GNUSTEP CONCRETE 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 Gorm or the applications built on GNUstep. The primary architectural style of GNUstep is layered style with three layers: the Foundation layer, Application layer, and Gorm layer. The secondary architectural style is object-oriented style, with components interacting as objects within the layers. An alternative style that was considered was publish-subscribe style, and some notable design patterns include MVC, and chain of responsibility. The Foundation layer holds GNUstep classes responsible for low-level functionalities unrelated to GUIs. The Application layer holds classes responsible for GUI-related functionalities for both the front and back-end. Gorm holds classes responsible for running the Gorm application and interacting with the GNUstep framework for building GUI applications. The *Back-end* subsystem lies within the Application layer, and is responsible for managing back-end functionalities such as drawing, event handling, and maintaining the main application environment. Reflexion analysis of the GNUstep system revealed three divergences, and reflexion analysis of the *Back-end* revealed two internal divergences. Two major use cases for the concrete GNUstep system involve a developer creating a button in Gorm, and a user pressing such a button through a GNUstep application, which then plays a sound.

Introduction

Purpose

The purpose of the report is to derive the concrete architecture of GNUstep, including its concrete components, architectural style, and design patterns. A deep analysis on the *Back-end* component is included as well, with explanations of internal sub-components and their interactions. It also presents the results of reflexion analysis, and notable use cases are also discussed. This report is useful for developers who wish to understand the underlying architecture within GNUstep. It is also useful to GNUstep contributors for improvements that could be considered in future updates.

Organization

This report is organized into six main sections. The first section is the introduction, which provides an overview of the report contents. The second section discusses the concrete architecture of GNUstep. The third section analyzes the *Back-end* subsystem in-depth. The fourth section discusses the results of reflexion analysis. The fifth section presents two use cases. Finally, the sixth section summarizes key conclusions and lessons learned.

Conclusions

The concrete architecutre of GNUstep adopts a primary layered style and a secondary object-oriented style. It also incorporates MVC and chain of responsibility design patterns. Three notable divergences were extracted from reflexion analysis on the overall system, and code refactoring is recommended to improve system robustness, evolvability, and maintainability. Two internal divergences were extracted from reflexion analysis on the *Back-end* component, however, no action needs to be taken in response. Careful consideration of divergence beneficiality should be considered by developers.

Derivation Process

The derivation of GNUstep's concrete architecture started with modifications to the original conceptual architecture. This was done by consulting both external sources, such as the Apple Cocoa documentation and the GNUstep documentation. The modifications included aggregating of certain sub-components together into higher-level components, as well as redefining the functionalities of certain sub-components. Next, files were mapped from the GNUstep source code to components in the revised conceptual architecture. This was done using the Understand Tool, and files were mapped based on knowledge about their internal code, in-line comments, and external documentation.[1] Finally, adjustments were made to the concrete architecture to fix incorrect mappings.

Concrete Architecture

Structural Components

Foundation

Base Functionalities

This component's purpose is to provide low-level services that support the whole GNUstep framework, as well as handling fundamental tasks. It contains several sub-components. The *Notifications* sub-component manages broadcasting notifications to and from objects asynchronously. *NSNotificationCenter* is the main notification class,[1] acting as a central communication hub. Objects register to the center and are informed whether a notification (created using the *NSNotification* class) was posted by another object to its matching criteria. The center processes notifications synchronously. Asynchronous processing of notifications is done by *NSNotificationQueue*.[2] *NSDistributedNotificationCenter* enables objects from different processes to communicate, though it is slower than normal notifications, as it involves inter-process and inter-host communication.[2] The *Runtime Utilities* sub-component provides all the functionalities needed for a running application, such as assertion handling (*NSAssertionHandler*), error handling, debugging (*NSExeption* and *NSError*), and task concurrency (*NSOperation*, *NSOperationQueue*).[2] These utilities serve to maintain application functionality and responsiveness to a user. The *Serialization and Archiving* sub-component provide functionalities to store and retrieve structured data. The top-level class, *NSCoder* encodes and decodes objects, and *NSArchiver* and *NSUnarchiver* implement *NSCoder* to serialize and deserialize objects.[1] Data values and structures are handled by the *Value Data* component. Raw data is stored by *NSData* and data formatting is handled by *NSFormatter*. Other forms of data storage, such as numerical values, textual data, and ordered and key-value collections, are managed by classes such as *NSValue*, *NSNumber* and *NSArray*, among others.[1]

OS Services

This component provides essential services for interaction with the OS, allowing GNUstep applications to access files, network resources and system-level task management. Similar to the *Base Functionalities* component, it consists of several sub-components. The *File Management* sub-component provides services for handling file storage, access, deletion, and creation through classes such as *NSFileManager* and *NSFileHandle*, as well as for handling metadata and filesystem operations.[1] It enables applications to interact with local and remote files with classes like *NSPathUtilities*, which manipulates file

directories, and *NSHTTPFileTypes* and *NSHost*, which support file transfers.[1] Applications can communicate over local and remote networks using the *Networking* sub-component. This component provides services for HTTP requests, URL handling and network protocol support, using classes such as *NSURLRequest* and *NSURLConnection*.[1] The *Task Management* component is responsible for managing the different processes and threads running in the application. Classes such as *NSPipe* and *NSMessagePort* aid in inter-process communication, while classes such as *NSThread*, *NSLock*, *NSTimer*, and *NSAutoReleasePool* provide services for managing and distributing system resources.[1]

Application

View Elements

This component provides all the essential GUI services and elements in GNUstep applications. It defines the appearance and behavior of all interface elements with classes such as *NSButton*, *NSSlider*, *NSSwitch* and *NSSegmentedControl,* which create interactive user elements and event-driven controls.[1] These classes all depend on the *NSView* class, which defines the drawing and event handling functionalities. As such, any application that needs to print or display events will have to use *NSView* objects.[1]

Display Management

This component is responsible with handling windowing, panels and other display-related services using classes such as *NSWindow*, *NSPanel* and *NSWindowController*. Classes like *NSMenu* or *NSMenuItem* provide services for menu creation, and *NSCursor*, *NSColorPanel* and *NSFontPanel* handle user input tools and interactive displays.[1] Other important classes are *NSViewController* and *NSTabViewController*, which control switching between views.[1] Ultimately, this component presents and manages all displays.

Back-end

The *Back-end* component handles core system functionalities that support GUI and application behavior. It includes various sub-components that manage sound input and output, graphical rendering and image processing, drawing operations, handling user input events, wrapping and linking files, managing control flow, and connecting to external services. The *Back-end* is further discussed in a subsequent section of this report.

Gorm

Gorm is GNUstep's graphical interface builder and contains two sub-components, the *Gorm UI* and the *Gorm Main Application*. The *Gorm UI* provides the interface for creating GNUstep apps, enabling users to build GUI applications without directly modifying source

code. This is provided through editor and inspector classes such as *GormButtonEditor* and *GormMenuEditor*.[6] The *Main Gorm Application* manages the underlying logic that powers Gorm's UI-building capabilities, connecting user actions to back-end processes and ensuring seamless interaction between UI components and app logic. For example, *GormNibWrapperBuilder* and *GormXIBPlugin* facilitate the conversion and management of internal files, and *GormPluginsManager* handles external resources.[6]

Architectural Styles

Layered Style

The main architectural style of GNUstep is the layered style, comprising of three main layers: Foundation, Application, and Gorm.[3] This linear design incorporates separation of concerns, abstraction, encapsulation, and modularity. The architecture separates its concerns by dividing each major responsibility amongst different layers. The Foundation deals with the OS and base functionalities, and it contains the *libs-base* and *libs-corebase* libraries.[3] The user does not interact directly with this layer, and all operations in this layer are hidden from the user. The Application layer handles GUI elements, displays, and the main application environment, and it contains the *libs-back* and *libs-gui* libraries.[3] This layer is primarily responsible for providing functionality to the application running the graphical interface, and it acts a way for Gorm to interact with the Foundation layer. Lastly, Gorm pertains to the Gorm GUI and the main Gorm application which a developer would interact with when creating an application, and it contains the *apps-gorm* library.[6]

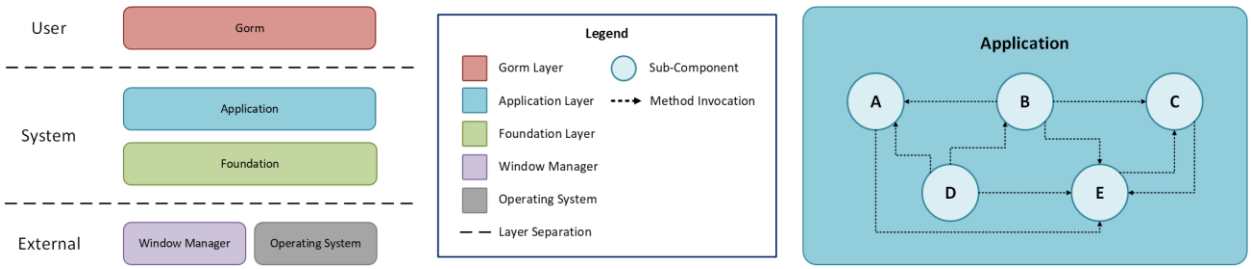


Figure 1. Layered and object-oriented architectural styles in GNUstep

Object-Oriented Style

A secondary architectural style used in the concrete architecture of GNUstep is object-oriented style. This is implemented within each layer, where each layer's services are encapsulated within different objects that invoke methods between one another. These classes are then grouped into modules, sub-components, and components within the overall layer. For example, the Application layer in GNUstep consists of various classes that handle various tasks, such as *NSView* for managing GUI views, *NSWindow* for managing windowing, and *NSApplication* for managing the main application.[1]

Alternative Styles

Publish-Subscribe

Publish-subscribe style in GNUstep is expressed through the *Notifications* functionalities provided in the *Base Functionalities* component. These functionalities are provided by a group of classes such as *NSNotification* and *NSNotificationCenter*, which are responsible for broadcasting notifications to objects in GNUstep applications.[1] However, this style is not suitable as an architectural style, as objects also communicate through direct method invocation, making it fit the role of a design pattern better than an architectural style.

Design Patterns

Model-View-Controller (MVC)

GNUstep provides classes that support the design of GUI applications that follow the MVC design pattern.[5] 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 through classes such as *NSView* and *NSController*.[1] It should be noted that MVC includes the composite, strategy, and observer design patterns as well.

Chain of Responsibility

GNUstep incorporates the chain of responsibility due to its nature of being a graphical interface where multiple actions can occur simultaneously.[5] The chain of responsibility allows for multiple requests to be handled by multiple handlers rather than just one. This ensures all events are handled in a timely manner and not bogged down in a convoy of requests. GNUstep implements this through *NSResponder,* which is the backbone of the responder chain.[1] Upon receiving an event, the event is passed down the chain until a suitable handler is found. If no handler works, the system applies a default response.

Component Interactions

Foundation

All components in GNUstep rely on *Base Functionalities* and *OS Services*, as they provide essential system-level capabilities for a running application. These include memory management, file handling, networking, threading, and inter-process communication (IPC). In addition, all components depend on notifications for reliable communication and response to events. This ensures smooth control flow, as background processes can update the user interface without direct intervention.

Application

Within the Application layer, both the *View Elements* and the *Display Management* components depend on the *Back-end*. This is because the *Back-end* provides all of the necessary functionalities for running the main application, meaning the *View Elements* and *Display Management* components need the *Back-end* to be able to sustain the GUI. The *Back-end* depends on *Display Management*, which in turn depends on the *View Elements* for sending events down the responder chain to the correct handlers.

Gorm

Gorm facilitates GUI application construction by integrating with the various GNUstep components. It utilizes the services provided by components in the Application layer to both build other GUI applications and sustain itself. The *View Elements* and *Display Management* components provide Gorm with GUI-related functionalities, while the *Back-end* provides Gorm with services related to internal application management.

Subsystem Analysis: Back-end

Components

Main Application Environment: Responsible for providing functionalities related to starting and terminating the application, managing the main run loop, managing memory and bindings of view elements, and connecting to external interfaces. Some classes that implement these functionalities include *NSApplication* and *NSNib*-related classes.[1]

Event Handling: Responsible for dealing with input events initiated by the user in a GNUstep application, such as mouse clicks and key presses. Its main functionalities include detecting user events and determining how different objects should react in response. Some classes that implement these include *NSEvent* and *NSResponder*.[1]

Drawing: Responsible for defining how to draw graphical elements of a GNUstep application on the user's screen. Its main functionalities include loading and displaying images, displaying colours, and applying transformations and special effects. Some classes that implement these functionalities include *NSDrawer*, *NSColor*, and *NSImage*.[1]

Text: Responsible for managing how textual elements of a GNUstep application appear on the user's screen. Its main functionalities include applying fonts, formatting and rendering text, and fitting text in containers. Some concrete architecture classes that implement these functionalities include *NSTextContainer*, *NSFont*, and *NSLayoutManager*.[1]

File Data: Responsible for connecting a GNUstep application to files located in the machine's filesystem. Main functionalities include loading file data into the application, retrieving data about files, and mapping edits done in the application to the file. Some classes that implement these functionalities include *NSDataAsset* and *NSDataLink*.[1]

Audio: Responsible for managing audio input and output in a GNUstep application. Its main functionalities include converting data between text and speech, connecting to external audio sinks, and managing audio files in the application. Some classes that implement these functionalities include *NSSound* and *NSSpeechSynthesizer*.[1]

Interactions

Since the *Main Application Environment* initializes the main application and connects it to all its required external services, all other components in the *Back-end* subsystem must depend on it to retrieve the necessary resources required to perform their functionalities.

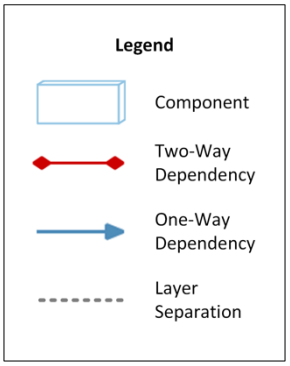
The *Main Application Environment* and the *Event Handling* component are dependent on each other, as they constantly communicate with each other to manage input events coming into the application. The *Main Application Environment* depends on the *Event Handling* component to determine where user events should be sent to in the system. At the same time, the *Event Handling* component depends on the *Main Application Environment* to provide the functionalities and external interfaces for input handling.

The *Drawing* and *Text* component are dependent on each other because they are both responsible for determining how text in the application appears on screen. The *Drawing* component depends on the *Text* component to give it details about the characteristics of different text fonts and styles, while the *Text* component depends on the *Drawing* component to represent the text in a format that can be displayed on the user's screen.

Reflexion Analysis

GNUStep

Before performing reflexion analysis, we made some modifications to our original conceptual architecture from A1. These changes mainly consisted of grouping related lower-level subsystems together into higher-level components that aggregated related functionalities. We chose to make these modifications because they better reflected the library partitioning in the source code, and because it allowed us to perform more meaningful reflexion analysis on higher-level components.



Divergences

The first divergence involves a new dependency resulting from a single reference from one of the *Back-end* classes to one of the *View Elements* classes. Specifically, the *NSStoryboardSegue* class in the *Back-end* invokes the *setViewController* method provided by view item classes in the *View Elements* component. Storyboards manage the relationships between controllers across a GNUstep application, and within a Storyboard, the *NSStoryboardSegue* class is responsible for managing transitions between window controllers.[1] However, in GNUstep, Storyboards also manage view controllers, meaning the *NSStoryboardSegue* class will sometimes invoke the *setViewController* method within a view item. This results in the appearance of a single new dependency.

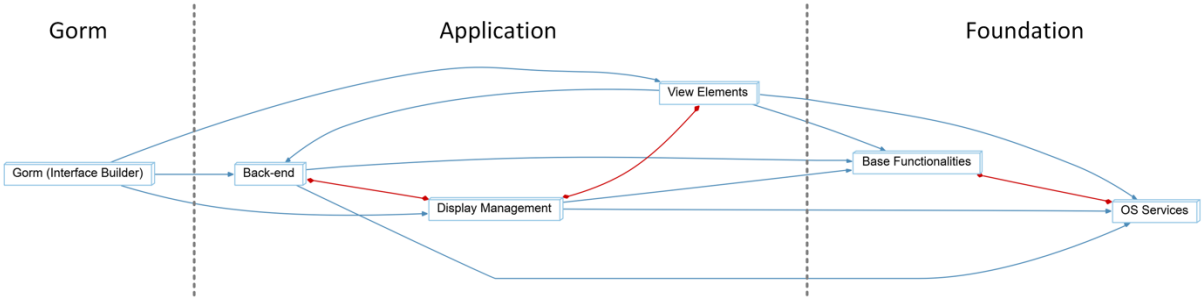


Figure 2. GNUstep conceptual architecture box-and-arrows diagram

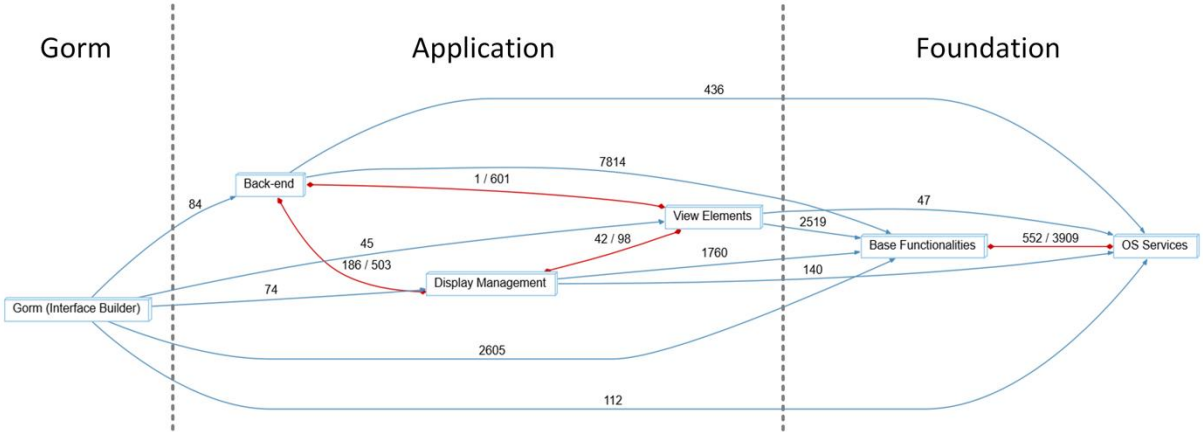


Figure 3. GNUstep concrete architecture box-and-arrows diagram

This two-way dependency is not ideal for GNUstep, as it unnecessarily couples the *Back-end* and *View Elements* components together. If the *View Elements* component were to experience an increased request load or a bug that impaired its functionality, this could lead to unwanted effects that could impair the *Back-end* as well. Thus, it would be recommended to modify the Storyboard implementation in GNUstep to eliminate this single dependency. This could be done by dividing the Storyboard functionalities across

the *Back-end* and *Display Management* components. Transitions between window controllers could be managed within the *Back-end* component, while transitions between view controllers could be managed within the *Display Management* component. This new implementation would be beneficial, as the two components would no longer be so tightly coupled, and would simplify future bug fixes, evolution, and integration for developers.

With respect to the other two divergences, these new dependencies come from *Gorm* classes that directly reference classes in the Foundation layer. Upon inspection of the source code, the main reason for this divergence is due to the implementation of additional features that are added on top of the GNUstep application layer classes. *Gorm* uses many classes from the GNUstep Application layer "as-is" without modification, but it also implements some additional functionalities on top of the pre-built classes. In the implementation of these additional functionalities, *Gorm* needs to access the Foundation layer classes to access lower-level services. For example, in the *NSGormAppDelegate* class, additional functionalities are implemented to manage XLIFF documents, which requires services related to file management and networking from the *OS Services*, and services related to resource management from the *Base Functionalities*.[1]

These new dependencies are not ideal for GNUstep, as they deviate from the intended layered architectural style. By having the *Gorm* component take a "short-cut" to directly reference the Foundation layer, the intended abstraction created by the Application layer is nullified. This could create issues in the maintenance and evolution of the GNUstep system, such as in the case where new updates are added to the Foundation layer, meaning both the Application and Gorm layers would need to be modified accordingly. This would increase costs from both a money and time perspective. Therefore, it is recommended that these divergences be removed, and that any functionalities in *Gorm* referencing the Foundation layer be managed by classes in the Application layer instead.

Back-end

In the process of performing reflexion analysis on the *Back-end* subsystem, we made some modifications to our original conceptual architecture from A1. These modifications mainly consisted of redefining the scope of sub-components and remapping dependencies. The reasons for these modifications are identical to those explained for the overall system.

Divergences

The first divergence involves *Event Handling* and *Drawing*, and is the result of a single reference. Specifically, according to the Understand tool, the *NSControl* class in *Event Handling* accesses the *type* variable in the *NSColor* class belonging to *Drawing*. However, upon closer inspection of the source code, this appears to be a false positive dependency.

Within the *NSControl* class, a conditional expression is used to check if the value of *type* is equal to the value *NSLeftMouseUp*, which is not a valid value for *type* defined in the *NSColor* class. However, the *NSEvent* class within the *Event Handling* subcomponent also has a variable named *type*, and this variable does have *NSLeftMouseUp* as a valid value. Thus, this dependency appears to be incorrectly identified by the Understand tool.

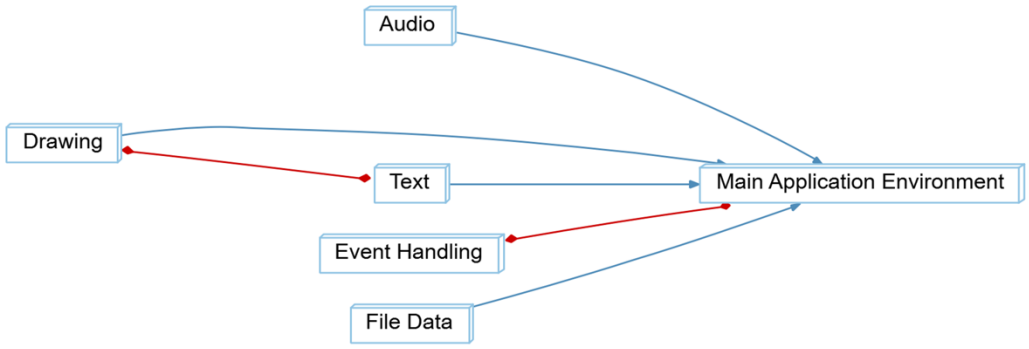


Figure 4. Back-end conceptual architecture box-and-arrows diagram

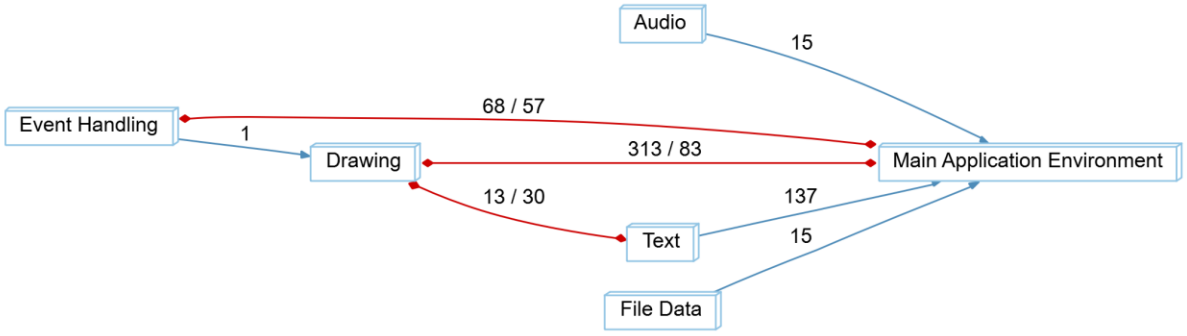


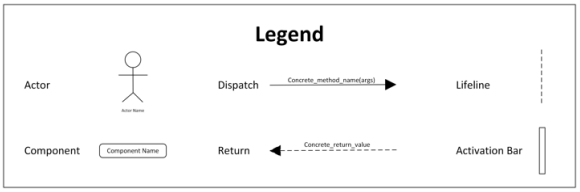
Figure 5. Back-end concrete architecture box-and-arrows diagram

With respect to the second divergence, the box-and-arrows diagram in figure 5 shows that this new two-dependency comes from *Main Application Environment* classes that directly reference classes in the *Drawing* subcomponent. Upon inspection of the source code, the rationale for this dependency relates to the role of the *Main Application Environment* in GNUstep. As specified in earlier sections, the role of the *Main Application Environment* is to manage the running of the GNUstep application. This includes functionalities related to printing and the application icon, provided by the *NSPrintOperation* and *NSWorkspace* classes respectively.[1] It also connects to the display server and initializes the display environment through specialized classes for different servers. These functionalities all depend on graphics contexts and image representations provided by classes in *Drawing*.

This divergence appears to be reasonable, and should not be considered as a problem in the implementation of GNUstep, as both subcomponents play an important role in updating and maintaining the current GUI display shown to the user. In this case, direct two-way communication between these subcomponents ensures that the display server is able to update the GUI quickly and reliably, enhancing real-time responsiveness to the user. Thus, although the implementation could be modified to decouple these two subcomponents, the results would be suboptimal, as this two-way dependency is crucial for minimizing display server delay and maintaining interaction with the user in real-time.

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: Adding a Button Through Gorm

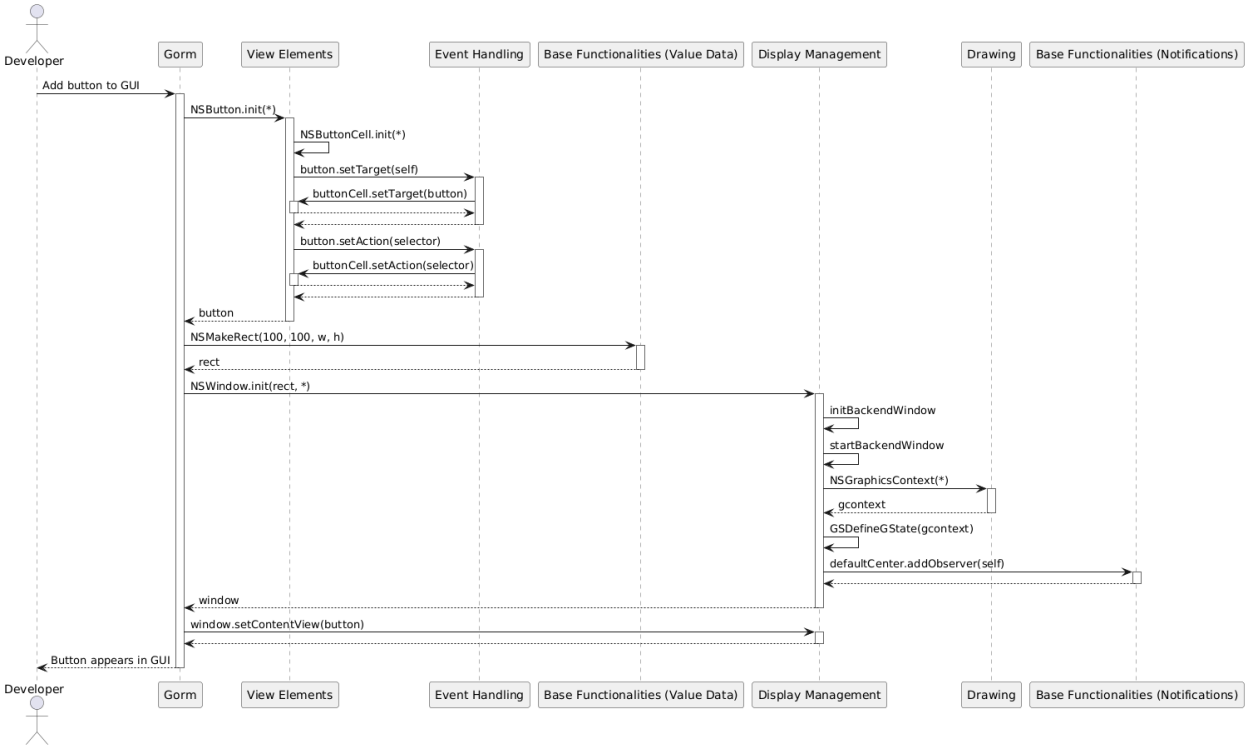


Figure 6. Sequence diagram for use case 1

This sequence diagram in Figure 6 illustrates control flow in GNUstep for the following use case: “a developer using Gorm adds a button to a GUI application being built.”

The first part of this sequence diagram shows how the button is created in GNUstep. First, the developer interacts with Gorm, which is prompted to create a new button. It does this using the *NSButton* class constructor in the *View Elements* component. When the new button is created, an instance of the *NSButtonCell* class is created with it. This cell handles the actual operations performed by the button. The *setTarget* and *setAction* methods are also called to add functionality to the button.

The second part of this sequence diagram shows how the button is added to a new window. First an instance of *NSRect* is created by referencing the *Value Data* sub-component in the *Base Functionalities* component. This is to define the area within the screen that the window owns. Next, the *NSWindow* instance is created through the *Display Management* component, and it initializes and starts the window. This requires for it to acquire a graphics context from the *Drawing* component. Finally, the window registers itself as an observer for future notifications from the *Main Application Environment*.

Use Case 2: Pressing a Button Through a GNUstep Application

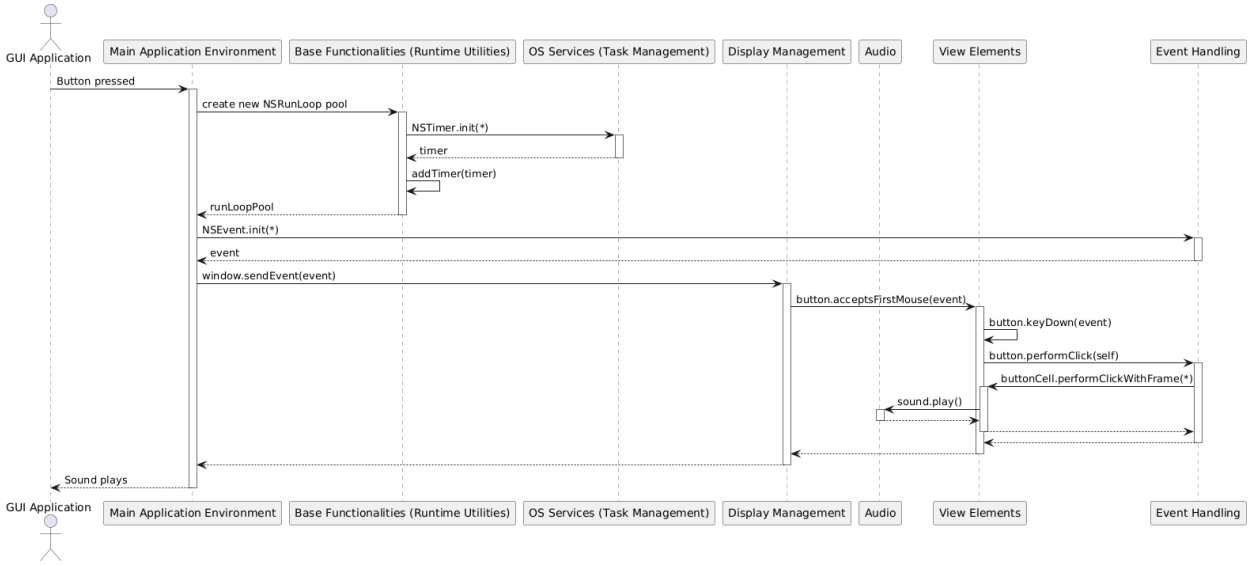


Figure 7. Sequence diagram for use case 2

The sequence diagram in Figure 7 illustrates control flow in GNUstep for the following use case: “a button in a GNUstep GUI application is pressed, which plays a sound effect.”

The first part of this sequence diagram shows how the *Main Application Environment* sets up run loops to receive user input. It does this by referencing the *Runtime Utilities* sub-component in the *Base Functionalities* component, which itself needs instances of *NSTimer* to control the run loops.

The second part of this sequence diagram shows how GNUstep responds to the event. When the *Main Application Environment* receives the button pressed event, it creates an

instance of *NSEvent* to pass the information down the responder chain. This is done using the *sendEvent* method to reach the window, and then the *acceptsFirstMouse* method to reach the view. Within the *View Elements* component, the button calls a method in the *Event Handling* component, which in turn calls a method on the button's cell to get a response. The button's cell calls a method to play a sound in response to the event.

Conclusions & Lessons Learned

The concrete architecture of GNUstep adopts primarily a layered architectural style, with a secondary object-oriented style within each layer. While the internal organization of the classes in GNUstep is relatively well-defined, reflexion analysis revealed three divergences that should be addressed to optimize the overall architectural design.

In particular, one major divergence involved the *Back-end* and *View Elements* components, where a two-way dependency was introduced on the basis of a single class reference. This introduced unnecessarily tight coupling between these two components, which increased the risk factor of potential bugs or request load issues during system operation. The elimination of this dependency through code refactoring is advised to decouple these two components, which would enhance system robustness and reliability.

Another major divergence involved the *Gorm* component and the Foundation layer components, where the source code did not properly adhere to the guidelines for layered architectural style. This reduced system evolvability and maintainability, and increased the costs associated with future modifications and bug fixes. The elimination of this dependency through the reallocation of additional Gorm functionalities to the Foundation layer is advised to improve system evolvability and reduce future costs of maintenance.

A deep analysis into the *Back-end* subsystem explored a variety of internal sub-components, and reflexion analysis on this component revealed two divergences, although they do not need to be addressed as urgently as those formerly mentioned.

One divergence was revealed to be a false positive within the Understand Tool, which failed to recognize the polymorphic nature of Objective-C. The other divergence involved the *Drawing* and *Main Application Environment* components, where an unexpected two-way dependency appeared. This was determined to be beneficial to the GNUstep system, as it ensures responsiveness to user events and facilitates real-time user interaction.

A key lesson learned from this study involves the importance of assessing the validity and rationale for divergences observed during reflexion analysis. The emergence of a false positive in the reflexion analysis process reminds us that software tools such as Understand need to be constantly scrutinized for correctness and accuracy, and that results should never be taken "as-is" without further investigation. Additionally, not all

divergences should be blindly determined to be harmful to the system, as some may simply be the result of oversights in the derivation of the conceptual architecture. Thus, thoughtful assessment of divergence beneficiality should always be conducted.

Data Dictionary

- **Concrete Architecture**: The actual implementation of the software system, including components, sub-components, classes, and their interactions
- **Component**: A modular and distinct part of the GNUstep system, serving a specific role, such as View Elements, Display management or the OS services.
- **Sub-Component**: A modular and distinct entity within a component, which performs tasks relevant to the component to which it belongs
- **Responder Chain**: A mechanism in GNUstep that passes user input events through a hierarchy of objects to the determine an appropriate handler.
- **Divergence:** A new dependency that appears in the concrete architecture that was not present in the original conceptual architecture

Naming Conventions

- **MVC:** Model-View-Controller
- **GUI:** Graphical User Interface
- **OS:** Operating System
- **Gorm:** Graphical Object relationship modeler
- **API:** Application Programming Interface
- **IPC:** Inter-process communication
- **NS**: Applies to anything with the “NS” prefix and stands for “Next Step”.

References

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