Lecture 25: Delivery

CS 5150, Spring 2025

Administrative Reminders

- Final Delivery May 14 12 PM
- For presentation, please use my calendly link to book 2 slots!
  - Reserve 1 hr

Topics to refresh for in-class exam

- Differences between various test generation techniques, their relative advantages, disadvantages
- Understanding how unit test generation algorithms work: Randoop, Korat, …
- Different dependency management systems
- Comparison of build systems
- Performance testing (Amdahl's law, flame graphs, profiling)
- Test minimization algorithm (delta debugging)
- How static analysis techniques work (annotations, types, etc.)
- Principles of user testing

Previously in 5150

Build Systems

What does a build system even do?

- Why something like `javac *.java` is not enough?
- How to handle:
  - Building libraries stored in different directories (shared libraries)
  - Code written in different programming languages (dependencies)
  - Third-party jar files (how to store them, version management)
  - Rebuilding part of the codebase after dependency upgrade
  - Target different systems/release builds (build configs)
  - (Implicit dependencies) Managing related artifacts/tasks: documentation, latest library version
- Write a shell script?

Desirable properties of build system

- Fast:
  - Run a single command to build and get the output binary in a short time (few seconds)
- Correct:
  - Reproducible: Should output same result for any developer/machine for the same input files

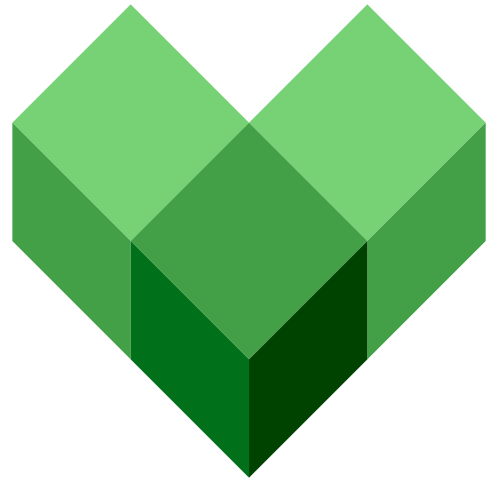
Task-Based Build Systems

- Task: Fundamental Unit of Work
- Tasks can have other tasks as dependencies
- Major build systems: Ant, Maven, Gradle, Grunt, Rake, …

Artifact-based build system

- Build: Tell the system “what” to build instead of “how”
- Implemented in Blaze/Bazel (Google), Pants, Buck
- Build files are declarative: specify set of artifacts to build, their dependencies, some build options (instead of exact steps)
- Blaze has full control over “how” build is run

- *(Stronger) correctness guarantee while being more efficient*



Bazel Advantages/Differences

- Parallelization:
  - Targets that only require java compiler (vs custom script)
- Reuse/caching:
  - If MyBinary.java changes, it will rebuild **MyBinary** but reuse **mylib**
  - If a source file for **//java/com/example/common** changes, Bazel knows to rebuild that library, **mylib**, and **MyBinary**, but reuse **//java/com/example/myproduct/otherlib**

Other Bazel Features

- Tools as dependencies, **toolchains** (platform-specific tool usage)
- Custom user-defined **actions:** specify inputs, outputs, and steps
- **Sandboxing**: isolating filesystem for each action
- Remote caching
- **Distributed build**: Remote build
- Making remote/external dependencies deterministic
  - Manifest file: Create **cryptographic hash** for each ext dependency, only redownload when hash changes, build fails if hash changes
  - What can go wrong?

Read more details in SWE@Google Chapter 18 build systems

PollEv.com/cs5150sp25

Which of these techniques is not useful for managing external dependencies?

A. Signature/hash verification
B. (Shared) Caching
C. Mirroring servers
D. Distributed build
E. Vendoring

Releasing software

When is software ready to release?

- When it is feature-complete?
- When it is bug-free?
- When it has soaked for long enough?
- On the release date?
- Continuously?

- "*The biggest risk to any software effort is that you end up building something that isn't useful… The earlier and more frequently you get working software in front of real users, the quicker you get feedback to find out how valuable it really is."*
  - Martin Fowler

Traditional release process

Example: GCC (Release managers)

1. Merge window (time-boxed, 4 months)
   - Branches that are "ready" are merged to trunk
2. Bug fixes (time-boxed) (now retired)
   - Cut release branch
   - No new (coupled) features
3. Regression & doc fixes (2 months)
4. Release
   - When all high-priority bugs are fixed

- Challenges
  - Need to coordinate process
  - Features that miss merge window must wait until next cycle
  - Problematic features are difficult to remove
  - Branch divergence (maintain dev branch)

Time-based vs. feature-based releases

Feature-based

- Product manager decides which major features define the next release
- Developers argue for their features to be included
- Which features should hold up release? (tendency for inflation)

Time-based

- All features that are completed (tested) by release deadline are included in release
- Features that are not ready must wait until next release
- Shorter release cadence reduces cost of missing deadline

Risks of long release processes

- Delay in providing value to client
  - May fall behind competition
- Slow feedback on feature utility
- Drain on morale
  - Churn among release managers prevents building expertise
- Difficult to diagnose issues
- Pain leads to over-conservatism, which leads to irrelevance
- Example: YouTube
  - Monolithic Python application
  - Manual regression testing (50 hours)
  - Requires release volunteers (lack of automation)
    - Burnout leads to loss of expertise
- Solution: use microservices?
- CD recommendation: don't slow down; speed up!

Principles of Continuous Delivery (CD)

- Agility
  - Release small updates frequently
  - Faster is safer! (CD@Google)
- Automation
- Modularity
  - Isolate changes
  - Enable delegation
- Data
  - Monitor health metrics
  - Evaluate feature effectiveness
  - A/B Testing
- Rollout control
  - Phased rollouts
  - Rollbacks

- Component-based design and microservice architectures provide modularity

- Most benefit comes from being *able* to release frequently, not necessarily from actually doing it

Launch and Iterate

"You get extraordinary outcomes by realizing that the launch never lands but that it begins a learning cycle where you then fix the next most important thing, measure how it went, fix the next thing, etc.—and it is never complete."

—David Weekly, Former Google product manager

Source: SWE@Google Chapter 24

Feature flags

- Tying new features to binary builds is risky
  - Binary rollout and rollback takes time
    - Risk of version skew
    - Can't synchronize feature availability with announcement
  - Fixing a regression requires rolling back *all* new features

- Runtime flags allow more granular control
  - Faster to propagate changes
  - Can enable for arbitrary subset of users
  - Can toggle independently of other features
- Build-time flags can be used to avoid *leaks*
- Config-driven enabling/disabling