# Prototype of RAGESS Which Is a Tool for Automatically Generating SwiftDiagram to Support iOS App Development

**Haruki Onaga[*], Tetsuro Katayama[*], Yoshihiro Kita†,**
**Hisaaki Yamaba[*], Kentaro Aburada[*], Naonobu Okazaki[*]**
[*]*Department of Computer Science and Systems Engineering, Faculty of Engineering, University of Miyazaki, 1-1 Gakuen-kibanadai nishi, Miyazaki, 889-2192 Japan*
[†]*Department of Information Security, Faculty of Information Systems, Siebold Campus, University of Nagasaki 1-1-1 Manabino, Nagayo-cho, Nishi-Sonogi-gun, Nagasaki, 851-2195 Japan*
*E-mail: onaga@earth.cs.miyazaki-u.ac.jp, kat@cs.miyazaki-u.ac.jp, kita@sun.ac.jp, yamaba@cs.miyazaki-u.ac.jp, aburada@cs.miyazaki-u.ac.jp, oka@cs.miyazaki-u.ac.jp*

**Abstract**

It is difficult to understand the structure of large and complex mobile applications. To support iOS app development, we proposed SwiftDiagram, a visualization of the static structure of Swift source code and confirmed its high usefulness. However, manually drawing SwiftDiagram is labor-intensive. This paper implemented a prototype of RAGESS (Real-time Automatic Generation of SwiftDiagram System), a tool that automatically generates SwiftDiagram by performing static analysis on Swift source code every time an iOS app build is successful.

*Keywords*: Software visualization, Mobile application, Swift

## 1. Introduction

The smartphone and tablet application market is expanding every year [1]. As a result, mobile applications are becoming larger and more complex. The following problems exist for developers of increasingly complex mobile applications.

- Difficult to understand the overall structure of the application
- Difficult to keep track of where changes to the source code may have an impact

To solve the above problems through software visualization, we have proposed SwiftDiagram. SwiftDiagram is a diagram that visualizes the static structure and impact scope of source code written in Swift, a programming language used to develop iOS applications. However, manually drawing a SwiftDiagram takes time and effort. Therefore, this paper implements RAGESS (Real-time Automatic Generation of SwiftDiagram System) to support the development of iOS applications in Swift. RAGESS is a tool that performs a static analysis of the Swift source code and automatically draws the corresponding SwiftDiagram when it detects that the target application builds successfully.

## 2. SwiftDiagram

SwiftDiagram is a diagram that targets source code written in the Swift programming language and supports the design and maintenance of iOS applications by visualizing the following:

- Static structure of type
- Affected scope when making changes to the type

In addition, SwiftDiagram is composed of the following four types of parts and one type of arrow. The appearance of each part and arrow is shown in Fig. 1.

- **Header Part**
  Shows a type identifier, such as type categories and name
- **Details Part**
  Shows type components such as properties, methods, protocol conformance, and so on
- **Extension Part**
  Shows a type extension
- **Nest Part**
  Shows a type that is declared nested inside another type
- **Affected scope (arrow)**
  Shows that changes made to the type at the root of the arrow may affect the component of the type pointed to by the arrow tip

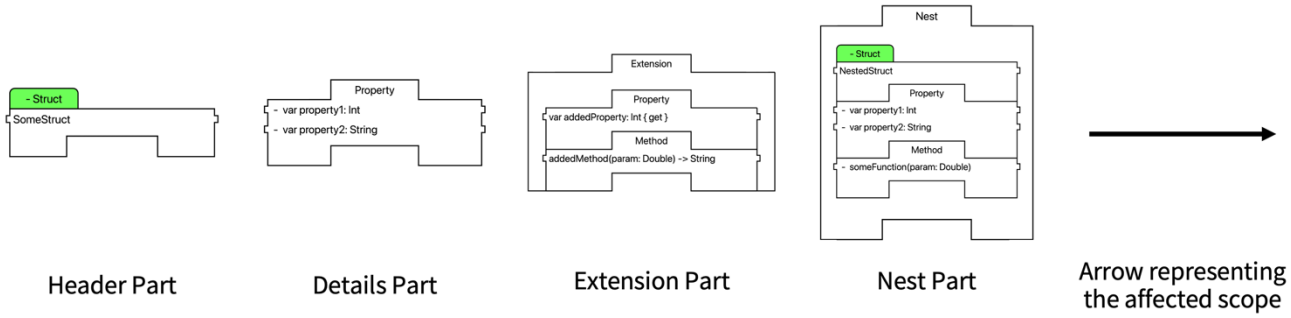


Fig. 1 Component parts of SwiftDiagram

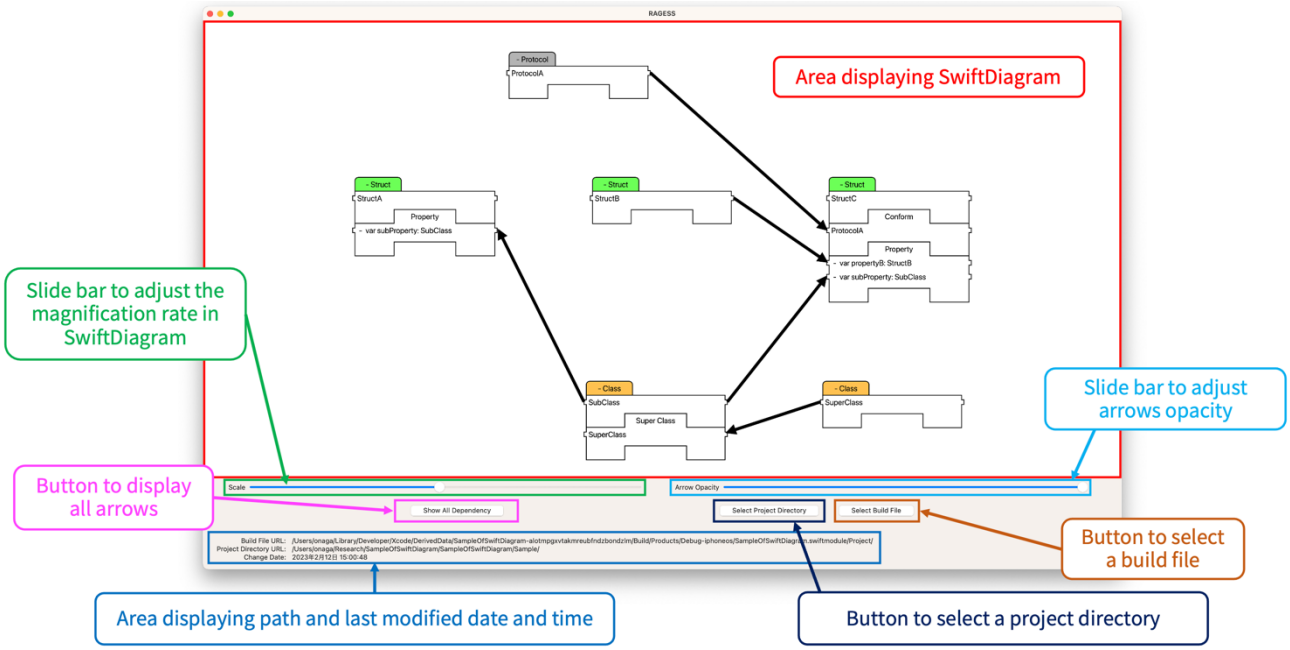


Fig. 2 Overview of RAGESS

## 3. RAGESS

We implement RAGESS as a macOS application in Swift. After launching RAGESS, the user can use RAGESS functions by selecting the path to the project and its build file for which the user wants to draw SwiftDiagram. This chapter describes the overview and functions of RAGESS.

### 3.1. *Overview of RAGESS*

Fig. 2 shows the overview of RAGESS implemented in this paper. RAGESS consists of two areas, two slide bars, and three buttons as shown below.

- Area displaying SwiftDiagram
- Area displaying path and last modified date and time
- Slide bar to adjust the magnification rate in SwiftDiagram
- Slide bar to adjust arrows opacity
- Button to display all arrows
- Button to select a project directory
- Button to select a build file

### 3.2. *Functions of RAGESS*

RAGESS has the following two functions.

- Automatic real-time drawing of SwiftDiagram corresponding the latest source code
- Narrowing down the affected scope

RAGESS monitors the build file of the target project. RAGESS statically analyzes the project's Swift source code every time it detects a change in the build file and automatically draws the corresponding SwiftDiagram. This saves the user from the trouble of updating the SwiftDiagram and means that the source code represented by the SwiftDiagram doesn't contain any syntax errors. Static analysis of Swift source code is

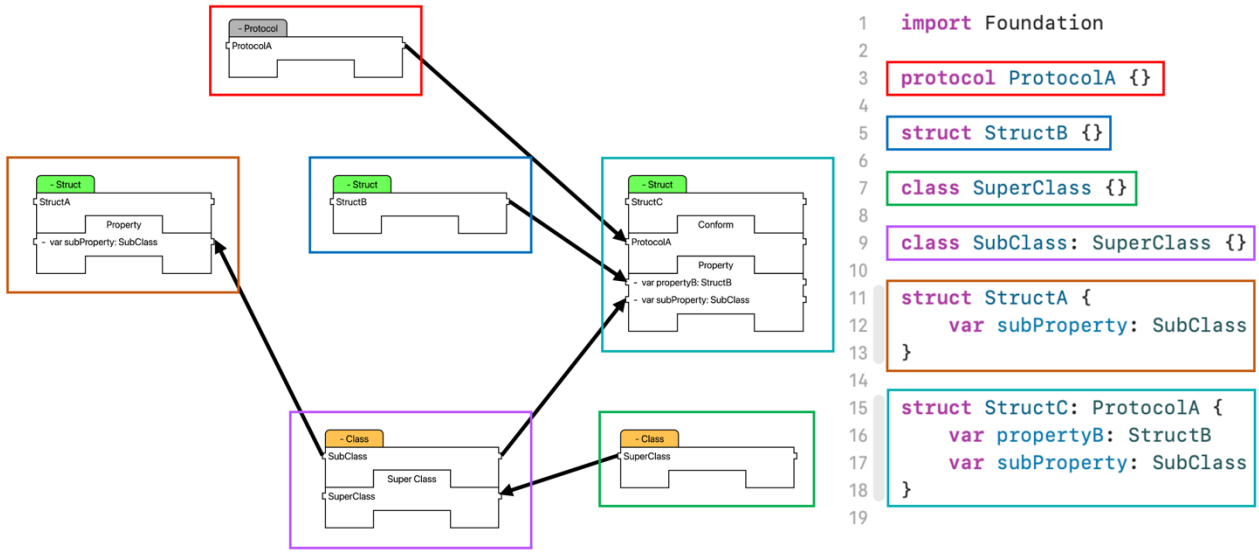


Fig. 3 Correspondence between SwiftDiagram and Swift source code in the sample project

performed by using SwiftSyntax [2], which is open source by Apple. The user can scale the drawn SwiftDiagram by operating the slide bar.

Representing the overall structure of a project in SwiftDiagram could complicate the arrows representing the affected scope and make it difficult for the user to understand the structure. When the user selects a header part, RAGESS narrows down the display to only those arrows that represent the affected scope of the changes made to the type. The user can also operate the slide bar to adjust the opacity of the arrows representing the affected scope. The button to display all arrows, returns the state of the narrowed down affected scope and the opacity of the arrows to their initial state, and displays all the arrows that represent the affected scope for the entire project.

## 4. Application Example

By using an application example, we verify that the RAGESS functionality works correctly for Swift source code that defines multiple types. Fig. 3 shows the correspondence between the SwiftDiagram drawn by RAGESS and the Swift source code in the example. Here, Fig. 2 shows a screen shot of RAGESS monitoring the build file of the project containing the Swift source code in the example, and when the build is successful.

By looking at the StructA of the SwiftDiagram and the Swift source code in Fig. 3, the structure name and properties of the structure in the SwiftDiagram correspond to the structure name and properties of the structure in the Swift source code, respectively. Similarly, looking at StructC, the structure name, properties, and dependencies by protocol conformance in the SwiftDiagram correspond to the structure name, properties, and dependencies by protocol conformance in the Swift source code, respectively.

Also, ProtocolA, SuperClass, SubClass, and StructB in the SwiftDiagram extend arrows toward components of the type that may be affected by changes made to the type.

Hence, it is clear that RAGESS can statically analyze Swift source code and draw the corresponding SwiftDiagram.

## 5. Related work

Emerge [3] is a tool for visualizing Swift source code. It is a browser-based tool that visualizes codebase and dependencies for multiple programming languages like Swift, Kotlin, and TypeScript. Because it visualizes the source code in a graph structure composed of nodes and edges, the user can explore and analyze it visually.

Emerge analyzes source files under the source directory that are written in a language set by the user and generates files for displaying graphs in a Web browser. The user can load the file with a Web browser to analyze the dependencies. However, Emerge cannot reflect changes made to the target source code after analysis in the graph displayed in the Web browser. Also, because it takes types and files as nodes, the user cannot analyze the dependencies of the type components. On the other hand, RAGESS automatically draws SwiftDiagram every time the target application builds successfully. In addition, because arrows representing the dependencies are connected to type components such as properties and methods, the user can understand the dependencies in more detail.

And another tool for visualizing Swift source code is Swiftcity [4], [5]. Swiftcity adapts the city metaphor, which has been studied for Java, C++, and other languages, to unique type extensions, structures, and more of Swift to map and visualize source code into cities and buildings. The height of the building represents the number of lines of code (LOC) of the type, while the width and depth of the building represent the number of methods of the type. The user can use Swiftcity by uploading a project file to the user's Web browser.

Swiftcity is useful for getting an overview of the application. For example, yellow buildings represent type extension, so if a city has many yellow buildings, it means that type extension are frequently used in its source code. The variety of building colors also means the use of all the standard features provided by Swift. However, Swiftcity visualizes type categories and the number of methods, but doesn't visualize elements such as properties and enumerated type cases, nor the dependencies such as function calls and protocol conformance. Therefore, use of Swiftcity is limited to understanding and analyzing an overview of the application in the maintenance process. On the other hand, RAGESS can visualize the size of a type by the height of the SwiftDiagram that combines each part and also components and dependencies of the type.

Therefore, RAGESS provides a more detailed visualization of the structure of applications than the other two tools that visualize Swift source code. In addition, it updates the SwiftDiagram in real-time every time the application builds successfully, thus reducing the time the user needs to modify the deliverables.

## 6. Conclusion

In this paper, we have implemented RAGESS, a tool statically analyzes the project's Swift source code every time it detects a change in the build file and automatically draws the corresponding SwiftDiagram, in order to support the development of iOS applications in the Swift programming language.

We have applied RAGESS to Swift source code and confirmed that it can draw the corresponding SwiftDiagram. We also have confirmed that RAGESS can visualize the structure of the application in more detail and in real-time compared to other tools that visualize Swift source code.

Consequently, RAGESS is expected to support the development of iOS applications in the Swift programming language.

Future works are as follows:

- **Evaluation of usefulness by subject experimentation**
  RAGESS cannot coexist with the current Swift and SwiftSyntax versions and is not buildable. Therefore, in this paper, we couldn't evaluate the usefulness of RAGESS through experimentation in which subjects actually use RAGESS. We need to implement RAGESS to adapt it to current Swift and SwiftSyntax versions and conduct subject experimentation to confirm its usefulness.

- **Implementation of SwiftDiagram editing function**
  We enable SwiftDiagram to edit in the area displaying SwiftDiagrams of RAGESS and reflect the edits in the source code. This function could assist the user in designing new developments or adding new function. We need to implement it because we believe it feature will further increase the usefulness of SwiftDiagram and RAGESS.

- **Implementation of search function**
  In the current RAGESS, the user must visually search for the target type in the area displaying SwiftDiagram to understand the structure of the application. We believe that implementing the function to search by type name or component will further reduce the user's time and effort.

- **Extension of supported syntax**
  RAGESS does not support type inference and can only extract properties whose types are explicitly indicated by type annotations. Since type inference is used in much Swift source code, we believe that supporting type inference would further increase the usefulness of RAGESS. And even if there are more nested types within nested declared types, RAGESS cannot extract that information. It is also not yet compatible with Swift Macros released with Swift 5.9. We plan to make RAGESS compatible with them.

## References

1. Ministry of Internal Affairs and Communications, Japan, Information and Communications in Japan WHITE PAPER 2022, https://www.soumu.go.jp/johotsusintokei/whitepaper/eng/WP2022/2022-index.html
2. GitHub, swift-syntax, https://github.com/apple/swift-syntax (Accessed 2023-12-14)
3. GitHub, emerge, https://github.com/glato/emerge (Accessed 2023-12-14)
4. Rafael Nunes, Marcel Rebouças, Francisco Soares-Neto, Fernando Castor, Poster: Visualizing Swift Projects as Cities, IEEE/ACM 39th International Conference on Software Engineering Companion, pp. 368-370, 2017.
5. Swiftcity, https://swiftcity.github.io/swiftcity-app/ (Accessed 2023-12-14)

## Authors Introduction

Mr. Haruki Onaga

Haruki Onaga received the Bachelor's degree in engineering (computer science and systems engineering) from the University of Miyazaki, Japan in 2023. He is currently a Master's student in Graduate School of Engineering at the University of Miyazaki, Japan. His research interests include software development support through software visualization.

Dr. Tetsuro Katayama

He received a Ph.D. degree in engineering from Kyushu University, Fukuoka, Japan, in 1996. From 1996 to 2000, he has been a Research Associate at the Graduate School of Information Science, Nara Institute of Science and Technology, Japan. Since 2000 he has been an Associate Professor at the Faculty of Engineering, Miyazaki University, Japan. He is currently a Professor with the Faculty of Engineering, University of Miyazaki, Japan. His research interests include software testing and quality. He is a member of the IPSJ, IEICE, and JSSST.

Dr. Yoshihiro Kita

Yoshihiro Kita received a Ph.D. degree in systems engineering from the University of Miyazaki, Japan, in 2011. He is currently an Associate Professor with the Faculty of Information Systems, University of Nagasaki, Japan. His research interests include software testing and biometrics authentication.

Dr. Hisaaki Yamaba

He received the B.S. and M.S. degrees in chemical engineering from the Tokyo Institute of Technology, Japan, in 1988 and 1990, respectively, and the Ph D. degree in systems engineering from the University of Miyazaki, Japan in 2011. He is currently an Assistant Professor with the Faculty of Engineering, University of Miyazaki, Japan. His research interests include network security and user authentication. He is a member of SICE and SCEJ.

Dr. Kentaro Aburada

He received the B.S., M.S, and Ph.D. degrees in computer science and system engineering from the University of Miyazaki, Japan, in 2003, 2005, and 2009, respectively. He is currently an Associate Professor with the Faculty of Engineering, University of Miyazaki, Japan. His research interests include computer networks and security. He is a member of IPSJ and IEICE.

Naonobu Okazaki

He received his B.S, M.S., and Ph.D. degrees in electrical and communication engineering from Tohoku University, Japan, in 1986, 1988 and 1992, respectively. He joined the Information Technology Research and Development Center, Mitsubishi Electric Corporation in 1991. He is currently a Professor with the Faculty of Engineering, University of Miyazaki since 2002. His research interests include mobile network and network security. He is a member of IPSJ, IEICE and IEEE.