Searching for Build Debt: Experiences Managing Technical Debt at Google

J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali
Google, Inc.
Mountain View, CA
{*jdm,gridman,ralucas,bhansali*}*@google.com*

***Abstract*—With a large and rapidly changing codebase, Google software engineers are constantly paying interest on various forms of technical debt. Google engineers also make efforts to pay down that debt, whether through special Fixit days, or via dedicated teams, variously known as janitors, cultivators, or demolition experts. We describe several related efforts to measure and pay down technical debt found in Google's BUILD files and associated dead code. We address debt found in dependency specifications, unbuildable targets, and unnecessary command line flags. These efforts often expose other forms of technical debt that must first be managed.**

***Keywords*-build system; technical debt; monolithic codebase;**

I. Introduction

Google's source code is big (hundreds of millions LOC [1]) and for the most part monolithic. This is both an advantage and a disadvantage for software development. One of the biggest advantages is that it allows a uniform development style to be adopted and enforced across the company. For example, there are written, language-specific style guides that all engineers follow, a single multi-language build system to build code for all projects, a single repository for the source code, a single continuous testing infrastructure that runs all unit tests, and another single repository for the results of every build and test run. Software tool writers have access to and can run analysis on the entire source code and everyone uses the same tools for code reviews and a single index to search the codebase.

On the other hand, the large, monolithic size of the codebase makes it easy to introduce technical debt and very hard to recover from it. For example, consider the scenario where someone introduces a low level API. Some time later, the author realizes that there are flaws in the original design, and wishes to deprecate the API in favor of a better design. But by then, there are already hundreds of projects in our codebase that have taken a dependency (directly or indirectly) on this API. In a small codebase, if the change is mechanical, it could be done easily using commonly available refactoring support in IDEs, e.g., Eclipse. But for large scale changes that span millions of lines of code, this can be very challenging. We have found that even simple changes like renaming a class or moving it from one package to another can take several days or weeks when operating at this scale. The problem is not just the size of the codebase but the rate at which the code changes (on average over a dozen new commits happen every minute [2]).

In this paper, we present the efforts of our team at controlling and repaying the technical debt in one part of our codebase, the build system. Our approach is guided by the following principles:

- *Automation.* Use automated techniques to analyze and (where possible) fix issues that contribute to our most egregious technical debt. Several teams are working on tools to make large scale changes easier. Our uniform development style makes it feasible to do this at scale.
- *Make it easy to do the right thing.* Many times technical debt is incurred because people are not aware of it. If we can analyze changes that developers are about to make as part of their normal workflow (during editing, browsing, or code review), we can prevent certain kinds of debt. The size of the codebase and the rate at which it changes makes this a non-trivial problem.
- *Make it hard to do the wrong thing.* This is similar to the above point, but with a greater emphasis on introducing stricter checks on the kinds of actions developers can do. For example, prevent people from taking a dependency on code that is not ready for prime time. Another example is to build stricter checks into the compilation process and make them compile time warnings or errors.

Our team is part of a common Developer Infrastructure group and does not own any of the application software or libraries that the various product teams build. As such, our focus is on finding and fixing technical debt that is domain independent and cuts across product boundaries. In this paper, we describe one such type of technical debt that we call *Build Debt* that has accumulated in Google's build specifications. We explain the notion of Build Debt, show why it is a painful debt for us, describe some of our ongoing efforts to reduce this debt, and results of some of our recent successes.

II. Google's Build System Debt

The specifications for building software at Google are encapsulated in BUILD files. BUILD files define modules of code (either at the library or binary level), list the source files and dependent libraries the module uses, and include additional metadata about building the project [3], [4]. BUILD files are for the most part manually maintained, and this lack of automation can be a particular pain point for engineers, requiring non-trival developer effort [5]. Among other things, these files specify the dependencies between different libraries or other software components, and over time these specifications can diverge from the actual dependencies needed to build, test, and execute our software. Technical debt accumulates unless engineers are diligent to keep the source code and the dependencies, or *deps*, of their build targets synchronized. Along with this build *dependency debt* associated with active, buildable targets, abandoned targets are another form of debt that tends to accumulate. At the extreme, we call targets that have not successfully built for several months *zombie targets.*

The original Google build system was completely open with no control over target visibility. This allowed any project to depend on the internal details of another project, and occasionally led to unwanted coupling between projects. This coupling creates technical debt when the lower-level project did not intend to expose these internal details, and now faces higher costs for any future modifications where encapsulation has been violated. Because dependencies are specified in only one direction, the project being depended on would only find out they had broken someone else when they received a complaint. At that point the technical debt would need to be repaid in order for both teams to move forward.

Although a feature to restrict target visibility was added to the build system to give projects control, it saw little use. New projects were in the same situation as legacy projects, until we tackled the problem of build *visibility debt.* Visibility debt is the cost of back-fitting visibility rules onto the existing build specifications, and re-establishing appropriate encapsulation. We determined that the first step to managing visibility debt was to stop the bleeding by automatically preventing new projects from accumulating it.

Dead code forms another type of technical debt that can confuse engineers looking for working APIs. We also discuss *dead flags* and a recent fixit designed to reduce some of this debt.

III. Dependency Debt

Dependency debt causes two types of pain for engineers. The first is slowdown of the build and test systems [6] due to extra work performed building *over-declared* dependencies, which are completely unneeded, or *underutilized* dependencies which are mostly unneeded. The second is brittleness of a project's build due to *under-declared* direct dependencies.

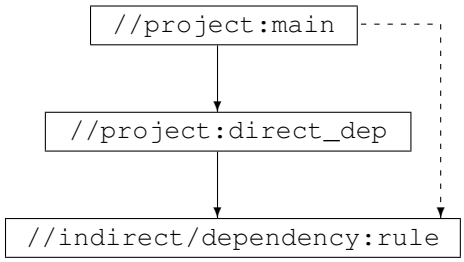

Figure 1. Target dependencies

Changes somewhere in the transitive closure of an under-declared target's dependences can cause the loss of some needed, but unspecified, dependency, resulting in a broken build. Brittleness pain is felt by both the project with the under-declared dependencies, and the projects on which they depend.

An example based on two BUILD files shows how this can happen. The first file consists of the specification to create a single C++ library named 'rule.'

```
/indirect/dependency/BUILD:
cc_library(name = 'rule',
           srcs = ['rule.h', 'rule.cc'])
```

The second file contains another library, and a binary target named 'main.' In addition to the declared dependencies shown in the `deps` attributes of these targets, `main.cc` and `direct_dep.h` also directly include `rule.h`.

```
/project/BUILD:
cc_binary(name = 'main',
          srcs = ['main.cc'],
          deps = [':direct_dep'])

cc_library(name = 'direct_dep',
  srcs = ['direct_dep.h', 'deps.cc'],
  deps = ['//indirect/dependency:rule'])
```

As shown graphically in Figure 1, build target `//project:main` directly depends on `//project:direct_dep`, which in turn directly depends on `//indirect/dependency:rule`. The solid lines represents a direct, declared, dependency between two targets. The dashed line shows an under-declared source code dependency from `//project:main` to `//indirect/dependency:rule`. Because transitive dependencies are allowed, the build works fine until someone changes the code for `//project:direct_dep` to no longer use `//indirect/dependency:rule`, and removes the unneeded dependency. This causes `//project:main` to break, since it no longer has access to a required dependency.

Generally speaking, under-declared dependencies in upstream libraries can cause breakages when lower level li-

braries remove dependencies. This can occur several levels away, across project boundaries, particularly given Google's single codebase. Teams may consider dependencies as 'internal,' but this can't currently be expressed in the build system.

Both of these pain points are felt widely across Google as each project depends on code from many others. Teams are not always aware that they are imposing the interest costs of their dependency debt on both their users and the teams on which their software depends. Core library teams in particular do not like to break their users. Under-declared dependencies in those user's build targets, though, make refactoring the internal dependencies of these libraries very painful due to the possibility of widespread build breakages. In general, the existence of under-declared dependencies makes safe removal of over-declared dependencies difficult.

A. Addressing Under- and Over-Declared Dependencies

One potential solution for addressing dependency debt would be to hold a global fixit day where engineers put aside their normal work and focus on fixing the build rules for their projects. This solution has several disadvantages. It is not automated, potential breakages make it hard for engineers to do the right thing, and easy for them to do the wrong thing. Without some way to prevent new debt from being incurred, whether via awareness, education, or enforcement, dependency debt will just recur in the future.

Instead, we have developed tools to partially automate the solution. These tools require language specific knowledge of the dependencies as expressed by the source code. Our first step was to find all under-declared dependencies, so that we could add these missing direct deps to all Google's BUILD files. We began with Java, leveraging the *javac* compiler to tell the build system the classpath element from which each referenced class is loaded. We also distinguish classpath elements that correspond to direct dependencies, as well as the target that created each element. We then generate a warning or error each time the source code references a class from an indirect, transitive dependency, including the name of that dependency.

This approach works for all Java rules that successfully build and generates a list of missing direct dependencies for each rule. We are currently paying down the principal balance by adding all these missing direct dependencies. We also created a target attribute to tell the build system to generate an error if it sees a missing direct dependency in the future, to make it hard to do the wrong thing.

The next step will be to find all the over-declared dependencies, further leveraging javac. Once under-declared dependencies are disallowed, the unneeded dependencies can be automatically and safely removed. The build system can then generate an error if it finds an unneeded dependency, effectively preventing this form of technical debt from recurring.

B. Underutilized Dependencies

A build target may contain lots of code that many of its clients don't need. A client's build is slower than necessary when these dependencies are poorly utilized. This type of dependency build debt causes engineers the same pain as over-declared dependencies. Both result in a transitive dependency closure for a build target that is larger than the strict minimum required by that target. However, while over-declared dependencies can be handled by rewriting the BUILD files alone, increasing utilization may require both code and BUILD file refactoring. The solution usually involves repartitioning large, underutilized libraries and their build targets. There is less opportunity for a fully automated solution, since engineers need to make encapsulation decisions when breaking up large libraries. We are instead working on discovery tools that will help these engineers find and understand low dependency utilization of their targets.

Figure 2 shows a screenshot of a dependency refactoring assistant, Clipper, currently under active development. There has been a lot of work around software visualization frameworks, see [7], [8] for quality metrics, or [9] for build performance in particular. Clipper attempts to fill the gap between the visualization tools and IDEs by giving engineers refactoring guidance. While Clipper can serve as a dependency exploration tool to look at the structure of the dependency graph, an engineer can use it as a refactoring advisor to suggest specific dependencies that would be good cleanup candidates.

Given the large number of dependencies in the transitive closure of an average target in Google's build system, it can be challenging to even decide where to begin. Clipper makes suggestions by ranking dependencies in terms of high cost and low removal effort. Factors contributing to the dependency cost are the number of symbols defined by the target, the cumulative number of symbols in the transitive closure of target's dependencies, and the utilization of both kinds of symbols. Factors contributing to the difficulty of refactoring are the dependency proximity or depth counted as the number of hops between the target and its dependency and dependency interconnectedness or density counted as the total number of paths leading to the dependency.

By using Clipper to highlight the low hanging fruit we hope to encourage engineers to start cleaning up their projects, which will begin to pay down the dependency debt and allow us to improve the tool as we learn from our early adopters. Though our work is still in the very early stages, the feedback from the initial beta testers is encouraging.

IV. Zombie Targets

In a system that changes as much as Google's (a large percentage of files change every month [2]), some things are bound to break. If they are important, broken targets and tests are quickly fixed. However, older, sometimes

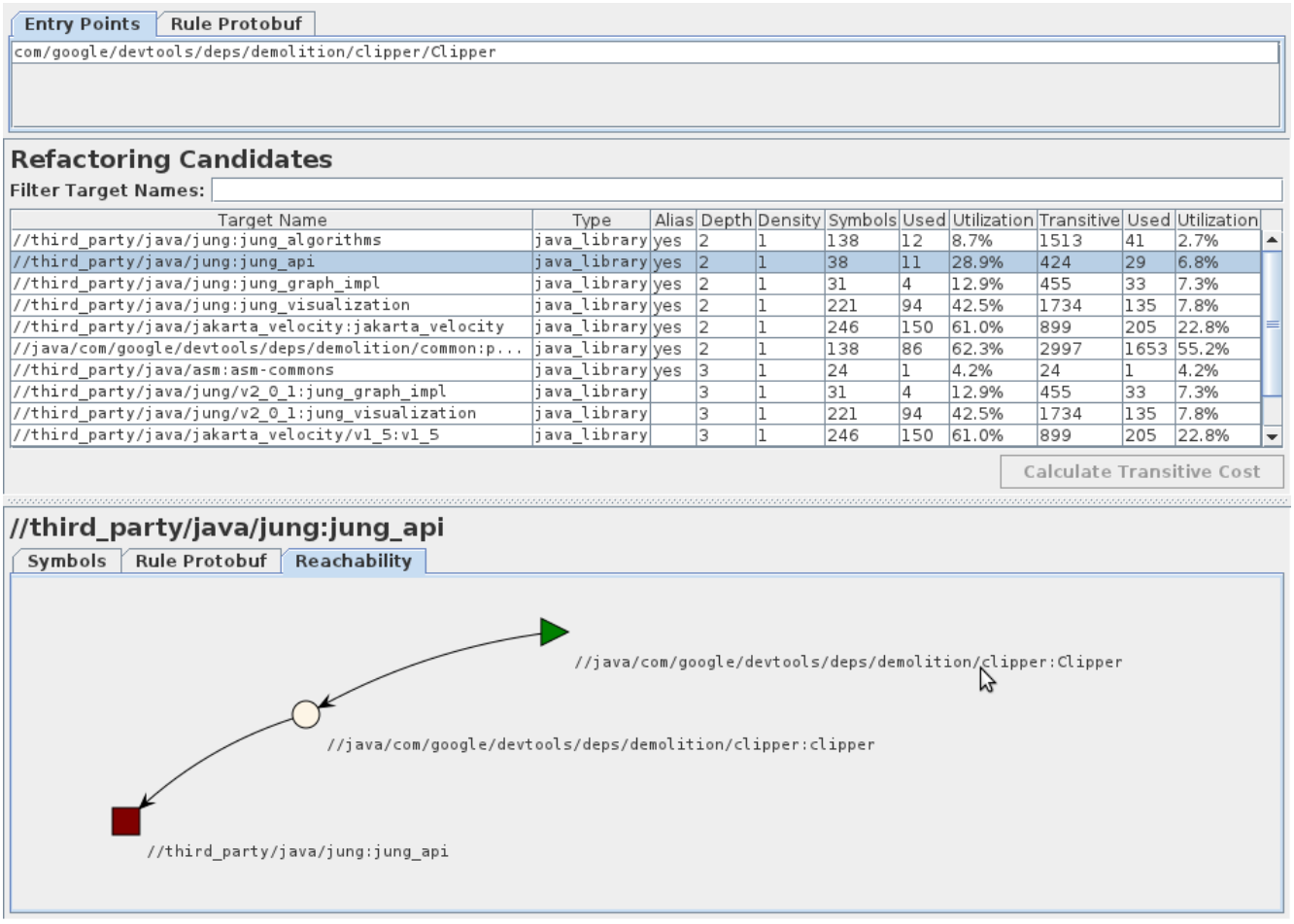

Figure 2. Clipper prototype

abandoned code and the build targets that compile it can slip under the radar. Because Google's codebase is monolithic, these dead targets are visible to everyone. That can cause pain for anyone attempting a global refactoring or even just looking for existing functionality to reuse. Depending on existing code helps projects move faster, but only if an engineer does not need to repair existing build breakages to get the unmaintained code running.

This debt adds interest cost to every refactoring that affects the dead code, as engineers waste time modifying unused, and perhaps unusable, libraries. Broken targets add to the burden of testing a global change. There is also a cognitive cost to understanding a codebase where some portions are effectively unusable.

Repaying this debt involves first categorizing broken targets as either transient breakages or long-term zombies. The results of every build and test run at Google are stored, indexed, and queryable. We use this query capability to determine the last time that each target was successfully built, as well as the last time it was attempted to be built. Our automation updates this data nightly, and generates a list of the day's zombie targets that existed at head in the source control system at that time. During frequent build system upgrades, the build team attempts to build every target in every BUILD file in order to validate the new features, so these data are even available for completely abandoned projects. After internal discussions, if all build attempts have failed for at least 90 days, a target can be officially declared 'dead.' Until then, breakages are assumed to be transient and left to their owners, if any, to handle.

After 90 days, we can 'terminate' a zombie target by deleting its definition from the BUILD file, along with any source files that are only reachable through that target. All code changes, including these deletions, must be reviewed by the project's owner of record before submission, providing a final check that prevents the automation from running amok and wiping out large swaths of Google's live code.

V. Visibility Debt

To eliminate unwanted project coupling, where one project depends on the internal implementation details of

another, we decided to change the existing default visibility of all targets from public to private. When the visibility feature was originally added to Google's build system a few years ago, no target had a visibility specification, so it made sense to consider the default, unspecified behavior as *publicly visible*, effectively making the new feature opt-in. Visibility debt is, unfortunately, invisible to a project until after it becomes a problem. Google's culture meant that locking down code wasn't immediately seen as necessary; teams only realized that explicitly stating visibility was useful when it was too late – when clients were unexpectedly found, and changes either required time-consuming modification to previously unknown clients, or couldn't be made at all. The idea of controlling visibility was also somewhat controversial within Google, as we discovered.

We reasoned that if the default was changed to private (or opt-out) visibility instead of public (or opt-in), awareness, use and the benefits of this feature would all increase. At issue were the tens of thousands of BUILD files with no visibility specification. We devised a plan to mark these files with a special `legacy_public` visibility and then change the default to private in the build system. The idea was to encourage engineers creating new BUILD files to consider whether they wanted other engineers to be able to depend on their new code immediately. In order to use a private target, potential clients need to communicate with the target's owners to agree on whether the API should be shared, and under what conditions. Teams need to retain the ability to determine what constitutes the stable, supported API of their project, and what constitutes private implementation details.

The initial announcement of the upcoming change was greeted with a fair amount of criticism by engineers. Some were concerned that visibility was 'un-Googley,' that it would make sharing more difficult and would generate a large number of code clones when teams refused to allow others to depend on their code. Others worried widespread build breakages would result from the change. The breakages were a possibility, because at first we had asked teams to fix their own BUILD files, with a hard deadline. The feedback we received convinced us to reconsider our plan and make the necessary changes ourselves, adding `legacy_public` default visibility to all existing BUILD files. After verifying the vast majority of targets would build correctly, we were able to change the default with little drama.

We successfully made this transition in June 2011 and as of mid-February 2012, the default visibility of the 2000 most recently created BUILD files breaks down as shown in Table I. Nearly half of the new files start out publicly visible despite the new default, a reminder of Googler's attitude toward openness.

These results show that Google engineers are explicitly overriding the default private visibility with some appropriate value in over 80% of new BUILD files. Since the default was changed, we have seen no evidence of widespread code copying, nor heard any complaints of teams refusing to allow other engineers to take dependencies on their code. Lesson learned: any change will always be opposed by someone and in a large organization it is important to have an independent group that is empowered to make decisions based on a global cost-benefit analysis.

Table I
PACKAGE VISIBILITIES OF THE LATEST 2000 NEW BUILD FILES

none specified (`private` by default)	16%
`public`	40%
selectively visible	32%
`private`	9%
`legacy_public`[1]	3%

VI. DEAD FLAGS

Google developed its own command-line parsing utilities, along with custom mechanisms for defining the set of recognized command-line flags for libraries and binaries. Core libraries, striving to be as reusable and customizable as possible, define many more flags than the regular developer will ever use or know of. Some flags control test-only behaviors, such as the address of a dummy backend vs. the real production backend. Others control experimental code, or protect the addition/removal of such code. Consequently, there are now more than half a million command-line flags defined in Google's code base across C++, Java, and Python projects. The question is, how many of these are still useful, and how many can be declared dead and replaced with constants?

The technical debt incurred by dead flags is perhaps the hardest to quantify. They won't slow down execution or increase the source/binary size enough to be noticed. However, in many cases they guard dead code, which in turn depends on other dead code which gets linked into the final binary unnecessarily. They also complicate refactorings and make the code harder to understand. To gather more insight into the possible solutions, we have recently co-organized a Dead Flag Fixit.

The goals for the fixit were twofold. First, we wanted to estimate the number of dead flags without expensive static/dynamic analyses. We developed a simple analysis pipeline to identify flags that have always been set to the same value. We looked at the command lines of all the binaries running in production for the past year and some of the locally-invoked binaries (i.e., binaries running on the developers' workstations), and aggregated these data sources. We ended up with 150,000 possible candidates, a surprisingly high number of constant-valued flags. As a

[1]Intended for existing files only, this is sometimes blindly copied into new BUILD files – a new form of build debt.

side-effect, the candidate set also includes flags from dead projects which are still being built and tested but no longer run in production (otherwise missed by the zombie target effort). Second, we wanted to observe how dead flags are typically removed, as their refactoring is non-trivial.

During the Fixit, around 6,000 flags were evaluated and about 60% of them were marked as 'not dead.' Engineers removed 2,300 flags from the codebase and also deleted 272,000 lines of code as a result. We collected valuable data, both for refining our definition of a dead flag and for suggesting automated code fixes. The high ratio of deleted lines of code per dead flag motivates a refined taint-like analysis to look for flags that influence control flow, as replacing them with constant values would make some code paths unreachable. Flags that are passed verbatim to underlying libraries, such as server addresses, port numbers, etc., cannot be removed due to the nature of their data. We will next try a simple constant folding algorithm to quantify the amount of dead code per constant-valued flag. In summary, we seek automation for detecting and ranking the offending flags; the ultimate decision of deprecating and killing a flag still belongs to the developer. We believe dead flags are the coarsest level of dead code identification, and perhaps the one most readily accessible.

VII. Uncovering Additional Debt

Each of the above efforts also uncovered additional technical debt in the build system itself. Adding thousands of missing dependencies is a task best automated, and in principle, that should be easy. However, BUILD files can be automatically edited only if they meet certain syntactic criteria. When originally deploying the current build system, there were a few BUILD files that did not meet these criteria, and to speed adoption, the new system had to build these, too. This was handled by using a preprocessor which performs a one-way transformation on the files. As no mechanism existed to enforce these criteria, over time a larger and larger proportion of BUILD files failed to comply and required preprocessing. Managing this debt became a prerequisite for cleaning up dependencies, zombie targets, and dead flags.

VIII. Conclusion

We have described a type of technical debt found in Google's build system artifacts that we call Build Debt. We have explained how this debt hurts the company in terms of a) lower productivity of engineers — slower builds, more brittle targets, maintenance of abandoned, or broken libraries and b) increased computation costs for our build and test infrastructure — building and running unnecessary code and tests. We described some of our efforts to reduce this debt, including dependency debt, visibility debt, and zombie targets, along with their results. We also described a fixit organized to delete dead command line flag definitions and the dead code they often guard.

Our experience has suggested that prioritizing and dealing with technical debt cannot always be left to individual teams, since many engineers resist these efforts on the grounds that it would slow them down or encourage code duplication. In addition, as the size of a codebase increases, the cost of recovering from technical debt increases non-linearly. Therefore, it is imperative to pay attention to these debts early and invest in tools, policies, and mechanisms that make an organization aware of the debt being incurred and make it easy to continually repay/avoid the debt as part of each engineer's normal workflow.

Acknowledgment

The authors would like to thank many of our co-workers for help getting our efforts off the ground. For help editing this paper, we thank Ulf Adams, Robert Bowdidge, and Russ Rufer.

References

[1] N. York. (2011, June) Build in the cloud: Accessing source code. Google Engineering Tools. [Online]. Available: http://google-engtools.blogspot.com/2011/06/build-in-cloud-accessing-source-code.html

[2] A. Kumar. (2010, December) Development at the speed and scale of Google. QCon. [Online]. Available: http://www.infoq.com/presentations/Development-at-Google

[3] C. Kemper. (2011, August) Build in the cloud: How the build system works. Google Engineering Tools. [Online]. Available: http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html

[4] M. Barnathan, G. Estren, and P. Lebeck-Jobe. (2012, March) Building software at google scale tech talk. Google. [Online]. Available: http://www.youtube.com/watch?v=2qv3fcXW1mg

[5] S. McIntosh, B. Adams, T. Nguyen, Y. Kamei, and A. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 141–150.

[6] P. Gupta, M. Ivey, and J. Penix. (2011, June) Testing at the speed and scale of Google. Google Engineering Tools. [Online]. Available: http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html

[7] J. Bohnet and J. Döllner, "Monitoring code quality and development activity by software maps," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 9–16.

[8] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. Society Press, 2007, pp. 92–99.

[9] A. Telea and L. Voinea, "A tool for optimizing the build performance of large software code bases," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*, 2008, pp. 323–325.