CISC322 A3

Architectural Enhancement of GNUstep.

3, April, 2025

Quantum Loop

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

Table of Contents

1 Abstract

This report presents a proposed architectural enhancement to GNUstep through the introduction of a Unified Event Logging System. Current logging across GNUstep's components: libs-gui, libs-back, GORM, and libs-base has limitations. It lacks consistency, centralized control and structured output which hinders maintainability, testing and performance optimization. Two architectural alternatives are examined: a centralized logger and modular logger interfaces. Through SAAM (Software Architecture Analysis Method), the modular approach is identified as the most effective solution due to its alignment with GNUstep's object-oriented and layered architecture. This enhancement delivers immediate value by offering developers a unified and extensible logging structure. This will help them improve system transparency, simplifies troubleshooting, and supports future scalability. The logging system not only improves performance diagnosis and user support but also enhances test automation and long-term maintainability of GNUstep applications.

2 Introduction

The proposed enhancement is the integration of a Unified Event Logging System into GNUstep. It is designed to provide structured, consistent and maintainable logging across all major subsystems. The current approach relies on ad hoc logging using NSLog and inconsistent mechanisms across components like libs-base, libs-gui, libs-back, and GORM, which leads to fragmented log data and reduced system observability.

Two architectural alternatives were considered:

1. **Centralized Logger Component:** A single EventLogger service used by all components to capture and manage logs.
2. **Modular Logger Interfaces:** Each subsystem implements its own logging module that adheres to a shared LoggerInterface, promoting modularity and flexibility.

Value and Benefits:

- **Improved Debugging & Troubleshooting:** Developers gain clear, consistent logs across all layers, making it easier to identify and resolve issues quickly.
- **Enhanced Testability:** Logging modules can be mocked or isolated during tests, enabling precise validation of component behaviors.
- **Future-Proof Design:** New GNUstep components can easily adopt logging by implementing the interface, avoiding code duplication and integration headaches.

- **Improved User support:** The Diagnostics Panel provides end users with human-readable logs, allowing for more effective technical help and self-troubleshooting.
- **Performance Optimization:** Fine-tune logging granularity for each subsystem to identify and resolve performance issues.

3 Current State of GNUstep

Without a central logging component in GNUstep, it is still possible to develop GUI applications and text-based tools, but there is no easy or consistent way of keeping track of errors or other events that may happen. While libs-base provides basic logging functionality through classes like NSLog [1], it lacks advanced features like structured logging or configurable outputs. Libs-gui, libs-back and GORM may also implement logging in their own ways when needed, leading to inconsistent formats and fragmented messaging across different layers. A unified logging system helps developers quickly trace issues and optimize performance, makes testing/validation easier, and keeps GNUstep maintainable as complexity grows.

4 Architectural Alternatives and SAAM Analysis

To support the introduction of a unified event logging system in GNUstep, we evaluated two architectural alternatives that differ in complexity, modularity, and coupling. This section presents these alternatives, the architectural styles they follow, and a SAAM (Software Architecture Analysis Method) evaluation to determine the best fit for the system's needs.

4.1 Architectural Alternatives

Alternative 1: Centralized Logging Service Component

This approach introduces a single EventLogger component placed in the Core and Utility Layer alongside libs-base, serving as a global logging service for the system. It acts as a global logging service, with all other subsystems—libs-gui, libs-back, gorm, and libs-corebase—making direct calls to it. The logger handles message formatting, timestamping, filtering, and output (e.g., to file or console). This design ensures a consistent logging format and centralized management of logging configurations.

Architectural Style:

- Primarily layered, the logger is placed in the Core and Utility Layer alongside libs-base as a shared service component, which already supports other subsystems.

- Follows a shared service pattern, where a single utility is reused across multiple clients.

Advantages:

- Simple to implement and integrate.
- Promotes consistency across log entries.
- Reduces code duplication for logging logic.

Drawbacks:

- Introduces tighter coupling between libs-base and higher layers.
- May become a bottleneck if logging is too frequent or poorly managed.
- Harder to test components in isolation due to shared state.

Alternative 2: Modular Logger Interfaces (Distributed Logging)

In this design, each major subsystem includes its own logging module that conforms to a common LoggerInterface. This allows subsystems like libs-gui or libs-back to implement their own logging logic or delegate to a shared backend. The interface-based approach allows future subsystems to integrate logging with minimal effort and promotes decoupling.

Architectural Style:

- Object-oriented, with a focus on interface-based modularity.
- Each subsystem implements its own logging class that conforms to a common LoggerInterface defined by the EventLogger component. Inspired by dependency injection and plugin-based architecture for extensibility.

Advantages:

- Improves separation of concerns and testability.
- Subsystems remain loosely coupled and can log independently.
- More adaptable for unit testing, mocking, and future extensions.

Drawbacks:

- Higher initial complexity and boilerplate.
- Slight duplication of effort across modules.
- May result in inconsistent formats if not centrally enforced.

4.2 SAAM Analysis

4.2.1 Stakeholders and their concerns

Table 1: Stakeholders and Their Concerns

Stakeholder	Concern	Non-Functional Requirements (NFRs)
Developers	Easier debugging and system observability	Maintainability, Modularity, Understandability
End Users	Stable and responsive applications	Performance, Reliability
Testers	Accurate, reproducible testing and traceability	Testability, Traceability, Simplicity

4.2.2 Evaluation of Alternatives

Table 2: Evaluation of Architectural Alternatives

Criteria / NFR	Centralized Logger (Alt 1)	Modular Logger Interfaces (Alt 2)
Maintainability	Easy to update in one place	High overhead, updates across interfaces
Modularity	Tight coupling to libs-base	High – loggers can evolve independently
Testability	Shared state complicates testing	Interfaces allow mocking and unit testing
Performance	May create bottlenecks	Localized logging can be optimized per subsystem
Traceability	Centralized logs = clear audit trail	Requires aggregation from multiple modules
Extensibility	Hard to scale for new subsystems	Easy to extend by implementing interface
Simplicity	Very straightforward to implement	Requires setup of interfaces and boilerplate

4.3 Recommended Alternative

Based on the SAAM evaluation, Alternative 2: Modular Logger Interfaces is the preferred solution. While more complex to implement initially, it aligns more closely with GNUstep's hybrid object-oriented and layered architecture. It supports long-term maintainability, modularity, and testability, which are crucial for a mature system like GNUstep. Developers can tailor logging granularity per subsystem, while shared interface definitions help maintain consistency.

5 Effects on Conceptual Architecture

The implementation of a standardized event logging feature for developers proposed in this document relies on the introduction of a new software system component. Such an addition implicitly demands a restructuring of the conceptual architecture. This restructuring can be examined from two perspectives, from the high level, and from the low level.

5.1 High-Level Effects on Conceptual Architecture

At the high level, implementing the centralized event logger involves the creation of the EventLogger software system component. This is a component separate from the previously presented libs-base, libs-corebase, libs-gui, libs-back, and GORM. The most suitable layer to place this new component in is the Core and Utility Layer along with libs-base and libs-corebase; this placement is due to the component being a utility for developers. The EventLogger component requires the addition of some new dependencies as well. Primarily, EventLogger depends on libs-base for classes to handle strings and memory allocation among others. This is similar in nature to the dependencies of all other components on libs-base. libs-back, libs-gui, and GORM will all depend on EventLogger at a high level. The reason for their dependency on EventLogger is explained in more detail in the Low-Level Effect on The Conceptual Architecture section below. As EventLogger will act as an interface which other components can implement to their specifications, this new component introduces an Object-Oriented aspect into the overall architectural style, as well as a Plugin-Based architectural style. The idea of modularity and compartmentalization are the main motivators for both additions.

5.2 Low-Level Effects on The Conceptual Architecture

At the low level, the dependencies created by the addition of EventLogger must be explained. The focus of this addition is to maintain modularity, the advantage of which is explained in detail in the Architectural Alternatives section. The components libs-back, libs-gui, and GORM will all depend on EventLogger for their own implementation of the logging interface. Each component has their own personal report types that will stay within itself, thus there is no reason not to allow the customization of the report message, storage method, and report types. Each component will be given a new class that will be the link between it, and EventLogger. This is a part of the Object-Oriented architecture style addition. Furthermore, little to no changes need to be made in existing classes within the depending components, and the event logger is not necessary for the well-functioning of a component. The catalyst for this design feature is the Plugin-Based architecture style of this upgrade. As a result, future components can easily add or omit an event logger as they see fit.

6 Interactions with Features

The implementation of the modular logging system (as per the recommended SAAM analysis) introduces structured interactions between the new logging interfaces and existing GNUstep components. This involves each subsystem implementing its own logger that follows a common LoggerInterface defined within the EventLogger component. Below is a breakdown of these interactions:

Dependencies:

- Libs-base: the EventLogger component would need to access the foundational classes (strings, dates, dictionaries, file I/O, error objects etc.) like NSString or NSError that are available in libs-base [1].

Dependents:

- Libs-Gui: logging GUI events like clicking buttons, keyboard input or tracking rendering errors
- GORM: logging design-time errors like invalid UI configurations or debugging resource loading failures like missing images
- Libs-back: logging backend-specific issues like graphics driver errors or hardware compatibility warnings

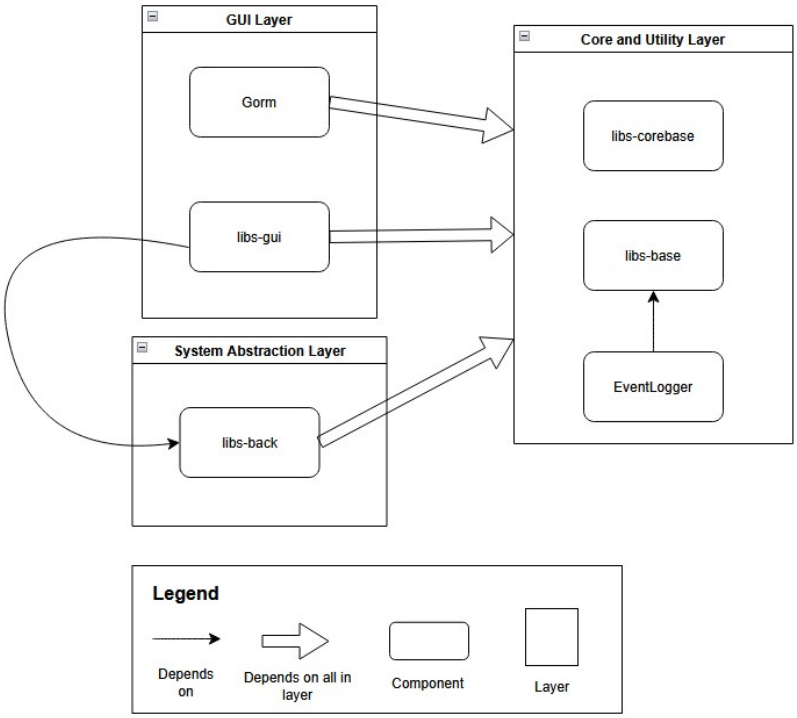

Figure 1: Diagram showing the interactions between EventLogger and other features

7 Impacted Directories and Files

The EventLogger introduces changes to the following subsystems, ensuring each implement the LoggerInterface:

libs-gui

- Source files like NSWindow.m [2] which implements GUI-specific logging (e.g. window resizing errors, mouse events); it would use a dedicated GUIEventLogger class that conforms to the LoggerInterface
- Existing methods would need to use the new local logger instead of direct calls to libs-base utilities like NSLog [1]

GORM

- Files like GormDocument.m [3] would need to integrate a GormDesignLogger to track UI design issues (e.g. missing resources) through the LoggerInterface

libs-back

- Files like XGServerWindow.m [4] would use a BackendEventLogger to record backend errors like GPU failures
- Custom log storage methods (GPU-specific buffers) implemented via the LoggerInterface, maintaining performance while ensuring compatibility with core standards

8 Effects on Maintainability, Evolvability, Testability, and Performance

Introducing the Unified Event Logging System into GNUstep significantly impacts four essential aspects of the system: maintainability, evolvability, testability, and performance. These effects are explained clearly, referencing insights obtained from our concrete architecture analysis.

8.1 Maintainability

Currently, our concrete architecture demonstrates that components such as libs-gui, libs-back, and Gorm communicate indirectly using mechanisms like NSNotification. This makes issue tracking and debugging challenging because event logs are fragmented and inconsistent across the system. Implementing a centralized or modular logging approach addresses this directly by providing standardized and structured logs across all subsystems. Developers would gain a single, consistent log format, significantly reducing the complexity and time spent identifying and resolving issues. Thus, the logging enhancement directly contributes to more effective long-term maintenance practices.

8.2 Evolvability

Our proposed logging system enhances the system's evolvability by supporting future growth and adaptation. The modular logging interface, as discussed in the alternatives analysis, offers clear integration pathways for future subsystems or new components. By defining standard logging interfaces, future components can seamlessly incorporate logging functionality without modifying the underlying architecture. This reduces future integration effort and helps maintain the modularity and scalability of GNUstep as its complexity increases.

8.3 Testability

The addition of the Unified Event Logging System will also substantially improve testability. With standardized logging methods, test scenarios can precisely verify whether specific events are being triggered and logged correctly. The modular design enables individual logger implementations to be easily replaced or mocked during unit and integration testing. This isolation capability significantly simplifies testing practices, enabling more efficient, accurate, and comprehensive validation of each subsystem's functionality.

8.4 Performance

Introducing logging inherently involves a performance trade-off due to the overhead of capturing and recording events. However, our recommended logging solution—the modular logging interfaces approach—mitigates performance concerns effectively. This distributed method reduces bottlenecks by spreading logging responsibilities across different subsystems, ensuring each component handles logging asynchronously and independently. Logging verbosity can also be dynamically configured based on context, such as development versus production environments, further reducing any noticeable performance impacts during typical usage.

9 Testing impacts of Interactions

To thoroughly assess the interaction of the new Unified Event Logging System with GNUstep's existing components (libs-gui, libs-back, and Gorm), we propose the following comprehensive testing plan:

9.1 Unit Testing

Each subsystem affected by the logging enhancement will undergo detailed unit testing:

- Verify that methods within key components (NSApplication.m, NSWindow.m, XGServerWindow.m, GormDocument.m) correctly call the new logging interfaces.
- Utilize mock logging objects to validate component behavior without dependency on logging implementations.

9.2 Integration Testing

Full integration tests will evaluate end-to-end interactions between components and the logging system:

- Simulate common user interactions such as button presses, UI navigation, and error conditions, verifying proper logging of these events.

- Confirm the logging output maintains correct formatting, timestamps, and severity levels across components.

9.3 Performance and Stress Testing

Performance tests will ensure that the logging system does not adversely impact system responsiveness or stability:

- Simulate high-frequency scenarios, such as rapid user input or intensive rendering activities, and measure the overhead introduced by logging.
- Monitor memory consumption, CPU usage, and responsiveness, ensuring that logging remains efficient even under heavy load.

9.4 UI and Usability Testing (Troubleshooting panel)

The newly proposed user-accessible Troubleshooting Panel will undergo usability tests to verify its effectiveness in real-world scenarios:

- Evaluate user interaction with log viewing, filtering, and exporting functionalities.
- Conduct feedback sessions with non-technical users to ensure clarity and ease of use, refining user-facing log descriptions to improve overall usability.

Through this rigorous and structured testing approach, we aim to validate the robustness, performance, and usability of our proposed enhancement, ensuring it integrates smoothly within GNUstep and significantly improves both developer experience and end-user support capabilities.

10 Sequence diagrams

Use Case 1: A user attempts to open a GNUstep-based application, but it unexpectedly fails to launch. To identify the issue, the user accesses the built-in "Diagnostics" panel from the application's Unified Event Logger. They filter logs to view recent error messages, discovering a clear, user-friendly log entry indicating a permissions issue with a required configuration file. Guided by this information, the user resolves the permissions problem or exports the detailed logs to technical support for additional help. After resolving the issue, the user successfully starts the application.

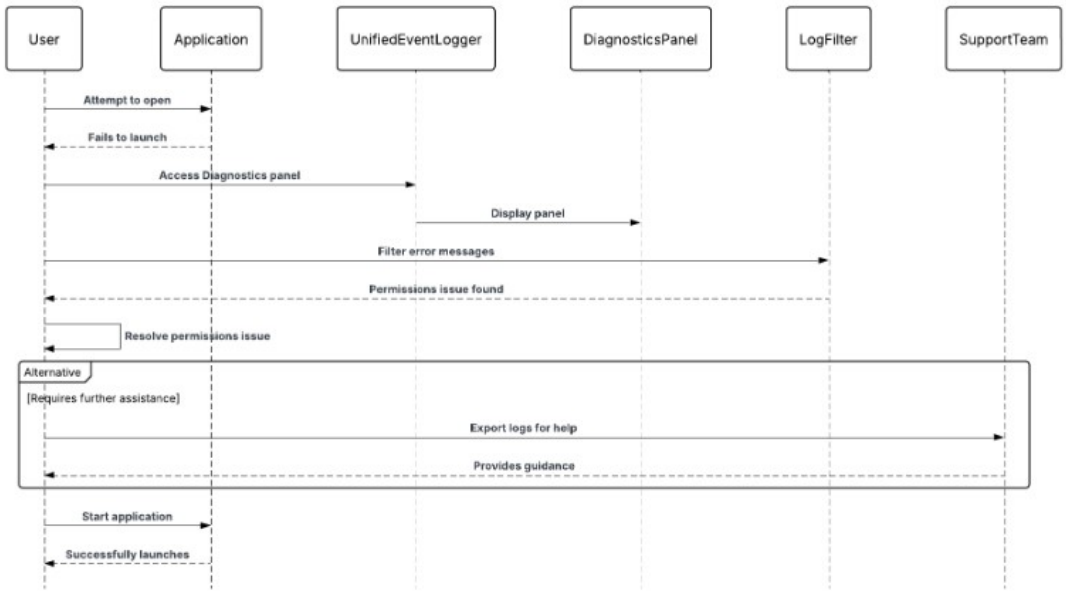

Figure 2: Sequence diagram for use case 1

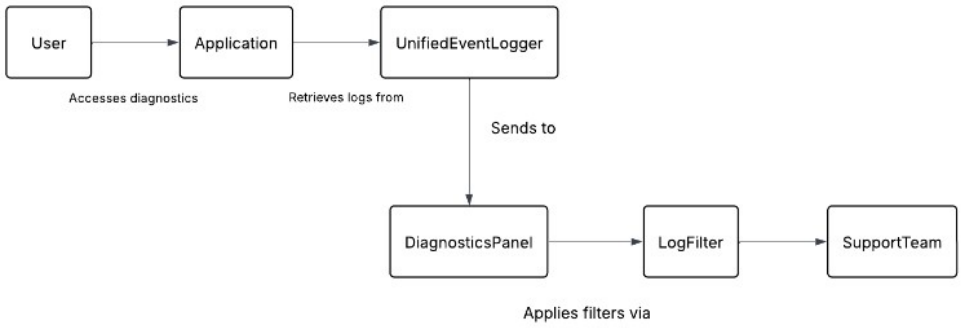

Figure 3: Box and arrow diagram for use case 1

Use Case 2: A developer notices performance issues, such as occasional GUI lag, in a GNUstep-based application. To diagnose this, the developer accesses the built-in Diagnostics GUI Panel of the Unified Event Logger. The developer retrieves and filters logs to view entries at informational and warning levels. The developer identifies recurring performance warnings related to GUI

rendering methods within the interaction between the libs-gui and libs-back modules through detailed timestamps and descriptions in the logs. Using this information, the developer pinpoints the source of performance bottlenecks, optimizes problematic methods, and verifies performance improvements by analyzing updated logs and observing enhanced application responsiveness.

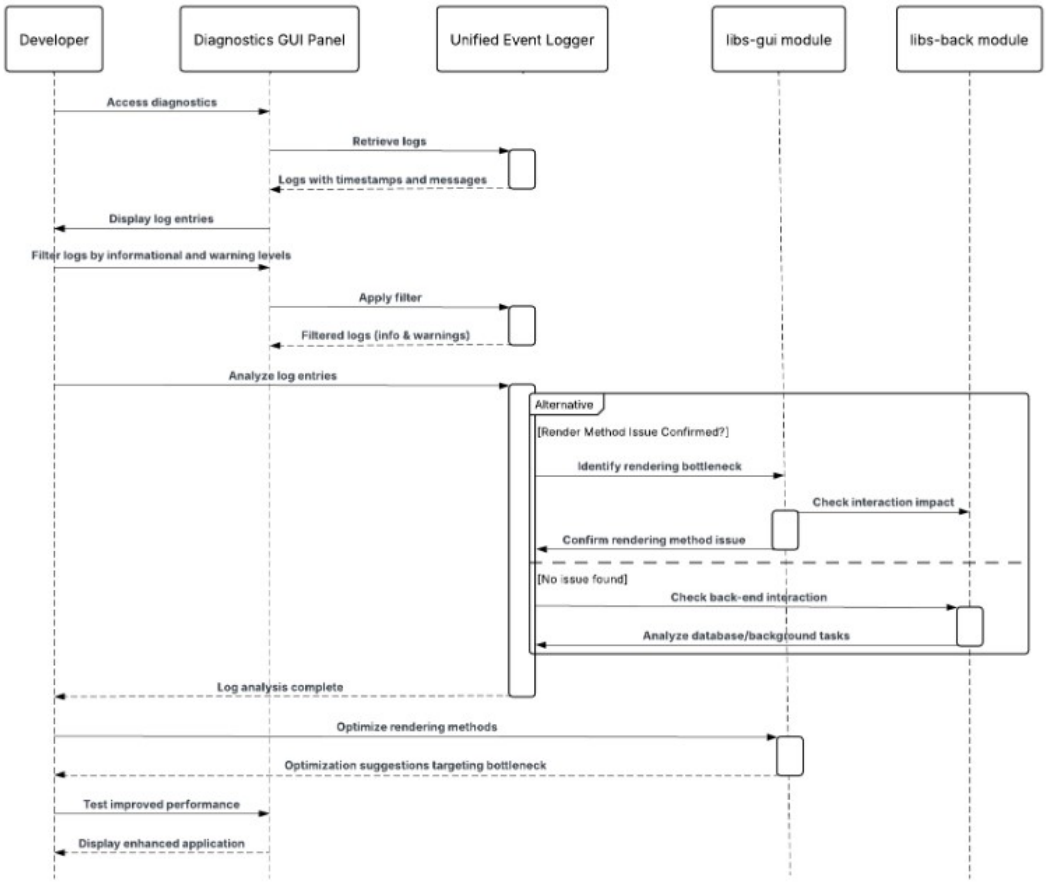

Figure 4: Sequence diagram for use case 2

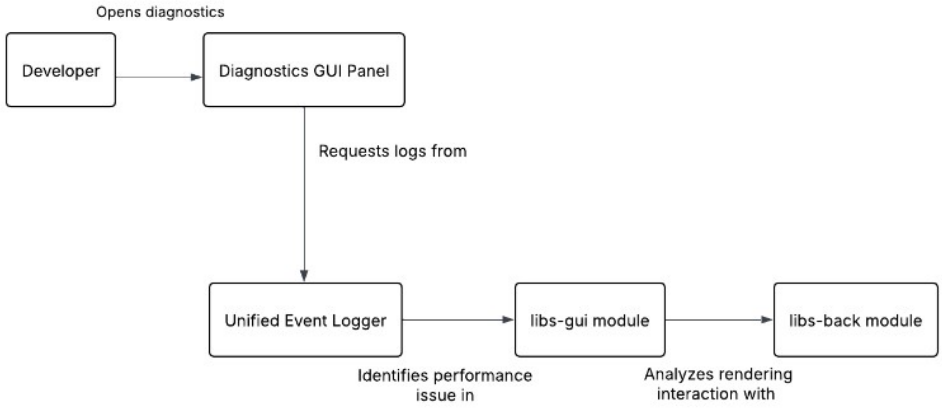

Figure 5: Box and arrow diagram for use case 2

11 Conclusion

The addition of a Unified Event Logging System to GNUstep provides important architectural and practical benefits. Through detailed analysis and comparison of alternatives, the modular logger interface approach has emerged as the optimal choice for its long-term benefits in maintainability, flexibility and system evolution. This enhancement addresses critical shortcomings in the existing logging mechanisms by introducing structured, testable, and subsystem-specific logging functionality. It enables developers to trace issues more efficiently and improves system observability. It also facilitates future enhancements with minimal architectural disruption. The proposed system reinforces GNUstep's scalability and robustness, ensuring it remains a viable platform for both development and deployment of complex Objective-C applications.

References

1. Gnustep. “Gnustep/Libs-Base: The Gnustep Base Library Is a Library of General-Purpose, Non-Graphical Objective C Objects.” GitHub, github.com/gnustep/libs-base. Accessed 3 Apr. 2025.
2. Gnustep. “Gnustep/Libs-GUI: The Gnustep Gui Library Is a Library of Graphical User Interface Classes Written Completely in the Objective-C Language; the Classes Are Based upon Apple’s Cocoa Framework (Which Came from the Openstep Specification).” GitHub, github.com/gnustep/libs-gui. Accessed 3 Apr. 2025.
3. Gnustep. “Gnustep/Apps-Gorm: Gorm Is a Clone of the Cocoa (Openstep/Nextstep) ‘interface Builder’ Application for GNUstep.” GitHub, github.com/gnustep/apps-gorm. Accessed 3 Apr. 2025.
4. Gnustep. “Gnustep/Libs-Back: The Gnustep Gui Library Is a Library of Graphical User Interface Classes Written Completely in the Objective-C Language; the Classes Are Based upon Apple’s Cocoa Framework (Which Came from the Openstep Specification).” GitHub, github.com/gnustep/libs-back. Accessed 3 Apr. 2025.
5. Kazman et al., “SAAM: A Method for Analyzing the Properties of Software Architectures”, ICSE 1994. `https://www.researchgate.net/publication/3560794_SAAM_a_method_for_analyzing_the_properties_of_softwarearchitectures`