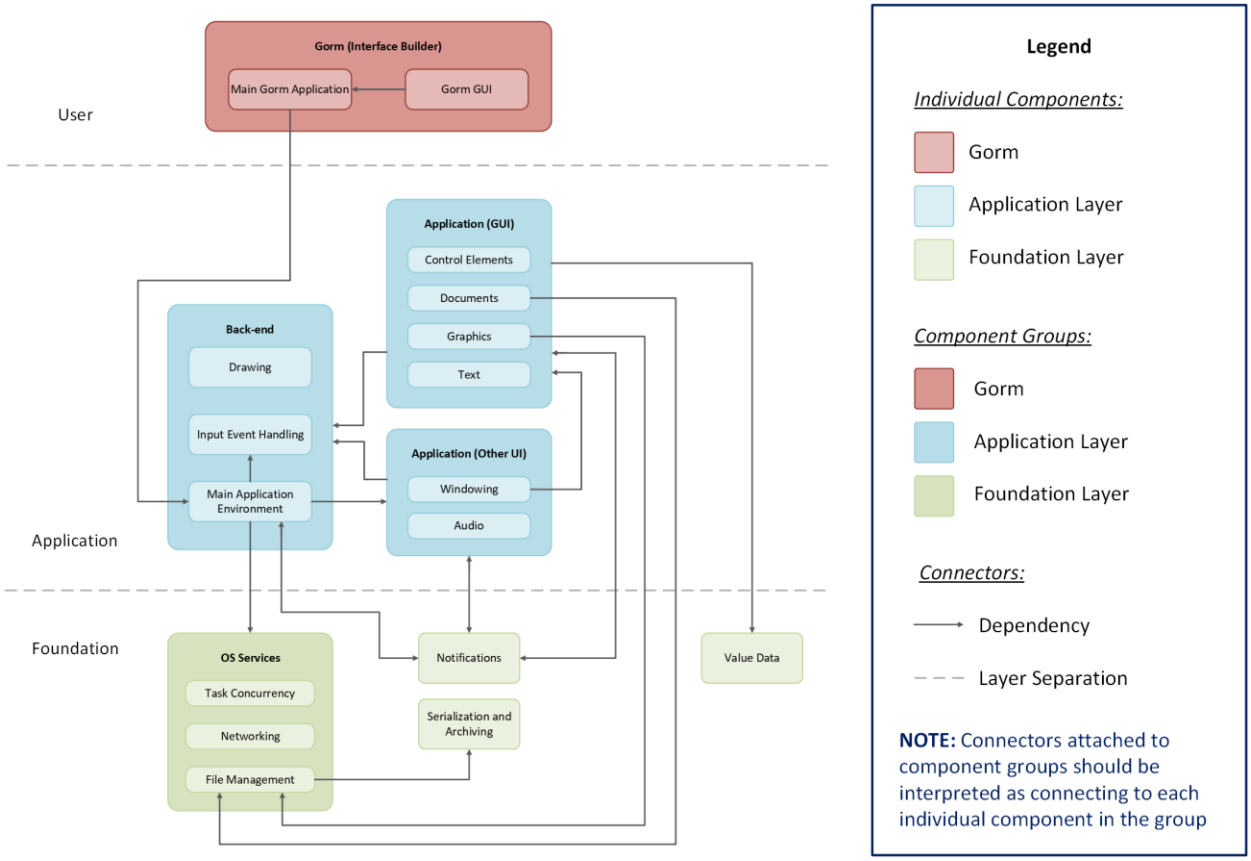


Figure 2. Box-and-lines diagram for GNUstep

The *Drawing* component is responsible for determining how objects are drawn on the screen. Thus, both the *Windowing* and GUI element components depend on this component to allow them to determine how to draw themselves. The GUI elements also depend on the *Value Data* component when they are created or edited, as their internal data depends on how data representations are defined in the *Value Data* component.

*T*he *File Management* component depends on the *Serialization and Archiving* component whenever it needs to save or load a file to or from the file system. This is so that any data structures can be converted correctly between archived and unarchived formats.

In the conceptual architecture, Gorm is built on GNUstep, meaning the Gorm application uses the components in the GNUstep framework to both run itself, and build other GUI applications. This is very reasonable conceptually, as GNUstep is designed for GUI applications, making it possible for Gorm to be built entirely with it. It would also be highly efficient, as the components in GNUstep could be used to simultaneously run Gorm and build other GUI applications, instead of separating the two tasks across different

frameworks. The *Gorm GUI* component serves as the interface a developer interacts with, and it depends on the *Main Gorm Application* component for performing the actual operations in Gorm. Thus, the *Gorm GUI* component is the front-end, and depends on the *Main Gorm Application* component, which is the back-end. The *Main Gorm Application* component depends on the *Main Application Environment* component in GNUstep to actually run the Gorm application, as well as to get access to other components for GUI building. Thus, all of Gorm's operations would be facilitated by the GNUstep services.

Alternative Architectural Styles

Object-Oriented Style

Certain aspects of object-oriented style are present in the conceptual architecture of GNUstep. The services provided by GNUstep are encapsulated within abstract data types that Gorm and other applications use when they run. These services can be grouped together into components, which act as "objects" that "invoke" one another to provide services to one another as needed. This style is then integrated into the layered style, where each layer has its own service "objects". Thus, object-oriented design works as a possible secondary architectural style within the layered style, but not as the main style.

Publish-Subscribe Style

Certain aspects of publish-subscribe style are present in the conceptual architecture of GNUstep. This is most notably seen in the *Notifications* component, which broadcasts notifications in GNUstep applications. Observers subscribe to a notification center object, which broadcasts notifications to them. Upon receiving the notification, the observers can respond as needed. This style is integrated together with the object-oriented style within the layered architecture, as "objects" in a layer can communicate either directly or through event broadcasting. However, in the context of GNUstep, publish-subscribe style is more of a design pattern than an encompassing architectural style, which is why it was rejected.

Design Patterns

Model-View-Controller

The model-view-controller (MVC) design pattern is notably used in the design of GNUstep GUI applications, as MVC is directly implied among the different GNUstep components. Gorm would also likely implement MVC design, as it is built on GNUstep. With respect to MVC, the Foundation layer provides services related to value data for building models, while the Application layer provides services related to control and display for controllers and views. It should be noted that through MVC, GNUstep automatically uses the composite, strategy, and observer design patterns as well. These design patterns are all implicitly present within MVC, and are thus relevant design patterns in GNUstep.

Chain of Responsibility

The chain of responsibility design pattern is notably used in the event handling services provided by GNUstep. This design pattern allows objects to either handle an event, or pass the event along a responder chain to allow other objects to handle it. This makes it so that no object needs to handle all possible events, and instead, different objects can handle different events, with the event always reaching the correct handler. As an example within GNUstep, consider the *Main Application Environment*, which upon receiving an event, either handles the event itself, or passes the event along the responder chain to the correct *Window*, and then the correct GUI element that is able to handle the event.

Component Interactions

Global Control Flow

The global flow of control in GNUstep is mostly facilitated by responder chains, which are described in the section above. This ensures that given an event, the correct component is given control to handle that event properly. For control flow among concurrently operating components, the *Task Concurrency* component is mainly responsible for ensuring that global control flow between threads is maintained. This is done through multithreading and IPC functionalities, which ensure that responder chains across different threads are still able to propagate events between threads to the correct responder. The *Notifications* component also plays an important role in global control flow, as it disseminates events to the various components in the system. Control is then transferred to these components to allow them to perform operations in response.

Data Flow

From an information view, data flow in GNUstep mainly involves the *File Management* component, as it is responsible for retrieving, loading, and transferring data from files to the various other components in the system. This transfer of data may also be facilitated by the *Task Concurrency* component, which provides IPC mechanisms, such as pipes, for efficient data transfer across components in a running application.

Concurrency

The *Task Concurrency* component is primarily responsible for concurrency within the system. It manages multithreading in the system, allocating and deallocating threads to accommodate the needs of different components, and handles synchronization through the use of locks and other resources. It also schedules tasks to be performed in the required order, such as ensuring that GUI elements are built before pixels are redrawn on the screen. Concurrent operations, such as graphics and audio services that happen simultaneously, and IPC between multiple threads are also managed by this component.

External Interfaces

Operating System

GNUstep communicates with different OS environments through system calls and APIs. On Linux and BSD, it relies on POSIX-compliant commands for process management, memory allocation, and file operations. Windows applications interact with the Win32 API for registry access, GUI event handling, and system resource management. GNUstep includes Cocoa compatibility functionalities, which translate macOS method calls and UI properties into equivalents for GNUstep applications. These interactions enable GNUstep applications to run on multiple platforms, leveraging system functionalities while maintaining cross-platform flexibility with minimal modifications.

Window Manager

GNUstep does not include a built-in window manager but instead relies on an external one to handle window placement, resizing, and decorations. In an X11-based environment, any X11-compatible window manager can be used, such as Window Maker, Openbox, XFCE, Fluxbox, or i3. Among these, Window Maker is commonly associated with GNUstep because it was designed to resemble the NeXTSTEP windowing environment, providing a consistent user experience. The window manager interacts with the X server, managing window placement, focus, and user interactions while ensuring that GNUstep applications function within a structured desktop environment.

Graphics & Rendering

GNUstep utilizes external rendering backends to manage graphical output. The X11 display server processes drawing commands and event messages, enabling windowed applications on Unix-like platforms. The Cairo graphics library provides vector-based rendering, allowing high-quality graphics and scalable text output. The Windows GDI handles pixel-based rendering, ensuring compatibility with native Windows applications. These backends allow GNUstep to maintain consistent rendering across different systems, ensuring GUI responsiveness and adaptability in various display environments. GNUstep applications also rely on these interfaces to generate graphical content dynamically.

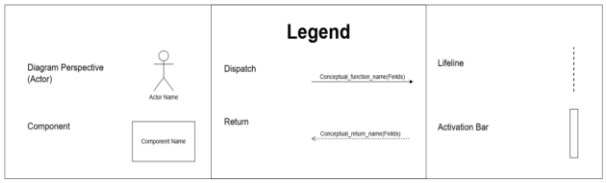
Packaging & Distribution

GNUstep applications are deployed through external package management systems to simplify installation and updates. APT and RPM handle dependency resolution and software installations for GNUstep libraries, ensuring compatibility with Linux

distributions. Flatpak and Snap offer sandboxed environments for application distribution, providing additional security and portability. These package management systems standardize software deployment, ensuring GNUstep applications remain accessible and easily maintainable across diverse computing environments.

Use Cases

This section will cover two use cases for users and stakeholders that interact with GNUstep. The provided legend (right) describes both sequence diagrams.



Use Case 1: Gorm

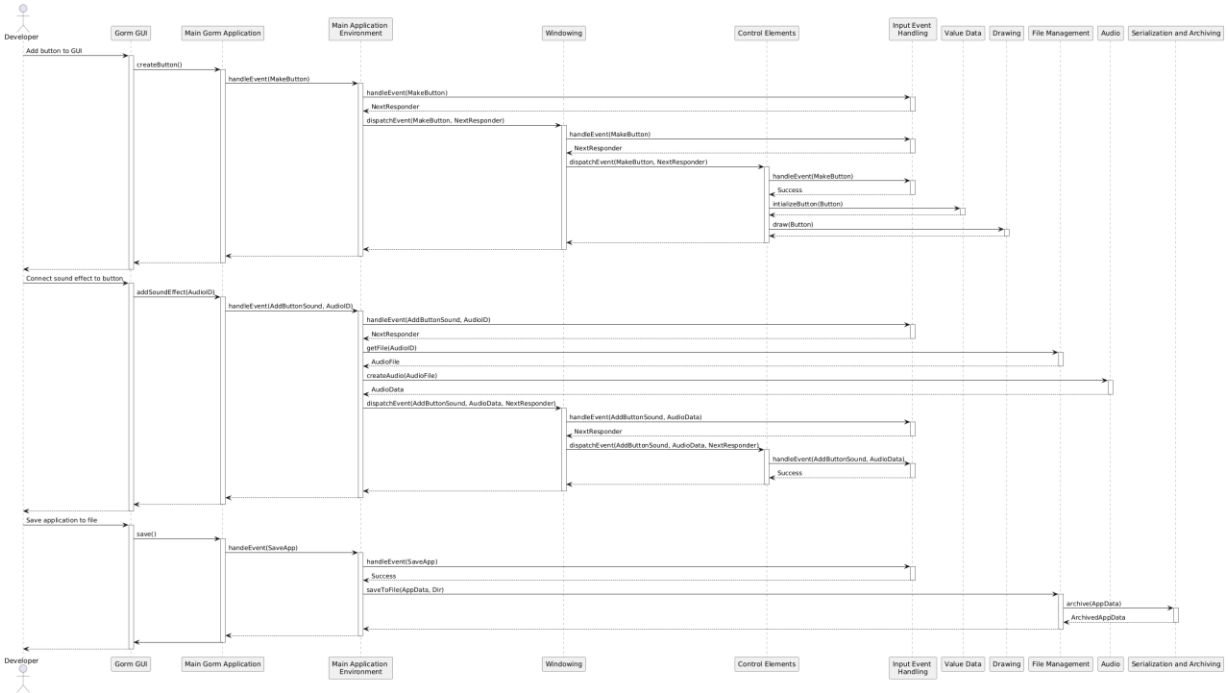


Figure 3. Sequence diagram for a developer adding a button with a sound effect in Gorm

This sequence diagram in Figure 3 illustrates control flow in GNUstep for the following use case: “a developer using Gorm adds a button to the GUI application they are building, and connects a sound effect to the button to be played when the button is pressed.”

In the conceptual architecture, Gorm builds applications using the GNUstep framework, but it itself is also built on GNUstep. Thus, the components in the sequence diagram in Figure 3 have two roles: running Gorm’s own interface, and building the application that is being constructed by a developer through Gorm. The *Control Elements* component is first used by Gorm’s own GUI for displaying the button that the developer can insert using a

drag-and-drop interface. Then, when the developer chooses to add that button, the actual instance of the button is created through the responder chain.

The first part of this sequence diagram shows how the button creation event is passed along the responder chain in Gorm, from the *Main Application Environment*, to the *Windows*, to the GUI *Control Elements*. Each component attempts to handle the event, and if it is unable to, it passes it to the next responder. In this case, the event is eventually handled by the *Control Elements* component, which creates the button for the GUI using the *Value Data*. The new button is then drawn on the screen with the *Drawing* component.

The second part of this sequence diagram shows how the event for connecting a sound effect to the button is passed along the responder chain. Before passing the event to the *Windows* the *Main Application* gets the required audio file from the *File Management* component, and stores it using the *Audio* services. The final connection is handled by the button that the sound effect is being connected to.

The third part of this sequence diagram shows how the event for saving the application file is passed along the responder chain. The *File Management* component handles the file creation, saving, and storing in the file system. It uses services provided in the *Serialization and Archiving* component to convert the GUI application's data into a file-storable form.

Use Case 2: GNUstep GUI Application

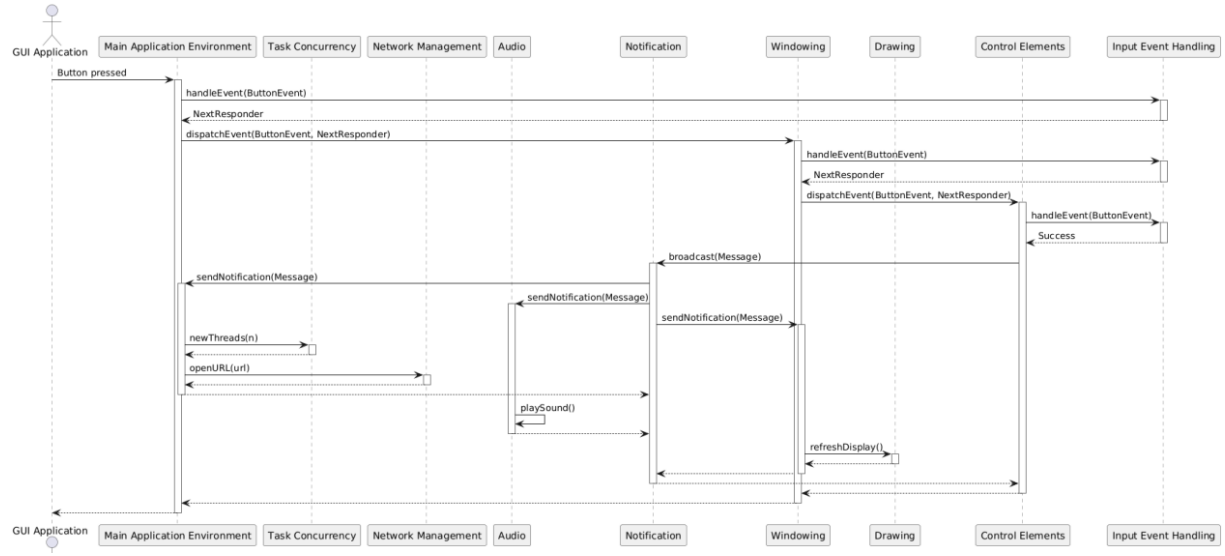


Figure 4. Sequence diagram for a user pressing a button in a GNUstep GUI application

The sequence diagram in Figure 4 illustrates control flow in GNUstep for the following use case: "a user of a GNUstep GUI application presses a button in the GUI, which plays a sound effect and then opens a webpage through a URL."

Similar to the first use case, the first half of this use case diagram shows the passing of the button press event along the responder chain from the *Main Application Environment*, to the *Windows*, to the GUI *Control Elements* to handle the actual event. Once the *Control Elements* component knows how to handle the event, it broadcasts a message through the *Notifications* component to let the other components respond. In response to the notification, the *Task Concurrency* component starts up some new threads to allow for the different components to operate concurrently, the *Main Application Environment* opens the URL through the *Networking* services, the *Windows* refresh the UI display using the *Drawing* services, and the *Audio* component plays the button's sound effect.

Conclusions & Lessons Learned

The conceptual architecture of GNUstep uses layered style and design patterns such as MVC and Chain of Responsibility. It is also reliant on external interfaces for OS interactions, rendering displays, and deploying applications. While ensuring cross-platform compatibility, GNUstep faces performance constraints and architectural dependencies that require optimization to improve efficiency and usability.

A major finding is that the *Main Application Environment* serves as a key bottleneck, as it handles all user input events before delegation. The *Notifications* component also needs to be developed strategically to reduce redundancy and enhance real-time event handling. Optimizing these components, along with the *Input Event Handling* component, would significantly enhance responsiveness. Non-functional requirements involving throughput, response time, and scalability should be kept in mind when designing these components.

Reliance on external window managers and rendering backends introduces potential inconsistencies across platforms. An additional standardized abstraction layer managing this connection could be considered to optimize evolvability and integrability.

To stay relevant, GNUstep must evolve alongside Apple's Cocoa framework. A dedicated community effort to track Cocoa updates and implement missing features will ensure long-term viability. Strengthening modern development practices, enhancing compatibility with contemporary IDEs, and improving documentation and tutorials will make GNUstep more accessible to developers and encourage broader adoption.

Key lessons learned from this study include the importance of early identification of architectural bottlenecks and the need for careful management of cross-platform complexity through abstraction layers. Additionally, optimizing notification systems for efficiency and responsiveness is crucial for maintaining smooth interactions. Keeping pace with external frameworks remains a challenge, highlighting the necessity of structured

updates and feature integration. Comprehensive documentation should also be a priority to increase accessibility to a wider user group.

By addressing these challenges and implementing the proposed optimizations, GNUstep can continue to provide a robust, scalable, and flexible framework for GUI application development. With strategic improvements in performance, compatibility, and developer support, it can remain a competitive option for cross-platform application development.

Data Dictionary

- **Application layer:** GNUstep layer providing GUI elements for building applications
- **Foundation layer:** The base layer of GNUstep architecture that provides non-GUI functionalities such as data storage, file management, and networking.
- **Gorm:** The graphical interactive interface builder for GNUstep
- **Graphical user interface:** The visual application interface that users interact with
- **Model-View-Controller:** A design pattern used to separate data management (Model), user interface (View), and logic (Controller)
- **Serialization and Archiving:** The process of converting objects into a file-storable format, and then restoring them to a usable form when needed
- **Use case:** A specific scenario demonstrating a stakeholder or developer interacting with a GNUstep application to achieve a goal

Naming Conventions

- **MVC:** Model-View-Controller
- **11:** X Window System
- **GUI:** Graphical User Interface
- **OS:** Operating System
- **URL:** Uniform Resource Locator
- **Gorm:** Graphical Object relationship modeler
- **POSIX:** Portable Operating System Interface
- **API:** Application Programming Interface
- **IPC:** Inter-process communication
- **Win32:** Refers to the Windows API
- **GNU:** “GNU’s not Unix”, GNUstep (“g-noo-step”) is just “GNUstep”

References

[1] Introduction to GNUstep
[2] GNUstep Frameworks
[3] GNUstep System Overview
[4] OpenStep Specification
[5] Apple Cocoa Documentation
[6] Gorm Application Guide