cs304
Software Engineering

TAN, Shin Hwei
陈馨慧
Southern University of Science and Technology

Slides adapted from cs427 (UIUC) and cs409 (SUSTech)

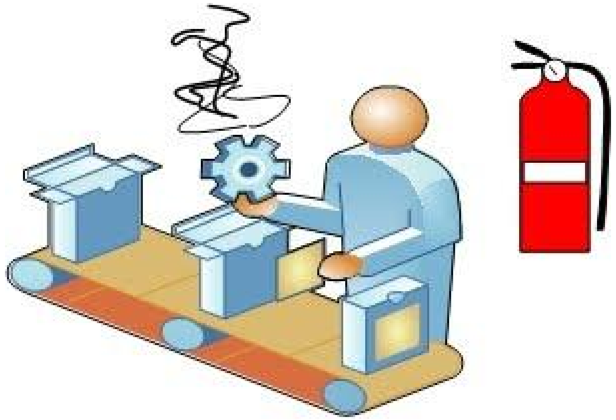
Reminder

- **Project Final Presentation Uploaded:**
  - **due on 22 May 2021, 11.59pm**
- **All lab exercises due on 24 May 2021,11.59pm**
  - **coverage lab: https://classroom.github.com/a/S8hiiL1J**
  - **junit lab(Pair programming): https://classroom.github.com/g/trrjQPYs**
  - **metrics lab: https://classroom.github.com/a/qMOJKfSm**
  - **reverse engineering lab: https://classroom.github.com/a/4Dfhej1A**
  - **ui-ci: https://classroom.github.com/a/Yuzgn5gJ**
  - **security (this week): https://classroom.github.com/a/RwgDwI38**

DevOps and Continuous Integration

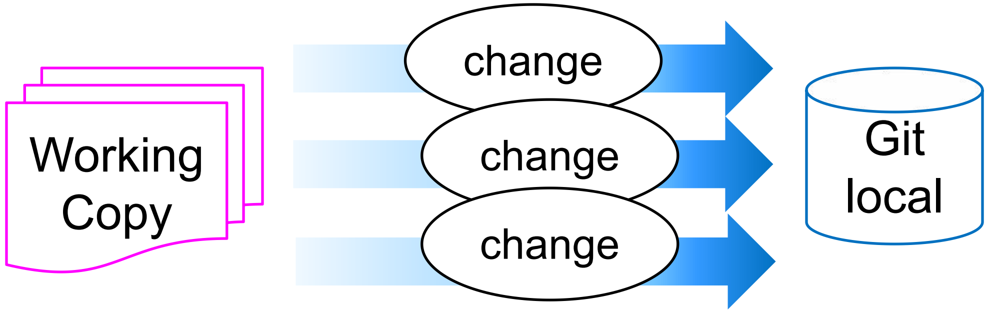
Smoke Test

- A quick set of tests run on the daily build.
  - Cover most important functionalities of the software but NOT exhaustive
  - Check whether code catches fire or “smoke” (breaks)
  - Expose integration problems earlier



Daily Commits

- Submit work to main repo at end of each day.
  - Idea: Reduce merge conflicts
  - This is the key to "continuous integration" of new code.



  - *Caution:* Don't check in faulty code (does not compile, does not pass tests) just to maintain the daily commit practice.
  - If your code is not ready to submit at end of day, either submit a coherent subset or be flexible about commit schedule.

Continuous Integration server

An external machine that automatically pulls your latest repo code and fully builds all resources.

- If anything fails, contacts your team (e.g. by email).
- Ensures that the build is never broken for long.

Examples of CI Server

First CI Server: *CruiseControl*

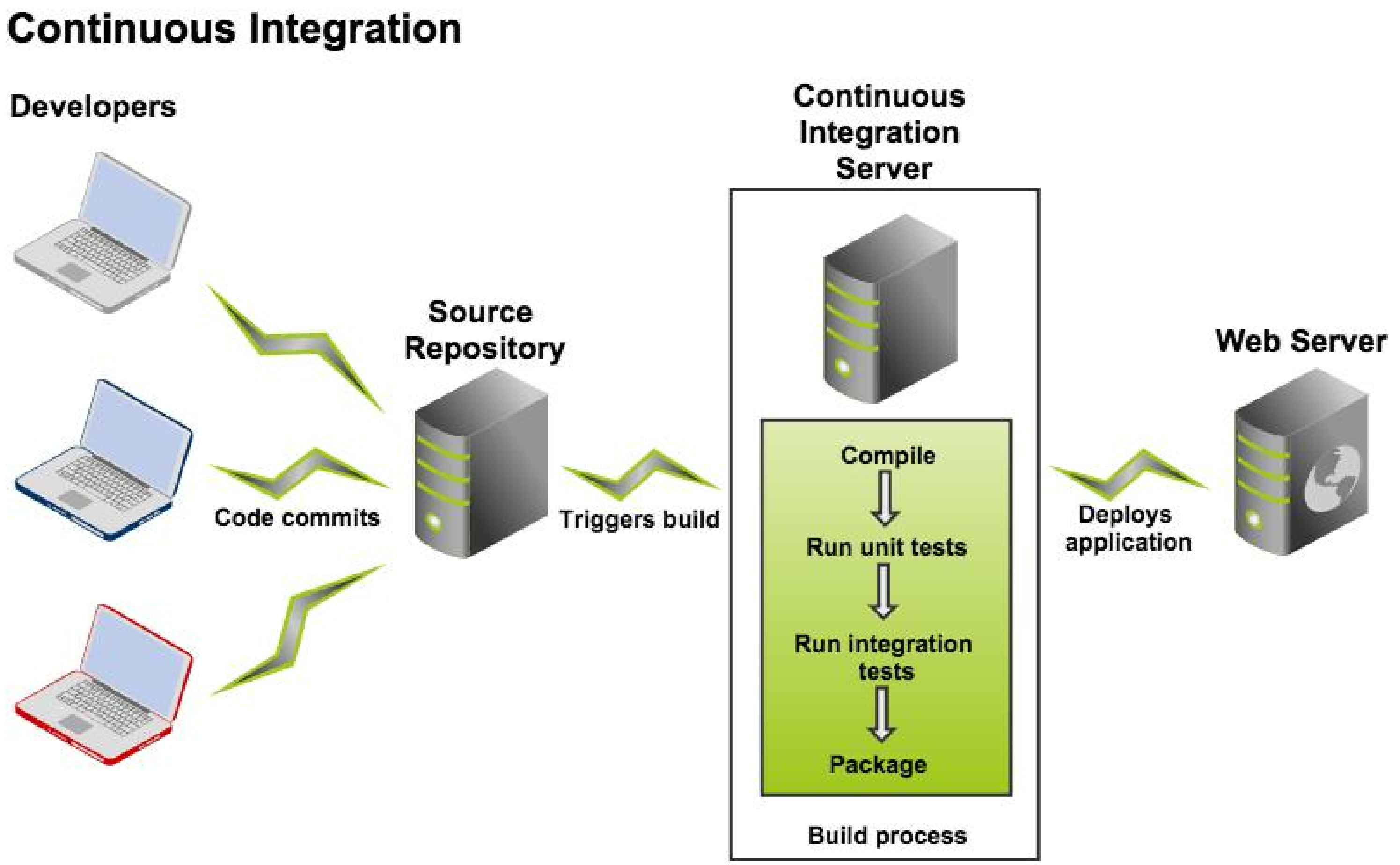
Jenkins

An extendable open source continuous integration server

Bamboo

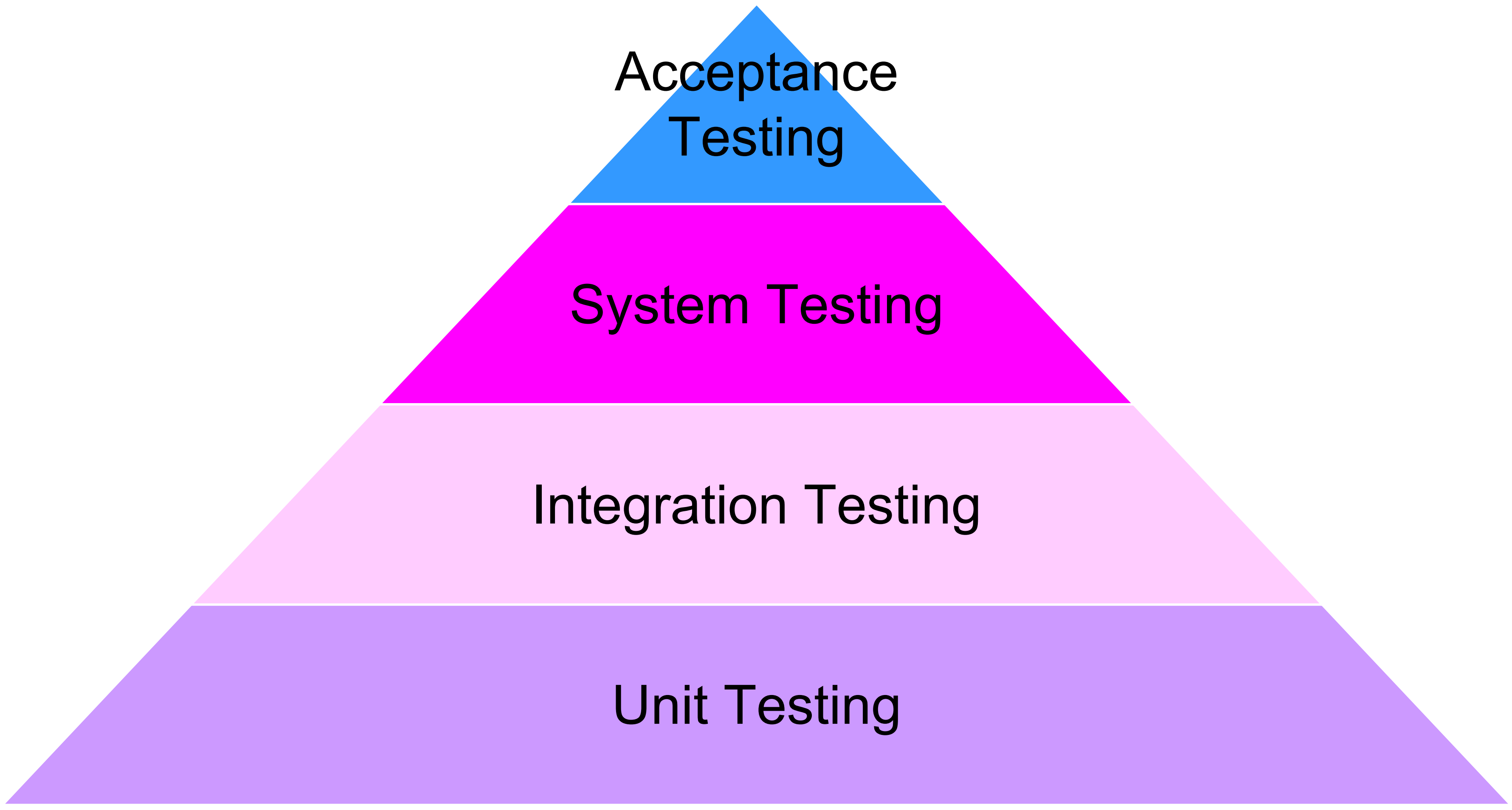
Travis CI

What happen in CI Server?



Pic from: https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html

Levels of Software Testing



Unit Testing

- Test individual units of a software
- A unit: smallest testable part of an application
  - E.g., method

```
public static double div(int x, int y){
        return x/y;
}
```

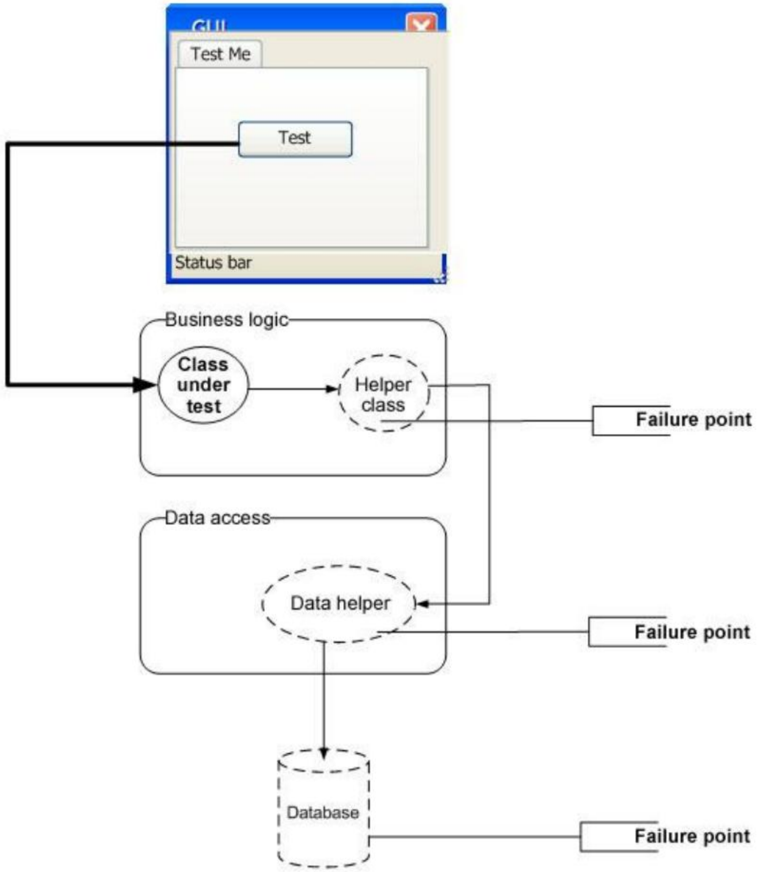
Input 1	Input 2	Output	Unit Test
1	2	0.5	`assertEquals(0.5, div(1, 2));`
1	1	1.0	`assertEquals(1.0, div(1, 1));`

1	0	ArithmeticException	`@Test(expected=java.lang.ArithmeticException.class)`
			`public void testDivideByZero() {`
			`div(1,0)`
			`}`

```
assertEquals(expected, div(1, 2));
```

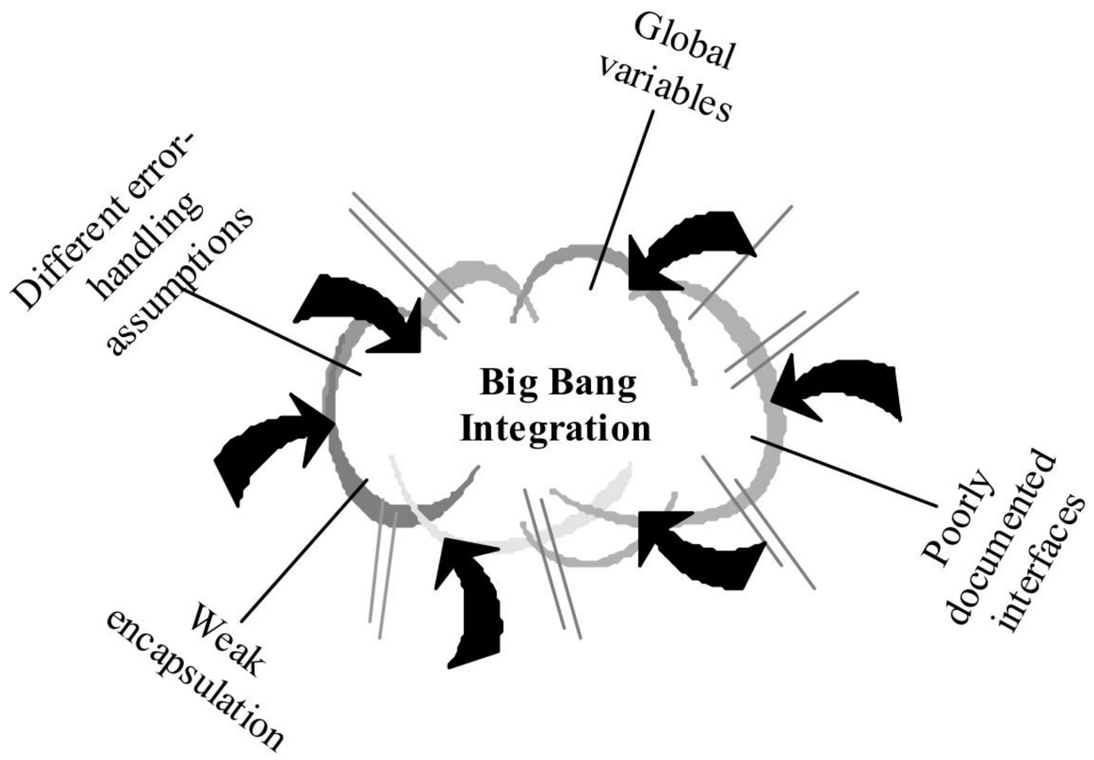
Integration testing

- **Integration testing**: Verify software quality by testing two or more dependent software modules as a group.
- Challenges:
  - Combined units can fail in more places and in more complicated ways.
  - How to test a partial system where not all parts exist?
  - How to properly simulate the behavior of unit A so as to produce a given behavior from unit B?



Big-bang Integration Testing

- *All* component are integrated together at *once*



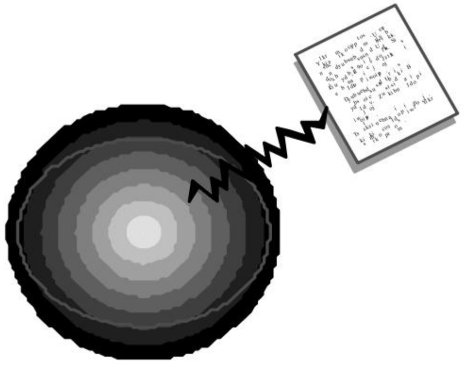
Big-bang Integration Testing

- Advantages:
  - Convenient for small systems.
- Disadvantages:
  - Finding bugs is difficult.
  - Due to large number of interfaces that need to be tested, some interfaces could be missed easily.
    - Testing team need to wait until everything is integrated so will have less time for testing.
    - High risk critical modules are not isolated and tested on priority.

Incremental Integration Testing

- **Incremental integration**:
  - Develop a functional "skeleton" system
  - Design, code, test, debug a small new piece
  - Integrate this piece with the skeleton
    - test/debug it before adding any other pieces

Big-bang Integration



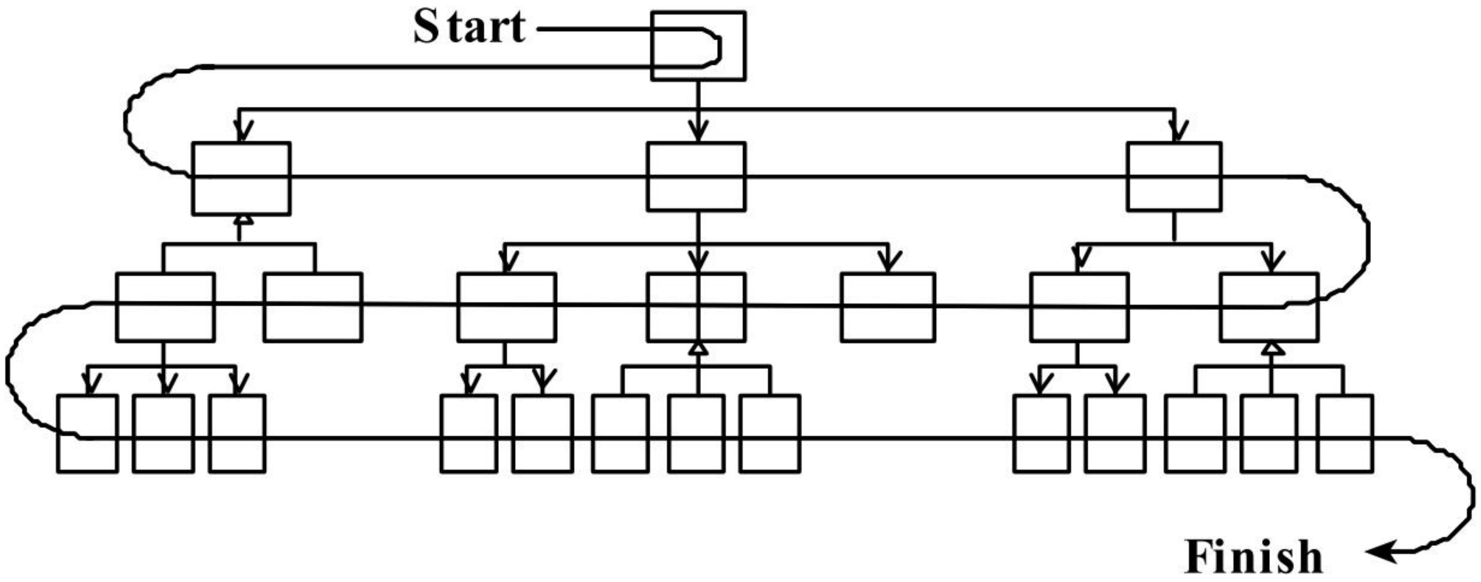
Incremental Integration

Incremental Integration Testing

- Advantages:
  - Errors easier to isolate, find, fix
    - Reduces developer bug-fixing load
  - System is always in a (relatively) working state
    - Good for customer relations, developer morale
- Disadvantages:
  - May need to create "stub" versions of some features that have not yet been integrated

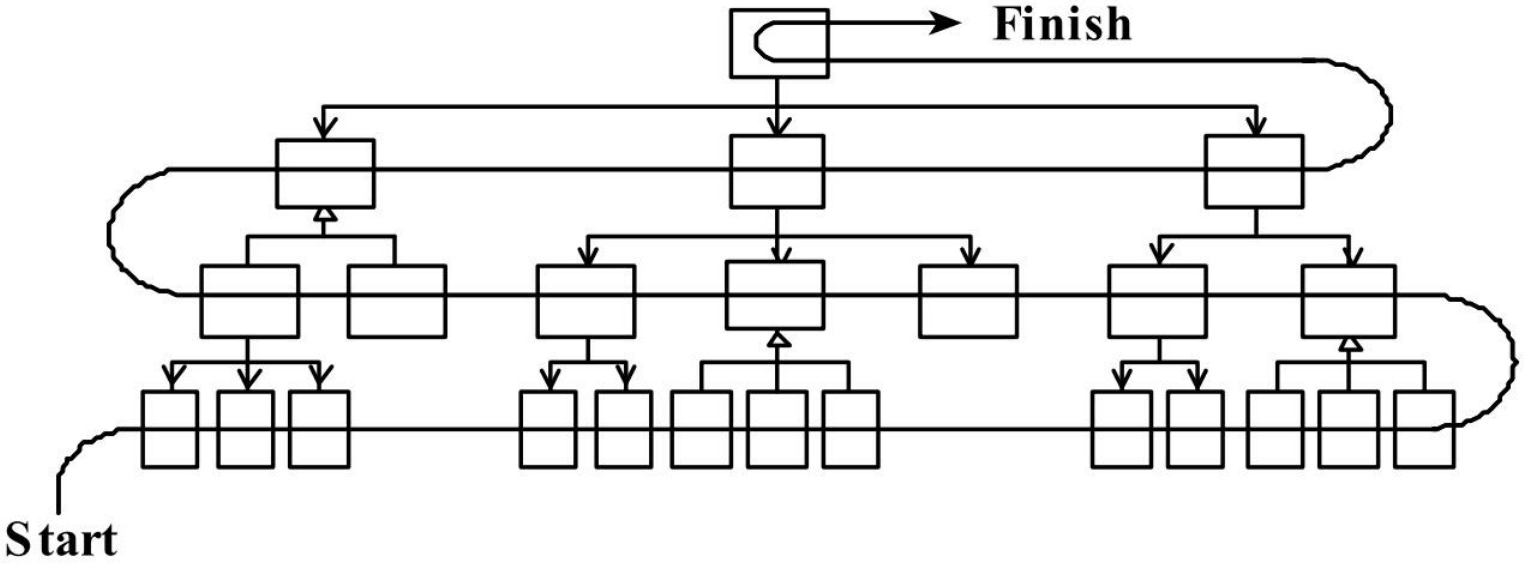
Top-down integration

- Start with outer UI layers and work inward
  - Must write (lots of) stub lower layers for UI to interact with
  - Allows postponing tough design/debugging decisions (bad?)



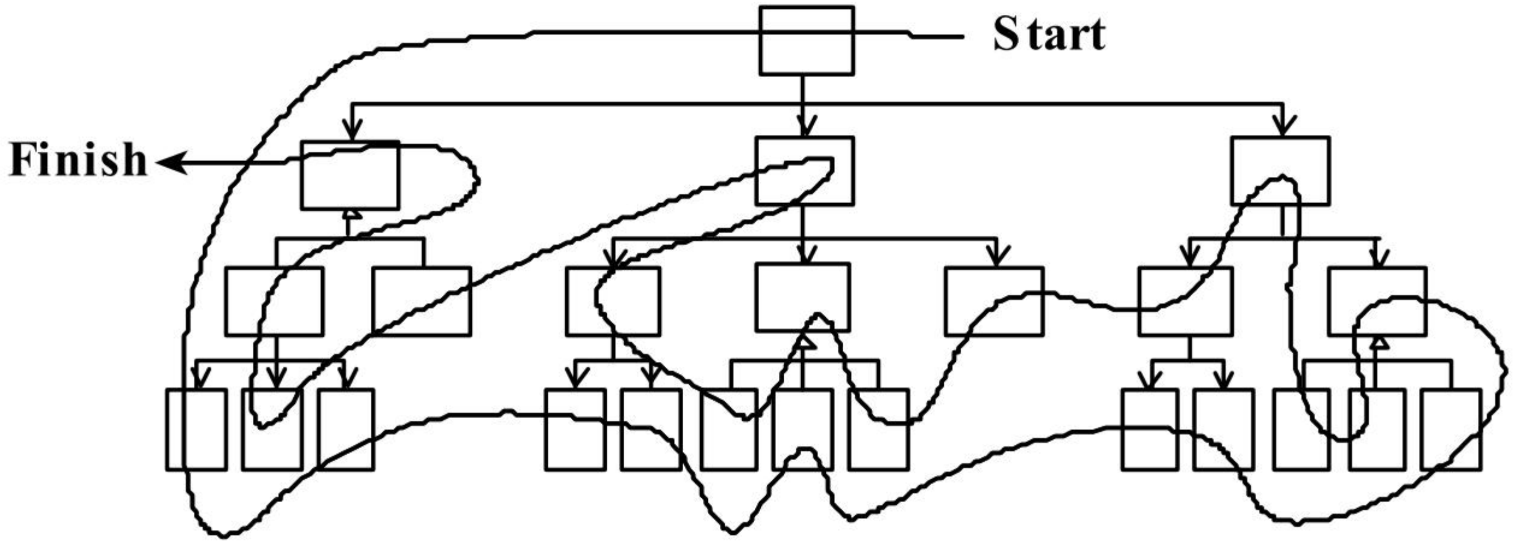
Bottom-up integration

- Start with low-level data/logic layers and work outward
  - Must write test drivers to run these layers
  - Won't discover high-level / UI design flaws until late



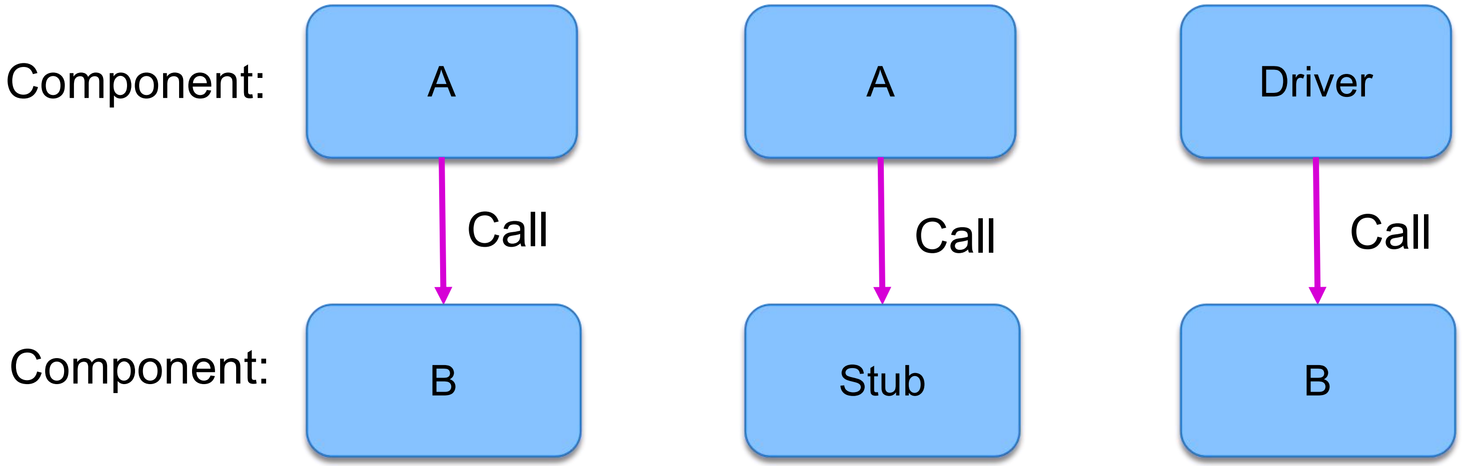
"Sandwich" integration

- Connect top-level UI with crucial bottom-level classes
  - Add middle layers later as needed
  - More practical than top-down or bottom-up?



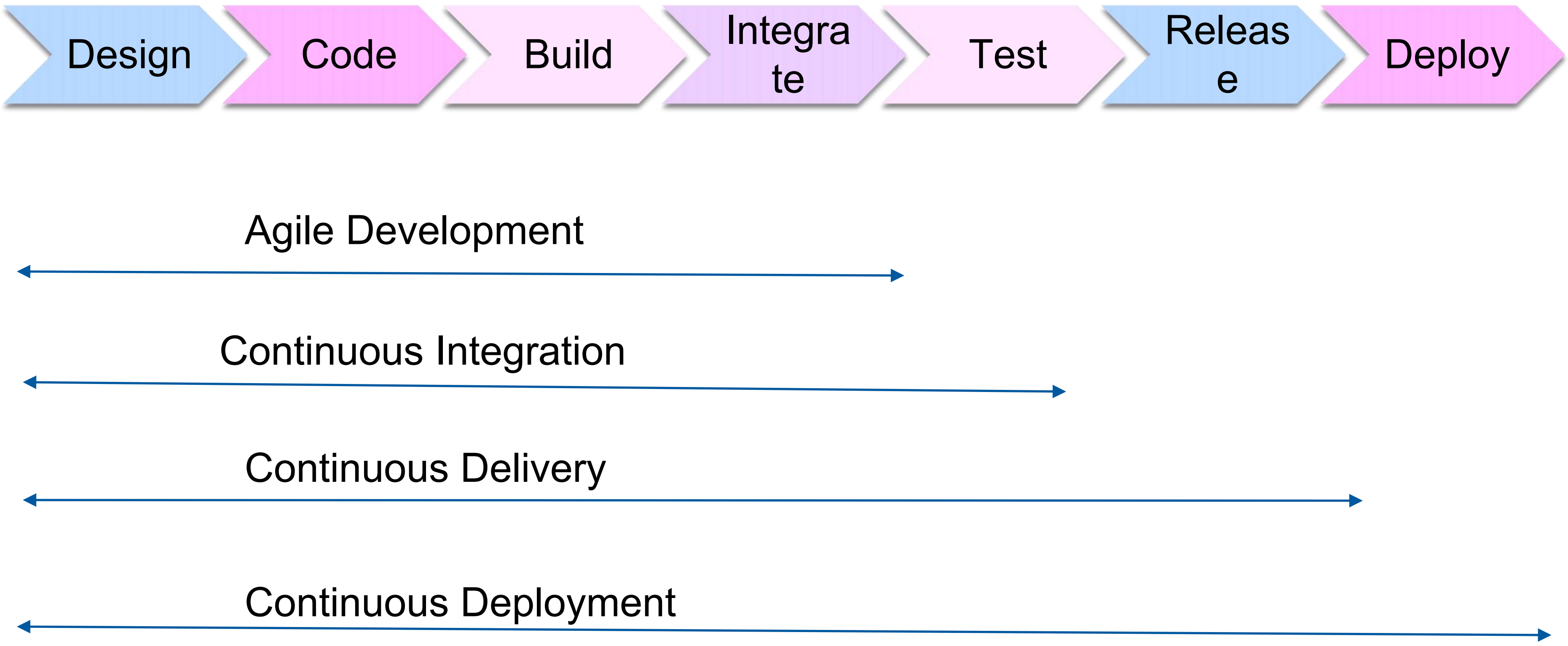
Stub versus Driver

- Both are used to replace the missing software and simulate the interface between components
  - ➢ Create dummy code



Stub: Dummy function gets called by another function
Driver: Dummy function to call another function

Continuous Integration & Continuous Deployment



Continuous Delivery?

“The essence of my philosophy to software delivery is to build software so that it is always in a state where it could be put into production. We call this ***Continuous Delivery*** because we are continuously running a *deployment pipeline* that tests if this software is in a state to be delivered.”

Jez Humble

Author of "Continuous Delivery"

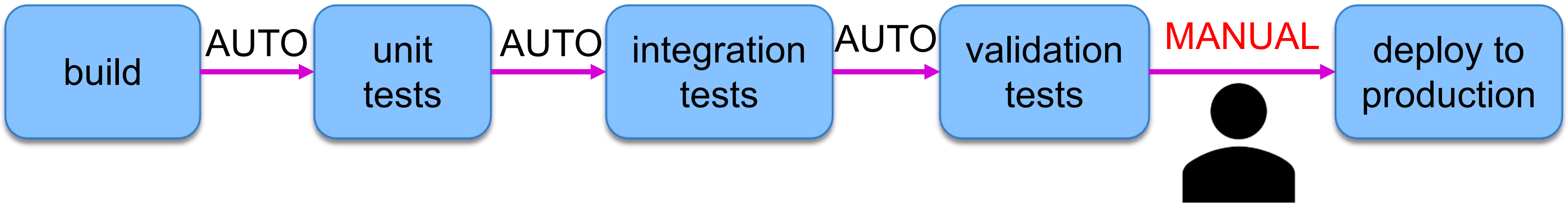
Continuous Integration != Continuous Delivery: CI != CD

- $Continous\ Delivery\ =\ CI\ +\ automated\ test\ suite$
- Not every change is a release
  - Manual trigger
  - Trigger on a key file (version)
  - Tag releases!

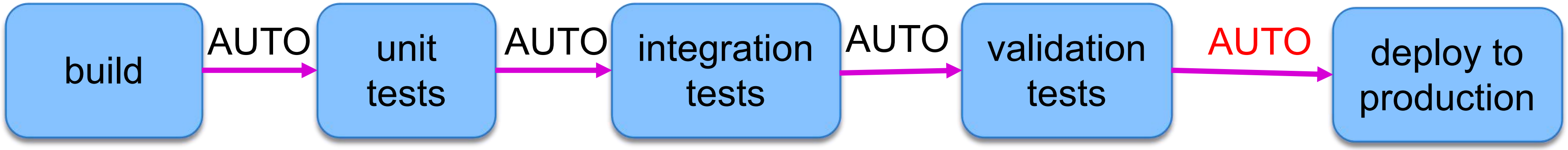
- *CD* – The key is *automated testing.*

Cont. Delivery vs. Deployment

Continuous Delivery



Continuous Deployment



Continuous Deployment

- $Continous\ Deployment\ =\ CD + Automatic\ Deployment$
- Every change that passes the automated tests is deployed to production automatically.
- Deployment Schedule:
  - Release when a feature is complete
  - Release every day
- $Continous\ Deployment\ =\ CD + Automatic\ Deployment$

Deployment strategies

Strategy 1: Zero-downtime deployment

1. Deploy version 1 of your service
2. Migrate your database to a new version
3. Deploy version 2 of your service in parallel to the version 1
4. If version 2 works fine, bring down version 1
5. Deployment Complete!

Strategy 2: Blue-green deployment

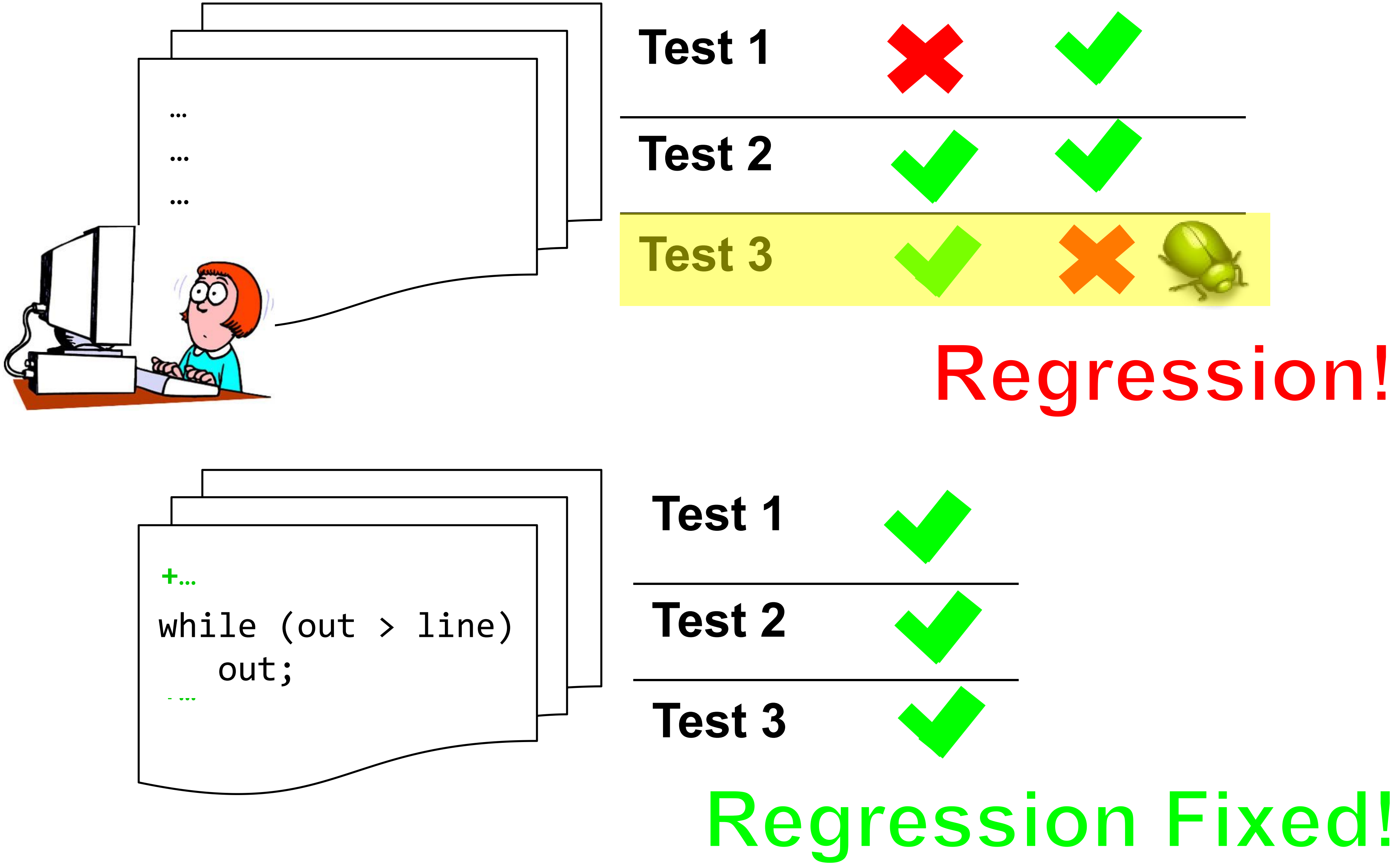
1. Maintain two copies of your production environment (“blue” and “green”)
2. Route all traffic to the blue environment by mapping production URLs to it
3. Deploy and test any changes to the application in the green environment
4. “Flip the switch”: Map URLs onto green & unmap them from blue.

Zero-downtime & Blue-green Deployment

- Advantage:
  - No outage/shut down
  - User can still use the application without downtime
- Disadvantages:
  - Needs to maintain 2 copies
  - Double the efforts required to support multiple copies
  - Migration of database may not be backward compatible

Safer Strategy: Shut down→Migrate → Deploy

Regression Testing



…
…
…
Test 1
Test 2
Test 3
Regression!
+…
while (out > line)
out;
Test 1
Test 2
Test 3
Regression Fixed!

What is Regression?

- Software undergoes changes
- But changes can both
  - improve software, adding feature and fixing bugs
  - break software, introducing new bugs
- We call such “breaking changes” regressions

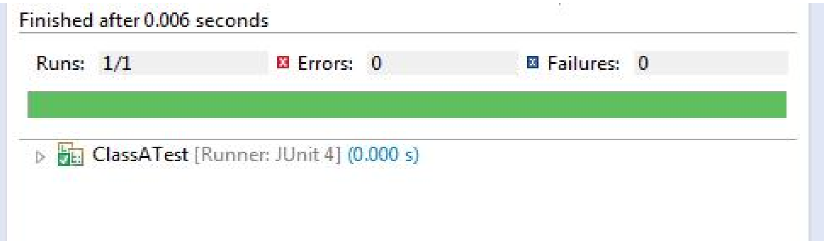
What is Regression testing?

- Testing that are performed to ensure that changes made does not break existing functionality
- It means re-running test cases from existing test suites to ensure that software changes do not introduce new faults

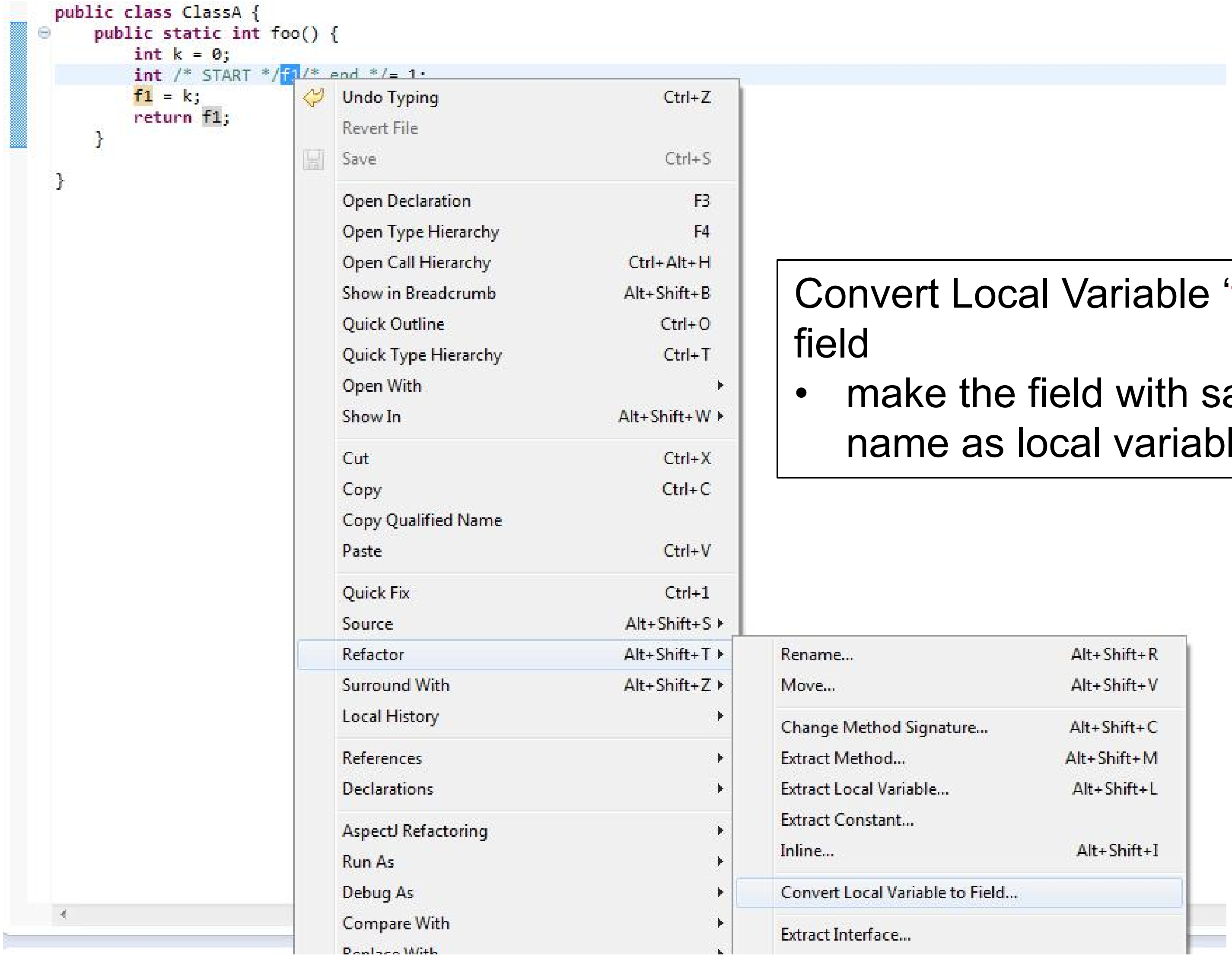
Example:
How can regression test helps in Refactoring?

Class	`public class ClassA {`
	`    public static int foo() {`
	`        int k = 0;`
	`        int /* START */f1/* end */= 1;`
	`        f1 = k;`
	`        return f1;`
	`    }`
	`}`
		• foo() returns 0 before refactoring

Test	`import static org.junit.Assert.*;`
	
	`public class ClassATest {`
	`    @Test`
	`    public void testFoo() {`
	`        assertEquals(0,ClassA.foo());`
	`    }`
	`}`
		• Existing test serves as regression test
		• Run regression test before refactoring

Test Resu lt		• All test passes!

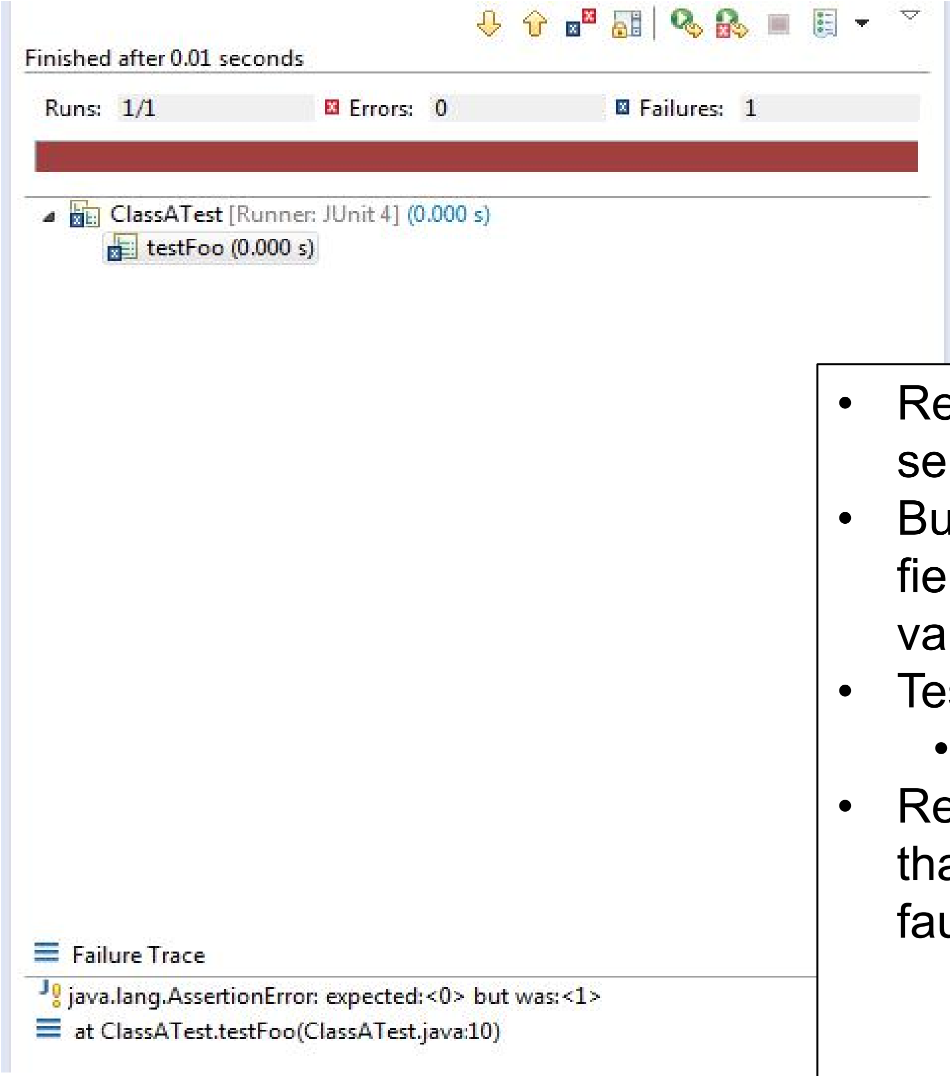
Perform Refactoring: Convert Local Variable to Field



Convert Local Variable ‘f1’ to field

- make the field with same name as local variable ‘k’

After refactoring




```java
public class ClassA {
    private static int /* START */ k;

    public static int foo() {
        int k = 0;
        k = 1;
        k = k;
        return k;
    }

}
```

- Refactoring should not change the semantic meaning of the code
- But the refactoring allows the new field to overshadow the existing local variable
- Test now fails
  - Regression occurs!
- Re-running regression test ensure that refactoring did not introduce new faults

How to fix regression bug?

What has changed?

- Do a diff against the program version that worked
  - Use version control system to see the changes between version
    - E.g., svn diff, git diff

Which changes are wrong?

- Find which changes are related to the tests failure
  - Use debugger to locate the failure inducing changes
  - E.g., Eclipse Debugger

Fix the incorrect changes

Check if regression bugs are fixed

- Run tests again to see that the fix passes all tests
- Add new tests to reflect the changes

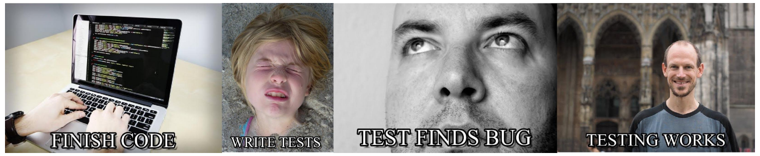
If you are interested in automated approach, read:
http://www.shinhwei.com/relifix.pdf

The State of Continuous Integration Testing @Google

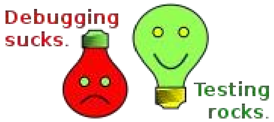
Adapted from https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45880.pdf

Testing Scale at Google

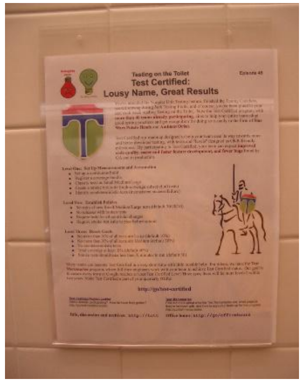
- 4.2 million individual tests running continuously
  - Testing runs before and after code submission
- 150 million test executions / day (averaging 35 runs / test / day)
- Distributed using internal version of bazel.io to a large compute farm
- Almost all testing is automated - no time for Quality Assurance
- 13,000+ individual project teams - all submitting to one branch
- Drives continuous delivery for Google
- 99% of all test executions pass



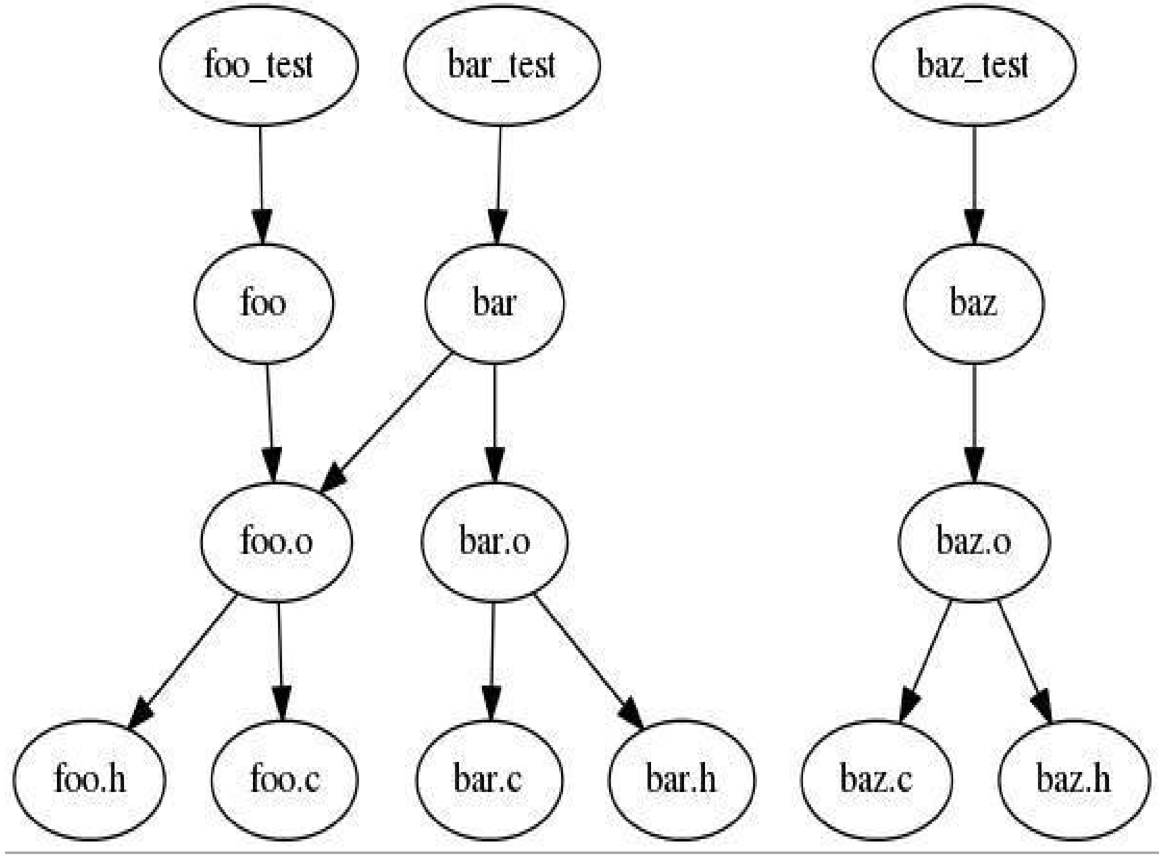
Testing Culture @Google



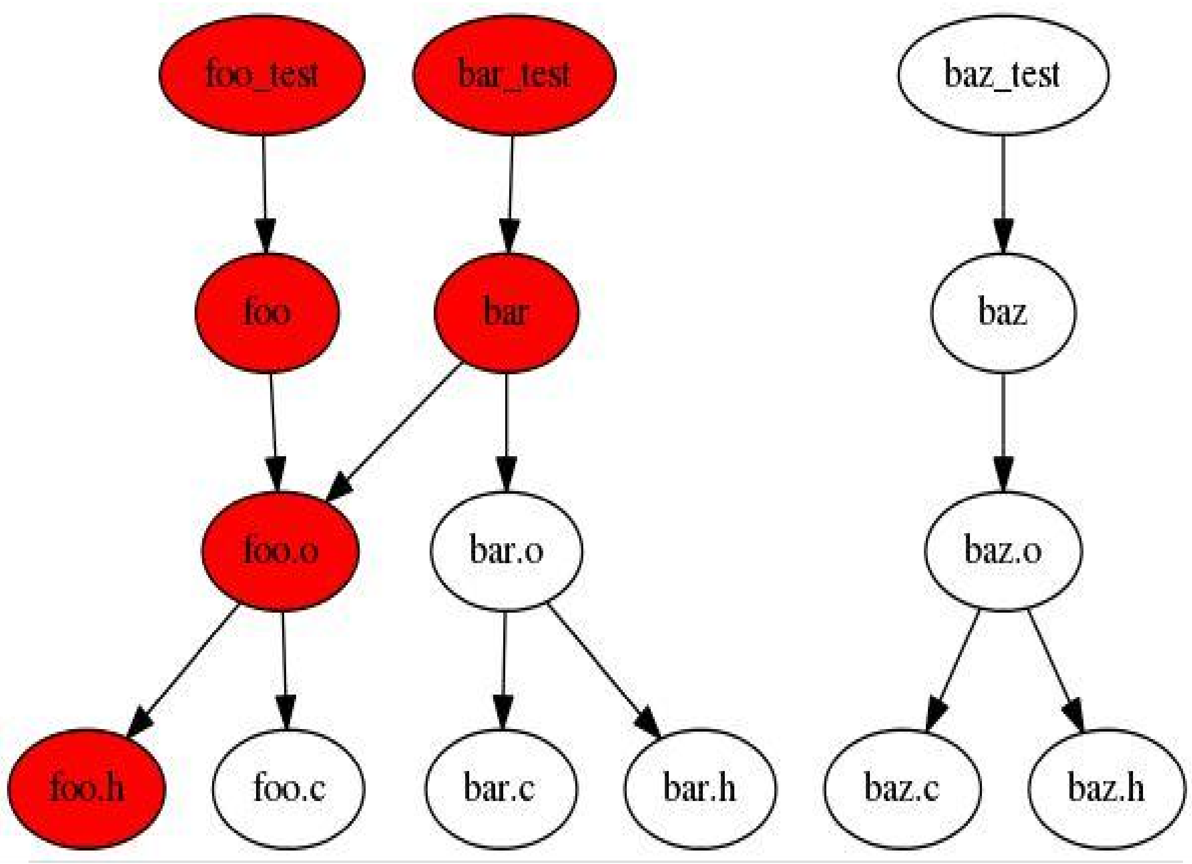
- ~10 Years of testing culture promoting hand-curated automated testing
  - Testing on the toilet and Google testing blog started in 2007
  - GTAC conference since 2006 to share best practices across the industry
  - Part of our new hire orientation program
- SETI role
  - Usually 1-2 SETI engineers / 8-10 person team
  - Develop test infrastructure to enable testing
- Engineers are expected to write automated tests for their submissions
- Limited experimentation with model-based / automated testing
  - Fuzzing, UI waltkthroughs, Mutation testing, etc.
  - Not a large fraction of overall testing



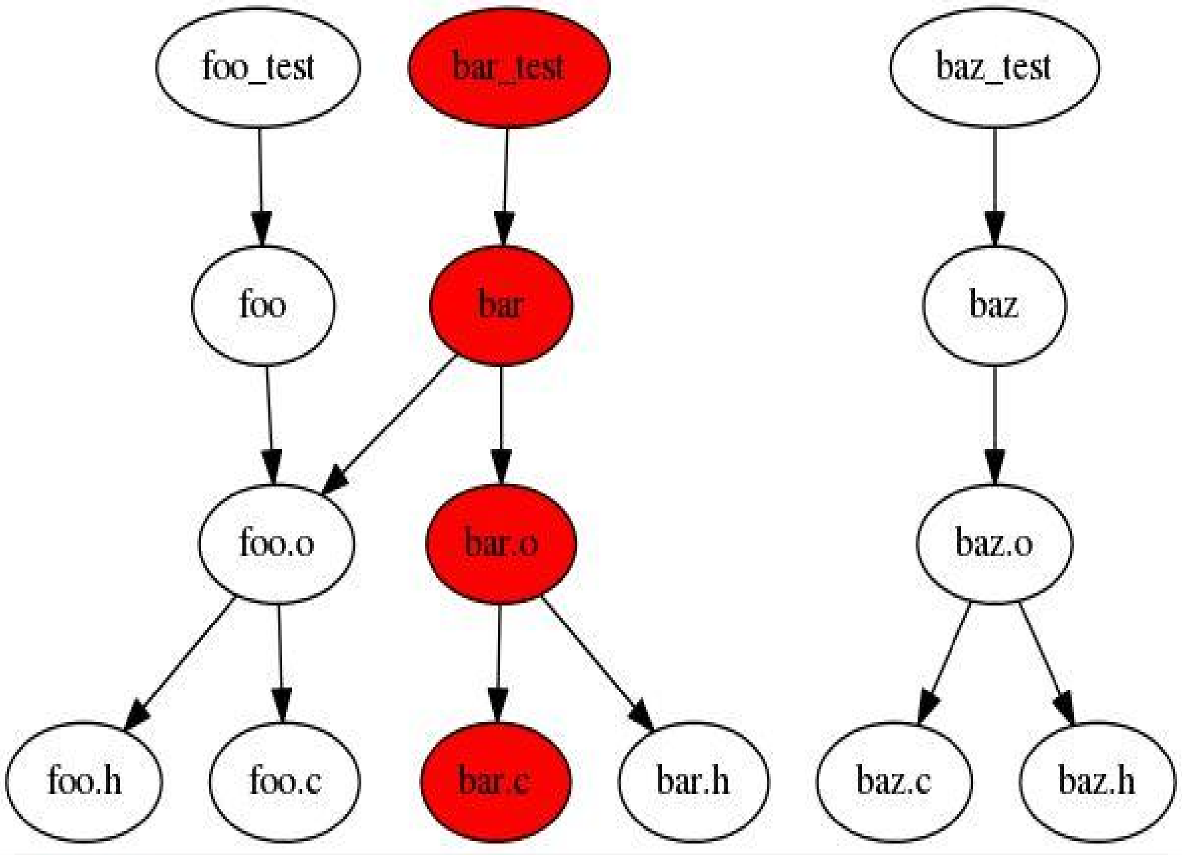
Regression Test Selection (RTS)



Regression Test Selection (RTS)



Regression Test Selection (RTS)



Presubmit Testing

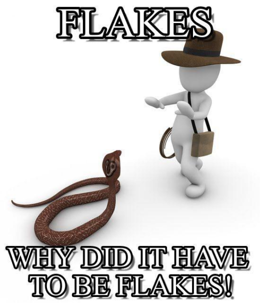
- Uses fine-grained dependencies
- Uses same pool of compute resources
- Avoids breaking the build
- Captures contents of a change and tests in isolation
  - Tests against HEAD
- Integrates with
  - submission tool - submit iff testing is green
  - Code Review Tool - results are posted to the review

Example Presubmit Display

Postsubmit testing

- Continuously runs 4.2M tests as changes are submitted
  - A test is affected iff a file being changed is present in the transitive closure of the test dependencies. (Regression Test Selection)
  - Each test runs in 2 distinct flag combinations (on average)
  - Build and run tests concurrently on distributed backend.
  - Runs as often as capacity allows
- Records the pass / fail result for each test in a database
  - Each run is uniquely identified by the test + flags + change
  - We have 2 years of results for all tests
  - And accurate information about what was changed

Analysis of Test Results at Google



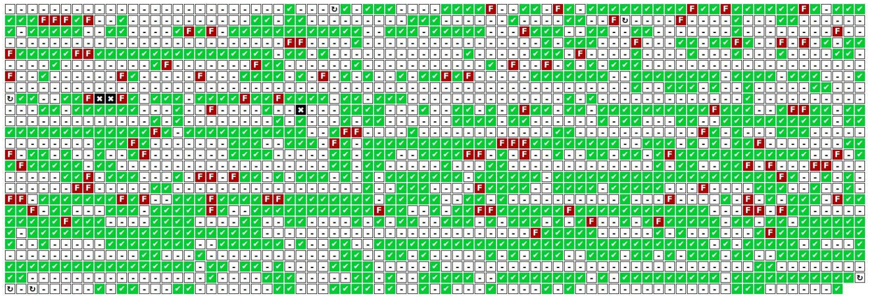
- Analysis of a large sample of tests (1 month) showed:
  - 84% of transitions from Pass -> Fail are from "flaky" tests
  - Only 1.23% of tests ever found a breakage
  - Frequently changed files more likely to cause a breakage
  - 3 or more developers changing a file is more likely to cause a breakage
  - Changes "closer" in the dependency graph more likely to cause a breakage
  - Certain people / automation more likely to cause breakages (oops!)
  - Certain languages more likely to cause breakages (sorry)

Problems of CI in Google: Flaky Tests

- Flaky Tests
  - Test which could fail or pass for the same code
- Sources of test flakiness:
  - Concurrency
  - Environment / setup problems
  - Non-deterministic or undefined behaviors

Flaky Tests

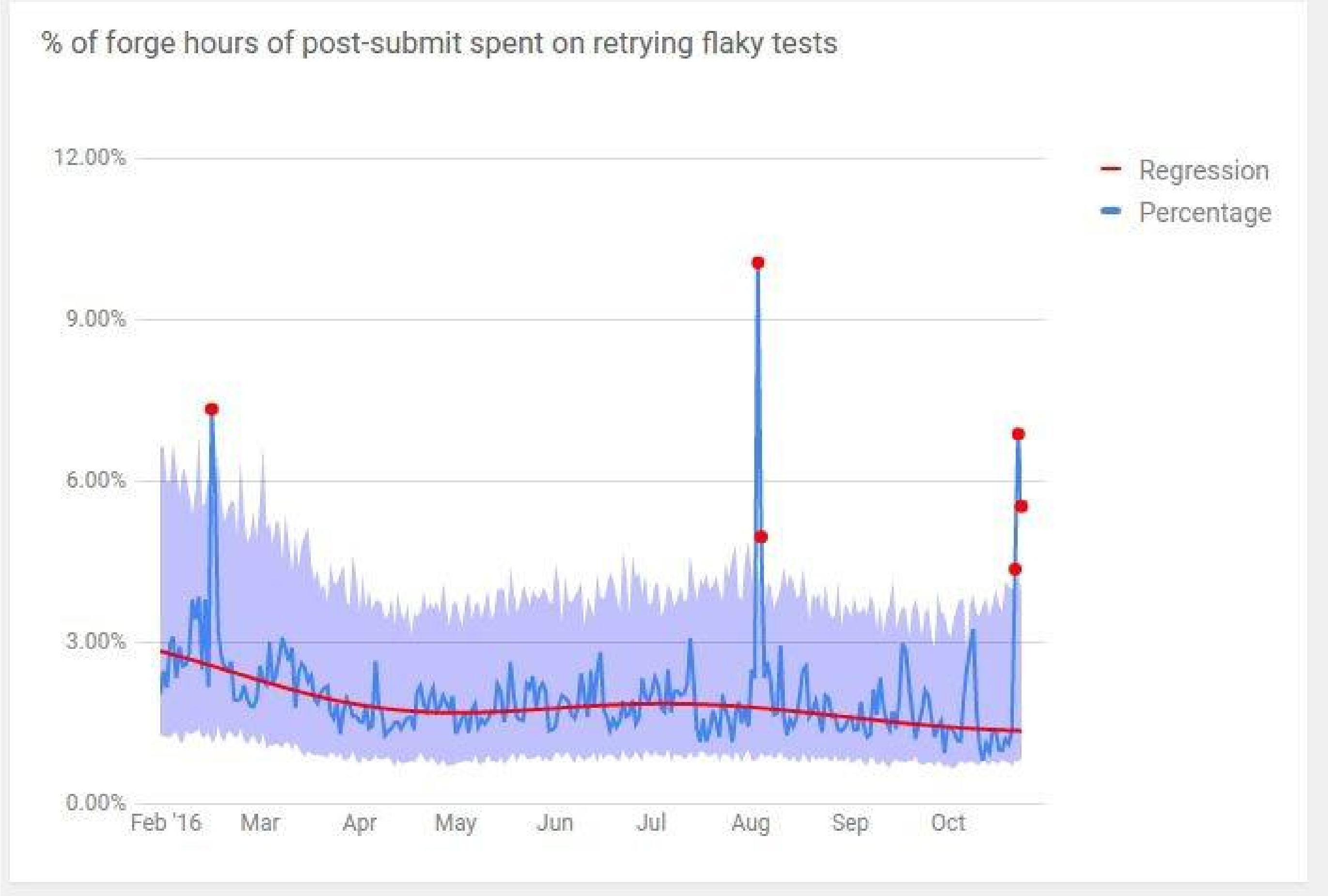
- Test Flakiness is a huge problem
- Flakiness is a test that is observed to both Pass and Fail with the same code
- Almost 16% of our 4.2M tests have some level of flakiness
- Flaky failures frequently block and delay releases
- Developers ignore flaky tests when submitting - sometimes incorrectly
- We spend between 2 and 16% of our compute resources re-running flaky tests



Flaky test impact on project health

- Many tests need to be aggregated to qualify a project
- Probability of flake aggregates as well
- Flakes
  - Consume developer time investigating
  - Delay project releases
  - Waste compute resources re-running to confirm flakes

Percentage of resources spent re-running flakes



Sources of Flakiness

Factors that causes flakiness

- Async Wait
- Concurrency
- Test Order Dependency
- Resource Leak
- Network
- Time
- IO
- Randomness
- Floating Point Operations
- Unordered Collections

See: https://pdfs.semanticscholar.org/02da/46889ee3c6bc44bfa0fc45071195781b99ce.pdf

Example of flaky tests

```java
@Test
public void testRsReportsWrongServerName() throws Exception {
MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
MiniHBaseClusterRegionServer firstServer =
(MiniHBaseClusterRegionServer)cluster.getRegionServer(0);
HServerInfo hsi = firstServer.getServerInfo();
firstServer.setHServerInfo(...);

// Sleep while the region server pings back
Thread.sleep(2000);
assertTrue(firstServer.isOnline());
assertEquals(2,cluster.getLiveRegionServerThreads().size());
... // similarly for secondServer
}
```

- Test fails if the server does not respond fast enough, e.g., because of thread scheduling or network delay

Flakes are Inevitable

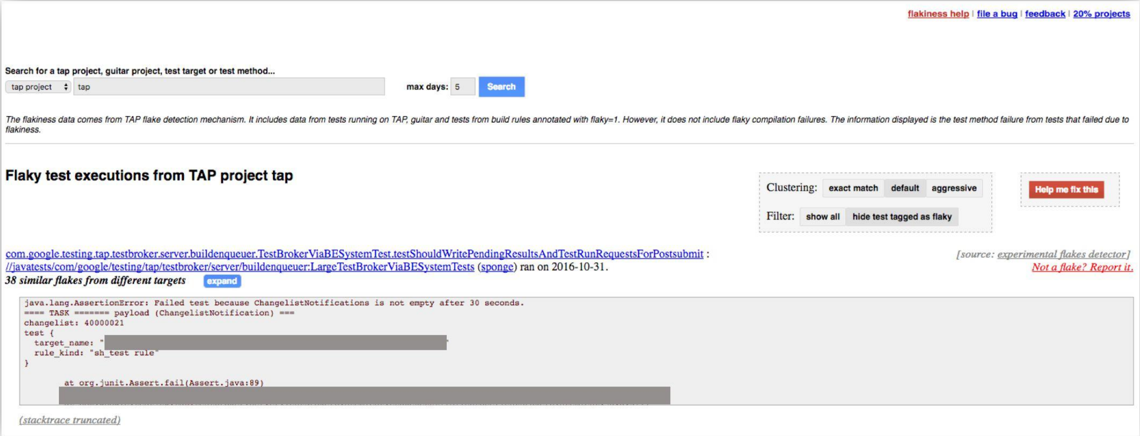
- Continual rate of 1.5% of test executions reporting a "flaky" result
- Despite large effort to identify and remove flakiness
  - Targeted "fixits"
  - Continual pressure on flakes
- Observed insertion rate is about the same as fix rate

Conclusion: Testing systems must be able to deal with a certain level of flakiness. Preferably minimizing the cost to developers

Flaky Test Infrastructure

- We re-run test failure transitions (10x) to verify flakiness
  - If we observe a pass the test was flaky
  - Keep a database and web UI for "known" flaky tests

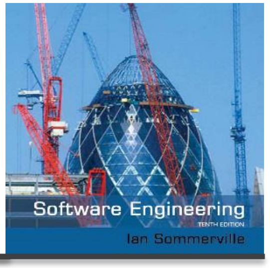
le



Continuous Integration at Google Scale

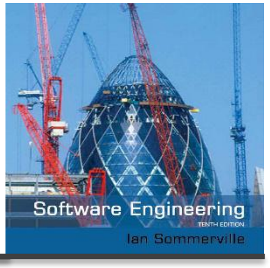
- 5000+ projects under active development
- 17k submissions per day (1 every 5 seconds)
- 20+ sustained code changes per minute
- 50% of code changes monthly
- 100+ million test cases run per day

Taken From: https://www.eclipsecon.org/2013/node/1251.html



Chapter 13 – Security Engineering

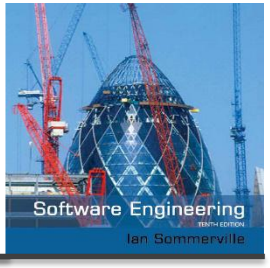
Slides Adapted from Software Engineering Book
by Ian Sommerville



Topics covered

- ✧ Security and dependability
- ✧ Security and organizations
- ✧ Security requirements
- ✧ Secure systems design
- ✧ Security testing and assurance

Security engineering



- ✧ Tools, techniques and methods to support the development and maintenance of systems that can resist malicious attacks that are intended to damage a computer-based system or its data.
- ✧ A sub-field of the broader field of computer security.

Security dimensions

✧ *Confidentiality*

- Information in a system may be disclosed or made accessible to people or programs that are not authorized to have access to that information.

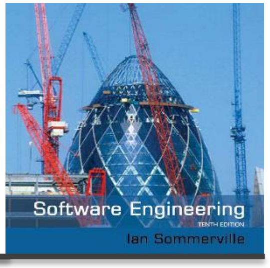
✧ *Integrity*

- Information in a system may be damaged or corrupted making it unusual or unreliable.

✧ *Availability*

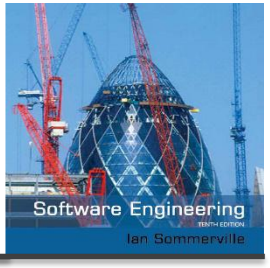
- Access to a system or its data that is normally available may not be possible.

Security levels

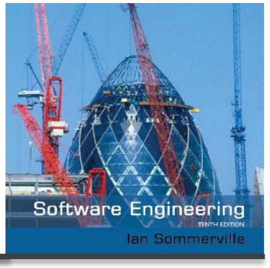


- Infrastructure security, which is concerned with maintaining the security of all systems and networks that provide an infrastructure and a set of shared services to the organization.
- Application security, which is concerned with the security of individual application systems or related groups of systems.
- Operational security, which is concerned with the secure operation and use of the organization’s systems.

System layers where security may be compromised

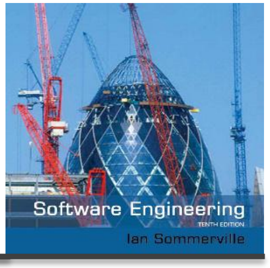


Application	
Reusable components and libraries	
Middleware	
Database management	
Generic, shared applications (browsers, e--mail, etc)	
Operating System	
Network	Computer hardware



Application/infrastructure security

- Application security is a software engineering problem where the system is designed to resist attacks.
- Infrastructure security is a systems management problem where the infrastructure is configured to resist attacks.
- The focus of this chapter is application security rather than infrastructure security.



System security management

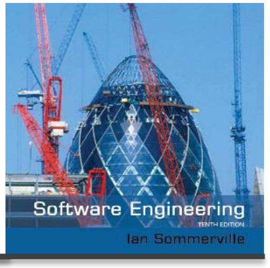
- ✧ User and permission management
  - Adding and removing users from the system and setting up appropriate permissions for users
- ✧ Software deployment and maintenance
  - Installing application software and middleware and configuring these systems so that vulnerabilities are avoided.
- ✧ Attack monitoring, detection and recovery
  - Monitoring the system for unauthorized access, design strategies for resisting attacks and develop backup and recovery strategies.

Operational security

- ✧ Primarily a human and social issue
- ✧ Concerned with ensuring the people do not take actions that may compromise system security
  - E.g. Tell others passwords, leave computers logged on
- ✧ Users sometimes take insecure actions to make it easier for them to do their jobs
- ✧ There is therefore a trade-off between system security and system effectiveness.

Security and dependability

Security

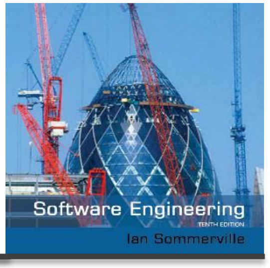


- The security of a system is a system property that reflects the system's ability to protect itself from accidental or deliberate external attack.
- Security is essential as most systems are networked so that external access to the system through the Internet is possible.
- Security is an essential pre-requisite for availability, reliability and safety.

Fundamental security

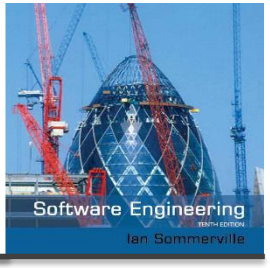
- ✧ If a system is a networked system and is insecure then statements about its reliability and its safety are unreliable.
- ✧ These statements depend on the executing system and the developed system being the same. However, intrusion can change the executing system and/or its data.
- ✧ Therefore, the reliability and safety assurance is no longer valid.

Security terminology



Term	Definition
Asset	Something of value which has to be protected. The asset may be the software system itself or data used by that system.
Attack	An exploitation of a system’s vulnerability. Generally, this is from outside the system and is a deliberate attempt to cause some damage.
Control	A protective measure that reduces a system’s vulnerability. Encryption is an example of a control that reduces a vulnerability of a weak access control system
Exposure	Possible loss or harm to a computing system. This can be loss or damage to data, or can be a loss of time and effort if recovery is necessary after a security breach.
Threat	Circumstances that have potential to cause loss or harm. You can think of these as a system vulnerability that is subjected to an attack.
Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm.

Examples of security terminology (Mentcare)



Term	Example
Asset资产	The records of each patient that is receiving or has received treatment.
Exposure暴露	Potential financial loss from future patients who do not seek treatment because they do not trust the clinic to maintain their data. Financial loss from legal action by the sports star. Loss of reputation.
Vulnerability漏洞	A weak password system which makes it easy for users to set guessable passwords. User ids that are the same as names.
Attack攻击	An impersonation of an authorized user.
Threat威胁	An unauthorized user will gain access to the system by guessing the credentials (login name and password) of an authorized user.
Control控制	A password checking system that disallows user passwords that are proper names or words that are normally included in a dictionary.