Principles of Software Construction: Objects, Design, and Concurrency

Software development at scale
Bonus slides: Unseen GoF design patterns

(The end)

Michael Hilton   Bogdan Vasilescu
Administrivia

- Final exam Monday May 6\textsuperscript{th} 5:30-8:30 GHC 4401
- Review session Saturday May 4\textsuperscript{th} 1pm NSH 3305
Part 1: Design at a Class Level
  Design for Change: Information Hiding, Contracts, Unit Testing, Design Patterns
  Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns

Part 2: Designing (Sub)systems
  Understanding the Problem
  Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies
  Design for Reuse at Scale: Frameworks and APIs

Part 3: Designing Concurrent Systems
  Concurrency Primitives, Synchronization
  Designing Abstractions for Concurrency
SOFTWARE DEVELOPMENT AT SCALE
Releasing at scale in industry

- Facebook: [https://atscaleconference.com/videos/rapid-release-at-massive-scale/](https://atscaleconference.com/videos/rapid-release-at-massive-scale/)
- Google: [https://www.slideshare.net/JohnMicco1/2016-0425-continuous-integration-at-google-scale](https://www.slideshare.net/JohnMicco1/2016-0425-continuous-integration-at-google-scale)
- Why Google Stores Billions of Lines of Code in a Single Repository: [https://www.youtube.com/watch?v=W71BTkUbdqE](https://www.youtube.com/watch?v=W71BTkUbdqE)
- F8 2015 - Big Code: Developer Infrastructure at Facebook's Scale: [https://www.youtube.com/watch?v=X0VH78ye4yY](https://www.youtube.com/watch?v=X0VH78ye4yY)
Pre-2017 release management model at Facebook
Diff lifecycle: First, local testing

Test and lint locally
Diff lifecycle: Next, CI testing (data center)
Diff lifecycle: Then, diff ends up on master
Release every two weeks
Quasi-continuous push from master (1,000+ devs, 1,000 diffs/day); 10 pushes/day
Aside: Key idea – fast to deploy, slow to release

Dark launches at Instagram

- **Early**: Integrate as soon as possible. Find bugs early. Code can run in production about 6 months before being publicly announced (“dark launch”).
- **Often**: Reduce friction. Try things out. See what works. Push small changes just to gather metrics, feasibility testing. Large changes just slow down the team. Do dark launches, to see what performance is in production, can scale up and down. "Shadow infrastructure" is too expensive, just do in production.
- **Incremental**: Deploy in increments. Contain risk. Pinpoint issues.
Aside: Feature Flags

Typical way to implement a dark launch.

http://martinfowler.com/bliki/FeatureToggle.html
Issues with feature flags

Feature flags are “technical debt”
Example: financial services company with nearly $400 million in assets went bankrupt in 45 minutes.

Diff lifecycle: Finally, in production
What’s in a weekly branch cut? (The limits of branches)
Post-2017 release management model at Facebook

Quasi-continuous web release

- Release
- Test
- Build
- Deploy
Google repository statistics
As of Jan 2015

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of files*</td>
<td>1 billion</td>
</tr>
<tr>
<td>Number of source files</td>
<td>9 million</td>
</tr>
<tr>
<td>Lines of code</td>
<td>2 billion</td>
</tr>
<tr>
<td>Depth of history</td>
<td>35 million commits</td>
</tr>
<tr>
<td>Size of content</td>
<td>86 terabytes</td>
</tr>
<tr>
<td>Commits per workday</td>
<td>45 thousand</td>
</tr>
</tbody>
</table>

*The total number of files includes source files copied into release branches, files that are deleted at the latest revision, configuration files, documentation, and supporting data files.
Exponential growth
Google™  Speed and Scale

- >30,000 developers in 40+ offices
- 13,000+ projects under active development
- 30k submissions per day (1 every 3 seconds)

- All builds from source
- 30+ sustained code changes per minute with 90+ peaks
- 50% of code changes monthly
- 150+ million test cases / day, > 150 years of test / day
- Supports continuous deployment for all Google teams!
Google code base vs Linux kernel code base

Some perspective

**Linux kernel**
- 15 million lines of code in 40 thousand files (total)

**Google repository**
- 15 million lines of code in 250 thousand files *changed per week, by humans*
- 2 billion lines of code, in 9 million source files (total)
How do they do it?
1. Lots of (automated) testing

Google workflow

- Sync user workspace to repo
- Write code
- Code review
- Commit

- All code is reviewed before commit (by humans and automated tooling)
- Each directory has a set of owners who must approve the change to their area of the repository
- Tests and automated checks are performed before and after commit
- Auto-rollback of a commit may occur in the case of widespread breakage
2. Lots of automation

**Additional tooling support**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critique</td>
<td>Code review</td>
</tr>
<tr>
<td>CodeSearch*</td>
<td>Code browsing, exploration, understanding, and archeology</td>
</tr>
<tr>
<td>Tricorder**</td>
<td>Static analysis of code surfaced in Critique, CodeSearch</td>
</tr>
<tr>
<td>Presubmits</td>
<td>Customizable checks, testing, can block commit</td>
</tr>
<tr>
<td>TAP</td>
<td>Comprehensive testing before and after commit, auto-rollback</td>
</tr>
<tr>
<td>Rosie</td>
<td>Large-scale change distribution and management</td>
</tr>
</tbody>
</table>

** See "Tricorder: Building a program analysis ecosystem". In International Conference on Software Engineering (ICSE), 2015
3. Smarter tooling

- Build system
- Version control
- ...

3a. Build system
Standard Continuous Build System

- Triggers builds in continuous cycle
- Cycle time = longest build + test cycle
- Tests many changes together
- Which change broke the build?
Google Continuous Build System

- Triggers tests on every change
- Uses fine-grained dependencies
- Change 2 broke test 1
Which tests to run?

- **GMAIL**
  - Test Target:
    - name: //depot/gmail_client_tests
    - name: //depot/gmail_server_tests

- **BUZZ**
  - Test targets:
    - name: //depot/buzz_server_tests
    - name: //depot/buzz_client_tests

Diagram:
- buzz_client_tests
- gmail_client_tests
- gmail_server_tests
- buzz_server_tests
- youtube_client
- gmail_client
- gmail_server
- youtube_server
- common_collections_util
Scenario 1: a change modifies common_collections_util

When a change modifying common_collections_util is submitted.
Scenario 1: a change modifies common_collections_util

When a change modifying common_collections_util is submitted.
Scenario 1: a change modifies common_collections_util

When a change modifying common_collections_util is submitted.
Scenario 1: a change modifies common_collections_util

All tests are affected! Both Gmail and Buzz projects need to be updated.

When a change modifying common_collections_util is submitted.
Scenario 2: a change modifies the youtube_client
Scenario 2: a change modifies the youtube_client

Only buzz_client_tests are run and only Buzz project needs to be updated.

When a change modifying youtube_client is submitted.
3b. Version control

- Problem: even git can get slow at Facebook-like scale
  - 1M+ source control commands run per day
  - 100K+ commits per week

![Cloning with git: iOS Today](image)
3b. Version control

- Solution: redesign version control
3b. Version control

- Solution: redesign version control
  - Query build system's file monitor, Watchman, to see which files have changed
3b. Version control

- Solution: redesign version control
  - Query build system's file monitor, Watchman, to see which files have changed → **5x faster “status” command**
3b. Version control

• Solution: redesign version control
  – Sparse checkouts?? (remember, git is a distributed VCS)
3b. Version control

• Solution: redesign version control
  – Sparse checkouts:
  – Change the clone and pull commands to download only the commit metadata, while omitting all file changes (the bulk of the download)
  – When a user performs an operation that needs the contents of files (such as checkout), download the file contents on demand using existing memcache infrastructure
3b. Version control

- Solution: redesign version control
  - Sparse checkouts → **10x faster clones and pulls**
  - Change the clone and pull commands to download only the commit metadata, while omitting all file changes (the bulk of the download)
  - When a user performs an operation that needs the contents of files (such as checkout), download the file contents on demand using existing memcache infrastructure
4. Monolithic repository
Monolithic repository – no major use of branches for development

Trunk-based development
Combined with a centralized repository, this defines the monolithic model

- Piper users work at “head”, a consistent view of the codebase
- All changes are made to the repository in a single, serial ordering
- There is no significant use of branching for development
- Release branches are cut from a specific revision of the repository
Did it work? Yes. Sustained productivity at Facebook

Lines Committed Per Developer Per Day

Growth of the size of the Android and iOS dev teams
MONOREPO VS MANY REPOS
A recent history of code organization

- A single team with a monolithic application in a single repository
- Multiple teams with many separate applications in many separate repositories
- Multiple teams with many separate applications microservices in many separate repositories
- A single team with many microservices in many repositories
- Many teams with many applications in one big Monorepo
What is a Monolithic Repository (monorepo)?

A **single** version control repository containing multiple

- projects
- applications
- libraries,

often using a common build system.
Monorepos in industry

Google (computer science version)

COMMUNICATIONS OF THE ACM

CONTRIBUTED ARTICLES

Why Google Stores Billions of Lines of Code in a Single Repository

By Rachel Potvin, Josh Levenberg
Communications of the ACM, Vol. 59 No. 7, Pages 78-87
10.1145/2854148
Comments (3)

Early Google employees decided to work with a shared codebase managed through a centralized source control system. This approach has served Google well for more than 16 years, and today the vast majority of Google’s software assets continues to be stored in a single, shared repository. Meanwhile, the number of Google software developers has steadily increased, and the size of the Google codebase has grown exponentially (see Figure 1). As a result, the technology used to host the codebase has also evolved significantly.

Back to Top
Key Insights
Advantages and Disadvantages of a Monolithic Repository

A case study at Google

Ciera Jaspan, Matthew Jorde,
Andrea Knight, Caitlin Sadowski,
Edward K. Smith, Collin Winter
Google
ciera,majorde,aknight,supertri,edwardsmith,
collinwinter@google.com

ABSTRACT
Monolithic source code repositories (repos) are used by several large tech companies, but little is known about their advantages or disadvantages compared to multiple per-project repos. This paper investigates the relative tradeoffs by utilizing a mixed-methods approach. Our primary contribution is a survey of engineers who have experience with both monolithic repos and multiple, per-project repos. This paper also backs up the claims made by these engineers with a large-scale analysis of developer tool logs. Our study finds that the visibility of the codebase is a significant advantage of a monolithic repo: it enables engineers to discover APIs to reuse, find examples for using an API, and automatically have dependent code updated as an API migrates to a new version. Engineers also appreciate the centralization of dependency management in the repo. In contrast, multiple-repository (multi-repo) systems afford engineers more flexibility to select their own toolchains and provide significant access control and stability benefits. In both cases, the related tooling is also a significant factor; engineers favor particular tools and are drawn to repo management systems that support their desired toolchain.

CCS CONCEPTS
• Software and its engineering → Software configuration management and version control systems;

1 INTRODUCTION
Companies today are producing more source code than ever before. Given the increasingly large codebases involved, it is worth examining the software engineering experience provided by the various approaches for source code management. Our study finds that the visibility of the codebase is a significant advantage of a monolithic repo: it enables engineers to discover APIs to reuse, find examples for using an API, and automatically have dependent code updated as an API migrates to a new version. Engineers also appreciate the centralization of dependency management in the repo. In contrast, multiple-repository (multi-repo) systems afford engineers more flexibility to select their own toolchains and provide significant access control and stability benefits. In both cases, the related tooling is also a significant factor; engineers favor particular tools and are drawn to repo management systems that support their desired toolchain.

One approach to scaling development practices is the monolithic repo, a model of source code organization where engineers have broad access to source code, a shared set of tools, and a single set of common dependencies. This standardization and level of access is enabled by having a single, shared repo that stores the source code for all the projects in an organization. Several large software companies have already moved to this organizational model, including Facebook, Google, and Microsoft [10, 12, 17, 21]; however, there is little research addressing the possible advantages or disadvantages of such a model. Does broad access to source code let software engineers better understand APIs and libraries, or overwhelm engineers with use cases that aren’t theirs? Do projects benefit from shared dependency versioning, or would engineers prefer more stability for their dependencies? How often do engineers take advantage of the workflows that monolithic repos enable? Do engineers prefer having consistent, shared toolchains or the flexibility of selecting a toolchain for their project?

In this paper, we investigate the experience of engineers working within a monolithic repo and the tradeoffs between using a monolithic repo and a multi-repo codebase. Specifically, this paper seeks to answer two research questions:

1. What do developers perceive as the benefits and drawbacks to working in a monolithic versus multi-repo environment?
2. To what extent do developers make use of the unique advantages that monolithic repos provide?
Monorepos in industry

Scaling Mercurial at Facebook

With thousands of commits a week across hundreds of thousands of files, Facebook’s main source repository is enormous—many times larger than even the Linux kernel, which checked in at 17 million lines of code and 44,000 files in 2013. Given our size and complexity—and Facebook’s practice of shipping code twice a day—improving our source control is one way we help our engineers move fast.

Choosing a source control system

Two years ago, as we saw our repository continue to grow at a staggering rate, we sat down and extrapolated our growth forward a few years. Based on those projections, it appeared likely that our then-current technology, a Subversion server with a Git mirror, would become a productivity bottleneck very soon. We looked at the available options and found none that were both fast and easy to use at scale.

Our code base has grown organically and its internal dependencies are very complex. We could have spent a lot of time making it more modular in a way that would be friendly to a source control tool, but there are a number of benefits to using a single repository. Even at our current scale, we often make large changes throughout our code base, and having a single repository is useful for continuous
Monorepos in industry

Microsoft claim the largest git repo on the planet

The largest Git repo on the planet
05/24/2017 by Brian Harry MS // 59 Comments

It’s been 3 months since I first wrote about our efforts to scale Git to extremely large projects and teams with an effort we called “Git Virtual File System”. As a reminder, GVFS, together with a set of enhancements to Git, enables Git to scale to very large-repos by virtualizing both the .git folder and the working directory. Rather than download the entire repo and checkout all the files, it dynamically downloads only the portions you need based on what you use.

A lot has happened and I wanted to give you an update. Three months ago, GVFS was still a dream. I don’t mean it didn’t exist – we had a concrete implementation, but rather, it was unproven. We had validated on some big repos but we hadn’t rolled it out to any meaningful number of engineers so we had only conviction that it was going to work. Now we have proof.

Today, I want to share our results. In addition, we’re announcing the next steps in our GVFS journey for customers, including expanded open sourcing to start taking contributions and improving how it works for us at Microsoft, as well as for partners and customers.

Windows is live on Git

Over the past 3 months, we have largely completed the rollout of Git/GVFS to the Windows team at Microsoft.

As a refresher, the Windows code base is approximately 3.5M files and, when checked in to a Git repo, results in a repo of about 300GB.
Monorepos in open-source

foresquare public monorepo
Monorepos in open-source

The Symfony monorepo

43 projects, 25,000 commits, and 400,000 LOC

https://github.com/symfony/symfony

- Bridge/
  - 5 sub-projects
- Bundle/
  - 5 sub-projects
- Component/
  - 33 independent sub-projects like Asset, Cache, CssSelector, Finder, Form, HttpKernel, Ldap, Routing, Security, Serializer, Templating, Translation, Yaml, ...
Common build systems

**Bazel from Google**

**Buck from Facebook**

**Pants from Twitter**
Some advantages of monorepos
High Discoverability For Developers

- Developers can read and explore the whole codebase
- `grep`, IDEs and other tools can search the whole codebase
- IDEs can offer auto-completion for the whole codebase
- Code Browsers can links between all artifacts in the codebase
Code-Reuse is cheap

Almost zero cost in introducing a new library

- Extract library code into a new directory/component
- Use library in other components
- Profit!
Refactorings in one commit

Allow large scale refactorings with one single, atomic, history-preserving commit

- Extract Library/Component
- Rename Functions/Methods/Components
- Housekeeping (phpcs-fixer, Namespacing, ...)

Qafoo
passion for software quality
Copyright Qafoo GmbH, All Rights Reserved
Another refactoring example

- Make large backward incompatible changes easily... especially if they span different parts of the project

- For example, old APIs can be removed with confidence
  - Change an API endpoint code and all its usages in all projects in one pull request
Some more advantages

• Easy continuous integration and code review for changes spanning several projects
• (Internal) dependency management is a non-issue
• Less context switching for developers
• Code more reusable in other contexts
• Access control is easy
Some downsides

- Require collective responsibility for team and developers
- Require trunk-based development
  - Feature toggles are technical debt (recall financial services example)
- Force you to have only one version of everything
- Scalability requirements for the repository
- Can be hard to deal with updates around things like security issues
- Build and test bloat without very smart build system
- Slow VCS without very smart system
- Permissions?
Summary

- Software development at scale requires a lot of infrastructure
  - Version control, build managers, testing, continuous integration, deployment, ...
- It’s hard to scale development
  - Move towards heavy automation (DevOps)
- Continuous deployment increasingly common
- Opportunities from quick release, testing in production, quick rollback