**6.3 Class, File, User Interface, and Data Structure Descriptions**

**6.3.1 Class and File Descriptions**

The Swift Compiler Project that we are working in as a part of Project Cleanup: Swift Property Wrappers is massive, being about 138.09 GB. It includes a multitude of classes and files, where some files will include multiple (sometimes even nested) classes, since most of these files are a few thousand lines long. In addition, each class will include a multitude of fields and methods, some of which are not directly modified as a part of Project Cleanup: Swift Property Wrappers. For this reason, this document will mainly be describing the purpose of some of the major files in the project that contain some of the most important classes that contribute to the creation of Swift’s constraint graph and constraint system, that allows for type checking and general dynamic semantic analysis associated with the compiler. This is crucial to describe because the constraint system helps generate the fixes associated with the errors created as a part of Project Cleanup: Swift Property Wrappers that need to eventually be transformed into diagnostics and displayed to the user, so that they can fix any issues in their code associated with property wrappers.

**6.3.1.1 CSGen.cpp**

This file implements constraint generation for the type checker portion of the constraint system. This is a part of the main set up for creating the constraint system.

**6.3.1.2 ConstraintSystem.cpp**

This file implements the constraint-based type checker, which is anchored mainly by the **ConstraintSystem** class. This class describes a system of constraints on type variables, as well as the solution, which assigns concrete types to each of the type variables. Constraint systems are typically generated given an expression. It also contains the class, **SolutionApplicationTarget,** which describes the target to which a constraint system’s solution can be applied.

***Main Fields for the ConstraintSystem class:***

* ***Context***: Memory address of an ASTContext object
  + Creates and owns AST nodes, as well as its own thread. This thread maintains local storage for every source file provided to the compiler, along with its own ASTContext.
* ***DC*:** Pointer to a DeclContext object
  + Creates a context object for variable declarations.
* ***Options*:** ConstraintSystemOptions object
  + Stores a list of options, in the form of ConstraintSystemFlags, which handles a variety of specific issues associated with the constraint system, such as whether or not to suppress diagnostics, meaning that any diagnostics generated from the constraint system would not be made visible to the user.
* ***SolverScope:*** embedded/nested class
  + Introduces a new solver scope, which stores a ConstraintSystem, and only tries to solve a certain portion of it, or within a certain scope.

***Main Fields for the SolutionApplicationTarget class:***

* ***Kind***: enum class
  + Labels the kind of solution to which the constraint system can be applied. The possible solutions include expressions, functions, statement conditions, a labeled item, pattern bindings, and uninitialized wrapped values. The class also contains a data structure representing each possible kind, and how to apply it.

**6.3.1.3 TypeCheckPropertyWrapper.cpp**

This file implements semantic analysis for property wrappers. Any findings from this process can be passed back to ConstraintySystem.cpp.

**6.3.1.4 ConstraintLocator.cpp**

This file implements the **ConstraintLocator** class, and its related types, which is used by the constraint-based type checker to describe how a particular constraint was derived. Each locator locates a given constraint system within the expression being type checked, which may refer down into subexpressions and parts of the types of those subexpressions. Each locator is anchored at some expression and contains a path that digs further into the type of that expression. The findings provided by the ConstraintLocator can be passed back to the constraint system to perform further type checking.

***Main Fields for the ConstraintLocator class:***

* ***PathElementKind*:** enum
  + Describes the kind of a particular path element, for example, tuple element, call result, or wrapped value (in the case of property wrappers).
* ***PathElement***: embedded/nested class
  + Represents one element in the path of a locator, which can include both a kind and a value used to describe specific kinds further, for example, the position of a wrapped value element.

**6.3.1.5 CSSimplify.cpp**

This file implements simplifications of constraints within the constraint system, meaning that it uses a variety of heuristics in order to try and simplify the equations applied to each node of the constraint system to speed up the process of analyzing the graph, as well as help solve portions of the constraint system’s system of equations.

**6.3.1.6 CSFix.cpp**

This file implements the **ConstraintFix** class and its related types, which is used by constraint solver to attempt to fix constraints to be able to produce a solution which is easily diagnosable. It also includes **FixKind,** an enum class, which describes the kind of fix applied to a given constraint before visiting it. For example, a kind could include a forced optional or a contextual mismatch.

***Main Fields for the ConstraintFix class:***

* ***CS*:** address of an instance of a ConstraintSystem object
  + Stores the constraint system which contains the fix that needs to be applied
* ***Kind*:** FixKind enum object
  + Stores the kind associated with a given constraint prior to visiting it
* ***Locator:*** pointer to a ConstraintLocator object
  + Location of a given constraint in the constraint graph

**6.3.1.7 CSSolver.cpp**

This file implements a constraint solver used in the type checker. It plays a role in Project Cleanup: Swift Property Wrappers, because it’s responsible for trying to apply or resolve any fixes made in the constraint system.

**6.3.1.8 CSDiagnostic.cpp**

This file implements diagnostics for a constraint system, should a fix not be resolved after re-analyzing the constraint graph. These diagnostics will be made available to the user either in the compiler output, which is visible in the console, or directly as fix-its in the user’s test file.

**6.3.1.9 CSApply.cpp**

This file implements an application of a solution to the constraint system to a particular expression, resulting in a fully-type-checked expression.

**6.3.2 Detailed Interface Descriptions**

**6.3.2.1** **Running from the Swift Compiler project**

Should the user decide to clone a fork of the Swift Compiler project, and follow the [user manual](https://github.com/Strieker/lmu-cmsi-401/blob/master/README.md), they should have a local version of the Swift Compiler project on their computer that passes in a Swift file to the compiler. Then, the user can select the run button from within Xcode when they have the Swift Compiler project open. If the user chooses to hit the run button, at launch time, they will build a version of the compiler to start. Then, if the build succeeds, the compiler will take in the file passed in as an argument at launch time and parse it into an abstract syntax tree. This is then passed to the second phase of the compiler, which is where the majority of Project Cleanup: Swift Property wrappers does its work: the dynamic semantic analysis phase. This phase of the compiler passes the AST to the constraint system generator. From there, the constraint system generator creates values necessary to perform type checking from within the constraint system. As it does this, the type checker begins to check that any types in the constraint graph can be verified, or determined without the use of type inference, and the constraint locator searches for any places in the code where certain pieces of data associated with the AST is needed for a part of the constraint system. After getting all of this information, a variety of heuristics are applied to the constraint system via the constraint system simplifier. As the constraint system is being generated, should any errors occur, a new constraint fix is created. This constraint fix can be applied to the constraint graph when the constraint solver attempts to simplify the system. Should these fixes have issues after reanalyzing the system, these fixes are translated into constraint system diagnostics, which are applied to the constraint graph. This means that the solution for the constraint system is created, which includes any error messages that were stored into diagnostics. These diagnostics can then be viewed from within the compiler’s output, which will show up in Xcode’s embedded console, which can be viewed from within the logs section editor, when the user clicks the most recent launch, or running, time stamp. Should their original test file have contained any errors associated with property wrappers that were added or modified as a part of Project Cleanup: Swift Property Wrappers, then these errors will display in the output. To visually see where the user can view the compiler’s output, or selects the run button, see the [architectural design document](https://github.com/Strieker/lmu-cmsi-401/blob/master/Sage%20Strieker's%20Architectural%20Design.docx) or the [user manual](https://github.com/Strieker/lmu-cmsi-401/blob/master/README.md).

**6.3.2.2 Running from a test file**

Should the user decide to just run a test file directly from within Xcode, without cloning a fork of the Swift Compiler project, then the process that the compiler went through will be same as the last heading. The only differences are that the user would not have needed to pass the file as an argument in the compiler’s scheme editor to pass in at launch time, and the diagnostics would be applied as fix-its, which would display from within the test file itself, as long as it was opened in Xcode.

**6.3.3 Detailed Data Structure Descriptions**

This project mainly involves manipulating an existing abstract syntax tree into a constraint system, and modifying the errors, or fixes, associated with particular constraints in the constraint graph. An abstract syntax tree is created in the parsing phase of a compiler, which involves translating a token stream, with the help of a grammar and a variety of backend frameworks, such as LLVM, into a data structure called a tree. This tree is unambiguous. Should this tree get created successfully, that means that the syntax, or the structure, of the given program passed into the compiler is sound. Now, it is necessary to begin traversing this AST and translating it into a constraint graph in order to perform type checking and ensuring that the meaning behind the program is sound. This system will consist of a graph, in which each node can have constraint systems. Each system’s job is to maintain a system of equations used to solve a given type of a value, or determine if the types provided in a program are legal within Swift. Swift uses a form of the Hindley-Milner algorithm to develop this system of equations after traversing the AST bi-directionally to figure out what the types are for different expressions within a given program. In doing this, type inference is more accurate. Type inference in Swift takes up the bulk of semantic analysis, so it is acceptable to use the words type inference and semantic analysis interchangeably. Any errors that were created as a part of Project Cleanup: Swift Property Wrappers would be added as fixes and placed in the part of any of the constraint system’s systems of equations where the errors originally occurred during the first analysis of the constraint graph. If after a second analysis, the errors cannot be resolved, then the fixes are translated into diagnostics to be made visible to the user.

**6.3.4 Detailed Design Diagrams Section**



