Poster: Visualizing Swift Projects as Cities

Rafael Nunes, Marcel Rebouças, Francisco Soares-Neto, Fernando Castor
Center of Informatics
Federal University of Pernambuco
Recife, Brasil
{rngs,mscr,fmssn,castor}@cin.ufpe.br

***Abstract*—Human's natural ability to perform software maintenance is compromised as a project gets bigger, older, and more complex. Software visualization tools can be used to mitigate this problem, easing software understanding. However, no such tools are available for Swift, a new programming language that is experiencing widespread adoption by developers. In this paper we present SwiftCity, a software visualization tool that uses the City Metaphor. Visualizing Swift projects as cities is different from projects in other languages, such as Java and Javascript. Swift employs a number of different units of modularity that are not available in these languages, such as extensions and structs.**

I. Introduction

Software development is often a complex activity that involves the coordination of teams, clients' requirements and limitations of infrastructure, time and human resources. As a project ages and gets more complex, it becomes harder to understand, manage, and identify system behavior by looking just at its source code. To mitigate this problem, data visualization can be utilized. In the context of systems, tools can provide a visual representation of the code used to support development, maintenance, inspection, and debugging phases[1]. Many approaches of visualization were already proposed, both in 2D [2][3] and 3D [4]. By having an extra dimension, 3D visualizations allows the display of more information, but also bring different problems such as navigation, occlusion, and comparison of elements, among others.

Wettel and Lanza[4] proposed the *city metaphor*, where code elements are mapped and visualized in the context of neighborhoods and buildings. The metaphor has already been experimentally analyzed, with positive results [5]. Also, there are already implementations of the metaphor for JavaScript [6] and for the analysis of Java concurrency [7].

This paper presents SwiftCity, a realization of the *city metaphor* for Swift. Since its release, Swift's increasing adoption already puts it as one of the most popular programming languages[1]. Swift is also bound to be further adopted, as it is being developed to replace Objective-C in Apple's ecosystem. Notwithstanding, to the best of our knowledge, there are currently no visualization tools targeting Swift. Furthermore, it is not possible to directly apply other realizations of the city metaphor to Swift because the language employs a number of different units of modularity that are not available in these languages, such as extensions and structs.

[1]December 2016 - http://www.tiobe.com/tiobe-index/

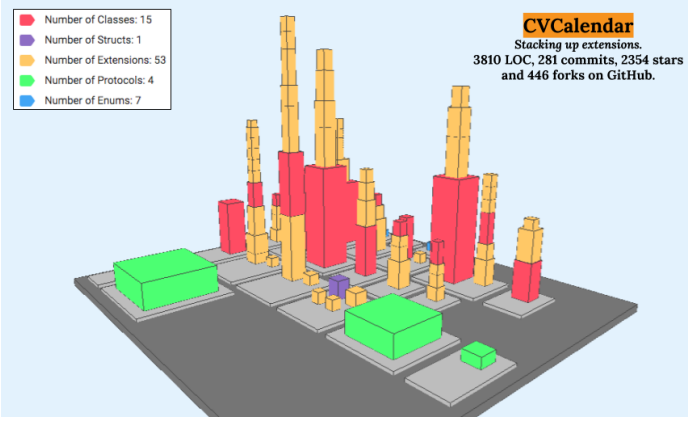

Fig. 1: City of the CVCalendar Project

II. SwiftCity

Swift. Swift is a modern, multiparadigm language that combines imperative, object-oriented, protocol-oriented, and functional programming. It contains most elements that are well-known in object-oriented languages, such as *classes*, *structs*, *protocols* (which are similar to Java's *interfaces*), and *enums*.

Less known, Swift also contains *extensions*, which are elements that add new functionality to an existing class, structure, enumeration, or protocol type. This element even enables the extension of types in situations where the original source code is not available, such as adding new methods to *String* or *Int*, or making an existing type conform to a protocol. Extensions are not unlike inter-type declarations from aspect-oriented programming languages [8]

Visual Aspects. A block is the fundamental element of the metaphor, representing a city building. In the original metaphor, a block was mapped to represent a class of the system. However, Swift has several elements that must be represented that were not available in Java or Javascript (*e.g.,* structs and extensions). We decided to differentiate the types of elements by using different colors for each kind since, according to Zanolie [9], forms are usually associated with the representation of elements of different nature, yet all elements are Swift types that belong to different categories. Moreover, the use of different forms would go against the metaphor, translating the elements into objects that do not necessarily create the immersion of a city.

Several metrics can be applied in order to define the measurements of a block. For this work, the block height

is given the number of lines of code of the element, while the width and length are defined by the number of methods. These metrics are usually used in analysis tools [10]. We chose to differ from the original metrics, which used the number of methods as height and number of attributes as width and length. We believe number of LOC is a more general, albeit imperfect, approach to represent the complexity of an element, since not every element has methods and enumerations, in particular, have *cases* which are neither methods nor fields, but can still be captured by LOC.

The city topology is usually mapped to elements of the system hierarchy. Since there are no packages, namespaces or anything equivalent in Swift, the mapping limits itself to two clear levels: the project and the files. The lowest layer represents the city limits as being the whole project (dark gray), and each element of the second layer represents a district, which translates to a file within the project (light gray).

Another challenge was how to represent the relations between *extensions* and other elements. Two approaches were considered satisfactory: (i) adding another layer that would contain the element and its *extensions*, and (ii) stacking *extensions* to the element, so they would become a single building. There are, however, advantages and disadvantages of each. The first, for example, allows a cleaner view as the city elements are more spread. On the other hand, the second better translates the complexity of the elements.

Technical Aspects. From the collection of data to its visualization, three tools had to be implemented. Firstly, we modified the Swift compiler in order to get the contents of its *-dump-ast* flag. This flag is responsible for printing an AST representation of each source-code file. This representation, however, was irregular, not being originally meant by Swift's developers to be machine-parseable, so it had to be modified. The result is a custom compiler and *toolchain* that we used integrated into Xcode. Secondly, we implemented a parser that could translate the AST representation to the metadata that we wanted to visualize, such as information about classes and extensions. Lastly, we implemented SwiftCity[2], a client-based web tool meant to consume the produced metadata and dynamically present its visualization through the city metaphor. More information about the tools and their repositories can be found at their website[3].

III. Case Study

In this section, we present an analysis of the visualizations of two popular Swift projects selected from GitHub. Their cities are shown in Figure 2.

Alamofire. Alamofire is an API for HTTP network written in Swift. While the project seems to utilize extensions as a pattern, some really big classes are present in the city. There are also two strangely large blue towers (enums). Although enumerations may contain instances of pre-computed methods and properties, it is difficult to see such an implementation

[2]https://swiftcity.github.io/web-app
[3]https://swiftcity.github.io/website/

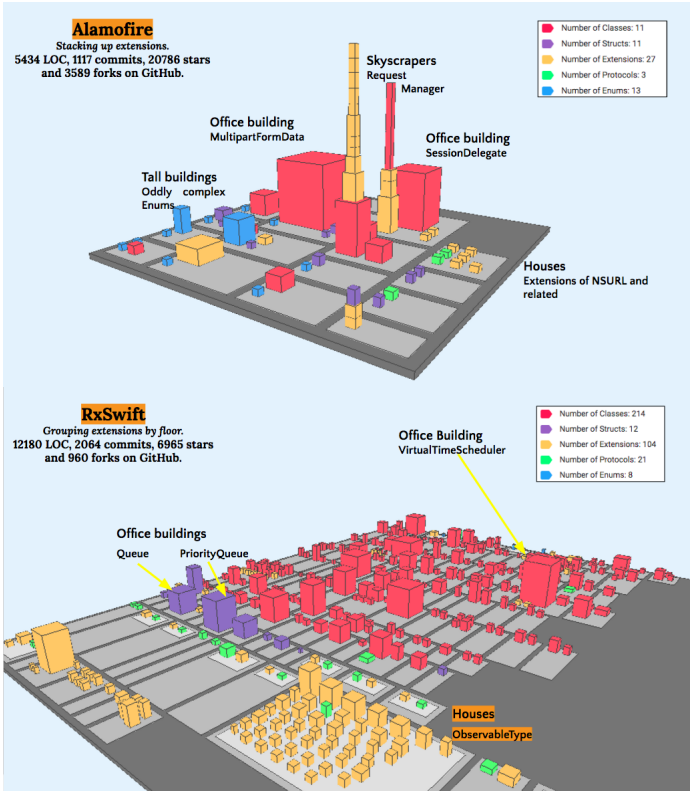

Fig. 2: Cities of Alamofire and RxSwift

with about 200 LOC. Therefore, it may be interesting to analyze why such enums have grown so large.

RxSwift. RxSwift is an implementation of reactive programming for Swift. It uses extensions in a different way from Alamofire, since there are few occurrences that extend classes. There are also more extensions to native types of Swift, shown as the ungrouped yellow blocks in the city. More interestingly, however, is that the `ObservableType` protocol has 53 extensions to its definition. We also noticed that structs are not used so much in comparison to classes, differently from the Alamofire project, which can be a practice imposed by the developers or a provisional decision.

We can identify some patterns from the analyzed projects:

- Yellow buildings throughout the city: Extensions are widely used in projects, for decoupling and modularity. However, some extensions are even more complex than extended element.
- Variety in the colors of the blocks: The system's design seems to make use of all the of the standard elements provided by Swift.

IV. Conclusions and Future Work

In this paper, we presented SwiftCity, a tool to visualize Swift projects as cities. Contributions include a mapping for elements of the language to the city metaphor and a useful application. Our next steps include improvements to the tool, the visualization, and the completeness of the translated metaphor.

REFERENCES

[1] D. E. Fyock, "Using visualization to maintain large computer systems," *IEEE Computer Graphics and Applications*, vol. 17, no. undefined, pp. 73–75, 1997.

[2] M. Wilhelm and S. Diehl, "Dependencyviewer - a tool for visualizing package design quality metrics," in *In VISSOFT*, 2005.

[3] M. Lanza and S. Ducasse, "The class blueprint: Visually supporting the understanding of classes," *IEEE Transactions on Software Engineering*, vol. 31, no. undefined, pp. 75–90, 2005.

[4] R. Wettel and M. Lanza, "Visualizing software systems as cities," ser. VISSOFT 2007, 2007, pp. 92–99.

[5] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: A controlled experiment," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 551–560. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985868

[6] M. Viana, E. Moraes, G. Barbosa, A. Hora, and M. T. Valente, "JSCity: Visualizao de sistemas JavaScript em 3D," in *III Workshop de Visualização, Evolução e Manutenção de Software (VEM)*, 2015, pp. 73–80.

[7] J. Waller, C. Wulf, F. Fittkau, P. Dohring, and W. Hasselbring, "Synchrovis: 3d visualization of monitoring traces in the city metaphor for analyzing concurrency," *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, vol. 00, no. undefined, pp. 1–4, 2013.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997, pp. 220–242.

[9] K. Zanolie, S. Teng, S. E. Donohue, A. C. van Duijvenvoorde, G. P. Band, S. A. Rombouts, and E. A. Crone, "Switching between colors and shapes on the basis of positive and negative feedback: An fmri and {EEG} study on feedback-based learning," *Cortex*, vol. 44, no. 5, pp. 537 – 547, 2008, special Issue on Selection, preparation, and monitoring: Current approaches to studying the neural control of action. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010945207001177

[10] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 131–142. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390648