Google's Innovation Factory: Testing, Culture, And Infrastructure

Patrick Copeland, Google
copeland@google.com

ABSTRACT

Google's external mythology has been one of a brilliant and chaotic innovation machine that produces new products and features at an amazing rate. Behind the curtain of public perception is a company that takes quality seriously and is reinventing how software is created, tested, released, and maintained; a reality that's even more interesting than the myth.

At Google we've learned a lot in the last few years about accelerating very large scale software development; in this paper we'll share what has worked and what hasn't worked for us.

1. DYNAMIC EQUILIBRIA

Since humans began writing software in the middle of the last century, the process has been cumbersome, error prone and has more often than not created an end product that is low in quality. Most companies are better at talking about software quality than implementing it.[1]

This is clearly not a new problem. In 1962 the "most expensive hyphen in history" forced the destruction of the Mariner I rocket only 293 seconds after it was launched. Instead of its intended flyby of Venus, the rocket ended up in the Atlantic Ocean.[2]

Such events have been a mainstay of computing history ever since. In fact, Googling the search term "software bug" turns up over 80 million hits. Buggy software is part of the industry's fabric.

1.1 TORRENTIAL PROCESS

There have been numerous attempts over the prior decades to build more reliable software and these have come under many guises. Total Quality, Zero Defect, Six Sigma and Cleanroom have all borrowed ideas that were successful in manufacturing, specifically prescribing more methodical and process-driven approaches to software development. Yet here we are in 2010 still talking about software quality! It's hard to come to any other conclusion than that the lessons learned from manufacturing don't translate well to software.

Quality is still very hard to evaluate in software and we end up with estimations that focus on quantifying the measurable and rely on subjectivity for the rest. As Niklaus Wirth recently said,

> *"The experience, judgment, and intuition of programmers who have survived the rigors of testing are what make programs of the present day useful, efficient, and correct."* [3]

1.2 EMERGING LEFTISM

One thread common to formal models are that they focus on a few of the many variables: improving efficiency, predictable process, estimation of quality, or others. As most practitioners know, a development process is a polynomial wrapped inside of a culture, and solving for a few variables only achieves a momentary local maxima.

While process-heavy development models may work well for manufacturing airplanes and have been successfully applied by some companies[4], they have been viewed by many developers as burdensome and contrary to the creative nature of writing innovative software. Conversely, "process-less process", can lead to a heroic culture that's unable to repeatedly deliver. There needs to be balance.

Consider the physics of flight as an analogy to software process. In addition to reasonable flying conditions and an experienced pilot, the key to getting airborne is having an appropriate balance of factors that match the situation: too much weight or too little thrust can be disastrous depending on the situation. Similarly, teams, products and process all have virtual physics. For instance, adding engineers late in a product cycle doesn't necessarily provide more lift[5]. Adopting a new process may give a team more thrust momentarily, but may also incur a longer term drag that makes them incapable of innovation.

The popularity of Agile, while not a wholesale rejection of more rigid processes, indicates that developers desire more balance and creativity. Whatever we do to make software higher quality and more predictable to build, we must maintain a balance that encourages the innovative aspects of the art form. We need to motivate smart minds to solve hard problems and deliver rich features to customers. In other words, we need to focus on staying airborne for the long term.

1.3 PARADIGM SHIFTS

A lot of software is now released as services and deployed to data centers controlled by the software producers rather than being installed on customer-owned servers/clients of infinite configurations scattered around the globe. Software can be released to early adopters and beta users, bug fixes can be deployed to all users simultaneously or to a small percentage, maintenance and updates are handled centrally by experts and not by end users. With more control of the end product, development teams can experiment and take more risks providing innovation faster and with less fear. When problems appear, they can be identified and fast-fixed before impacting large groups of users.

But the cloud paradigm is only part of the equation. We also need to think differently about using these capabilities for software development itself. Can we align our culture, tools and processes to take full advantage of this new model? Can we use automation to solve, once and for all, the repetitive, mundane and downright boring aspects of building products? Can we integrate development and testing so tightly that writing good code is easier than writing bad code? Can we encourage big thinking that leads to new ideas? Can we do all of this at the scale and hypercompetitive pace of the Internet?

We've been tackling this problem for several years at Google and this paper is a report of our progress. Our approach has been to automate those development tasks that shouldn't require a human in-the-loop, to focus on building a culture around quality, to promote multiple approaches to innovation, reduce bureaucratic creep, and to invest in reusable infrastructure.

2. INNOVATION FACTORY

We encourage our engineers to focus on innovation. Eric Schmidt, has said, "We take our jobs to be innovators and we are failing if we are not innovating quickly enough.[6]" Many of our best ideas were envisioned by engineers who were passionate about solving a problem. Popular products, like Gmail, were initially developed by a few passionate engineers outside of their normal work.

Linus Pauling is commonly quoted as saying, "The best way to have a good idea is to have lots of ideas." Google has made its mark on the industry with new approaches to old problems. For example, our systems are built on "flaky" commodity hardware and an infrastructure that dynamically compensates for that flakiness. Initially this was a subversive idea, as other companies at the time were building servers that attempted to eliminate all failures (like the foolproof HAL9000 from 2001). We expect everything to fail and use redundancy and automated compensation techniques to maintain overall reliability.

2.1 BUILDING FOR SCALE

Outside the walls of Google, this innovation factory has created desirable products for our users. Inside the walls, it has created large repositories of code, data, dependencies and information that must be managed closely. Consider the logistics of delivering at Google's current pace:

- More than 6,000 engineers and >40 offices.
- 2,500 ongoing projects (2.5 developers / project).
- 1,600 active external release branches for products.
- 59,000 builds / day each with 10-1000 targets..
- 1.5 million tests / day, both manual and automated.
- Most products localized into 40 languages.
- At least bi-weekly release cycles.

2.2 FLAT & AUTONOMOUS

The organizational structure we use is atypical in the industry. For one, Google is a flat organization with many Nooglers being no more than 2-3 steps below senior executives. The company structure can be characterized as: flat and autonomous.

At Google, managers are not controllers, they are connectors charged with ensuring that teams make effective use of information and tools. Many managers have 15 or more direct reports, introducing some chaos and reducing the time available to micromanage. Managers are judged on their ability to enable smart people to get things done.

Teams are aligned along business lines we call "focus areas" rather than around strict product lines. People doing similar work, no matter what products they are contributing to, will find themselves in close reporting proximity to their colleagues. This matrix encourages some amount of competition, but also the reuse of good ideas.

Projects live and die based on free-market Darwinism, where successful projects are further funded and less successful ones face atrophy. We take many short and long term bets, but projects must produce value to survive.

2.3 AVOIDING PLAUSIBLE DENIABILITY

The entire product team is responsible for quality, and is judged on their ability to enable innovation, anticipate problems, make plans, and implement high quality software. Teams adopt processes that are in their own self interest and that allow them to focus on innovation.

The role of someone doing testing in this environment is structured slightly differently than other technology companies. Testers avoid becoming codependents within this system and generally do not write unit tests or other activities that are best done by the developer. Testing teams focus on higher abstractions, like identifying latencies, system or customer focused testing, and enabling the process.

Code is expected to have high reliability as it is written and we adhere to a socially reinforced code review and check-in practices. Development teams write good tests because they care about the products, but also because they want more time to spend writing features and less on debugging. Teams with good testing hygiene upstream have more time to innovate, and are thus more adaptable and competitive. In addition, there is one source tree and poorly written code is quickly identified because it breaks other people's tests and projects. Aggressive rolling back is employed to keep the tree building "green."

Unlike traditional testing approaches, teams do not focus on the tail end of the process or pad the schedule for special testing phases. Instead, they look for ways to anticipate issues and solve them proactively in real time. Within each project are experts in the field of software quality and they ensure that the right tools, test cases and test procedures are in place throughout the product lifecycle. When bugs do slip through, or more commonly

unanticipated complex behavior situations occur, we aggressively do postmortems and quickly put in place solutions that prevent them from reoccurring.

2.4 VIRAL ADOPTION

At an individual project level, uniformity is rarely mandated and adoption of tools and process is left to an internal "market" to decide. Apart from our core systems, discussed later, a large portion of our tools are developed by motivated individuals to solve local challenges. Similarly, process is tailored specifically to projects. While this leads to a healthy amount of chaos, good ideas tend to spread quickly, because they have been proven useful by others. Engineers decide what's best for engineering, to articulate the right vision, and to drive initiatives in the most sustainable fashion, and *then* others follow after grassroots successes. We've found that positive experience is an effective means of persuasion.

An example of viral adoption is a "fix it", or an event organized by engineers, that encourages Googlers to work on the same problem at the same time. The idea is to get a large amount of work done in a short amount of time by leveraging the power of masses. In the past, these have been focused on fixing 1000 TODOs in the code-base or fixing tests to take advantage of new infrastructure improvements.

Testing on the Toilet is another example of a viral adoption. It started as an offhand joke and it became a world wide sensation, making headlines in the Wall Street Journal. The idea was to communicate ideas about testing and to do it in a place where we know people would have the time to read it. It's is now published in hundreds of stalls in most Google offices, taking submissions from different programming languages and application domains, and appears on Google's public testing blog[7]. While the articles themselves need to be short enough for people to read while they "do their business", the ideas create a buzz about specific topics and that would otherwise be difficult to achieve.

2.5 CMM WITH A TWIST

One popular grass roots initiation that resembles a more traditional process methodology is called the Test Certified Program. Test Certified is a series of increasingly advanced levels, each defined by a list of measurable testing goals and capabilities. These goals are set by testers and present quality practices, advanced techniques and quality-oriented goals for a development team to strive to achieve. As a development team achieves more goals, using whatever techniques that suit their team culture and problem domain, they move up through the Test Certified ladder levels from TC1 to TC5.

At the initial stages of the process, teams are asked to clean up and do several remedial actives, all of which are designed to get them seeing the benefits of testing immediately. Establishing a continuous build that runs a set of fast deterministic tests is the most important aspect of the first phase. Speed is achieved by focusing on small unit level tests and using practices like mocking and distributed execution.

In subsequent levels, code coverage goals are explicitly defined, rules about releasing on "non-green" builds are imposed, and a broader array of testing is expected, such as integration, system level, and various other techniques.

This process is defined not to dictate to developers what to do but to identify goals that will help them develop better software faster and spend less time in later phases fixing bugs. Advancement won't guarantee higher quality software but it does pattern a roadmap that makes good quality more probable.

2.6 ELEMENTS OF CONTROL

As a counterbalance to the randomness incurred by our relatively freeform process are a set of release standards and guidelines. These "launch reviews and criteria" are outlined to ensure that products answer common sense questions before release. A few examples are:

- Is the design secure and customer data private?
- Will the service scale with the anticipated load?
- Does the UI meet standards?
- What are the data center utilization estimates?
- What are the latency estimates?

The point is that the release process is not friction free. There are many high standards that must be met and it can be frustrating for teams that procrastinate. Pain can be avoided by driving change up stream as early as possible. Given the forewarning, teams can meet the standards in a way appropriate to their constraints.

3. FASTER DEVELOPER WORKFLOW

The build/test system is at the core of day-to-day activity for software engineers at Google. Almost everything deployed in production is developed, tested, and built using this system. Thus, the performance and usability of the build tools has a large impact on engineer productivity, where even small changes are multiplied by the total number of tool interactions.

As traditional companies scale, sub-organizations begin to maintain separate code silos, build tribal release and integration procedures, and duplicate effort. But, more disturbing, they end up sinking a large amount of time into maintenance issues. Time that should be spent adding value is instead used to atone for past sins.

Engineering teams should be able to concentrate a maximum of their time on quality and innovation. At Google that time is achieved, at least in part, by making the hard and the mundane simple and automatic. As a case-in-point, consider our build and deployment infrastructure.

3.1 DESIGN CONSIDERATIONS

Prior to 2006, Google employed a fairly slow build and test process that was designed for a much smaller company. Back then, builds might be broken for days or weeks, the

"unpaid mortgage" of new code would build up, and then would be followed by lengthy debugging and stabilization phases. We needed an approach that provided developers nearly instant feedback on every code check-in.

We designed the system with the following principles:

- Speed: All test and analysis systems need to return results very fast. If it takes too long, engineers will either ignore or not bother looking for that data.
- Feedback: The focus of test systems must be on high quality feedback. We want engineers to keep code at production quality at all times, not adding time to fix code that was broken earlier.
- Simplicity: Engineers should not have to understand how the underlying build and test systems work. All data and feedback must be easy to understand, integrated into commonly-used productivity tools, and presented in a workflow that allows them to take appropriate action.

Within milliseconds of a code check-in, our build process will automatically select the appropriate tests to run based on dependency analysis, run those tests and report the results. By reducing the window of opportunity for bad code to go unnoticed, overall debugging and bug isolation time is radically reduced. The net result is that the engineering teams no longer sink hours into debugging build problems and test failures.

3.2 ESTIMATING IMPACT

We created a more holistic approach to estimate the overall impact of improvements. We know the general workflow of engineers, and we can estimate how much time engineers spend in each area of the workflow. From this we can model a "representative engineer" which provides a framework for estimating where engineers spend their time with tools. With this model we can measure the effect of improvements on each area of the workflow to estimate overall impact.

TABLE: ESTIMATED MONTHLY ACTIVITY PER DEVELOPER

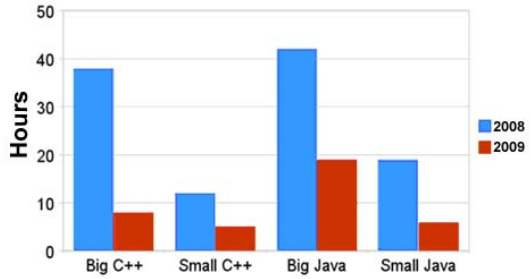
ACTIVITY	INITIAL CHECK-OUT	CLEAN BUILD	BUILD AFTER EDIT	BUILD AFTER SYNC	RUN TESTS
FREQUENCY	2	4	160	20	60

From the workflow we can identify five key activities involving build tools. These are: Initial Checkout, Clean Build, Build After Edit, Build After Incremental Sync, and Run Tests. The tricky part is estimating the frequency of these actions. This is subjective since the details vary for each engineer. Some do a clean checkout and build for every task. Others never do a full sync/clean build after the initial build is created. By collecting the data and identifying the most frequent use cases, we were able focus on the largest productivity wins.

3.3 RESULTS

We were able to save the company about 600 person years of time that would otherwise have been spent waiting on tools. We did this improving the highest traffic workflows with caching, distributed execution, and avoiding bottlenecks.

CHART: TIME WAITING ON TOOLS IN HOURS/MONTH/DEVELOPER.



More importantly we were able to change how products are produced with an emphasis on continual improvement. The chart below show the number of hours "saved" per month per developer on different types of projects. For instance "big" are defined as having more than 20k or more files.

5. CONCLUSION

Just as we are witnessing a paradigm shift to cloud computing that stretches our imagination and challenges the limits of software, our process for developing that software is going through an equally dramatic revolution. We are reconsidering the appropriateness of the lessons we've taken from manufacturing. We believe that software development models require a new set of physics.

Google has experimented with this new physics with innovative new tools, processes and infrastructure. While there is no magic bullet, there is a pragmatism that can be applied to software development that seeks to balance the art form of creating software with the needs for repeatability, efficiency, and quality. At Google that has meant eliminating the tedious and repetitive tasks with automation and streamlined processes allowing testers to engage the full extent of their creativity on innovation and meeting the challenges of modern software development.

6. ACKNOWLEDGEMENTS

James Whittaker, Alberto Savoia, Nathan York, Mark Striebeck, and the following teams from Google: Engineering Productivity, and the Test Grouplet.

7. REFERENCES

[1] David N. Wilson, Tracy Hall, Perceptions of software quality: a pilot study, Software Quality Journal 7, (1998) 67–75.
[2] Eric Roberts, Mariner I, The Risk Digest, Volume 5, Issue 66, (1987).
[3] Niklaus Wirth, Opening Talk GTAC 2009, (2009).
[4] Michael Diaz, Joseph Sligo, How Software Process Improvement Helped Motorola, IEEE, 0740-7459, (1997) 75-81.
[5] Fred Brooks, Mythical Man Month, (1995).
[6] Ravi Mattu, World exclusive interview with Google!, ft.com/managementblog, July 8, 2009.
[7] Introducing "Testing on the Toilet", Google Testing Blog, January 21, 2007.