Non-Atomic Refactoring and Software Sustainability

Titus Winters
Google, Inc
titus@google.com

ABSTRACT

Sustainability is the ability of a project / codebase / organization to react to necessary changes over its expected lifespan. At a large enough scale, or with enough disconnect between dependencies, sustainability comes from application of both technical and non-technical approaches. On the technical side, I advocate for restraint among API providers on making arbitrary changes, and use of non-atomic refactoring techniques when more invasive changes are required; such techniques are employed in many Google projects, and in programming languages like Go and C++, to allow more flexible changes to language standards over time. On the non-technical side, I argue for a clear separation of responsibilities (providers need to do the bulk of the work for the update), as well as a growing need to document acceptable usage of an API, be it a library or programming language. In many languages, there are very few changes to an API that are provably safe without this idea: just because a user's code currently works does not mean that it is supported and can be expected to continue to work indefinitely under maintenance. Taken together, these two approaches form what I believe to be a minimum set of requirements when approaching software sustainability.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; *Software maintenance tools*;

KEYWORDS

Software engineering, refactoring, libraries

ACM Reference Format:
Titus Winters. 2018. Non-Atomic Refactoring and Software Sustainability. 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/10.1145/3194793.3194794

1 INTRODUCTION

What is the difference between programming and software engineering? These are nebulous concepts and thus there are many possible answers, but my favorite definition is this: Software engineering is programming integrated over time. All of the hard parts of engineering come from dealing with time: compatibility over time, dealing with changes to underlying infrastructure and dependencies, and working with legacy code or data. Fundamentally, it is a different task to produce a programming solution to a problem (that solves the current instances of the problem) vs. an engineering solution (that solves current instances, future instances that we can predict, and - through flexibility - allows updates to solve future instances we may not be able to predict).

I have described software sustainability for years in very similar terms: your code is sustainable if you are capable of updating it to respond to necessary change for the expected lifespan of the project. For programming projects (quick hacks, school assignments, one-off academic analyses), it is likely that there are no necessary changes - there probably won't be a new language version published in the days or weeks that project is alive. For longer term projects, especially successful OSS or industry projects, it is entirely possible that the code in question needs to live for years or decades. It should come as no surprise there are differences in approach between these styles of projects.

From what I've seen, there is only limited awareness of sustainability issues on the part of API providers or consumers. Few providers are clear about what to expect from their APIs, and thus consumers assume that anything that works now will work indefinitely. Speaking as someone that has been involved in hundreds of thousands of changes, I can say that this approach only works if we stop everything and commit to stability over all else. However, in a world where programming languages, infrastructure libraries, and end-user code change continuously, it is foolish in the extreme to believe in perfect stability. Instead, what we need is a conceptual approach that makes the responsibilities of both providers and consumers clear, at least in the case where code needs to work in an engineering sense - over time. (Obviously if the problem lifetime is short and no dependencies are going to change, these issues become non-existent.)

This paper will demonstrate the applicability of these thoughts in three distinct situations: internal Google code, new Google-backed OSS projects providing explicit compatibility promises, and new approaches for the C++ standard.

2 THE GOOGLE MONOREPO

In the past 5 years, we have committed nearly 1 million commits to the Google codebase in service of large-scale refactoring changes - broad but shallow changes that require many updates but little or no semantic change. A single such Large Scale Change (LSC) tends to vary from 50 to

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

ACM ISBN 978-1-4503-5754-8/18/06.
https://doi.org/10.1145/3194793.3194794

10K commits. We execute roughly 20 LSCs per week across languages.

Importantly, we require that the team instigating a change needs to make the change. The developers involved in deciding that an API needs to be improved generally know far more about the details than casual users of that API. It is far more costly for non-expert developers to understand and evaluate a change. This turns out to be a pretty good guiding principle. Given the growth factors on dependencies, it is clear that transitive dependencies grow super-linearly. If project dependencies continue to require non-zero update work, eventually all available effort will be consumed just to keep up with churn caused by underlying dependencies. When instead the teams that are triggering the change are responsible for executing that change, the total cost is lower and projects are free to focus on the content of their project rather than the design details of every dependency. This also forces infrastructure teams toward stability or, at minimum, toward change that is feasible to automate.

On the flip side, consistent simple code is easier to refactor than unusual clever code. When automating a transformation, the more consistent the usage, the easier it is to generate correct transformations. This exerts a significant force on refactoring and API providers: if API providers are able to guide their users into consistent and simple uses, then the providers maintain greater flexibility if (when?) refactoring becomes necessary. As a direct result, extensive effort is spent on training, documentation, style guides, and static analysis to ensure that Google developers understand how to simply and consistently utilize programming languages and common APIs in the Google monorepo.

It unusually easy for Google to get that training to work because we have complete situational awareness: we control the build system, the languages / toolchains, and the indexing of code for searchability. We therefore understand how our code is used in the vast majority of cases.

Because of the scale of our repository, we almost exclusively use non-atomic refactoring techniques when executing large changes. The general nature of "non-atomic" refactoring is to execute a wide-ranging change as a series of largely-unsequenced steps. Most commonly: a new API is introduced, usage of the old API is changed piecemeal to the new form, and when there is no more use of the old API, it is deleted. This keeps the codebase building and functioning properly between every individual change. This approach is a practical necessity for us because of the limit on how many files can be included in a single change: it is often impossible to sync a single change fast enough to update more than a few thousand files at once (consider the expected amount of time to resolve merge conflicts and updates compared to the expected amount of time between some file being updated). It is also extra difficult to roll back a single huge change if anything goes wrong. Non-atomic refactoring breaks these down into simpler independent pieces. However, there are other reasons to rely on non-atomic techniques - if you lack perfect visibility and need to coordinate with other developers, or the code is visible but resides in disjoint repositories. Avoiding a breakage in those scenarios requires the same non-atomic approach. Both atomic and non-atomic approaches have value, but in software engineering that crosses project boundaries or involves large codebases, atomic approaches fail quickly.

Our scale was the guiding force behind us moving to policies of non-atomic refactoring, pushing toward simplicity and consistency, and forcing infrastructure teams to execute their own changes to user code. However, as we've refined those polices and techniques we have seen them become increasingly applicable in other domains - I believe this division of responsibilities and overall approach is applicable in most scenarios that require significant compatibility over time.

3 ABSEIL

Beginning in 2016, Google began a massive refactoring project that resulted in the open-source release of many of our C++ utility libraries. Released in 2017 [8], this codebase is known as Abseil [3]. One of the primary mandates for Abseil is to allow code sharing between our monorepo and the rest of the world, without reducing internal velocity or our ability to make changes to our common libraries. At the same time, we want to be respectful of the users of public Abseil and provide a stable target. This tension between velocity and stability is in many respects the core problem in dependency management and software engineering: when we move beyond a single definition of HEAD (via a monorepo or some other concept) or we lose perfect visibility in how our APIs are used, refactoring becomes manifestly more difficult.

Several observations in software engineering need to be accepted as axiomatic before discussing this in more depth:

- Dependency graphs are growing deeper and more complex [4].
- Hyrum's Law applies [9] - at scale, basically everything has the potential to be a breaking change for someone.
- When "breaking" changes are made in a lower-level library, it isn't just the direct dependencies of that library that are affected - all transitive dependencies are affected.
- Diamond dependencies [5] grow more common as the size of dependency graphs grows.
- Indirect users are the ones that are affected by diamond dependencies.
- Even if a break is relatively easy to fix, indirect users are unlikely to know anything about the library that is the source of the break OR the API where the breakage manifests.

There are, to the best of my knowledge, three strategies that API providers can take to avoid causing widespread damage for the users of those interfaces:

- Nothing ever changes. Over time this is unlikely to work in most situations - A few interfaces built upon the oldest, most fundamental, and most stable interfaces (libc, POSIX, etc) may be able to avoid change. Anything involving a computer network, untrusted data, complex dependencies, or more modern programming

languages is likely to require some sort of update over a long enough time frame.
- Release all dependencies as a single entity. This is sort of the Linux Distribution model of dependency management: build everything into one release that works together and allow outside developers to depend on that as a whole. Pieces within that entity can theoretically be updated so long as no breaking changes are introduced, but most users will likely not update until a new release of the whole entity is produced.
- Live at Head. All libraries are released against the most-current version of all of their (transitive) dependencies. Users are encouraged to use those most-current versions and to update frequently - syncing to current versions of your dependencies should be as easy as updating to HEAD in your repo. Practically speaking, this requires that library providers make changes carefully, unit tests are ubiquitous, and users understand what changes are and are not expected from their underlying dependencies.

This last idea is worth unpacking: a user of an API must be aware of what to expect over time. If the interface they are relying upon has issued a promise to never change, or the duration of the usage is brief (say, building a proof of concept or assignment or quick hack that will be discarded quickly), anything that works is "right." This is, in my terminology, merely a "programming dependency" - time doesn't matter.

Conversely, if this usage is expected to continue and the underlying API is not known to be indefinitely stable, a user of an API should attempt to understand what usage is allowable. Using a C++ example: it is likely that new releases of a library will introduce new APIs into the namespace for that library, and therefore a user adding symbols into that namespace directly may conflict with future upgrades. APIs may change by adding overload sets or default parameters; taking the address of a function or depending on the specific type signature from the underlying API will become a build break in the face of such changes. Relying on metaprogramming techniques to inspect the type signature of underlying APIs is similarly brittle in the face of changes to the underlying library.

To combat this, Abseil makes the following promise: we will explicitly enumerate to the best of our ability the things that users may not do [2]. If those rules are upheld, our consistent stream of minor releases won't break anybody. We further aim to provide static analysis checks to enforce this good usage whenever possible. In the rare instance that we are forced to make a change that can break well-behaved users, we will provide a tool to perform the upgrade. In the diamond-dependency scenario, the goal of such a tool is to allow a developer who is not expert in any of the underlying dependencies to perform the upgrade with near-zero knowledge of the code being modified. In the presence of good unit tests, such tooling provides (in theory) a low-cost solution to diamond dependencies and easier upgradability. This makes live-at-head possible, even in a non-monorepo world where we lack perfect visibility and control of users of our code.

It is important to recognize the importance and difficulty of adapting to time: A project that has no compatibility-over-time requirements is fundamentally different from one that needs to be maintained and adapt over time. We currently talk about correct programs, but that is simplistic and misleading: there are many programs that happen to be correct now but will break when their dependencies change. Developers must be aware of the expected lifespan of their outputs and of the difference between happens to work now and will reliably work for the foreseeable future. The latter is fundamentally harder and requires an understanding of the rights and responsibilities of a user of any given API - be it a library like Abseil or something more fundamental like the language version itself.

4 VELOCITY AND STABILITY IN THE C++ STANDARD

For the past year, Ive been pushing proposal P0684 [6] [7] through the C++ standards committee. This proposal is pushing the C++ language itself through the process of determining whether this is a language of exciting new features (fast evolution, some inevitable missteps, ability to change and fix mistakes over time) or a language of great stability (binary compatibility for 10+ years, API compatibility for longer timeframes, no existing code will ever have to change as a result of an update in the language). I argue that we cannot do a perfect job of both. Although the discussion is ongoing, preliminary results suggest that the C++ Standards Committee is likely to slightly prioritize velocity over stability.

Implicit in this prioritization is the notion that the release of a new language version is not an atomic operation that is expected to occur with zero effort. For comparison: even upgrading compiler versions between allegedly compatible language versions is an operation that requires some effort. Upgrading language versions is likely to require at least that much effort. Also note that any project or codebase that is going to upgrade to a new language release likely has someone performing that upgrade as a specific, intentional task. Going forward, the current proposal is that we assume two behaviors:

Compiler implementers will implement a warning mode, available in the current C++ version, to statically identify changes in behavior or APIs for the new version. Developers who are performing the language upgrade will first update to a current version of their compiler, enable those warnings, evaluate the safety of enabling the new language version, and then flip the switch.

This does not grant the committee complete freedom to make arbitrary changes. It is still highly preferable for changes to be made in a backward-compatible fashion. If that isn't possible, the best designs are those that allow explicit textual changes to the current version such that no instances of the old form remain when the upgrade is performed.

For example: In C++20 we could decide to make the assignment/initialization + conditional ill-formed, codifying the existing common warnings and instead relying on new if+initializer syntax from C++17. That is:

```
if (int i = Foo())
```

would become an error in favor of the new syntax:

```
if (int i; i = Foo())
```

or without declaration:

```
if (i = Foo())
```

would require the existing solution:

```
if ((i = Foo()))
```

This syntax is detectable, in that we can clearly issue a warning for it (we have done so for years in most/all compilers). This is opt-in, given that every existing instance can be converted to one of the two alternate syntaxes with no behavior change, in C++17 mode. Whomever is responsible for performing the upgrade to C++20 would use the compiler generated warning to identify affected cases and upgrade them appropriately - no instances of the old form would remain when C++20 mode is finally enabled.

By preferring that changes require static detection by previous-version compilers, the language is setting itself up to enable tool-assisted upgrades. By further preferring changes that can be opted-out in the previous language version, the language itself is acknowledging the necessity and preference for non-atomic refactoring patterns. The if+initializer example above works because we introduced the new syntax in C++17 - doing both operations (introducing a new API and removing the old) in the same step is far harder to adapt to or to use safely.

However, once again this approach is predicated on the idea that the language is able to make at least certain classes of changes safely. Introducing new APIs to the standard library is implicitly a breaking change if users have been ill-behaved in certain ways (adding APIs of the same name into namespace std, for example). Practically speaking, there are numerous operations at the standards level that the committee regards as safe, even though usages exist that would be broken in the face of that change. Once again we see that the answer here is clear communication about acceptable usage. Although the standard is far more accepting and offers greater stability promises than Abseil, there is significant overlap between what the standard practically disallows and what Abseil explicitly disallows. It is likely that the standard committee will vote on official endorsement of a similar (but shorter) list of compatibility requirements early in 2018.

5 SUMMARY

Although the examples above are predicated on the specifics of C++, I believe the underlying concepts are equally applicable across all languages. When maintaining code in a large and distributed fashion, a number of new behaviors become clear:

- If it is feasible to promise complete stability, that is (of course) preferable. Practically speaking, it is unlikely that perfect stability can be promised indefinitely - over a long enough time horizon, it is likely that everything will require critical consideration and change[1] On a long enough time-horizon, it is best if we are *able* to change everything - hopefully we do not have to.
- Non-atomic refactoring patterns are easier to adapt to and are applicable in a wide variety of contexts.
- Clear enumeration of acceptable usage should be provided - what types of compatibility does your public API promise?

I believe there is a rich and untapped vein of research surrounding non-atomic refactoring, dependency compatibility, live-at-head, and dependency management. There is a growing awareness that semantic versioning [1] is at best misleading, partly because of its focus on direct consumers of an API and blindness to the practical effects of networks of dependencies. It will take a concerted effort from both practitioners and theoreticians to get us to a world that more accurately understands how to deal with change over time.

REFERENCES

[1] 2010. Semantic Versioning: Technical Whitepaper. (2010). https://www.osgi.org/wp-content/uploads/SemanticVersioning.pdf

[2] abseil.io. 2017. Abseil Compatibility Guidelines. (Sept. 2017). Retrieved February 2, 2018 from https://abseil.io/about/compatibility

[3] abseil.io. 2017. abseil.io. (Sept. 2017). Retrieved February 2, 2018 from http://abseil.io/

[4] A. Decan, T. Mens, M. Claes, and P. Grosjean. 2016. When GitHub Meets CRAN: An Analysis of Inter-Repository Package Dependency Problems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 493–504. https://doi.org/10.1109/SANER.2016.12

[5] Alan Mycroft Mark Florisson. 2015. Towards a Theory of Packages. (2015). Retrieved February 2, 2018 from https://www.cl.cam.ac.uk/~mbf24/packages.pdf

[6] Daveed Vandevoorde Beman Dawes Michael Wong Howard Hinnant Titus Winters, Bjarne Stroustrup. 2017. *C++ Stability, Velocity, and Deployment Plans*. Technical Report P0684r0. ISO C++ Standardization Committee. http://wg21.link/P0684r0

[7] Titus Winters. 2017. *C++ Stability, Velocity, and Deployment Plans*. Technical Report P0684r1. ISO C++ Standardization Committee. http://wg21.link/P0684r1

[8] Titus Winters. 2017. CppCon Keynote: C++ as a Live At Head Language. Video. (26 Sept. 2017). Retrieved February 2, 2018 from https://www.youtube.com/watch?v=tISy7EJQPzI

[9] Hyrum Wright. 2014. Hyrum's Law. (2014). Retrieved February 2, 2018 from http://hyrumslaw.com/

[1] Consider the recent impact of Spectre and Meltdown.