Can I Remove This Method? How Live Feedback from the Ecosystem Supports Co-Evolution

Manuel Leuenberger
Software Composition Group
University of Bern
Bern, Switzerland
manuel.leuenberger@inf.unibe.ch

Abstract

Albeit open-source projects have been co-evolving since years, upgrading a library can still be a tedious task for developers. APIs change over time, and breaking changes require precious developer time to adapt a dependent project's code. We present our vision on how embracing co-evolution in general, and library upgrades in specific, as a first-class citizen in our IDE, can support both API authors and API users. In a tiny self-experiment we show how API authors can profit from live feedback on the compatibility of their changes. API users can profit by being provided migrations to be applied on the dependent's code.

CCS Concepts • **Software and its engineering → Software maintenance tools**; *Patterns*;

Keywords co-evolution, API migration, refactoring, IDE, API usage

preprint

1 Introduction

When a library's API changes over time, its dependents have to adapt to the new version of the API. While many of these changes are refactorings that can be recovered and replayed [3], some changes are of a more complex nature and require careful changes in a dependent [2]. Due to the non-triviality of the co-evolution of a library and its dependents, library developers are reluctant to change their public APIs [2], and developers of dependents are reluctant to upgrade to new versions of a library [7]. This situation leaves unexploited potential for improvement on both sides. Once a version of an API is defined and published, library developers may value backwards-compatibility higher than changing and improving an API. Developers of dependent projects stick to older versions of a library because an upgrade would require changes in their code and extensive retesting. Sticking to an old version of a library is not without risk, *e.g.*, outdated dependencies potentially reveal known security issues.[1]

Although a library and its dependent projects co-evolve, communication of changes across projects is usually only possible by rudimentary means. Libraries may publish a new version along with a change log or a migration guide in textual form.[2] Developers of dependents have to filter out the relevant changes and assess the impact on their projects individually and mostly without tool support. Developers of a library dependent can provide bug reports in case they observe broken behavior in a new version. Library developers on the other side are eager to know how their APIs are used [5]. Yet, to learn about how their library is used, developers have to explore the ecosystem. All of the aforementioned tasks lack proper support by the IDE, therefore developers have to rely on external resources, *e.g.*, issue trackers, wikis, mailing lists, Q&A communities, which can be hard to search and extract the relevant aspects for a specific project. We believe that is in the interest of the actors on both sides of the story to establish a tighter feedback loop that is well integrated into their tooling. Adapted tools for the ecosystem-aware co-evolution context should support library developers in exploring the ecosystem and assessing the impact of changes. Library dependents should be able to collect the changes affecting themselves quickly, as well as they could profit from a migration path reified as an executable, inspectable, and debuggable entity.

In this paper we present our vision on how an ecosystem-aware, tool-supported co-evolution could look like. We present a small self-experiment around the post-mortem analysis of a breaking change introduced by the author of this paper. The redesign of a dropdown widget in a widget library lead to the removal of a method, which in turn broke a dependent that relied on this method. As a mean to avoid or mitigate the effects of a breaking change, we propose an integrated approach that enables the observation and control over the effects of a change in a library on its dependents. By running the dependent's examples or tests relevant to a library change automatically, we can provide live feedback on the effects of the change to the library developer. This makes

IWST'19, August 27-29, 2019, Cologne, Germany
2019.

[1] https://help.github.com/en/articles/about-security-alerts-for-vulnerable-dependencies
[2] For an example of the Elasticsearch project, see https://www.elastic.co/guide/en/elasticsearch/reference/current/breaking-changes-7.1.html

the dependent system both inspectable and debuggable for the library developer, which in turn opens the opportunity to create migration scripts to be used by the developers of the dependent.

The rest of the paper is structured as follows. In section 2 we present a tiny case study of a breaking change. Our implementation of a live feedback prototype for co-evolution is outlined and discussed in section 3. Related work is summarized in section 4, and conclusions and future work can be found in section 5.

2 A Tiny Case Study

Our tiny case study is based on the experience of the author of this paper.

While working on a UI widget library for Pharo Smalltalk[3], the dropdown widget required a redesign to make it more flexible and easier to use. As the widget included some duplicated code as well as unused state and methods, the goal of the first phase was to clean up the dropdown code. In the course of this clean up, the method `BrDropdownLook >> #dropdownTarget:` was removed, which sets the element the dropdown is attached to. Running hundreds of examples that exercise the widget library confirmed that the method was indeed unused by these examples. Thus, the change was considered viable and the changes were pushed. A few days later, it was discovered that the removal of the method broke the tests in a dependent project that used the method. As a consequence, the initially removed method was added again to support the needs of the dependent project.

An analysis of this story reveals multiple moments where a lack of information leads to suboptimal decisions. First, it was unclear that the removed method was public. The method was removed because it was believed to be internal, as it was neither documented as public, nor exercised in the library's tests. Better tests and better documentation could help to mitigate this problem. Second, it was unclear that a dependent used the method. This is generally the point where the integration in IDEs breaks. IDEs usually assume a closed ecosystem around the projects in the local workspace. This assumption does not hold in the case of libraries. The ecosystem includes its dependents that live outside of the common IDE workspace.

In total it took several days from the initial question "Can I remove this method?" to the conclusive answer "No, it is used.". We want to answer this question and similar ones quicker, live in the best case.

3 Live Feedback from the Ecosystem

We want to give library developers quick feedback on the effects of their changes, with only minimal changes to their development setup. To achieve this we need to be able to query the ecosystem in a similar way as it is done in the Ecosystem Monitoring Framework. [10]

3.1 Architecture

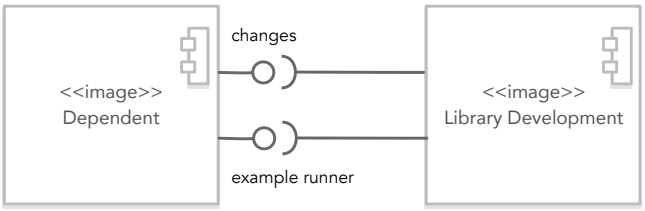

Figure 1. Architecture to connect and synchronize a library image with a dependent image running dependent code.

Our prototype is based on a library image, in which the library developer performs the library changes, and a dependent image, which has a dependent installed together with the library. Figure 1 shows how the two images are connected. The dependent image runs on the same machine as the library image, but it could also run on another machine, *e.g.*, as on-demand images on a continuous integration server. In our prototype the dependent image offers two endpoints. Through the *changes* endpoint the library image pushes all code changes to the dependent image to synchronize the library code in both images. The *example runner* endpoint provides an interface to evaluate a code snippet in the dependent image and answers the snippet's result or exception. The communication relies on Seamless[4], which enables interaction with remote objects by proxying message sends through a network tunnel.

3.2 Tool Integration

Our main goal is to provide live feedback on the effects of library changes. We extend the system browser with badges on classes and methods, indicating whether the class or method is used by the dependent, as well as a green/red colors representing success or failure of the executed dependent examples, see Figure 2 and Figure 3.

Whenever some code changes in the library image, the changes are automatically pushed and applied on the dependent image. Additionally, a change also triggers the execution of examples to exercise the dependent system and gather feedback on the effects of the change. In our case study, we are only interested in examples exercising the code of the dropdown widget. Hence we instrument the methods in the dropdown widget package and track which examples in the dependent use a specific method. After the first change in the library image the 75 relevant method are instrumented and all of the 189 examples of the dependent are run in 25.2s, yielding 51 examples that exercise the dropdown widget. The instrumentation can be removed for subsequent

[3] http://www.pharo.org

[4] https://github.com/pharo-ide/Seamless

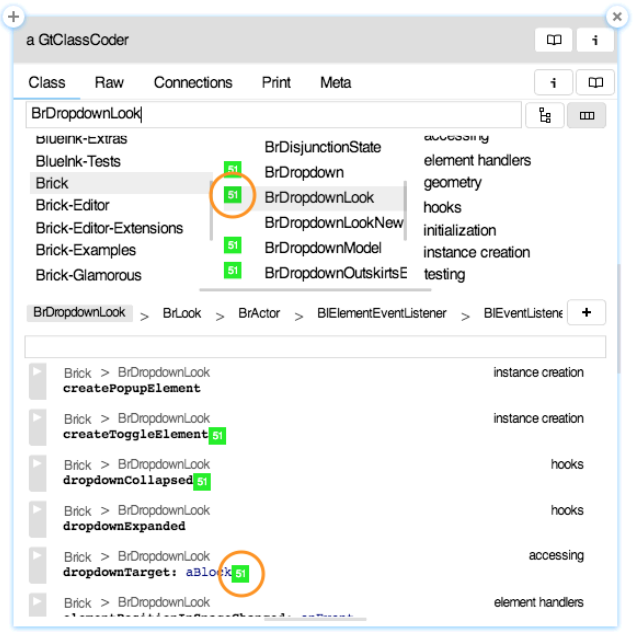

Figure 2. Class browser before removing `#dropdownTarget:`. Extensions visualize library usage and dependent health. Green badges show how many dependent examples execute successfully and use a class or a method. A click on the badges allows to inspect example return values, *e.g.*, badges circled in orange.

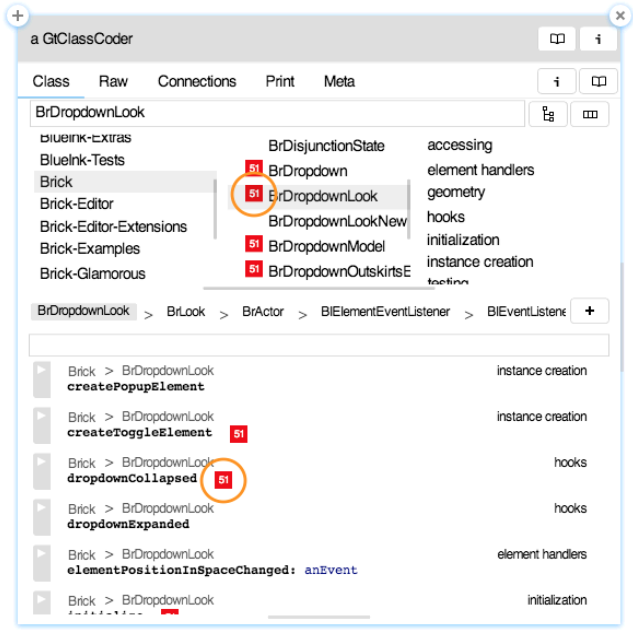

Figure 3. Class browser 4.7s after removing `#dropdownTarget:`. Extensions visualize library usage and dependent health. Red badges show how many dependent examples fail during execution, but use a class or a method. A click on the badges allows to inspect and debug exceptions, *e.g.*, badges circled in orange.

changes. From now on we only have to execute 51 examples without any instrumentation, which takes as little as 4.7s.

As the whole process is automated, the library developer receives live feedback on the effects on a dependent on of every change in the library.

3.3 Discussion

The presented prototype for live feedback from the ecosystem has several drawbacks. First, library dependents have to be detected. This could be solved by mining dependency relation in package repositories, such as GitHub. Second, running multiple images besides a library development image requires resources that can be allocated. In the setup of the case study the dependent image is running on the same machine as the library development image, which makes inspection and debugging of the effects of changes easy. We could also imagine a setup where dependent images are run on-demand on a continuous integration server, which makes inspection and debugging more complex though. Third, the feedback has to be treated with care as the quality depends on the quality of the dependent tests. Depending on how exhaustive the examples and tests of a dependent exercise the relevant library code can vary. No observed exceptions do not necessarily imply a non-breaking change. Likewise, not breaking one specific dependent cannot guarantee that no dependent is affected by a change. As a counter measure, static analysis techniques could be combined with the dynamic technique currently used to gain a more complete picture of the library usage in the ecosystem.

Embracing co-evolution as a first-class citizen in our IDE also offers new opportunities. First, live feedback tightens the feedback loop from a change in a library to its effects in its dependents. Breaking changes become immediately evident as such, given the information is gathered by running examples representative for the library use in the ecosystem. Second, exceptions can be inspected and debugged. This allows the library developer to design a migration script that can be applied when upgrading the library in the dependent. These scripts can then be distributed along with the new version of the library, potentially easing the upgrade of other dependents as well. Instead of readding a temporarily removed method again, such a script could rewrite the call sites of this method either statically or dynamically. Pushing the migration idea further, it might also be possible to design a library upgrade in a dependent as a pull request submitted automatically on the release of a new library version. The pull request could already apply the necessary code transformations on the dependent code, the maintainers of the dependent could review the changes.

4 Related Work

Our work is motivated by the observation that, besides the fact that many open-source project have been co-evolving and depending on each other for years, upgrading a library dependency is still costly.

Kula *et al.* performed a study on over 4 600 GitHub Maven projects, concluding that the majority of the projects have outdated dependencies, revealing security vulnerabilities. [7] Bavota *et al.* studied the evolution of the Apache ecosystem and find that dependency upgrades are mostly performed when many bugs are fixed or a new major version of the dependency provides new features or services. [1] Both studies report the reluctance of developers to upgrade dependencies as it is presumed to be risky and costly.

Automating the upgrade process has been an ongoing subject in the field of API usage research. These approaches attempt to derive the necessary code tranformations on the dependent of an API. Dig *et al.* reported that many API changes can be modelled by reconstructing basic refactorings on the API, which in turn can then be replayed on the code of a dependent. [3, 4, 8] Hora *et al.*, as well as Robilliard *et al.*, inferred additional API properties that support the migration from one version to another one. [6, 9]. All of these approaches have in common that they are applied post-mortem, *i.e.*, after a change in a library has already been published. Our approach tries to make the effects of changes more imminent, with the goal to enable more informed decisions when changing code. Tymchuk showed that timing for feedback matters, and developers appreciate contextualized and live feedback for code quality measures. [11]

Our current implementation of live ecosystem feedback draws a lot of similarities to the more generally applicable Ecosystem Monitoring Framework by SpasojeviÄĞ. [10]

5 Conclusions and Future Work

We present how the integration of co-evolution as a first-class citizen of the IDE can look like, and we discuss its benefits and drawbacks. Exemplified by a tiny case study of a breaking change, we show how live feedback on the effects of a change on ecosystem supports library developers by immediately revealing a change as a breaking change. We propose to further explore the integration of co-evolution with the goal to lower to cost of upgrading a library dependency.

5.1 Future Work

We believe that our approach also has the potential to enable exploration, creation, and validation of necessary migrations on library dependents for library upgrades. In this work we only hint at how the upgrade process might be improved, we intend to further investigate this research direction. Furthermore, the evaluation of the usefulness of our approach needs to be extended by applying it over a longer time within a real world experiment.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assistance" (SNSF project No. 200020-181973, Feb 1, 2019 - Apr 30, 2022).

References

[1] Gabriele Bavota, Gerardo Canfora, Massimiliano D. Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In *2013 IEEE International Conference on Software Maintenance*. 280–289. https://doi.org/10.1109/ICSM.2013.39

[2] A. Brito, L. Xavier, A. Hora, and M. T. Valente. 2018. Why and how Java developers break APIs. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 255–265. https://doi.org/10.1109/SANER.2018.8330214

[3] Danny Dig and Ralph Johnson. 2006. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)* 18, 2 (April 2006), 83–107.

[4] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison Wesley.

[5] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. 2014. A Quantitative Analysis of Developer Information Needs in Software Ecosystems. In *Proceedings of the 2nd Workshop on Ecosystem Architectures (WEA'14)*. 1–6. https://doi.org/10.1145/2642803.2642815

[6] Andre Hora, Anne Etien, Nicolas Anquetil, Stéphane Ducasse, and Marco Túlio Valente. 2014. APIEvolutionMiner: Keeping API Evolution under Control. In *Proceedings of the Software Evolution Week (CSMR-WCRE'14)*. http://rmod.inria.fr/archives/papers/Hora14a-CSMR-WCRE-APIEvolutionMiner.pdf

[7] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2017. Do developers update their library dependencies? *Empirical Software Engineering* (2017), 1–34.

[8] Don Roberts, John Brant, and Ralph E. Johnson. 1997. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)* 3, 4 (1997), 253–263.

[9] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *Software Engineering, IEEE Transactions on* 39, 5 (2013), 613–637. https://doi.org/10.1109/TSE.2012.63

[10] Boris Spasojević. 2016. *Developing Ecosystem-aware Tools*. PhD thesis. University of Bern. http://scg.unibe.ch/archive/phd/spasojevic-phd.pdf

[11] Yuriy Tymchuk. 2017. *Quality-Aware Tooling*. PhD thesis. University of Bern. http://scg.unibe.ch/archive/phd/tymchuk-phd.pdf