CISC 322 – Software Architecture

Queen's University

School of Computing

GNUstep Architectural Enhancement Proposal

Assignment 3

April 4, 2025

Spencer Hum
21seh19@queensu.ca

Aidan Flint
21akf3@queensu.ca

Tony Luo
tony.luo@queensu.ca

Brant Xiong
21zx38@queensu.ca

Anson Jia
21aj85@queensu.ca

Zhangzhengyang Song
20zs57@queensu.ca

Table of Contents

Abstract **1**

1 Introduction **1**

2 Proposed Enhancement **1**

3 Methods of Implementation **2**
3.1 Client-Server Style 2
3.2 Interpreter Style 3
3.3 Enhancements Interaction with Other Features 3

4 Current State of Subsystems **4**

5 Architecture Styles Used **4**

6 Enhancement's Effect on Non-Functional Attributes **4**
6.1 Maintainability 4
6.2 Evolutionary 4
6.3 Testability 5
6.4 Performance 5

7 SAAM Analysis **6**
7.1 Client-Server Style 6
7.2 Interpreter Style 7

8 Impacted files and directories **8**

9 Use Cases **9**

10 Plans for Testing **11**

11 Effects on Conceptual Architecture **12**
11.1 Low-Level Conceptual Architecture 12
11.2 High-Level Conceptual Architecture 12

12 Potential Risks **12**

13 Lessons Learned **13**

14 Conclusion **14**

15 Resources **14**
15.1 Data Dictionary 14
15.2 Name Conventions 15

References **15**

Abstract

This report proposes the introduction of hot reloading interface components as an enhancement to GNUStep. Currently, interfaced files are statically loaded, which requires the entire application to be reloaded when testing new changes to the code. To address this inefficiency, we begin analyzing the current architecture to incorporate our enhancement that would allow interface files to be dynamically reloaded and instantiated into a running application, recompiling only the parts of the interface that have been changed rather than starting from scratch. Next, we conduct an SAAM analysis to evaluate the impact of the enhancement on non-functional requirements and stakeholders. Additionally, we identify the effects of the proposed enhancement on existing features and identify potential risks. Finally, we will present two use cases with accompanying sequence diagrams to demonstrate examples of workflows and the activation of components during hot reloading.

1 Introduction

GNUstep is a free and open source object-oriented framework for developing desktop applications. In this report, we propose enhancing the application layer of GNUStep to support hot reloading, specifically through dynamic reloading and loading of interface files as the application is running. The primary motivation for this enhancement is to expedite iteration development and testing times. These components reload modified sections rather than the entire page, making it more efficient. To implement this, we first identify the key stakeholders that would benefit from the proposed enhancement and determine the most important non-functional requirements (NFRs) that the stakeholders would want.

2 Proposed Enhancement

A commonplace scenario during development is validating the UI controls to meet specific requirements. This includes functional requirements, such as ensuring that specific actions are triggered by certain navigation sequences, or design requirements, such as testing for accessibility support. Large applications often have a diverse set of UI components spread out in different areas or windows, and currently, each time the code is changed, the application must be recompiled to see the changes. Most notably, the current UI state is lost every time a recompilation occurs. For example, if a developer is testing a particular form that is hidden behind several navigation items for screen reader support, then they must restore the UI state to the one they are testing (i.e., perform manual input to get to the form) each time the code is modified. The manual process required from the developer to restore the UI state to the desired one they are testing is both incredibly time-consuming and inefficient. To address these issues, we propose the introduction of a hot reload component in libs-gui that can reload interface files and update the method bindings of the loaded control in real time as an application is running.

3 Methods of Implementation

We have identified two primary routes for implementing the proposed feature that builds upon GNUstep's current concrete architecture. We characterize these two methods by the architecture styles they follow: client-server and interpreter. The main implementation we have chosen to focus on is the interpreter style implementation. The client-server style implementation serves as an alternative implementation we use as a comparison for SAAM analysis.

3.1 Client-Server Style

This implementation method is influenced by the modern web technology stack, namely its partition into client and server components. To implement this, we introduce two additional low-level components: a hot reload server and the client. The server contains a wrapper for interface compilation functions to be embedded into GNUstep runtime libobjc2. It received packets from the client consisting of interface definitions to be loaded via some protocol such as UNIX sockets or NSNotifications. Once the interface definitions are received, it is loaded by calling the respective functions in libs-gui. The server then leverages the low-level access provided by the libobjc2 to swap the current GUI controls in the running application, which are presented to the runtime system as raw objects. We note that this manipulation completely bypasses Objective-C language constructs. At the language level, the underlying object a variable refers to has effectively been modified without any visible changes to the code. The client is a simple object that should be initialized by the application wishing to utilize the hot reload functionality. It is responsible for sending interface definitions to the server and observing the status of the requests. To carry out these functions, the client needs to know where to find the interface file to be loaded and when a request should be triggered. Specifically, it takes a list of NSFileHandle points to the interface files, defines a reload interface NSEvent and registers handlers for said event. It is up to the developer to pass this information and send the reload interface event when the interface should be reloaded.

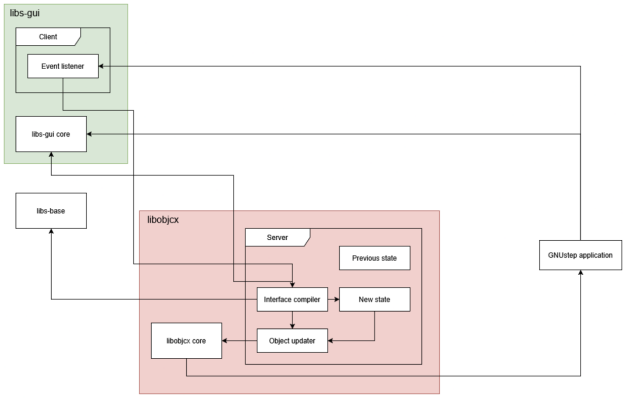

Figure 1: Client-Server Style

3.2 Interpreter Style

This method of implementation is based on the notion of an abstract state machine. Specifically, we view the hot reloader as an interpreter for interface files attached to the application. The inputs to the interpreter are interface definitions, and the outputs are the instantiated controls that the applications can use. To realize hot reloading, we introduce an interpreter component and an indirection layer for controls. The interpreter component is a regular object responsible for loading the interface definitions and deserializing the archived objects that are to be instantiated by the application with a list of NSFileHandle points to watch. The interpreter operates on a separate NSRunLoop with respect to the application and listens for reload triggers to be emitted by the application. Upon receiving such an event, the interpreter goes through the list of interface files, reads their content and deserializes them into an object graph to be passed to the indirection layer. On startup, a GUI application will have concrete methods attached to control objects. Hence, naively loading new objects will not persist these method registrations and lead to an inconsistent state. To address this issue, an object proxy or indirection layer is introduced. This layer takes the form of NSDocument subclasses that wrap control objects. The NSDocument subclasses internally point to a real control object and forward all functions to it. In addition, it tracks the state of the object it is proxying and provides a method to restore the state against a new object. Upon a reload of interface definitions by the interpreter, all the wrapped control objects will have their internal control object swapped with the old state reattached to the newly synthesized object. An application wishing to utilize the hot reload functionality will need to exclusively use these wrapped control objects instead of control objects provided by libs-gui.

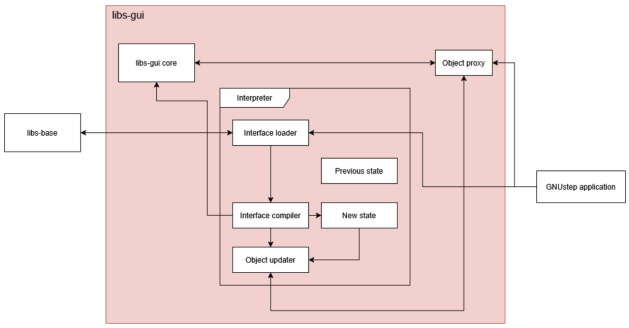

Figure 2: Interpreter Style

3.3 Enhancements Interaction with Other Features

The proposed enhancement is largely confined to the libs-gui component and does not interact with other parts of GNUstep in major ways. Following the interpreter style implementation, there would be two new auxiliary components introduced to libs-gui: interpreter and object proxy. The interpreter components depend on libs-base for file reading capabilities and existing parts of libs-gui which we have called libs-gui core, for loading interfaces. Following the client-server style implementation, there would be a new client component in libs-gui and a new server component in libobjc2. The client component acts as a one-way bridge between the GNUstep application and the server. The server component utilizes functions

in existing parts of libs-gui, which we have called libs-gui core, for loading interfaces and functions in libs-base for reading capabilities. This would correspond to a new dependency from libobjc2 to libs-gui.

4 Current State of Subsystems

The current GNUstep GUI layer consists of libs-gui and libs-back. libs-gui defines various graphical elements and auxiliary classes used for handling events. It also includes functionality for loading compiled interface files. libs-gui does not support live updates of changes to control objects. As such, each modification to interface elements at the code level requires recompilation, which makes prototyping UI a cumbersome experience. The current architecture of the GNUstep GUI layer revolves around a model where interface files are loaded statically along with the source code. Gorm is the primary tool for graphically designing UI. In a typical workflow, the interfaces are first compiled to a binary format with the extension .nib, which is then loaded on startup at runtime but is not dynamically updated.

5 Architecture Styles Used

As part of our enhancement design, we identified two architecture styles applicable to our implementation: client-server and interpreter. After evaluating both options against stakeholder requirements and NFRs, we adopted the interpreter style architecture. This approach encapsulates the hot reload logic within the application layer and preserves compatibility with GNUstep existing modular and layered architecture. The interpreter design incorporates aspects of the interpreter, proxy, and event driven architecture patterns, allowing dynamic interface updates while maintaining system integrity and modularity.

6 Enhancement's Effect on Non-Functional Attributes

6.1 Maintainability

From the maintainability point of view, hot reloading can improve maintainability by reducing the number of full applications rebuilt during development. This can make it easier to isolate and debug sections of the interface without inducing unintended changes elsewhere. However, this benefit does come at a cost, by introducing hot reloading it introduces a new component to the system architecture that must be maintained over time. This includes ensuring compatibility with Objective-C runtime, UI libraries, and event handling. To prevent this, it is important that the hot reload be clearly modularized and well documented.

6.2 Evolutionary

Implementing hot reloading significantly improves the evolvability of the system, as it allows the interface to be updated dynamically without requiring full recompilation. This opens the possibility for newer modular and flexible UI design. Over time, this will result in smoother

system updates, making it easier to add features. However, one important consideration is that GNUstep was not originally built with dynamic loading in mind. As a result, the reload system must be designed in a way that respects the existing architecture while also requiring careful integration to avoid instability or additional maintenance.

6.3 Testability

Testability is improved because hot reloading preserves the UI state during recompilation. This is useful when a developer or tester is deep within the application and doesn't want to restart from the beginning every time a UI file is reloaded. The ability to reload the interface and continue testing from the same state saves time. However, it can also introduce risks. The test environment may fail if a reload event were to occur in the middle of a test, or the system were not able to preserve or transfer correctly when a bad file is loaded. To reduce these risks, the system will retain the last known stable state of the interface and automatically fall back to it if a reload fails to improve testability.

6.4 Performance

Performance can be positively improved during development as the hot reload reduces compilation time. Only the changed interface needs to be reloaded instead of the whole application. This speeds up the feedback giving process, which in turn increases developer productivity. It must be noted that there is a runtime overhead incurred by the object proxy, with each method call on control objects needing to go through a pointer indirection. In addition, the process of watching interface files and detecting reload events also introduce minor performance penalties.

7 SAAM Analysis

Stakeholders	Key NFRs
GNUstep sponsor	**Usability:** The hot reload component should be "easy to integrate into existing applications." **Security:** The hot reload component should be optional and loosely coupled to the runtime system. In particular, the status quo static compilation should be supported. Other applications running on the system should not be able to call into the hot reload component to execute arbitrary code. **Performance:** Hot reloading should not add more than a 10% speed penalty than static loading.
Developers	**Development Speed:** This component should significantly shorten development cycles and increase iteration speed. **Modifiability:** No "major" code changes should be necessary to switch between static loading and hot reloading. **Manageability:** The hot reloader should record a log of reload results as if static loading were invoked to aid troubleshooting.
Testers	**Testability:** The hot reload component should not break the UI or change it in unexpected ways. The tester should still be able to run the test as usual.
GNUstep maintainers	**Reusability:** The hot reload component should be reusable and available as a standalone library. **Recoverability:**When a hot reload fails, the application should recover to the last working version without crashing.

Table 1: Stakeholders and Key Non-Functional Requirements

7.1 Client-Server Style

Modifiability: The client-server style implementation provides excellent modifiability. The only necessary boilerplate is the creation of the client and issuing reload events to the server. This is due to the fact that the actual logic and changes performed by the server operate beneath the language level in the runtime system.

Performance: The server component swaps the actual memory location pointed to by variables in the runtime system, which, in theory, incurs no performance impact. The swap phase may also block execution in other parts of the application, but because of privileged access to facilities in the runtime system, the server component can split the swap into subphases that still respect the order of execution.

Reliability: Due to the fact the server is at a lower level and the swap operation operates with raw memory, there are significantly more opportunities for subtle errors to be added that lead to an inconsistent state. It would be extremely difficult to debug such errors since they do not correspond to actual application code.

Security: The server component residing within the runtime system introduces a potential security issue where arbitrary processes in the development environment can, in theory, establish communication with the server component by mimicking the client to load malicious interface files that may lead to ACE.

7.2 Interpreter Style

Modifiability: The interpreter style implementation provides strong modifiability. The hot reloading component is completely encapsulated within the interpreter and object proxy. Developers only need to substitute traditional GUI components with proxy-wrapped equivalents to enable hot reload. Switching between static and hot reload can be done with minimum refactoring, making the system easy to adapt and extend.

Performance: The object proxy incurs a non-neglectable performance hit as every method call now goes through a pointer indirection. In addition, the phase where internal control objects are swapped may lead to sudden freezes in execution, as all other processing must be paused during this phase.

Reliability: Due to the fact that the interpreter and object proxy component do not have privileged access to the runtime system, there is a strong guarantee on memory safety during the swap operations. Although there are still potential bugs that may lead to an inconsistent state, these issues are relatively easier to debug and troubleshoot.

Security: The interpreter component being at the application level means that access is restricted to code residing in the same process. As the process address space is guarded by the operating system kennel, this eliminates the possibility for other processes to hijack interface loading facilities.

Based on our analysis of both implementation strategies with respect to requirements imposed by the stakeholders, we believe that the interpreter style implementation would be the more appropriate method to implement the hot reload feature and dedicate the remainder of the report to exploring it in depth. Although the client-server style implementation scores higher on the performance and modifiability front, it introduces new security and reliability concerns explicitly forbidden in the non-functional requirements by multiple stakeholders. Security and reliability are key non-functional requirements that directly ties into the architecture and can not be mitigated at a lower level, hence, the deficiency of the client-server architecture stood out to us. On the contrary, the interpreter style implementation ensures security and correctness at the cost of some performance penalties and modifiability. While performance and modifiability are key non-functional requirements identified by some stakeholders, we believe that these are issues that can be reasonably mitigated during implementation.

8 Impacted files and directories

In this section, we describe the concrete changes to libs-gui required to implement the hot reload following the interpreter style method of implementation.

Following the established naming convention of libs-gui, a new top-level directory called Reloader should be created to house the interpreter and object proxy component. All subcomponents of the interpreter should reside in a flat layout under the subdirectory Reloader/Interpreter while the object proxy component lives in the subdirectory Reloader/Proxy.

Interpreter Component

The interface reader is implemented in the file InterfaceReader.m/h which is responsible for fetching interface (.nib) files from a storage location. It does so by calling the appropriate file manipulation functions in libs-base. A new data structure called NSInterfaceState is defined in the file NSInterfaceState.m. This data structure stores the state of the current interface, which control objects are active and the state of each control. The interface loader is implemented in the file InterfaceLoader.m/h. It is responsible for deserializing the content of the interface file. It takes the previous state stored as an NSInterfaceState structure and applies it against the loaded object graph before handing off control to the object updater. The object updater is implemented in the file ObjectUpdater.m/h. It traverses the object graph and looks up each new object against proxied objects tracked by the object proxy component, specifically the ObjectProxyManager object. It then swaps the internal state on a successful match. Otherwise, a new proxy object is instantiated and added to the list of objects monitored by the ObjectProxyManager.

Object Proxy Component

There is a corresponding proxy wrapper named P<Classname>.m for each control class that is exposed by libs-gui under the Source directory. The proxy wrappers are subclasses of NSDocument and contain a pointer to the unwrapped control object as well as a table that tracks states that may be associated with the control object being wrapped. Moreover, the wrapper exports all methods of the corresponding control object but internally updates the state table before forwarding the call to the real control object. This makes the interface of the wrapper compatible with that of the control object. In addition, there is a manager class defined in the file ObjectProxyManager.m/h that collects all the proxy wrappers and exports them under a shared namespace. Furthermore, it contains methods to automatically track the wrappers and emit notifications to the object updater.

9 Use Cases

In this section, we present two use cases and their corresponding sequence diagrams that illustrate our proposed enhancement and its effects on GNUStep.

Our first use case illustrates the activation of components in a successful reload of interface files. In this scenario, a developer has created an example application called HotReload to experiment with hot reloading. This application has a simple interface consisting of a button that is initially blue and some images. For simplicity, we omit function calls that are not directly initiated by the hot reload components. To begin, the developer launches the application with hot reloading toggled on. The application first instantiates an object proxy manager and then registers all methods against proxied objects under the object proxy manager namespace. Next, the application calls the interpreter to load the base interface. The control flow first goes through the interface reader which calls the appropriate function in libs-base to read the interface file from disk. The content of the file is deserialized by the interface loader into an object graph. Since this is the initial load of the interface, no prior state exists. Hence, the initial state is created. The object updater then takes the object graph and loops through it for matching objects tracked by the object proxy manager. For every matched proxy object, it updates the internal pointer of the proxy object and records the change in the object proxy manager. Once all objects have been processed, the interpreter calls rendering functions in libs-gui to display the controls to the screen. When the developer changes the color of the button to red in the interface file on disk. The application detects the filesystem change and invokes the interpreter to reload the interface. The same sequence of steps occurs as for loading the initial interface except for an additional step. When the object graph is built, the object updater applies the previous state stored in the stable table to the newly constructed objects before querying the proxy objects and updating their internal pointers to objects in the object graph. Finally, upon completing the update operation for all proxy objects, the control objects are re-rendered, and the button's color is changed to red in the application.

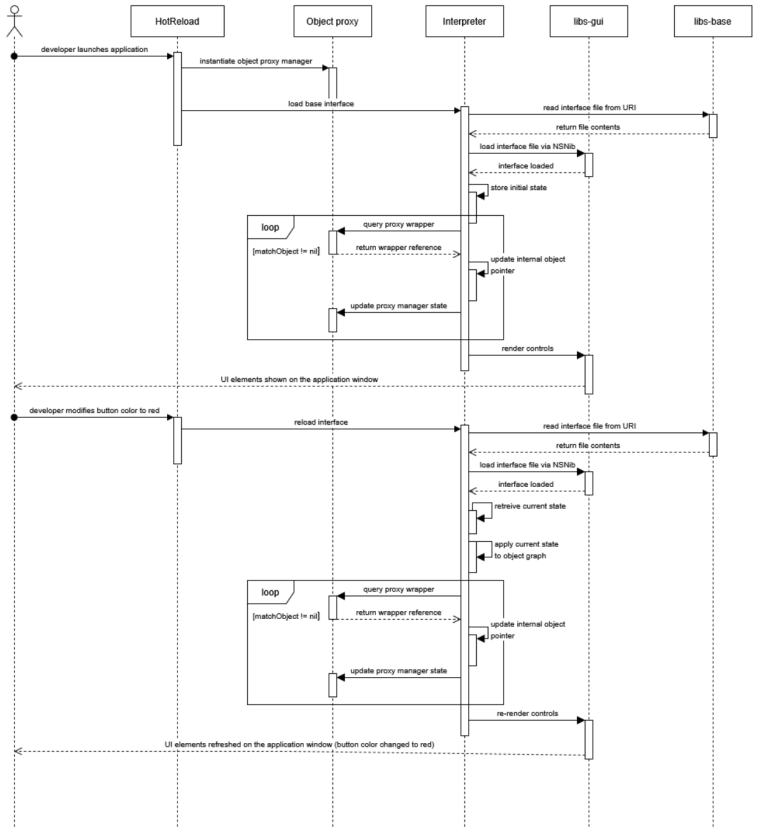

Figure 3: Successful Reload of Interface Files

Our second use case illustrates the activation of components in a failed reload of interface files and instated recovery mechanism. The situation is identical to our first use case, except that when attempting to reload the interface, some objects in the object graph turn out to be corrupted. Note that at this point, some of the proxy objects have already had their internal pointers replaced, while others are waiting to be processed. To ensure correctness, the interpreter must restore all objects to their corresponding state prior to attempting the reload. The activation following encountering the exception begins with the object updater retrieving the previous current state and the previous object graph cached in the interpreter object. It applies the current state against the previous object graph to obtain the last consistent state. Next, it walks through the object graph and gets every object tracked by the object proxy manager and restores its internal pointer to the objects in the old object graph. Once all objects have been processed, the application is once again in a consistent state, and execution can continue normally. The interpreter returns a status message that the reload has failed to the application who may wish to try the procedure again. The controls are not re-rendered, and the button's color remains blue.

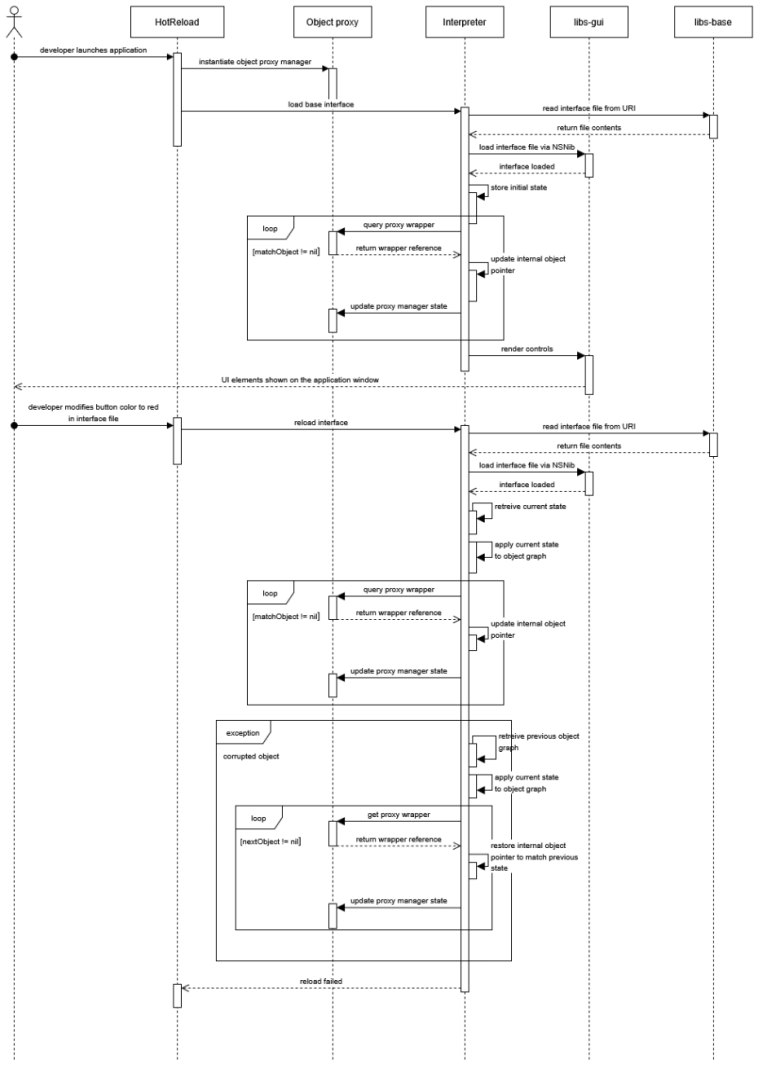

Figure 4: Failed Reload of Interface Files

10 Plans for Testing

To ensure the stability and performance of the new enhancement, we will implement a precise plan to cover unit testing, integration testing, performance benchmarking, and compatibility testing. Unit testing will focus on ensuring that the enhancement accurately detects modifications in interface files while ensuring only those modified files are reloaded. This will involve testing the file-watching mechanism to confirm changes are being registered in real-time as well and that newly loaded components can be injected without disrupting the existing state. Integration testing will evaluate how the enhancement interacts with the existing GNUStep components and test the UI updates in various application scenarios, simulating error cases so solutions can be made with this in mind. These error cases involve introducing syntax errors or invalid changes in files to ensure the appropriate fallback mechanisms are being applied. Testing these edge cases will minimize risks and disruptions in the application.

Performance benchmarking will compare the enhancement with the traditional approach, starting by examining the memory and CPU usage that will be introduced with the hot reloading components, as well as any potential trade-offs. By examining this, we ensure that performance is up to standard and analyze the latency of dynamic updates to ensure minimal delay without any excessive resources being used up in the process. Compatibility testing will ensure that the enhancement works correctly with many different development styles and functions as expected. This confirms that the system remains stable and existing functions continue as expected even with the enhancement disabled to maintain backwards compatibility. By implementing a structured testing approach, we can assess the stability and performance of the enhancement while mitigating any issues before deployment, keeping high standards for stability and performance to deliver an improvement to the development workflow.

11 Effects on Conceptual Architecture

Bringing in hot reloading in GNUStep will introduce new requirements for both high and low-level conceptual architecture, as it evolves to a more modular design. The change will improve development efficiency while it ensures stability and maintainability.

11.1 Low-Level Conceptual Architecture

At the low-level of conceptual architecture, modifications will become necessary in object loading and layer rendering to support real-time updates. This includes linking between interface components and the application to ensure no disturbances to ensure there are no state disturbances. Furthermore, there will need to be improvements in the memory to ensure components are handled safely, preventing leaks and ensuring a proper integration into GNUStep.

11.2 High-Level Conceptual Architecture

The proposed enhancement does not require changes to the high-level conceptual architecture of GNUStep. The interpreter and object proxy components are fully encapsulated within the existing libs-gui module and do not introduce new top-level dependencies or modify the interaction between major subsystems such as libs-back, libs-base or libobjc2. Therefore, the system retains its current layered and modular structure. The enhancement is designed to be self-contained and internal to the GUI layer, ensuring compatibility with the existing architecture and minimizing disruption.

12 Potential Risks

Extending the current GNUstep system to accommodate the proposed hot reload feature introduces several risks and drawbacks pertaining to the system's security, testability, maintainability and performance. In this section, we outline risks in each of the categories iden-

tified above. These risks must be thoroughly considered and balanced against benefits when deciding on adoption..

Testability Risk: By enabling hot reload at runtime, the system may enter unpredictable states between reloads. This variability can compromise the reliability of test results and reduce confidence in testing. Differences in application state before and after reload may lead to test failures that are not caused by actual defects, making debugging and validation more difficult and time-consuming.

Security Risk: The introduction of the hot reload function allows interfaces and components to be dynamically reloaded at runtime. It expands the application's runtime flexibility but costs the system's exposure to security threats. With hot reload, the system has less control over how the interface behaves at runtime. Without strict safeguards, dynamically loaded components may become an entry point for unauthorized modifications or unintended execution paths, especially in development environments where security boundaries are less enforced.

Increased System Complexity: Supporting both static and dynamic interfaces adds new layers of architectural complexity. These include additional responsibilities such as file change detection, state preservation and partial UI updates. The coexistence of two UI flows, static and hot reload, can result in code duplication, unexpected interactions and greater difficulty when introducing new developers or maintaining the system over time.

Performance Risk: The addition of hot reload introduces new background processes such as file monitoring, interface reload, and runtime component replacement. These operations, although helpful during development, may create performance overhead if not carefully managed. In particular, frequency reloads can cause temporary delays in UI rendering, increased memory usage or inconsistent frame rates. If parts of the hot reload system are mistakenly enabled in the developing process, it could negatively affect the responsiveness and efficiency of the application. The presence of both static and dynamic loading mechanisms may also make performance harder to predict or optimize over time.

13 Lessons Learned

One challenge we faced early on was our group's lack of knowledge regarding the full capabilities and functionalities that GNUStep had to offer. This issue was largely because GNUstep is an open source project over 30 years old, and the limited documentation regarding its functionalities required us to delve into source code written in Objective-C, a programming language none of our group members were familiar with, to figure out what each component does. To work around our team's gaps in knowledge, we devised a plan to put down any suggestions for enhancements and have a more experienced group member go through them and pick out valid suggestions where we would vote on the final enhancement for the report.

Another issue we had to deal with was balancing our workload toward the end of the semester, as several assignments and exams coincided with this project. This was particularly difficult for all of us since we had other priorities and deadlines to meet for different courses. To

manage this, we made sure to meet more frequently, which helped us stay on track. As a result, we were able to finish the report on time despite the tight schedule.

14 Conclusion

In this report, we proposed two ways to implement a hot reloading enhancement to GNUstep, which allows for automatic reloading of interface files whenever the code is modified. This enhancement would eliminate the need for the developer to recompile the program every time to test a change in the code, which would reset the state of the UI so the developer would need to manually restore the page to its desired state for testing. We presented hypothetical stakeholders and NFRs for the feature enhancement and performed an SAAM analysis to identify tradeoffs between the two potential implementations. For the remainder of the report, we concentrated on the interpreter style implementation. We described the concrete changes necessary to realize the enhancement, including impacted files and directories and developed two use cases with accompanying sequence diagrams to showcase examples of workflows and the activation of the added components during hot reloading. Lastly, we outline a plan for testing the implementation, the effects of the feature on the high-level and low-level conceptual architecture for the respective subsystems and the potential risks involved.

Through the completion of this project, we have been given the chance to explore and delve into the architecture of GNUstep, gaining valuable knowledge on the inner workings and subsystems of GNUstep, and gained hands-on experience with creating, analyzing and documenting software architecture.

This concludes our final report on GNUstep.

15 Resources

15.1 Data Dictionary

GNUstep: A set of cross-platform object-oriented frameworks written in Objective-C for developing applications.
Libs-base: The fundamental library of GNUstep, providing basic functions and routines for data manipulation, network and file operations, datetime handling and other primitive system functionalities.
Libs-corebase: An alternative library, following Apple's Core Foundation framework, implemented in C.
Libs-gui: The graphical user interface library, following Apple's Core APIs, serves as the frontend part of the GNUstep GUI library.
Libs-back: The graphical user interface backend component, serves to draw graphical elements as well as managing windows.
Gorm: A graphical interface builder for GNUstep.
Libobjc2: The runtime system for GNUstep, designed as a drop-in replacement for the GCC runtime, sets up the execution environment and provides raw primitives for stack

management, memory management and function closures, which operate behind the scenes at the language level.

15.2 Name Conventions

API - Application Programming Interface
NFR - Non Functional Requirements
SAAM - Software Architecture Analysis Method
UI - User Interface
ACE - Arbitrary Code Execution

References

[1] C. Armstrong. "Using the GNUstep AppKit," [Online]. Available: https://www.ict.griffith.edu.au/teaching/2501ICT/archive/resources/documentation/Developer/Gui/ProgrammingManual/AppKit.pdf Accessed: 02/12/2025.

[2] Apple Inc. "Nib Files," [Online]. Available: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/LoadingResources/CocoaNibs/CocoaNibs.html Accessed: 04/04/2025.

[3] R. Kazman, L. Bass, M. Webb, and G. Abowd, "Saam: A method for analyzing the properties of software architectures," in *Proceedings of the 16th International Conference on Software Engineering*, ser. ICSE '94, Sorrento, Italy: IEEE Computer Society Press, 1994, pp. 81–90, ISBN: 081865855X.