Exploring API / Client Co-Evolution

Anna Maria Eilertsen
Department of Informatics
University of Bergen
Norway
anna.eilertsen@uib.no

Anya Helene Bagge
Department of Informatics
University of Bergen
Norway
anya@ii.uib.no

ABSTRACT

Software libraries evolve over time, as do their APIs and the clients that use them. Studying this *co-evolution* of APIs and API clients can give useful insights into both how to manage the co-evolution, and how to design software so that it is more resilient against API changes.

In this paper, we discuss problems and challenges of API and client code co-evolution, and the tools and methods we will need to resolve them.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; *Software configuration management and version control systems*; *Software maintenance tools*;

KEYWORDS

API evolution, co-evolution, repository mining, software evolution, bytecode analysis

ACM Reference Format:
Anna Maria Eilertsen and Anya Helene Bagge. 2018. Exploring API / Client Co-Evolution. In *WAPI'18: WAPI'18: IEEE/ACM 2nd International Workshop on API Usage and Evolution , June 2–4, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3194793.3194799

1 INTRODUCTION

Increasing complexity and expectations of software motivates reuse of external software components when possible. An Application Programming Interface (API) defines how one such component offer functionality to clients [2]. APIs are found in the interface of web services, language workbenches, SDKs and bundles as well as both industrial level and homegrown libraries. APIs can also be internal, and serve as a less penetrable layer between local software modules. APIs abstracts, encapsulates and, ideally, defines an ergonomic way to interact with a module, independent of its internal workings [5].

Good API design is hard [2]. APIs change, and their usage patterns change as well [9]. For clients, updating to a new API version can be costly and off-putting; for API developers it is hard to be certain that you evolve the API in a good way, so that it fits existing or future usage patterns, while causing as little damage to client code as possible. Research and tools on API and client code co-evolution has the potential to save significant development time, prevent bugs, increase overall software quality and decrease the development cost.

One research approach to the co-evolution problem is to empirically analyse API usage data. By building a knowledge base of how APIs are used, and how clients respond to changes, researchers makes it easier to develop tools and methodologies that have real-world impact.

In this paper we give a short overview of background and terminology, and describe and discuss research questions that can aid API and client co-evolution. We lay out our experimental setup and tooling, based on analysing bytecode from public repositories, and discuss how it relates and improves on previous research.

Although we discuss APIs in the scope of *external software library interface in an object-oriented language, like Java*, we hope that this effort may aid not just API/client co-evolution, but also be generalised to aid evolution and migration of other kinds of APIs, like the "APIs" of software languages.

2 APIS & API (CO-)EVOLUTION

Research on API evolution and co-evolution is a fairly new field. It is closely related to software evolution research, and techniques related to software repository mining in the context of software evolution is also quite applicable here. However, in an API context we are looking at software components with independent release cycles [4], i.e. "co-evolving"[1]. We also look at a different "change granularity" than in the context of software evolution. Both will be explained in the following text.

2.1 Basic Terminology

An *API* is the interface through which a software module's functionality is accessible. *Client code* is the external code that uses the API. Client code can also be a library bundle, and can itself deliver an API that other clients consume. Typically, the internal implementation of the functionality will be *encapsulated* and not part of the API, so that it can be changed without affecting clients. In some cases, though, implementation details may still be accessible by or leak through to the client. Library functionality can be accessed by client code in several ways. We use the term *access point* about the exact code-expression of the client-API connection. Typical kinds of API access points are invocations of API methods or instantiating API classes (constructors can also be considered method calls); or through code annotations [4]. Other access may be through fields or constants. In other languages, data structure layouts and constant

[1] Though in some cases, such as the Eclipse eco-system we may have a "release train", with managed co-evolution and synchronised releases, rather than co-evolution "in the wild".

WAPI'18, June 2–4, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5754-8/18/06…$15.00
https://doi.org/10.1145/3194793.3194799

values may also be part of the API; e.g., `structs` and `#defines` in C. Code annotations, by their nature, are commonly used by logging frameworks, while method invocations can easier provide functionality directly to the code. The further discussion will focus on API use through method invocations and inheritance.

2.2 API/Client Co-Evolution

Software evolves, and APIs evolve. When client software must be modified to keep up with API changes, we see co-evolution. We may also co-evolve an API in response to changes in how it is used by clients.

Researching software co-evolution in this context means that we are looking at two separately changing artefacts, but we wish to extract *only, but exactly, the changes to each that are relevant to the other*. For the API that means looking at the publicly available API (technically including documentation and licenses), but not the internal changes to the code. For the client code that means disregarding the regular software maintenance and development, and only consider changes that are performed in response to an API change.

API evolution events can be categorised into *breaking changes* and *non-breaking changes*. Breaking changes includes changes to the API such that client code may break (not build, run erroneously) unless they rewrite their code correspondingly when updating. Non-breaking changes do not require client-side rewriting, although we may want to update the client to take advantage of new and better adapted access points.

The way we specify and write software has a clear impact on the deeper study of (both intended and actual) API use: "Surface" information, such as method names and parameters (the *syntax*) are directly coded in the programming language and easy to extract; but relationships between methods and critical information like required preconditions can often only be encoded in human readable documentation. Thus, deeper analysis of APIs and API use is needed to answer interesting questions about *semantics*, like "how will clients be impacted if I switch between returning `null` and throwing an exception"; something that can have a profound effect on clients, without necessarily being visible to a Java compiler. Thus, we may further divide changes (breaking or non-breaking) according to how they affect the client: *metadata changes* (e.g., renaming a library), *syntactic changes* (e.g, changing a method signature), and various degrees of *semantic changes*. Breaking semantic changes may require significant effort to repair (or even uncover), while simpler changes might be repaired mechanically.

As an example of API evolution, consider the popular unit testing framework JUnit.[2] Its API changed from version 4 to version 5, though code using JUnit 4 can continue to do so. We can also link against JUnit 5, and use its JUnit 4 compatibility layer (a simple change to the build metadata, rather than to the source code). If we upgrade to the new API, the changes we must make are mainly to the metadata (just import a different Java package) or the syntax (e.g., change `@Before` to `@BeforeEach`). However, if we want to take advantage of the new API, we might want to use the new support for parameterised and repeated tests instead of implementing this functionality ourselves; in this case JUnit has adapted its API to suit our testing style better, so that our client code becomes shorter and simpler.

In general, when API developers release a new version of an API that contains breaking changes, updating the client disrupts the development cycle and impose a cost on the project. The client developer may decide not to update (inducing *lag time* [8] which may be correlated with security issues and bugs), or updating and allocating development resources port the code accordingly.

3 EXPLORING APIS & CO-EVOLUTION

Research on API use is inherently empirical, focusing on mining public code repositories like GitHub or SourceForge, and extracting API usage information, potentially along an evolution axis [7, 9, 12–14]. Some researchers publish data sets [7, 12, 13]; other use mining to build tools that can aid in evolving client code [4, 9, 10]. It is unclear to which extent these tools are adopted and how well they work: several have limitations as pointed out by Nguyen et al. [9] and Jezek and Dietrich [4]. Synthetic data sets are useful for benchmarking tools [4], but do not help us in mapping out real API use.

Source code analysis induce challenges, like the need to resolve method bindings [13] against the API in question. Another alternative is analysing code after it has been built and names and method bindings are resolved. For Java, the bytecode is a nearly direct translation of the source code, with all names fully qualified, thus avoiding the binding issues of source code analysis. However, as source-control systems typically only deal with source code, one would need to build each commit to obtain the bytecode, which scales poorly [13] (unless one has access to the results of continuous integration). Although research on already-built artefacts is less common, tools like Clirr[3] can be used to analyse breaking changes in the API of Java libraries [11].

3.1 Working with Bytecode

We believe studying already-built artefacts rather than source code may prove to be the more fruitful approach. In particular, we may avoid issues with resolving bindings and extracting other semantic information as long as this information is available in the built code. In the case of Java, this is mostly true; though we may miss some information such as local variable names (this is optional debug information) and comments. Even so, we could still combine approaches as long as we are able to obtain the source code, e.g., by relating byte code to source code through debug information.

Our *Java Bytecode Fact Extractor* processes Java class files, either single, in bulk, or bundled in `jar` files. The tool is implemented using the ASM [3] bytecode analysis framework, and works by visiting both the declared access points in the class file as well as individual bytecode instructions. Extracted usage information includes all use of API access points, including method calls and field variable access. For fine-grained analysis we can distinguish individual calls, and it is also possible to extract context information, in order to answer questions such as "is the return value checked for `null` afterwards?" and "what other methods are called on the same object?" (limited, of course, by what can be achieved with static analysis).

[2] https://junit.org/junit5/

[3] http://clirr.sourceforge.net

Unlike source code analysis, where access to dependencies are need for name resolution, Java bytecode already uses fully resolved and qualified names. Of course, if we are building the code ourselves, we need the full build environment with all its dependencies; which may be tricky to set up, especially for older code. Fortunately, binary code repositories such as Maven Central[4] provide access to millions of already-built artefacts, in the form of `jar` library files, complete with metadata such as version number and dependency information. Since older versions of a library are kept when new ones are uploaded, it is possible to follow the their authors, and there are typically many versions available for each library, each with their own `jar` file, allowing us to examine how a library evolves over time.

3.2 API Measurement and Data Sets

One way to measure API use in client code, is through the number of invocations of API access points, either at run-time using a profiler, or by using static analysis to find and list API access points per method, class or project [7, 13]. Such access point counting is relatively easy to do, and the results are easily represented. Some literature however, make explicit the more complex relationship between APIs and client code. This is acknowledged in efforts to do API access prediction or API migration tools [4, 9, 14, 15]. Jezek and Dietrich [4] list semantics, quality of service and licensing as aspects of API use that are commonly not captured simply by looking at the API code. Nguyen et al. [9] list common complications in API use patterns, such as requirement in method call order, or method argument processing on client side. The simplified approach of listing method invocations is insufficient for reliably representing such data, and authors describe how they use a graph-based approach to extract "API use skeletons". Lämmel et al. [7] do a similar evaluation, but only represent access points in their data set. They do, however, a large data set, obtained by analysis of 69 APIs used by 1476 projects. In comparison, the data set produced by Sawant and Bacchelli [12] consists of the impressive number of over 20 000 projects, but collects API information only for five APIs.

In our work, we are less interested in counting API use, or extracting specific usage measurements; e.g., answering questions like "which APIs does this project use?" or "which classes call this method?". Rather, we would like to build a data set that we can use to explore things such as method relationships and how arguments and return values are processed by a client; for example, in order to guide further evolution of a library, or to see how clients respond to such evolution. Such as data set may be useful also for practitioners, if they are able to write their own queries against our it. To facilitate this, our tool stores facts as semantic triples[5] in the Graal knowledge base [1], making it possible to query them using the SPARQL[6] querying language or a logic/inference system such as Prolog or Datalog.

As an example, consider the following simplified triples (the full syntax uses URIs) describe "DustBunny" as a subclass of "Rabbit", with a method "getName" which calls Rabbit's "getName":

```
<DustBunny> <extends> <Rabbit> .
<DustBunny/getName> <methodOf> <DustBunny> .
<DustBunny/getName> <calls> <Rabbit/getName> .
<Rabbit/getName> <calls> <StringBuilder/append> .
```

Adding an inference rule for *transitivelyCalls* would allow us to see that *DustBunny/getName transitivelyCalls StringBuilder/append*. In a sense, we store raw facts, then build abstractions on top, rather than run query against the code and dump the answer in a database.

3.3 Change granularity

A significant difference between release versions and source-control versions is the *granularity* of versions changes, and change tracking.

When using projects from source-control systems the evolution data is collected along the axis of "commits". A commit may consist of anything between a small local code change of one or two lines of code, to a larger refactoring with far-reaching edits. As such source-control systems shows the software's rather fine-grained evolution over time. In traditional software evolution research this is a common and useful approach [6], and may in fact be consider rather coarse-grained (compared to looking at the programmer's editing session. However, the usefulness of the same granularity and code base has not been addressed in a API evolution context.

"Change granularity" of single commits are not homogeneous across source-control projects. Some commits may even undo changes made in previous ones, commits may introduce bugs, with later ones fixing them. At the same time, branches in e.g. GitHub is a significant part of the evolution environment. Some modular parts of the program evolution can be split into a new branch, which is later merged into the main branch. This complicates the otherwise linear software development, and is usually not addressed when presenting data collection. Sawant and Bacchelli [13] mention this problem, and choose to only collect data from the master branch.

Public software release versions have coarser change granularity than source-control systems. Analysing software along the axis of public releases, as opposed to commits, may provide us with a data set that is easier to analyse, while still being representative.

It is unclear what level of change granularity is ideal for researching API-client co-evolution. In our setup, we can in principle handle any granularity for which we can obtain build artefacts; this will of course be easier for release versions.

4 FUTURE EXPLORATION & CONCLUSION

4.1 Research Questions

Below we provide a selection of research questions for API-client co-evolution and API use. We hope for a discussion on relevance and usefulness, and welcome suggestions of questions we may address, or which may be of interest to practitioners or researchers.

R1. Does the release of an API version *without client code update* precede an increase in bug reports? (R1–R4 would also require mining bug reports.)

R2. When an API version contains only non-breaking changes, does upgrading the build version of the API *without client code updates* induce bug reports?

R3. Does closing issues/bugs relate to API updates in client code?

R4. Does API upgrade lag time predict an increase in bugs?

R5. Are some APIs used more commonly together, or, to what extent can the presence of one API indicate the need for the other?

[4] https://search.maven.org/
[5] Like RDF, https://www.w3.org/TR/rdf-concepts/#section-triples
[6] https://www.w3.org/TR/rdf-sparql-query/

R6. Can repetitive API use patterns indicate that API developers could extract the pattern into a single API access point?

R7. How different or similar is API use across different languages?

R8. How different or similar is API use across different APIs?

R9. How similar are the API footprints [7] in a random pool of open-source projects compared to Maven projects or industrial projects?

R10. Do some API functionality induce particular use patterns that are not documented in the API specification?

R11. Does it happen that a client use two different versions of the same API?

R12. To what extent do API functionality change without it being reflected in client code edits?

R13. Do clients respond differently to major releases versus minor or patches? How does the lag time differ?

R14. An interesting, but somewhat separate, research question would be laying out the exact benefits, drawbacks and limitations of the different data representations of these data sets.

The list of research questions laid out here are difficult to address, and we do not necessarily aim at answering all of them ourselves. Rather, we consider it important to develop and represent a data set such that they and other questions like them can be answered using it. The quality and abstraction level of information collected and represented in such a data set is vital for which answers it qualifies to answer. One must address how representative the collected code base is; how fine-grained (and correct, in the context of type-resolving method calls) the API use information one collect is; how large is the pool both of clients and of APIs; how homogeneous are the clients and/or the APIs; what change granularity do we use; how do we represent it in the data set (i.e. flattening many source control commits into larger change "chunks" could produce the same effect as looking at public releases). What ontology, i.e., what predicates or relations, do we use in our knowledge base, and how do we translate research questions into representative queries?

We also consider how easily the data set can be represented visually: graph data are more easily consumed visually than database tables.

4.2 Conclusion

In this paper we have given a short overview of the background for our exploration of API co-evolution. In effect, we propose a shift in the relationship between data granularity and change granularity that is commonly seen in these data sets. The data sets should contain more complex information about the API use, including the code context in which they happen, but can be generated from fewer versions is common in source-control systems. This should make the analysing process more scalable, while retaining the same quality of evolution data.

We believe that ontologies and knowledge representation technologies will be useful in building a tool set and data set that can be useful for a variety of purposes. For instance, we already see that our tool prototype may be useful in an educational setting, for exploring students' API use ("do they make good use of provide APIs?", "do they adapt or extend APIs to better suit their needs?"); in fact, the semantic triple example is extracted from student code.

ACKNOWLEDGMENTS

We thank the Ralf Lämmel, Tetiana Yarygina and the reviewers for valuable feedback. This work is supported by the Research Council of Norway under grant number 250683 *(Co-Evo)*, and by a travel grant from the Meltzer Research Fund.

REFERENCES

[1] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, and Clément Sipieter. 2015. Graal: A Toolkit for Query Answering with Existential Rules. In *Rule Technologies: Foundations, Tools, and Applications*, Nick Bassiliades, Georg Gottlob, Fariba Sadri, Adrian Paschke, and Dumitru Roman (Eds.). Springer, Cham, 328–344. https://doi.org/10.1007/978-3-319-21542-6_21

[2] Joshua Bloch. 2006. How to Design a Good API and Why It Matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 506–507. https://doi.org/10.1145/1176617.1176622

[3] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*.

[4] Kamil Jezek and Jens Dietrich. 2017. API Evolution and Compatibility: A Data Corpus and Tool Evaluation. *Journal of Object Technology* 16, 4 (Aug. 2017), 2:1–23. https://doi.org/10.5381/jot.2017.16.4.a2

[5] Ralph E Johnson and Brian Foote. 1988. Designing reusable classes. *Journal of object-oriented programming* 1, 2 (1988), 22–35.

[6] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software: Evolution and Process* 19, 2 (2007), 77–131. https://doi.org/10.1002/smr.344

[7] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. 2011. Large-scale, AST-based API-usage Analysis of Open-source Java Projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11)*. ACM, New York, NY, USA, 1317–1324. https://doi.org/10.1145/1982185.1982471

[8] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, Washington, DC, USA, 70–79. https://doi.org/10.1109/ICSM.2013.18

[9] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A Graph-based Approach to API Usage Adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 302–321. https://doi.org/10.1145/1869459.1869486

[10] Rahul Pandita, Raoul Jetley, Sithu Sudarsan, Timothy Menzies, and Laurie Williams. 2017. TMAP: Discovering relevant API methods through text mining of API documentation. *Journal of Software: Evolution and Process* 29, 12 (2017), e1845. https://doi.org/10.1002/smr.1845

[11] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2014. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 215–224. https://doi.org/10.1109/SCAM.2014.30

[12] Anand Ashok Sawant and Alberto Bacchelli. 2015. A Dataset for API Usage. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 506–509. https://doi.org/10.1109/MSR.2015.75

[13] Anand Ashok Sawant and Alberto Bacchelli. 2017. fine-GRAPE: fine-grained APi usage extractor – an approach and dataset to investigate API usage. *Empirical Software Engineering* 22, 3 (01 Jun 2017), 1348–1371. https://doi.org/10.1007/s10664-016-9444-6

[14] Tao Xie and Jian Pei. 2006. MAPO: Mining API Usages from Open Source Repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*. ACM, New York, NY, USA, 54–57. https://doi.org/10.1145/1137983.1137997

[15] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API Mapping for Language Migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 195–204. https://doi.org/10.1145/1806799.1806831