EVA: A Tool for Visualizing Software Architectural Evolution

Daye Nam, Youn Kyu Lee, and Nenad Medvidovic
University of Southern California
{dayenam,younkyul,neno}@usc.edu

ABSTRACT

EVA is a tool for visualizing and exploring architectures of evolving, long-lived software systems. *EVA* enables its users to assess the impact of architectural design decisions and their systems' overall architectural stability. (Demo Video: https://youtu.be/Q3bnIQz13Eo)

1 INTRODUCTION

As software systems grow in size and complexity, their high-level structure, i.e., *software architecture*, also grows in importance [17]. In practice, a system's architecture often decays over time as critical design decisions are added and modified [16]. Since architectural decay has been shown to incur technical debt and decrease a system's maintainability (e.g., [13]), understanding architectural changes and tracking the decay become critical.

A number of techniques have been proposed for analyzing architectural changes during the course of a system's evolution (e.g., [10, 11, 19]). However, existing techniques and the recent empirical studies (e.g., [8, 12]) only provide high-level summaries regarding architectural changes without analyzing underlying reasons and impact of those changes. Even for some existing techniques that support visualizing architectures, they only show an architecture at a single point in the system's evolution [12, 18]. Therefore, it is still difficult to track architectural changes.

To aid software architects in understanding an architecture's evolution and analyzing architectural changes, we present *EVA* (Evolution Visualization for Architectures). *EVA* provides three views: (1) *Single-Release Architecture*, (2) *3-D Architecture-Evolution*, and (3) *Pairwise Architecture-Comparison*. To present design decisions and/or rationales behind them, *EVA* collects relevant data from issue repositories and displays the data along with the architecture-evolution visualization. By supporting a variety of architecture recovery techniques, *EVA* enables users to see multiple architectural representations of a system. While the primary target for *EVA* are software system architects, the tool provides useful information for other software engineers as well, by showing the traceability of system decisions over time [15]. We evaluated *EVA*'s accuracy in the context of real systems to show its soundness.

EVA is distinguished from existing tools because (1) it visualizes software architectural evolution, (2) it allows architectural exploration, (3) it enables users to gauge the impact of a design decision that led to architectural changes, and (4) it enables users to assess the system's architectural stability [13].

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5663-3/18/05.

2 BACKGROUND

A software system's architecture captures a view of the key design choices made for that system. An architecture is represented as a set of interconnected *components*, each of which encapsulates a subset of the system's functionality and/or data [17]. An architecture of an existing software system can be extracted by architecture *recovery* techniques [10, 11, 19]. Recovery techniques cluster code-level entities (e.g., methods, classes) based on their functionalities and inter-relationships, and consider the resulting clusters as architectural components. While recent research, including our own [8, 12, 13], has begun investigating how recovery may be applied to understand the architectural changes in long-lived systems, the goals of that research have led to highly abridged and summarized results regarding system change and decay, preventing a deeper exploration of the collected data. *EVA* targets this very shortcoming.

3 DESIGN AND IMPLEMENTATION

EVA takes as input (1) a set of system releases and (2) the system's issue repository. It presents the output of its analysis in three different views. An example of the three views extracted from the implementation of Google *Guava* [4] is shown in Figure 1; the figure is further explained below.

EVA's architecture comprises two layers, as shown in Figure 2. (1) *EVA*'s *front-end* is the *Client* component responsible for interacting with the user and generating multiple visualizations. (2) *EVA*'s *back-end* obtains the issues that have induced the system changes, extracts the target system's architecture from its implementation, maps the contextual information to the architecture, and identifies architectural changes over a given time-span. The back-end consists of *Issue Processor*, *Architecture Extractor*, *Context Mapper*, and *Change Identifier* components. We describe each component next.

Client interacts with a user and the back-end. When a user provides a URL of the system issue repository (e.g., https://github.com/-google/guava) and uploads the source code for the desired releases, *EVA*'s *Client* forwards them to the back-end layer. As the back-end returns the architectural evolution information, *Client* depicts the information. It uses multiple types of notations and provides interactive visualization in order to effectively convey a large amount of information (e.g., many entities). For example, a small circle represents a code-level entity; the detailed information of each entity, such as its name, is hidden by default and is only displayed when a user's mouse hovers over the entity, as shown in Figure 1(c).

Issue Processor crawls the issue data from the issue repository and processes them. Its output is a set of issue information blocks. Each block contains the corresponding issue's title, tag, description, resolution date, and a list of code-level entities that have been changed to resolve the issue. *Issue Processor* consists of two sub-components, *Issue Crawler* and *Issue Tagger*.

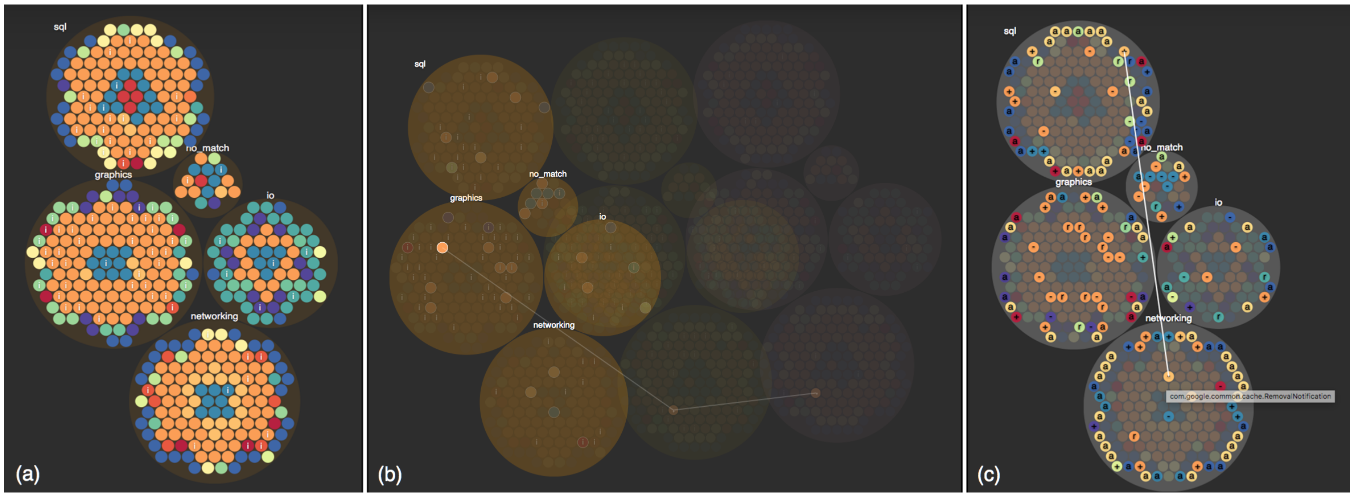

Figure 1: *EVA*'s Three Types of Visualization.

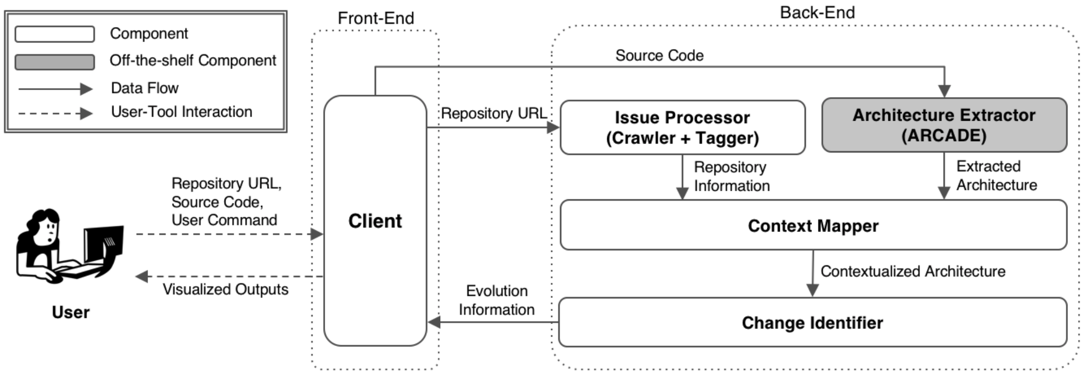

Figure 2: *EVA*'s Architecture

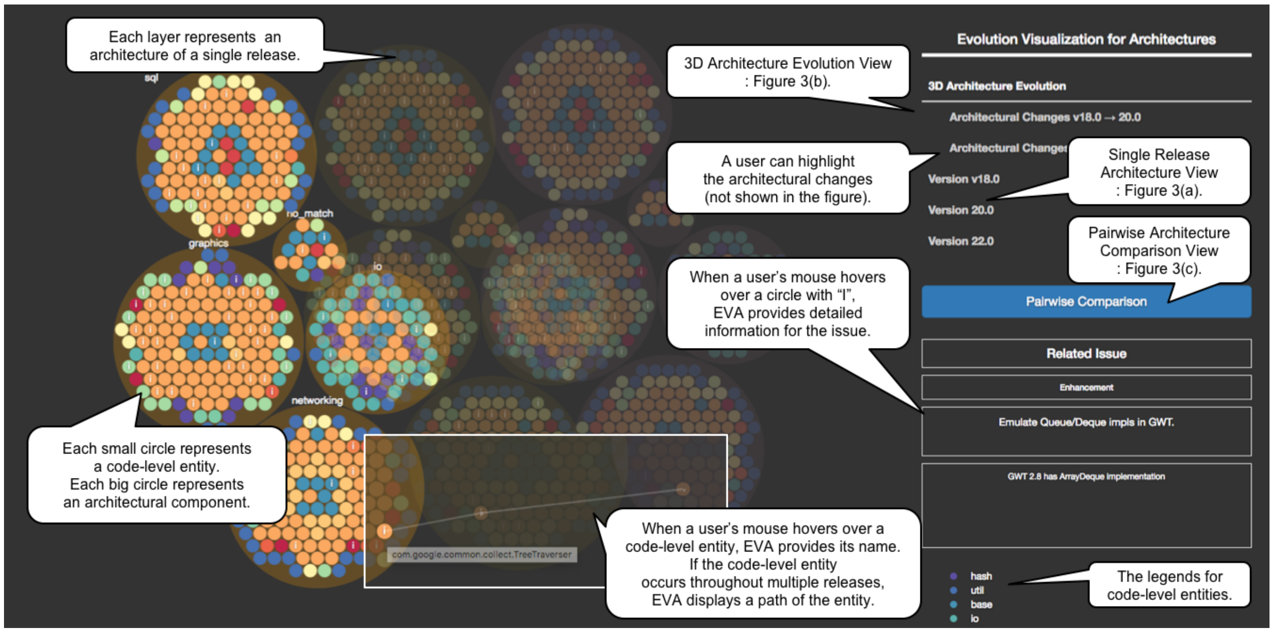
Each layer represents an architecture of a single release.
sql
no_match
graphics
io
networking
3D Architecture Evolution View : Figure 3(b).
A user can highlight the architectural changes (not shown in the figure).
When a user's mouse hovers over a circle with "I", EVA provides detailed information for the issue.
Each small circle represents a code-level entity. Each big circle represents an architectural component.
When a user's mouse hovers over a code-level entity, EVA provides its name. If the code-level entity occurs throughout multiple releases, EVA displays a path of the entity.
com.google.common.collect.TreeTraverser
Evolution Visualization for Architectures
3D Architecture Evolution
Architectural Changes v18.0 → 20.0
Architectural Changes
Single Release Architecture View : Figure 3(a).
Version v18.0
Version 20.0
Version 22.0
Pairwise Architecture Comparison View : Figure 3(c).
Pairwise Comparison
Related Issue
Enhancement
Emulate Queue/Deque impls in GWT.
GWT 2.8 has ArrayDeque implementation
hash
util
base
io
The legends for code-level entities.

Figure 3: Annotated Screenshot of *EVA*'s 3-D Architecture-Evolution View

Issue Crawler crawls issue data in a given repository. To collect issues that have actually induced system changes, it only crawls the issues that are resolved and whose subsequent changes have been reflected in the system. *Issue Crawler* also extracts code-level entities that have been changed when resolving the issue, and a list of release dates to map the issue to the architecture.

Issue Tagger adds tags (e.g., *bug*, *performance*) to each issue. This enables users to quickly classify the main concern of an issue. If an issue has been already tagged by system maintainers, *EVA* adopts that tag. In other cases, *EVA* uses a classifier that automatically tags an issue as *enhancement*, *bug*, *documentation*, or *performance*, based on the issue's title and description. The details of our classifier are provided in Section 3.1.

Architecture Extractor extracts architectural information in multiple releases from a code repository. Currently, this information includes architectural components and corresponding code-level entities because that is the information provided by existing recovery techniques. *EVA* is extensible to include additional information.

Context Mapper helps users infer the reasons behind architectural changes. A system's architectures and issue data are extracted from different sources. To use the issue data as context for architectural analysis, it is important to identify the relationship between them. As *Issue Processor* passes on a list of release dates and issue information blocks, *Context Mapper* maps the issues to code-level entities in the recovered architectures. When the resolution date of an issue falls between two consecutive release dates, *Context Mapper* relates the issue with the latter release. It then maps the code-level entities listed in the issue information block to the entities in the architecture of the related release.

Change Identifier identifies architectural changes between pairs of consecutive releases. It focuses on three types of architectural changes: (1) addition/removal of an architectural component, (2) addition/removal of a code-level entity, and (3) a change in code-level entity's assignment to an architectural component. The first type of changes indicate that significant functionality or data have been added/removed in the system, and/or that the system has undergone significant structural changes. The remaining two types of changes are indicative of structural shifts and functionality additions/removals that are localized in scope.

3.1 Implementation

We have implemented *EVA*'s *Issue Processor*, *Context Mapper*, and *Change Identifier* components using Python. Within the *Issue Processor* component, we used PyGithub [5] for *Issue Crawler*, and *gensim* [2] for *Issue Tagger*. For *Issue Tagger*, we trained our classifier on approximately 16, 000 issue-tag pairs collected from Github [3] repositories. We first generated vectors using Doc2Vec [14] for each issue and trained a logistic regression classifier. *Issue Processor*, *Context Mapper*, and *Change Identifier* consist of ~900 newly written SLOC, in addition to several off-the-shelf libraries. Intermediate data are transformed into JSON files with pre-defined formats.

EVA's *Architecture Extractor* wraps ARCADE [12], a workbench that employs a suite of architecture recovery techniques. Any recovery technique (e.g., [10, 11, 19]) can be wrapped in the same manner. The set of recovered architectures is transformed into a single JSON file with a pre-defined format.

Finally, we implemented *EVA*'s *Client* as a web application, using HTML5 with JavaScript libraries that include D3 [1] for data-driven visualization and TweenMax [7] for 3-D visualization. *Client* combines ~1,300 newly written SLOC with off-the-shelf libraries.

4 KEY FEATURES

EVA provides visualization capabilities, the broader context of architectural evolution, and multiple coordinated views.

Visualization. Given a set of source codes across multiple releases, *EVA* visualizes the system's architectural evolution. *EVA* depicts a code-level entity as a small circle contained within an architectural component, which is depicted as a larger circle. *EVA* distinguishes different groups of code-level entities (e.g., implementation packages) using color-coding. For example, in Figure 3, code-level entities extracted from package *common.util* are colored blue while entities extracted from *common.escape* are green. To enable users to analyze multiple aspects of architectural evolution, *EVA* provides three different types of views (recall Figure 1).

Single-Release Architecture View depicts the architecture of one release as shown in Figure 1(a). This view enables users to understand the functionality of each architectural component based on code-level entities. Label "i" on the code-level entity circle indicates that an implementation issue was mapped to the entity. When a user's mouse hovers above the "i", *EVA* displays the detailed information of the issue (e.g., its tag and title).

3-D Architecture-Evolution View presents the architectures of multiple releases in a single compositional view, as shown in Figure 1(b). In this view, the architecture of each release is shown in a layer, and the chosen set of releases is presented as stacked layers. A code-level entity that occurs across multiple releases is traced by a line that enables the user to track changes. When a code-level entity moves from one architectural component to another, *EVA* highlights the entity by adding a white border around its corresponding circle, as shown in the selected entity in Figure 1(b).

Pairwise Architecture-Comparison View shows the architectural differences between two releases, as shown in Figure 1(c). Labels "a" and "r" on the circles representing code-level entities indicate that a given entity has been added or removed. When a code-level entity is moved from one architectural component to another, the entity's circle is tagged with "-" in the earlier release and with "+" in the later release. Coupled with the color-coding, this allows a user to intuitively detect functionality shifts within architectural components. For example, the group of *yellow* circles labeled with "a" in Figure 1(c) quickly indicates to a user that functionality related to package *com.google.common.graph* is newly added to the system.

Since *EVA* visualizes a system's architecture and its changes in a single view, a user can easily get a sense of the amount and scope of changes. This allows the user to assess the architecture's stability.

Contextualization. Understanding a system's architectural evolution requires the ability to reason why a given change has been made. To this end, *EVA* displays implementation-issue data along with the architectures, as depicted in the lower-right portion of Figure 3. This information enables the user to infer the rationale behind an architectural change. Namely, when an issue is mapped to a code-level entity that was involved in an architectural change and *EVA* labels the entity's corresponding circle with an "i", the user can observe this easily. The user may reasonably assume that at least part of the rationale for the architectural change stems from the issue. Finally, the number of code entities that are labeled with a given issue is indicative of the scope of the architectural design decisions made in order to resolve the issue: the design decisions may be confined to a single code-level entity, multiple entities within a single architectural component, or multiple components.

Multiple Views. *EVA*'s *Architecture Extractor* (recall Figure 2) is an extensible module that can wrap different recovery techniques. For example, ARC [11] relies on information retrieval to isolate the concerns implemented by a system, while ACDC [19] leverages static dependency analysis. By relying on multiple recovery techniques, a user can view the system's architecture from multiple perspectives and improve her overall understanding of the system.

5 EVALUATION

We have assessed *EVA* for usability and accuracy. Due to length restrictions, we only illustrate *EVA*'s accuracy. Specifically, we highlight *EVA*'s (1) *precision*, i.e., its identified changes that are actually architectural, and (2) *recall*, i.e., whether *EVA* missed any of the actual architectural changes. As our subject, we use Google *Guava* [4], a set of libraries for Java comprising ~1 MSLOC. Our discussion in this section is based on using ACDC [19] as the recovery technique within *EVA*; similar results were obtained with other techniques available in ARCADE [12] (recall Figure 2).

EVA flagged more than 1,000 architectural changes in *Guava*'s six most recent major releases alone. There was no ground-truth available to calculate *EVA*'s **precision**, forcing us to manually inspect *EVA*'s output. Instead of inspecting every change which would have been time consuming, we selected 100 of the changes randomly; 92 of those were architectural (i.e., 92% precision). All false positives occurred because of ACDC's component-clustering decisions. In each case, the false positives can be mitigated by applying additional recovery techniques and correlating the results.

Open-source projects tend not to maintain lists of architectural changes [9]. For this reason, we decided to measure *EVA*'s **recall** using the *Guava* release notes. As the purpose of the release notes is to inform end-users of software updates, some architectural changes may be omitted and multiple related changes summarized in a single note. We manually inspected each note and the corresponding source code, to determine which notes are architectural. We found a total of 59 notes that are architectural across *Guava*'s six latest releases, of which *EVA* was able to identify 57 (i.e., 97% recall).

6 RELATED WORK

Tu and Godfrey [18] proposed an integrated approach to analyze a system's architectural evolution with metrics, visualization support, and techniques for reasoning about structural changes. Behnamghader et al. [8] described a large-scale empirical study of architectural changes across different system releases. ARCADE [12] employs third-party libraries to support visualizing architectures recovered from implementations. SonarQube [6], an open-source platform for continuous inspection of code quality, has several plug-ins for visualizing architectural evolution. All of the above approaches make general observations, provide highly summarized results, only support static visualization of the architecture of a single system release, and do not support focusing on specific design decisions or the rationales behind them.

7 CONCLUSION AND FUTURE WORK

EVA helps explore, visualize, and understand multiple facets of architectural evolution. It explicitly relates architecture with implementation details, and allows easy and intuitive system evolution tracking across multiple releases. We are in the process of deploying *EVA* to a large development organization, which will help assess and improve *EVA*'s scalability and guide our plans for updating it for more general use. We plan to extend *EVA* to provide additional information that can help users understand architectural evolution further. Our current work is focusing on determining and presenting explicit rationales behind architectural changes, with the goal to support tracking design rationale over time. We also aim to automatically identify architectural changes leading to different types of technical debt for more efficient analysis. *EVA* is available for download from https://github.com/namdy0429/EVA.

8 ACKNOWLEDGMENTS

This work is supported by the U.S. National Science Foundation under grants No. 1618231 and 1717963, U.S. Office of Naval Research under grant No. N00014-17-1-2896, and by Huawei Technologies.

REFERENCES

[1] 2017. D3. https://d3js.org. (2017).
[2] 2017. Gensim. https://radimrehurek.com/gensim. (2017).
[3] 2017. Github. https://github.com. (2017).
[4] 2017. Guava. https://github.com/google/guava. (2017).
[5] 2017. PyGithub. http://pygithub.readthedocs.io/en/latest/. (2017).
[6] 2017. SonarQube. https://www.sonarqube.org. (2017).
[7] 2017. TweenMax. https://greensock.com/tweenmax. (2017).
[8] P. Behnamghader, D. M. Le, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. 2017. A Large-scale Study of Architectural Evolution in Open-source Software Systems. *Empirical Software Engineering* 22, 3 (2017).
[9] J. Coelho and M. T. Valente. 2017. Why Modern Open Source Projects Fail. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering.*
[10] J. Garcia, I. Ivkovic, and N. Medvidovic. 2013. A Comparative Analysis of Software Architecture Recovery Techniques. In *Proceedings of the 28th International Conference on Automated Software Engineering.*
[11] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai. 2011. Enhancing Architectural Recovery Using Concerns. In *Proceedings of the 26th International Conference on Automated Software Engineering.*
[12] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. 2015. An Empirical Study of Architectural Change in Open-Source Software Systems. In *Proceedings of the 12th Working Conference on Mining Software Repositories.*
[13] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic. 2016. Relating Architectural Decay and Sustainability of Software Systems. In *Proceedings of the 13th Working Conference on Software Architecture.*
[14] Q. Le and T. Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on Machine Learning.*
[15] M. Paixão, J. Krinke, D. G. Han, C. Ragkhitwetsagul, and M. Harman. 2017. Are Developers Aware of the Architectural Impact of Their Changes?. In *Proceedings of the 32nd International Conference on Automated Software Engineering.*
[16] D. E. Perry and A. L. Wolf. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* 17, 4 (1992).
[17] R. N. Taylor, N. Medvidovic, and E. Dashofy. 2009. *Software Architecture: Foundations, Theory, and Practice.* Wiley.
[18] Q. Tu and M. W. Godfrey. 2002. An Integrated Approach for Studying Architectural Evolution. In *Proceedings of the 10th International Workshop on Program Comprehension.*
[19] V. Tzerpos and R. C. Holt. 2000. ACDC: An Algorithm for Comprehension-Driven Clustering. In *Proceedings of the 7th Working Conference on Reverse Engineering.*