SUGISAKA MASANORI

Journal of Robotics, Networking and Artificial Life Vol. 10(4); March (2024), pp. 336–341
ISSN (Online): 2352-6386; ISSN (Print): 2405-9021
https://alife-robotics.org/jrnal.html

Research Article

RAGESS: A Tool for Real-time Automatic Generation of SwiftDiagram to Support iOS Application Development

Haruki Onaga[1], Tetsuro Katayama[1], Yoshihiro Kita[2], Hisaaki Yamaba[1], Kentaro Aburada[1], Naonobu Okazaki[1]

[1]*Department of Computer Science and Systems Engineering, Faculty of Engineering, University of Miyazaki, 1-1 Gakuen-kibanadai nishi, Miyazaki, 889-2192 Japan*
[2]*Department of Information Security, Faculty of Information Systems, Siebold Campus, University of Nagasak, 1-1-1 Manabino, Nagayo-cho, Nishi-Sonogi-gun, Nagasaki, 851-2195 Japan*

ARTICLE INFO

Article History
Received 24 November 2023
Accepted 19 June 2024

Keywords
Software visualization
Mobile application
iOS
Swift

ABSTRACT

It is difficult for mobile application developers to understand the structure of large and complex mobile applications. To support iOS application development, we proposed SwiftDiagram: a visualization of the static structure of Swift source code, and demonstrated its usefulness. To further support them, this paper has implemented RAGESS(Real-time Automatic Generation of SwiftDiagram System), which is a software visualization tool. RAGESS performs static analysis on Swift source code and automatically generates the corresponding SwiftDiagram whenever the target project build succeeds.

1. Introduction

The market for mobile applications targeting smartphones and tablets is expanding every year [1]. As a result, mobile applications are becoming larger and more complex, driven by the demand for continuous releases in short cycles [2]. In such development, the following problems exist for mobile application developers.

- Difficult to understand the overall structure of the application by reading the source code
- Difficult to keep track of where changes made to the source code may have an impact within the project

We proposed SwiftDiagram to solve these problems with a software visualization approach. SwiftDiagram is a diagram that visualizes the static structure and impact scope of source code written in Swift [3]. Swift is a programming language used to develop iOS applications. However, manually drawing a SwiftDiagram is labor-intensive and error-prone. Therefore, to further support iOS application developers, we develop a prototype tool called RAGESS (Real-time Automatic Generation of SwiftDiagram System). RAGESS monitors the developer-selected project. When it detects that the project's build is successful, it performs static analysis on the source code and generates the corresponding SwiftDiagram. RAGESS immediately displays the generated SwiftDiagram, categorized into protocols, structures, classes, and enumerations.

2. SwiftDiagram

SwiftDiagram is a diagram that supports the design and maintenance of iOS applications for source code written in Swift. To achieve this, SwiftDiagram visualizes the following:

- The static structural composition of types
- The impact range resulting from modifications made to a type or its constituent elements

Corresponding author 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

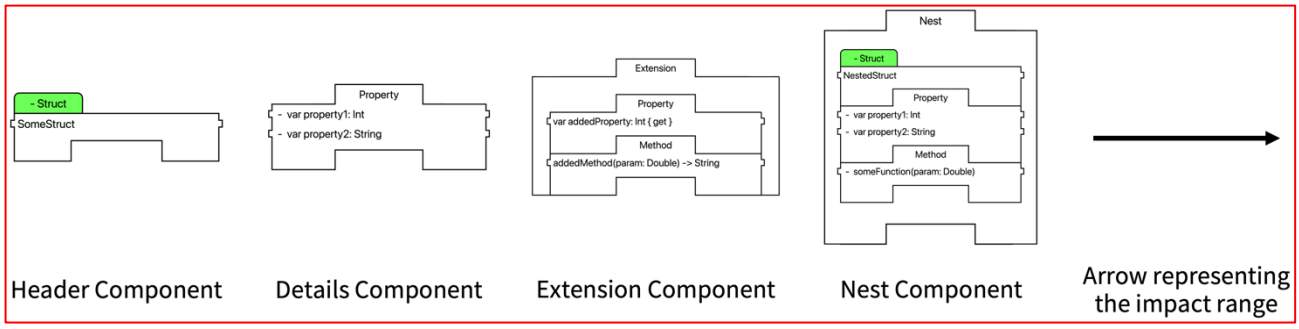


Fig.1 Component of SwiftDiagram

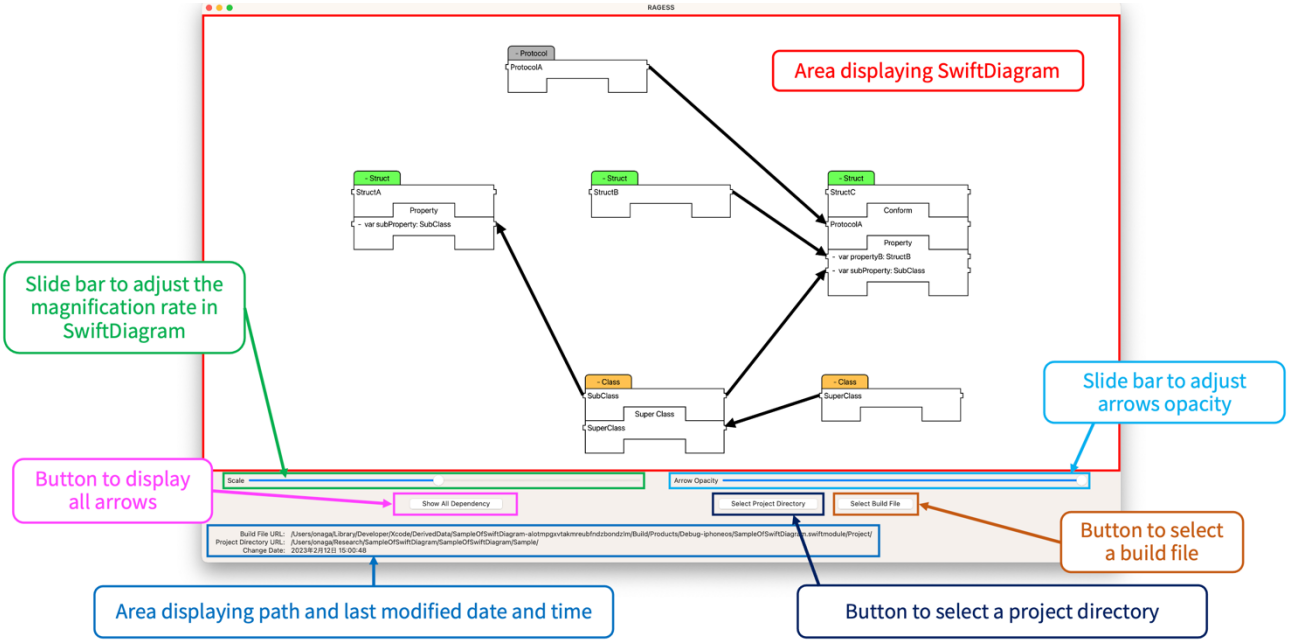


Fig.2 An appearance of RAGESS's user interface, with descriptions of its main components

SwiftDiagram is constructed using four different kinds of components and a single type of arrow, which are explained the below. The components of SwiftDiagram are shown in Fig. 1.

- **Header Component**
  Shows a type identifier, such as access level, type categories, and name
- **Details Component**
  Shows the components of a type, such as variables, functions, inheritance relationships, and so on
- **Extension Component**
  Shows type extensions defined in extension code blocks
- **Nest Component**
  Shows types that are declared within the scope of another type
- **Impact Range(arrow)**
  Shows that changes to the type at root of the arrow may impact the component of the other type of destination of the arrow

3. Prototype of RAGESS

We implement RAGESS as an application running on macOS using Swift and SwiftUI [4]. To use functions of RAGESS, the mobile application developer selects the project directory and the build file they want to visualize .111with SwiftDiagram using the standard macOS file selection dialog.

3.1. *Appearance of RAGESS*

Fig. 2 shows the appearance of RAGESS after it has generated and displayed SwiftDiagram. The user interface of RAGESS consists of the following components:

- An area displaying the generated SwiftDiagram
- An area displaying the path of selected project and the last change date and time

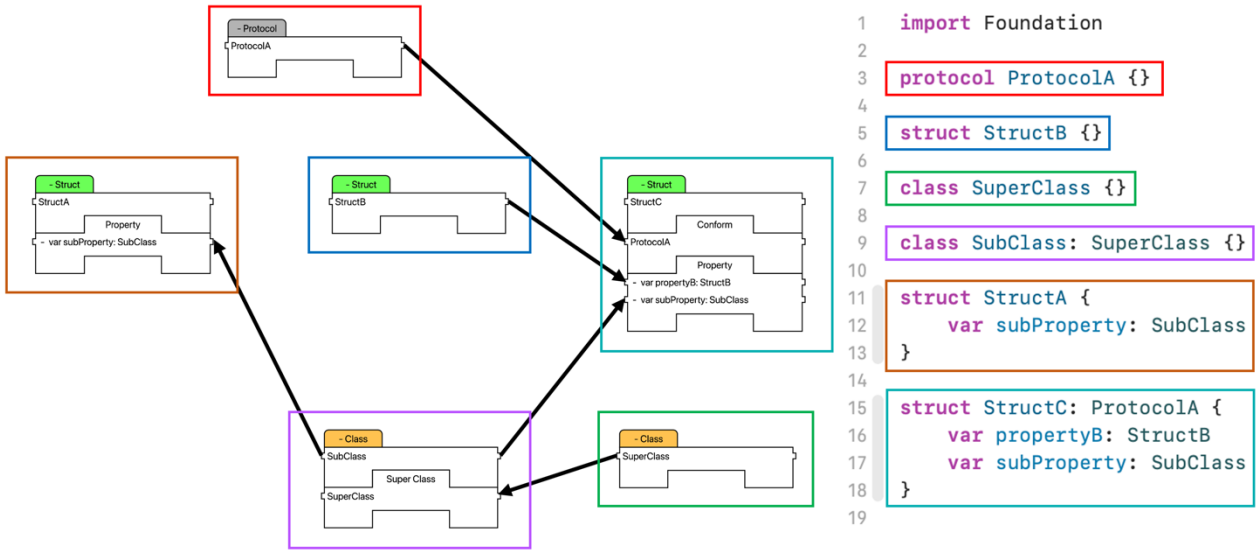


Fig.3 An example representing the correspondence between the SwiftDiagram and the Swift source code

- A slide bar to control the magnification level of the SwiftDiagram
- A slide bar to control the opacity of the arrows
- A button to make all arrows visible
- A button to select a directory of the target project
- A button to select a build file of the target project

3.2. *Features of RAGESS*

RAGESS has two primary features, which are as follows:

- Real-time automatic generation and display of the SwiftDiagram corresponding to the latest source code of the target project whenever its build succeeds
- Display of the impact range of a single type

RAGESS keeps track of the build file of the target project. Whenever RAGESS notices a change in the build file, it conducts a static analysis of the project's Swift source code and automatically generate the corresponding SwiftDiagram. This not only eliminates the need for developers to manually update the SwiftDiagram but also guarantees that the generated SwiftDiagram does not contain syntax errors. The static analysis of Swift source code is carried out using SwiftSyntax [5], an open-source tool made available by Apple. The developer can adjust the magnification level of the displayed SwiftDiagram by operating the slide bar.

Representing the entire structure of the target project in SwiftDiagram might result in complex arrows depicting the impact range, potentially hindering the developer's understanding of the structure. When the developer clicks on a header component of the SwiftDiagram generated by RAGESS, it narrows down the display to only the arrows representing the impact range of changes made to that specific type. Furthermore, the user can adjust the opacity of the arrows by using slide bar. Clicking the "Show All Dependencies" button reverts the impact range and arrow opacity to their initial state, displaying all the arrows representing the impact range for the entire project.

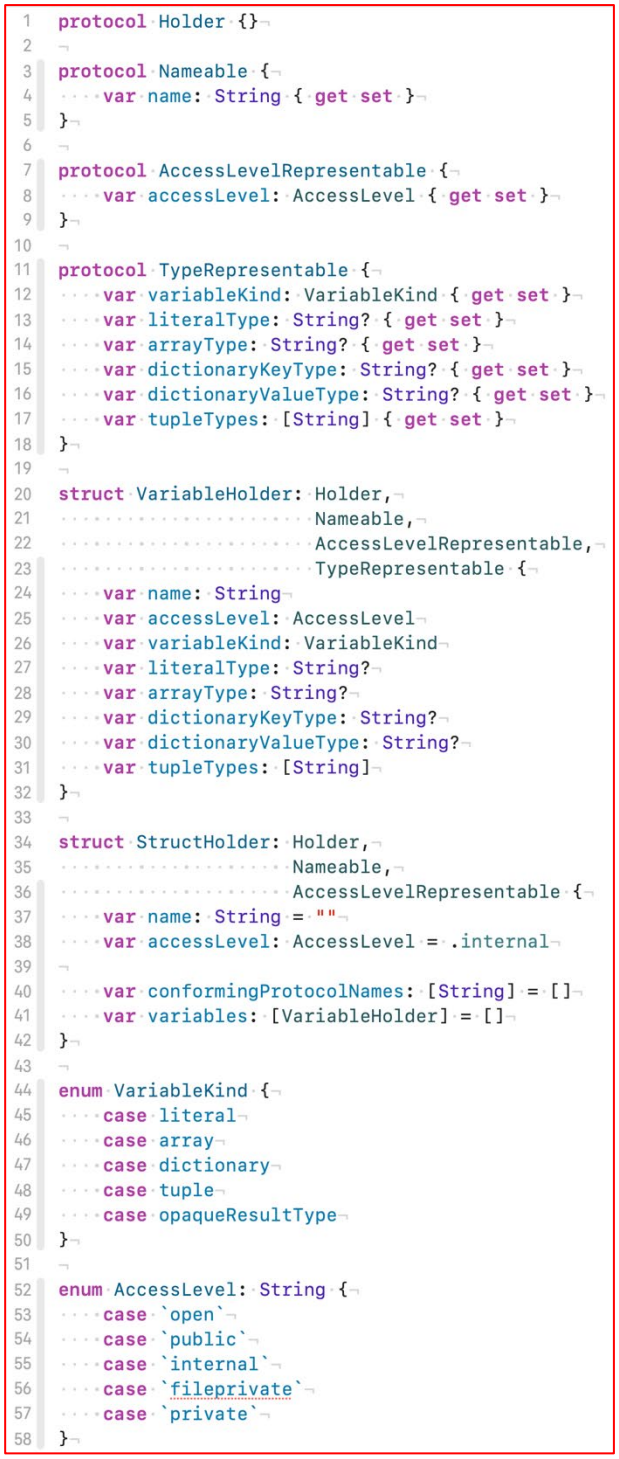
4. Application Example

As one of application examples, we verify that the features of RAGESS works correctly for Swift source code that defines multiple types. Fig. 3 shows the correspondence between the SwiftDiagram generated by RAGESS and the Swift source code in the example. Fig. 2 shows the screen of RAGESS when it monitors the build file of the project containing the Swift source code in the example and detects a successful build.

Comparing the SwiftDiagram and Swift source code in Fig. 3, we can see that the structure name and properties of StructA in the SwiftDiagram correspond to those in the Swift source code. Likewise, for StructC, the structure name, properties, and dependencies based on protocol conformance in the SwiftDiagram align with their counterparts in the Swift source code.

Additionally, in the SwiftDiagram, ProtocolA, SuperClass, and StructB have arrows pointing to the components of other types that could potentially be impacted by modifications made to these types.

Furthermore, as another application example, we verify that the RAGESS functionality works correctly for a simplified Swift source code that represents a portion of itself. Fig. 4 shows the simplified Swift source code for the data objects in RAGESS that hold structures and variables. Fig. 5 shows the screen after RAGESS monitors the build file of the sample project containing



```
protocol Holder {}

protocol Nameable {
    var name: String { get set }
}

protocol AccessLevelRepresentable {
    var accessLevel: AccessLevel { get set }
}

protocol TypeRepresentable {
    var variableKind: VariableKind { get set }
    var literalType: String? { get set }
    var arrayType: String? { get set }
    var dictionaryKeyType: String? { get set }
    var dictionaryValueType: String? { get set }
    var tupleTypes: [String] { get set }
}

struct VariableHolder: Holder,
                       Nameable,
                       AccessLevelRepresentable,
                       TypeRepresentable {
    var name: String
    var accessLevel: AccessLevel
    var variableKind: VariableKind
    var literalType: String?
    var arrayType: String?
    var dictionaryKeyType: String?
    var dictionaryValueType: String?
    var tupleTypes: [String]
}

struct StructHolder: Holder,
                     Nameable,
                     AccessLevelRepresentable {
    var name: String = ""
    var accessLevel: AccessLevel = .internal

    var conformingProtocolNames: [String] = []
    var variables: [VariableHolder] = []
}

enum VariableKind {
    case literal
    case array
    case dictionary
    case tuple
    case opaqueResultType
}

enum AccessLevel: String {
    case `open`
    case `public`
    case `internal`
    case `fileprivate`
    case `private`
}
```

Fig.4 RAGESS source code with simplified data objects holding structures and variables

the Swift source code from Fig. 4 and detects a successful build.

Comparing the source code in Fig. 4 with the SwiftDiagram in Fig. 5, the names of the types, variables, and enumeration cases match respectively. In the source code, the access levels for types and variables are omitted, and the SwiftDiagram generated by RAGESS correctly applies the default access level “internal”.

Moreover, the source code in Fig. 4 declares four protocols, two structures, and two enumerations. Among the protocols, Holder, AccessLevelRepresentable, and Nameable are conformed to by both structures. In the SwiftDiagram in Fig. 5, arrows are connected from each of these three protocols to the “Conform” parts of the two structures. The TypeRepresentable protocol is only conformed to by the VariableHolder structure, and correspondingly, in the SwiftDiagram in Fig. 5, an arrow is connected only to the “Conform” part of VariableHolder. Furthermore, the StructHolder structure has a property with an array of VariableHolder structures as its type. In the SwiftDiagram in Fig. 5, an arrow is connected from the VariableHolder structure to the property of the StructHolder structure. Similarly, the two enumerations also have arrows connecting to the properties for which they serve as types.

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

5. Related work

Another tool for visualizing Swift source code is Emerge [6]. Emerge is a web-based tool that provides visualizations of the codebase and dependencies for various programming languages, such as Swift, JavaScript, and Python. By visualizing the source code as a graph structure consisting of nodes and edges, it enables visual exploration and analysis.

Emerge examines source files in the specified directory that are written in a developer-defined language and produces file for rendering graphs in a web browser. By loading these files in a web browser, developers can analyze the dependencies. However, any modifications made to the source code after the analysis are not reflected in the graph displayed in the web browser. Moreover, because Emerge represents types and files as nodes, it is unable to analyze the dependencies at the level of type components. On the other hand, RAGESS automatically generates SwiftDiagram every time the target project’s build is successful. Additionally, since the arrows are linked to type components like properties and methods, developers can gain a more comprehensive understanding of the dependencies.

And another visualization tool targeting Swift is Swiftcity [7], [8]. This tool adapts the city metaphor to Swift’s unique features like type extensions and structures, visualizing the source code as cities and buildings. The height of each building corresponds to the number of lines of code (LOC) in a type, while the width and depth represent the number of methods.

However, Swiftcity doesn’t visualize components such as properties and enumeration cases, or dependencies like function calls and protocol conformance. On the other hand, RAGESS can represent the scale of a type through the combined height of the SwiftDiagram components and visualize type components beyond just methods, as well as the scope affected by changes.

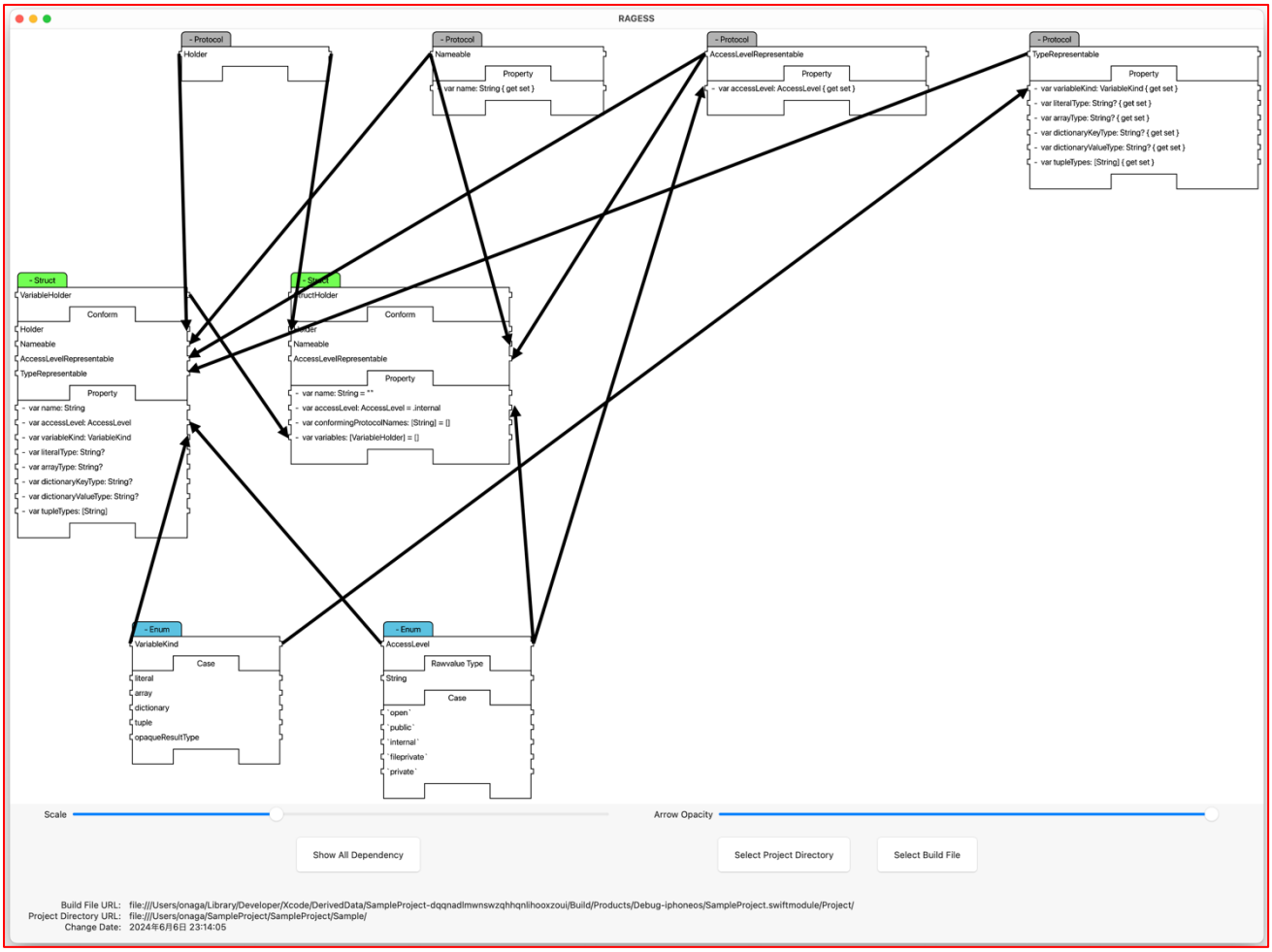


Fig.5 Screen after applying RAGESS to the source code in Fig.4

Therefore, RAGESS can visualize the structure of a project in greater detail. Moreover, by updating the SwiftDiagram in real-time every time the project build succeeds, RAGESS reduces the effort required by developers to modify the artifacts.

Therefore, RAGESS provides a more detailed visualization of the target project structure than the other two tools for visualizing Swift source code. Moreover, it automatically updates the SwiftDiagram after each successful build of the target project, reducing the amount of time the developer has to modify the artifacts.

6. Conclusion

In this paper, we have developed a prototype of RAGESS, a software visualization tool aimed at supporting iOS application development. RAGESS performs static analysis on source code written in the Swift programming language and generates and displays a SwiftDiagram. By monitoring the target project and detecting successful builds, RAGESS automatically updates the SwiftDiagram.

We applied RAGESS to two different examples of Swift source code and confirmed that it can generate SwiftDiagram corresponding to each example. Additionally, we compared RAGESS with other tools that apply software visualization approaches to Swift source code. The results showed that RAGESS can visualize that structure of a target project in more detail and in real-time compared to the other tools.

Consequently, RAGESS has the potential to support iOS application development using the Swift programming language.

Future works include the following:

- Evaluation of the usefulness of RAGESS through subject experiments involving experienced iOS application developers
- Implementation of search feature
- Visualization at different levels of abstraction

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. Ronald Jabangwe, Henry Edison, Anh Nguyen Duc. Software engineering process models for mobile app development: A systematic literature review. The

Journal of Systems & Software. 2018, 145, pp.98-111.
3. Apple, Swift, https://developer.apple.com/documentation/swift (Accessed 2024-6-13)
4. Apple, SwiftUI, https://developer.apple.com/documentation/swiftui (Accessed 2024-6-13)
5. GitHub, swift-syntax, https://github.com/apple/swift-syntax (Accessed 2023-12-14)
6. GitHub, emerge, https://github.com/glato/emerge (Accessed 2023-12-14)
7. Rafael Nunes, Marcel Rebouças, Francisco Soares-Neto, Fernando Castor. Poster: Visualizing Swift Projects as Cities. IEEE/ACM 39th International Conference on Software Engineering Companion. 2017, pp. 368-370.
8. Swiftcity, https://swiftcity.github.io/swiftcity-app/ (Accessed 2023-12-14)

Authors Introduction

Mr. Haruki Onaga

He 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

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

He 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.

Dr. 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.