**6.0 Architectural Design**

**6.1 Introduction**

This document presents the architecture and detailed design for the software for the Project Cleanup: Swift Property Wrappers project. The project involves making necessary modifications to the Swift Compiler, such that any programmer writing a program in Swift will see clear and detailed error messages associated their code, should it include Swift property wrappers. These error messages are significantly more specific, less ambiguous, and more concise than the error messages pertaining to property wrappers that were getting displayed to the user in Xcode prior to the conception of this project.

**6.1.1 System Objectives**

The objective of this project is to replace existing error messages associated with property wrappers with clearer ones. Prior to the start of this project, many errors associated with property wrappers would leave out crucial information associated with their initialization, their type, or their wrapped value type. When property wrappers are composed, or nested, errors tend to get even more confusing, since the error messages for regular property wrappers are so poor. In addition, more features are intended to be added to property wrappers, such as expanding the scope of where they can be passed into programs, such as in function and closure parameters (see the proposal for this feature [here](https://github.com/apple/swift-evolution/blob/main/proposals/0293-extend-property-wrappers-to-function-and-closure-parameters.md)). It will be very difficult to move forward with these projects prior to cleaning up error messages. Therefore, it was crucial for us to put Project Cleanup: Swift Property Wrappers into action.

**6.1.2 Hardware, Software, and Human Interfaces**

***Hardware***

The user will need a computer with 24.83 GB and 89.83 GB of available storage in order to download Xcode’s most recent beta version, clone a fork of the [Swift Compiler project](https://github.com/apple/swift) to their desktop, install cmake, sccache, and ninja, as well as generate build artifacts from building and running the project.

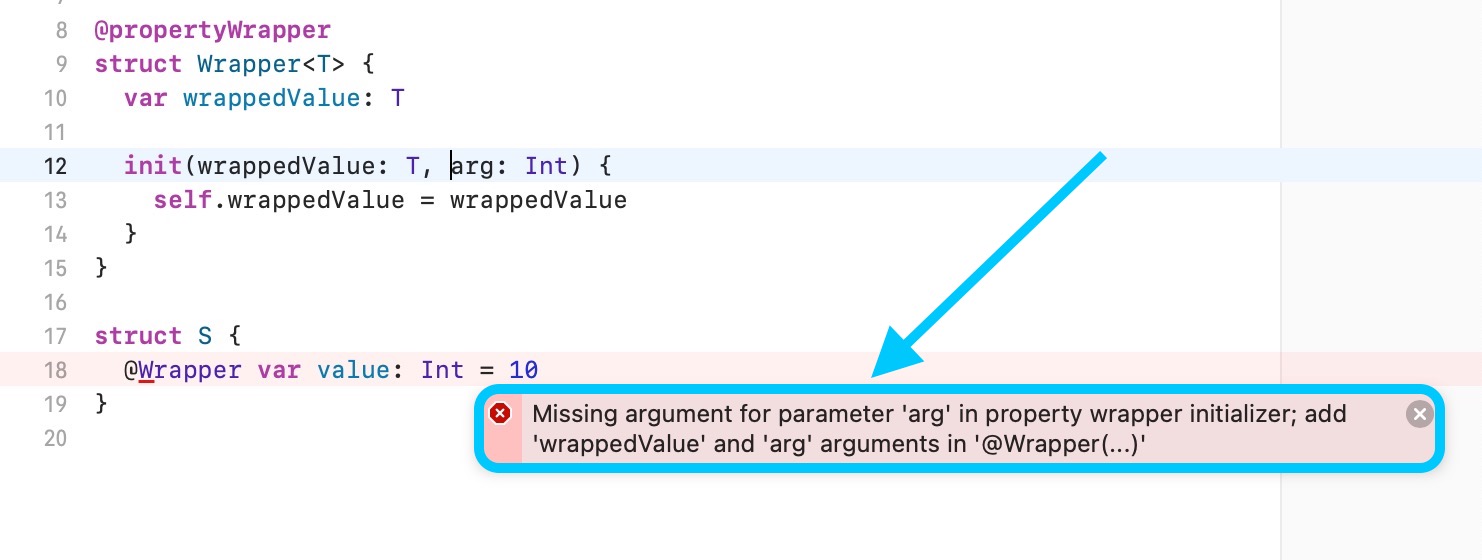
***Software***

The user will need to use a computer with Xcode’s most recent beta version installed. They can install this version [here](https://developer.apple.com/xcode/resources/). They will also need to follow the [user manual](https://github.com/Strieker/lmu-cmsi-401/blob/master/README.md) in order to access additional software to view the changes made to the Swift Compiler, as a part of Project Cleanup: Swift Property Wrappers, from within Swift’s Compiler project in Xcode. This additional software includes a clone of the Swift Compiler’s source code, should the user want to see the error messages associated with Project Cleanup: Swift Property Wrappers from within the Swift Compiler’s output after passing the compiler a test file.

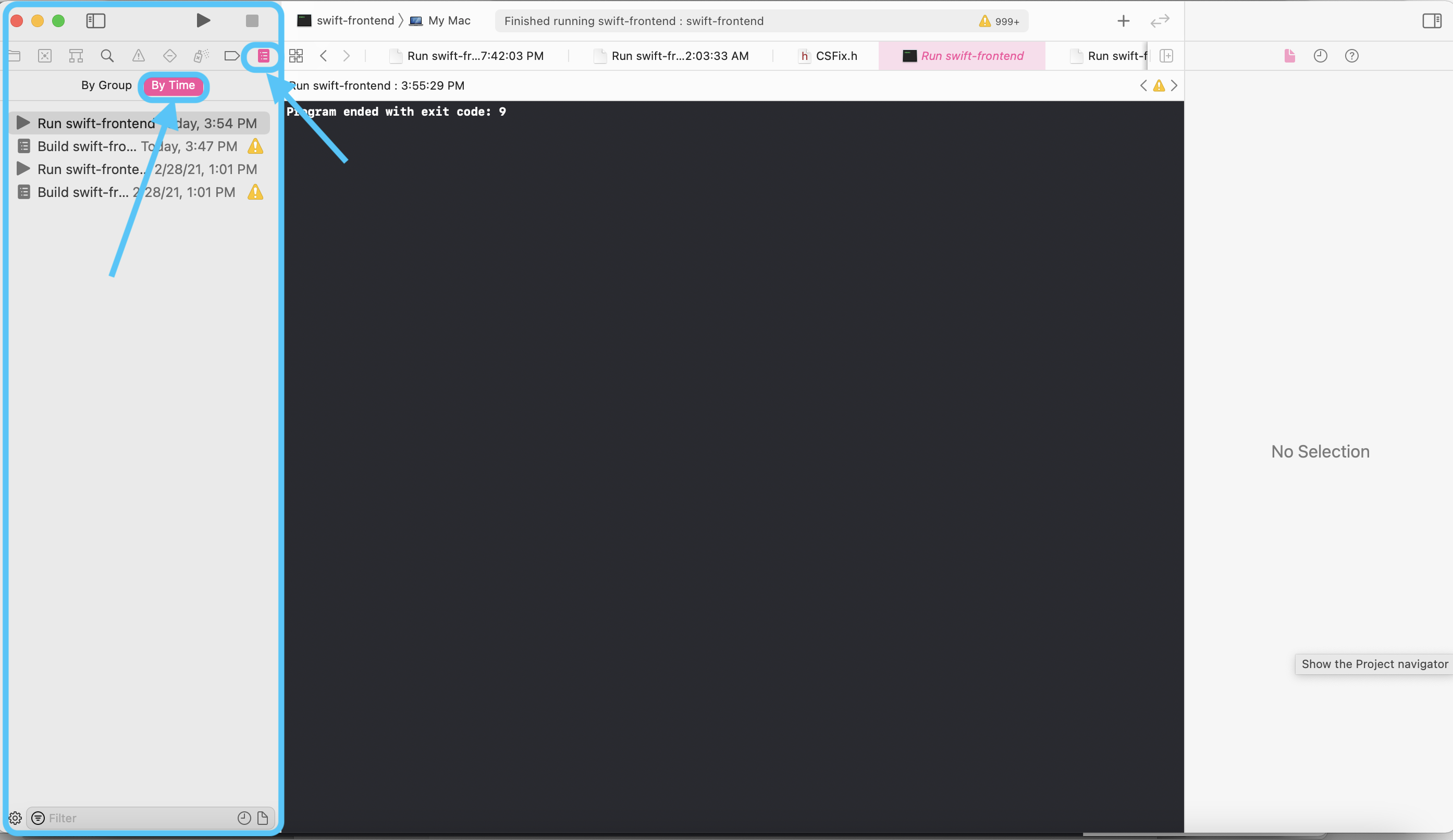
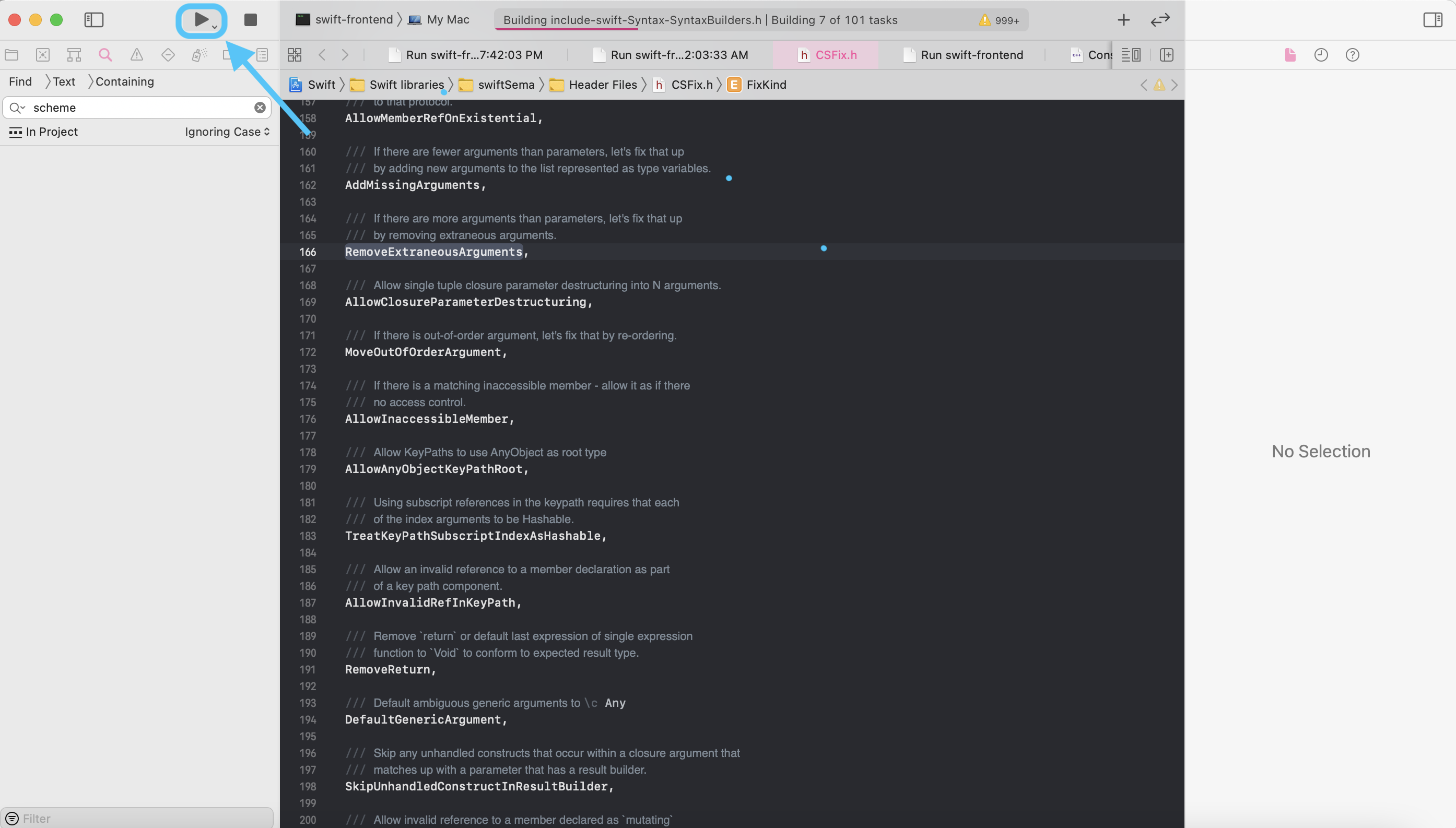
***Human Interfaces***

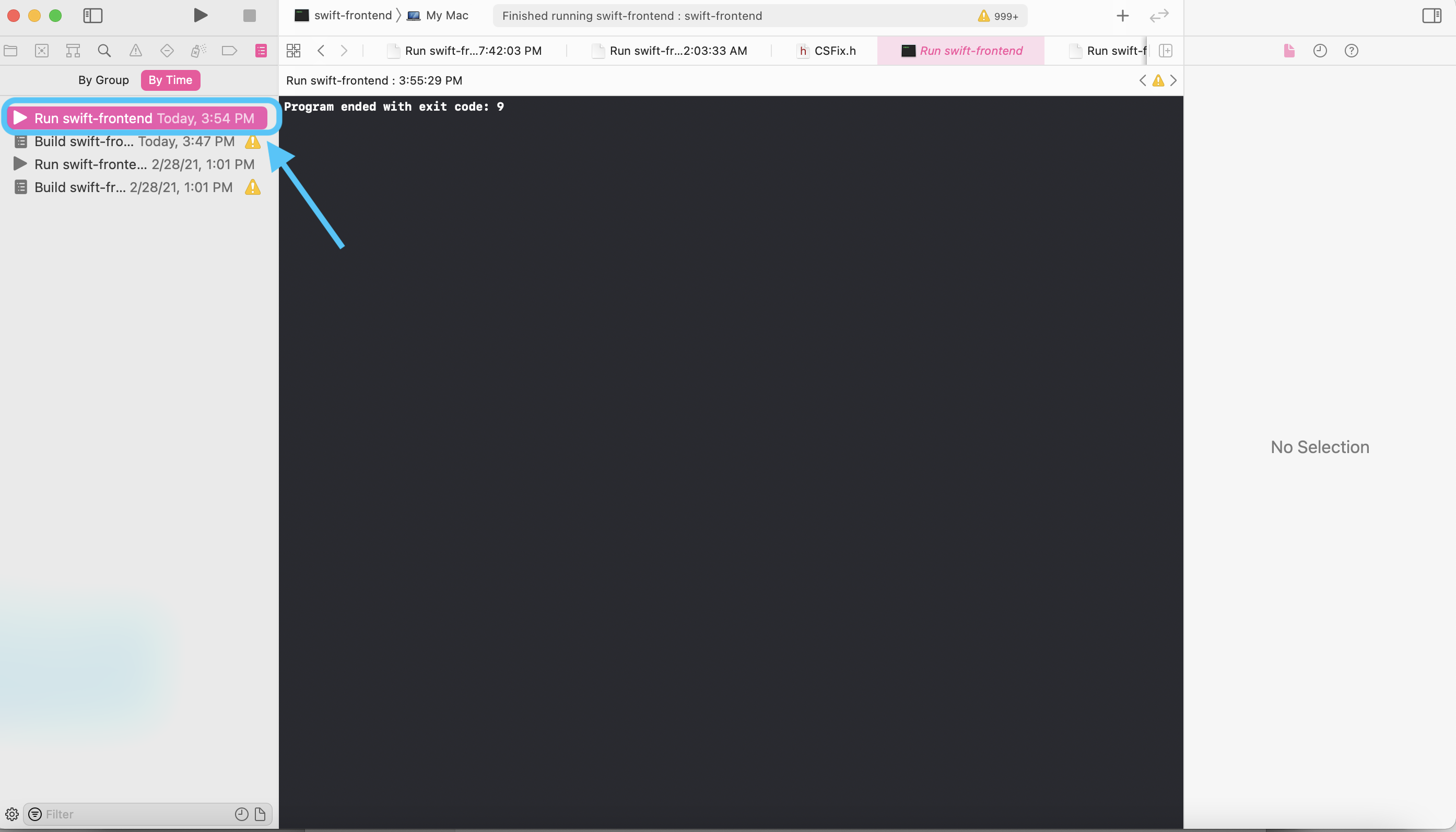
The user will use a mouse or trackpad, as well as a keyboard, in order to either

1. run their test program to view and apply the comments the fix-it results show from within Xcode (where the fix-it is labeled in blue, and it reflects any new error messages associated with property wrappers that were added as a part of Project Cleanup: Swift Property Wrappers)



1. run the Swift Compiler project from within Xcode to access the output of the Swift compiler, which could include error messages associated with property wrappers, which were added to the compiler as a part of Project Cleanup: Swift Property Wrappers



****

**6.2 Architectural Design**

**6.2.1 Major Software Components**

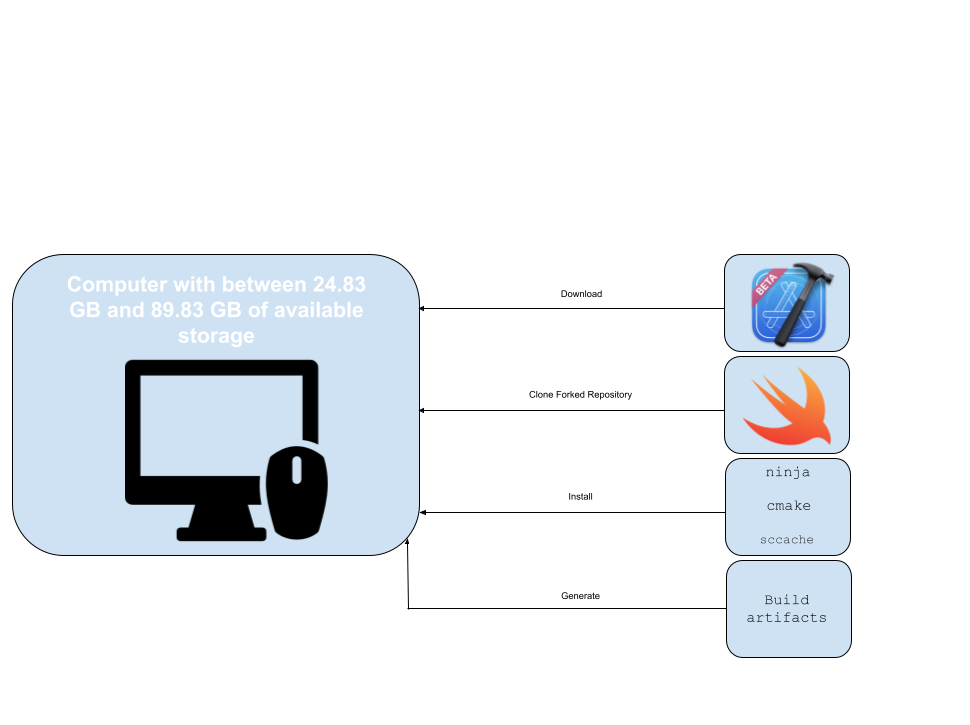
We will be using C++ to work on the compiler, as well as various LLVM methods from the LLVM standard library. We will also be working with different concepts, such as the Hindley Milner Algorithm, since it is a crucial part of the constraint system used for Swift’s type checking. During the process of type checking, the errors that we create for this project are stored into what are known as a fix. After performing a second round of type checking over the constraint system, if these errors cannot be resolved, or the fixes cannot be applied, then these fixes are wrapped into what are called diagnostics, which are displayed to the user.

**6.1.2 Major Software Interactions**

The main user interaction involves clicking a run button in order to either 1) run the compiler to access the output of the compiler, which includes error messages created as a part of Project Cleanup: Swift Property Wrappers, should there be errors associated with property wrappers in the test file they pass to the compiler or 2) run their test program in order to generate fix-its from within their editor, if they have errors associated with property wrappers, which will include the errors generated as a part of this project. Should the user have decided to perform the first action, prior to doing so, they would have had to go through various steps, or interactions, listed in the [user manual](https://github.com/Strieker/lmu-cmsi-401/blob/master/README.md). Such interactions include forking the Swift Compiler’s repository, and following the Swift Compiler’s [getting started guide](https://github.com/apple/swift/blob/main/docs/HowToGuides/GettingStarted.md). The getting started guide also includes a series of interactions, such as cloning the forked repository, installing cmake, ninja, and sccache, and doing a release build of the compiler for the first time, in order to generate a .xcodeproject file, which the user can open to begin interacting with the compiler. From there, the user will have to set up the project’s schemes, as well as set up the targets passed to the compiler at launch time, or at the time the compiler is ran manually by the user.

Regardless of what path the user chooses to go on (either running a test program to generate property wrapper errors, or downloading and setting up the Swift Compiler, and passing it a test file with code that will generate property wrapper errors) the compiler will perform the same actions. It will take the file in question, pass it through the parser to generate a new Abstract Syntax Tree (or AST), and then begin the process of semantically analyzing this tree. This semantic analysis will mainly involve type checking. This type checking entails creating a constraint system, which basically sets up a system of equations for each AST node, that tries to interpret and infer any given type in the program. Should there be any errors that arise during this process, these errors are noted and written as what is known as a fix. If this fix can be resolved during the process of traversing the tree a second time, then the fix is applied. Otherwise, it’s eventually wrapped into a diagnostic and sent to the user in the form of a fix-it (should the user decide to directly run their test file), or within the compiler’s output (should the user decide to pass the file to the compiler as an argument at launch time).

**6.1.3 Architectural Design Requirements**

****

