cs304
Software Engineering

TAN, Shin Hwei

陈馨慧

Southern University of Science and Technology

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

Administrative Info

- **Project Final Presentation Uploaded:**
  - **due on 22 May 2021, 11.59pm**
- All lab exercises due on 24 May 2021,11.59pm

UI Design

Recap: 10 rules of Good UI Design

1. Make Everything the User Needs Readily Accessible
2. Be Consistent
3. Be Clear
4. Give Feedback
5. Use Recognition, Not Recall
6. Choose How People Will Interact First
7. Follow Design Standards
8. Elemental Hierarchy Matters
9. Keep Things Simple
10. Keep Your Users Free & In Control

https://www.elegantthemes.com/blog/resources/10-rules-of-good-ui-design-to-follow-on-every-web-design-project

UI evaluation

☐ Be purposeful

- Decide on purpose of evaluation
  - "Is this menu confusing?"
  - "Can someone start using the system without reading a manual?"
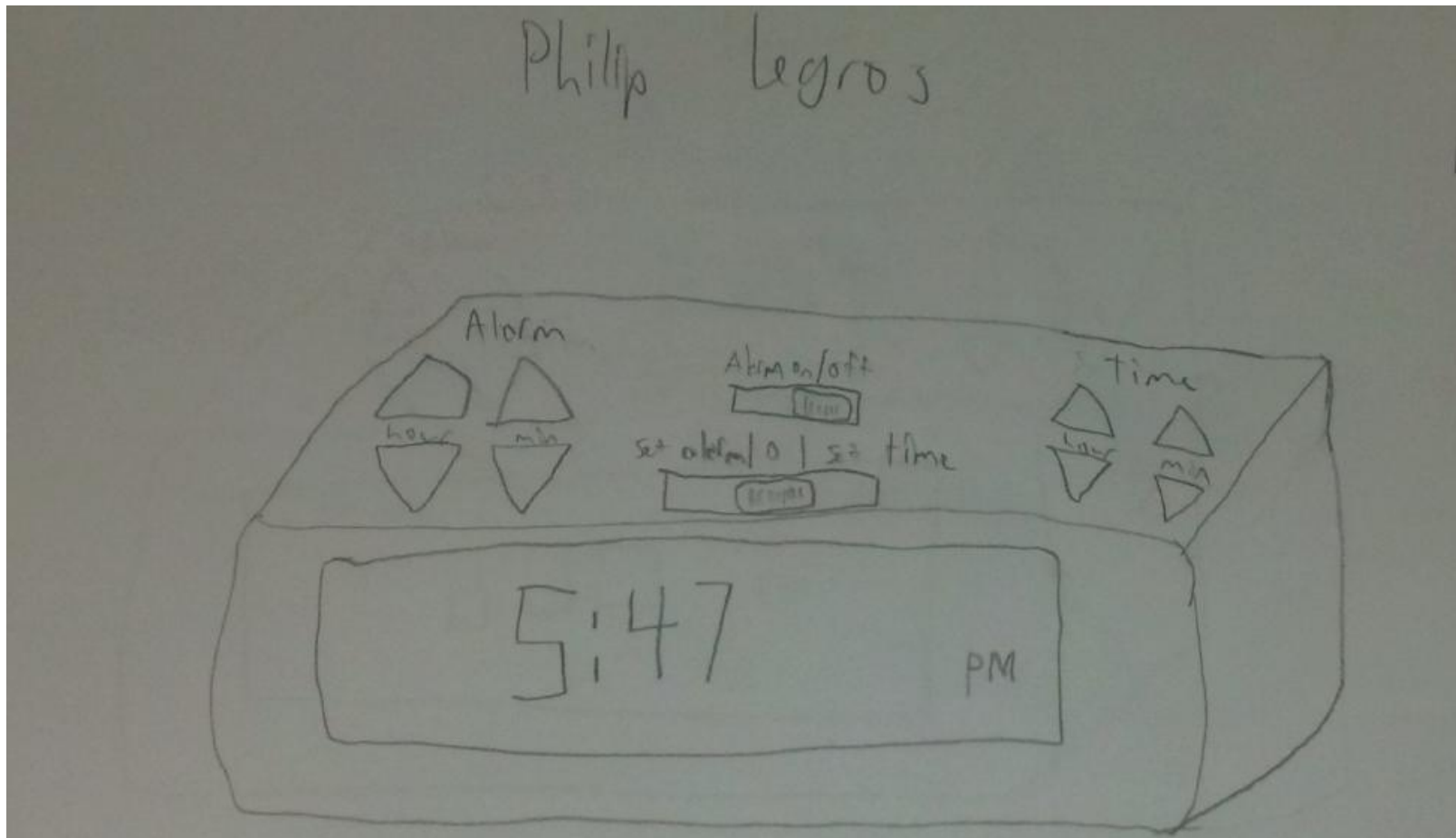- Choose tasks
- Make goals and measure to see if goals are met

Size of evaluation

☐Statistically valid sample maybe: 20-100

☐Most common size: 5

☐Purpose is to invent good UI

☐Perform evaluations after every iteration

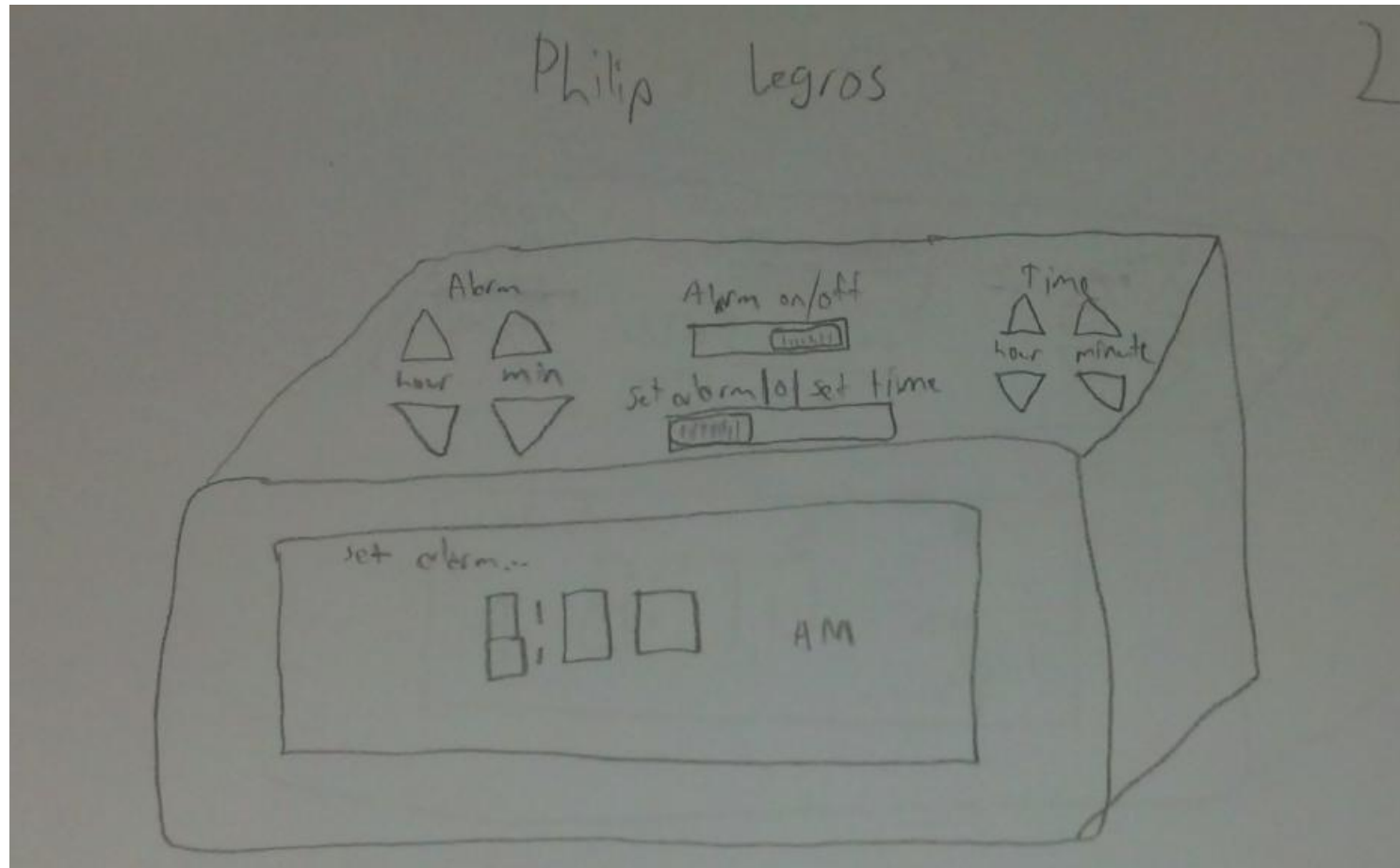# Alarm Clock Prototype - 1

1: Default Clock State   Alarm off, time set to 5:47PM.

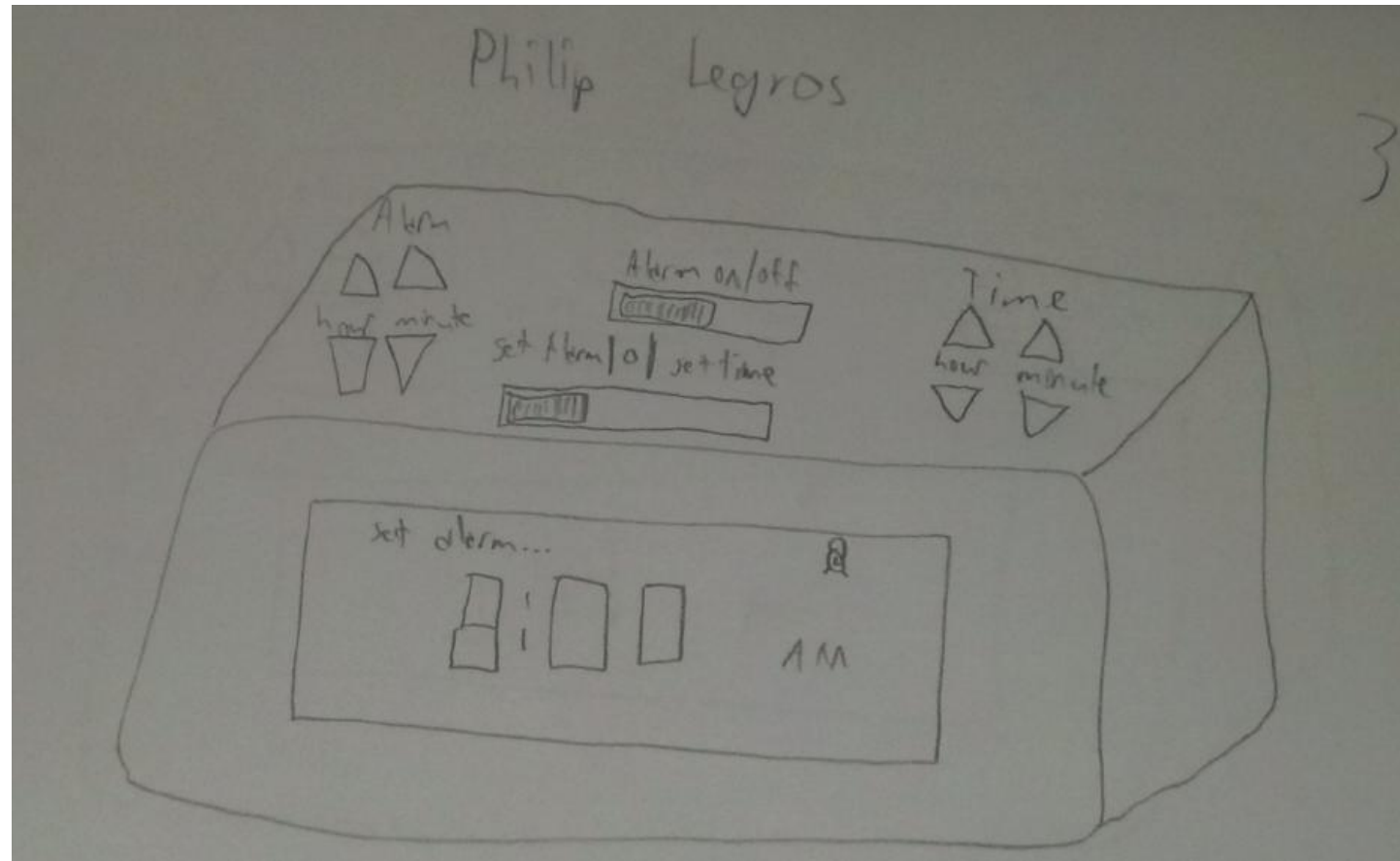# Alarm Clock Prototype - 2

2: Set Alarm

User sets alarm to 8AM.

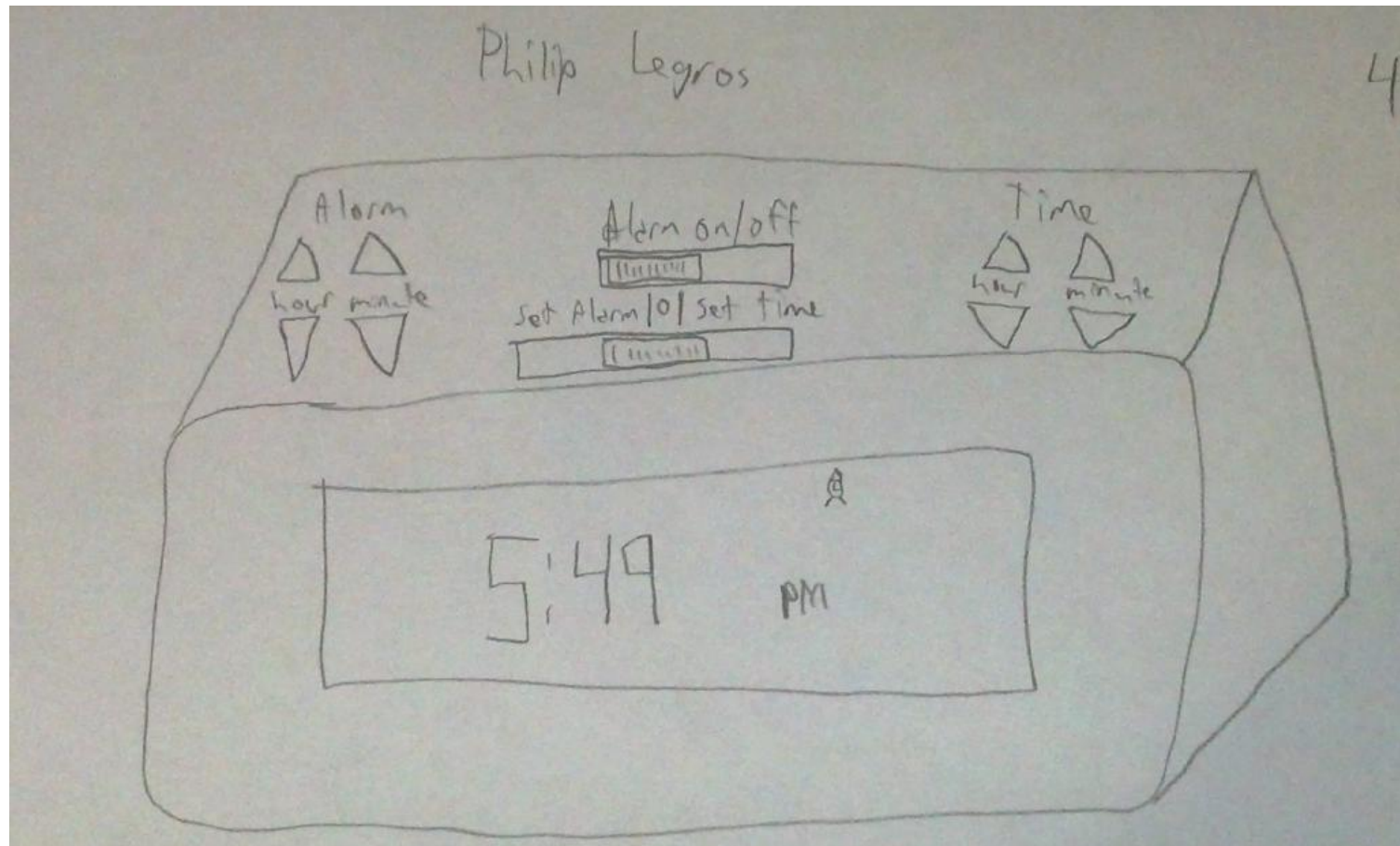# Alarm Clock Prototype - 3

3: Activate Alarm

User turns on the alarm.

Alarm Clock Prototype - 4

4: Return to Display

User returns to current time mode.

Group Exercise: Paper Prototyping

☐ Get into group of 2 students
  - or 3 if you there are an odd number of students in the lecture

☐ **Make a low-fidelity UI prototype of an alarm clock smartphone app OR part of your own project (6 mins)**
  - Each of you should draw your own alarm clock at the same time; don't discuss it with your partner yet

☐ **Then simulate your prototype (4 mins)**, acting as the phone, while your partner acts as user. Use these tasks:
  - Is the alarm set to wake me up at 9am?
  - Suppose not; set the alarm to wake me up at 9am
  - Set the current time one hour backward for a daylight savings time switch

☐ Then switch roles, so that the other person acts as the phone simulating their own prototype on you

Microsoft Customer Experience Improvement Program (CEIP)

- ☐ Multiple channels for user feedback
  - Usability test, surveys, focus groups & other field studies
  - Limited customer base
- ☐ CEIP
  - Providing all customers with ability to contribute to the design and development of Microsoft products
  - Voluntary participation
  - Anonymous

CEIP data

☐ Usage
- How software is used
- Examples: general feature usage, commands on Ribbon, actions taken in wizards, etc.

☐ Reliability and performance
- Whether software performs as expected
- Examples: assertions for logical inconsistency, measuring execution speed, etc.

☐ Hardware/software configuration
- Providing context for data interpretation
- Example: long document loading time only on machines with low RAM or a particular processor speed?

Questions answered by usage data

☐ **Command usage**
- How frequently is it used? *[Prominence on UI]*
- How many people use it? *[Impact]*
- What is the most frequent way of accessing it? *[Ease of access]*
- Does this command occur as part of a clear workflow? *[Better support]*

☐ **Feature usage**
- How many files contain a Table? *[Impact]*
- How big is the average Table? *[Optimization choice]*
- What are the most frequently used Table styles? *[Design choices]*
- What other features are used in files containing Tables? *[interaction with other features]*

Design alternatives

☐ Novice users
- Menus
- Make it look like something else
- Simple

☐ Expert users
- Commands
- Specialize to make users efficient
- Powerful

Design alternatives

☐ Standard IO vs. new IO

☐ Existing metaphors/idioms vs. new metaphors/idioms

☐ Narrow market vs. broad market

Implementation concerns

☐Simplicity

☐Safety

☐Use standard libraries/toolkits

☐Separate UI from application
- Model-View-Controller (MVC)
- Three-tier: presentation, application, data

1. Simplicity

☐ Tradeoff between number of features and simplicity

☐ Don't compromise usability for function

☐ A well-designed interface fades into the background

☐ Basic functions should be obvious

☐ Advanced functions can be hidden

Make controls obvious & intuitive

☐ Is the trash-can obvious and intuitive?

☐ Are tabbed dialog boxes obvious and intuitive?

☐ Is a mouse obvious and intuitive?

2. Safety

☐ Make actions predictable and reversible

☐ Each action does one thing

☐ Effects are visible
   - User should be able to tell whether operation has been performed

☐ Undo

3. Use standard libraries

☐ Don't build your own!
- If necessary, add to it, but try to use standard parts instead of building your own

☐ Provide familiar controls

☐ Provide consistency

☐ Reduce cost of implementation

☐ Library designers probably better UI designers than you are

When to build your own

- ☐ You are a platform provider or
- ☐ You have special needs and a lot of money and

- ☐ You are not in a hurry and
- ☐ You know what you are doing

4. Separate UI and application

☐ UI and application change independently

☐ UI and application built by different people
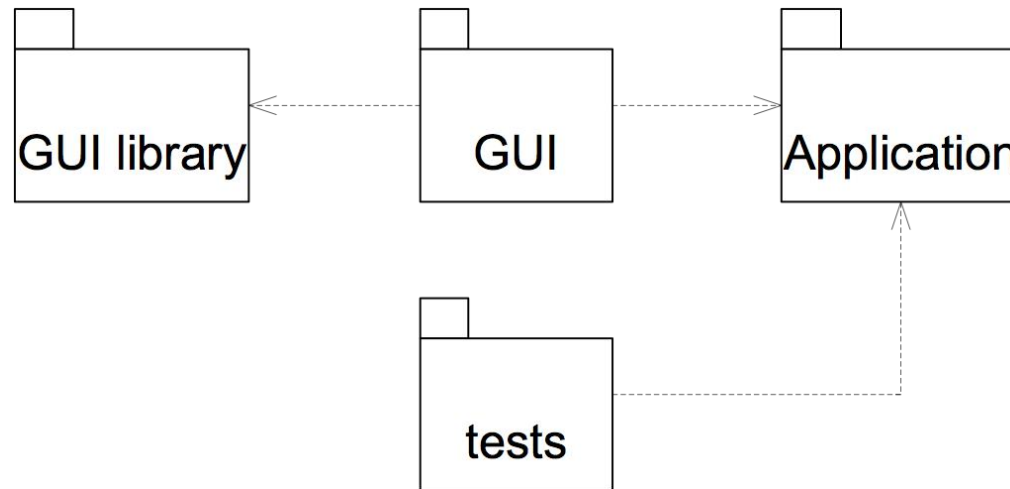
UI in Web: ASP/JSP/Rails…

☐ Embed code in your HTML
- VB code in ASP, Java code in JSP, Ruby code in Rails…

☐ Can call other code

☐ Need to decide on how much code goes in the web page, and how much goes outside

Separate UI from application

☐ HTML is UI

☐ Put as little code on web page as possible

☐ Web page has just enough code to call the actual application logic

Benefits

☐ Write automatic tests for application objects, not for UI

☐ People who write HTML don't need to know how to program well

☐ Programmers don't need to be good UI designers

Downside

☐ Application objects generate HTML
- But you can make standard set of "adapters" and so don't have to duplicate code
- Lists, radio buttons, etc.

☐ Code tends to creep into web pages
- Refactor
- Review

Results

☐Easier to test
- Automatic tests for application objects
- Test GUI manually or with automatic "smoke tests" or use something like Selenium

☐Easier to change
- Can change "business rule" independently of GUI
- Can add web interface, speech interface, etc.

One issue: selecting colors

☐Leave it to a graphic designer

☐Use system colors (actually pull them from config)

☐Use the company/university colors

☐Use a color palette generator

Summary

- UI design is hard
  - Must understand users
  - Must understand problems
  - Must understand technology
  - Must understand how to evaluate

- UI design is important
  - UI is what the users see
  - UI can "make it or break it" for software

DevOps and Continuous Integration

Popularity of DevOps

From：https://chinadevopsdays.org/

What is DevOps?

From: https://www.youtube.com/watch?v=_I94-tJIovg

What is DevOps?

A. Design + IT Operations
B. Design + Optimization
C. Development + IT Operations
D. Development + Optimization

What is Continuous Integration?

*"**Continuous Integration** is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible."*
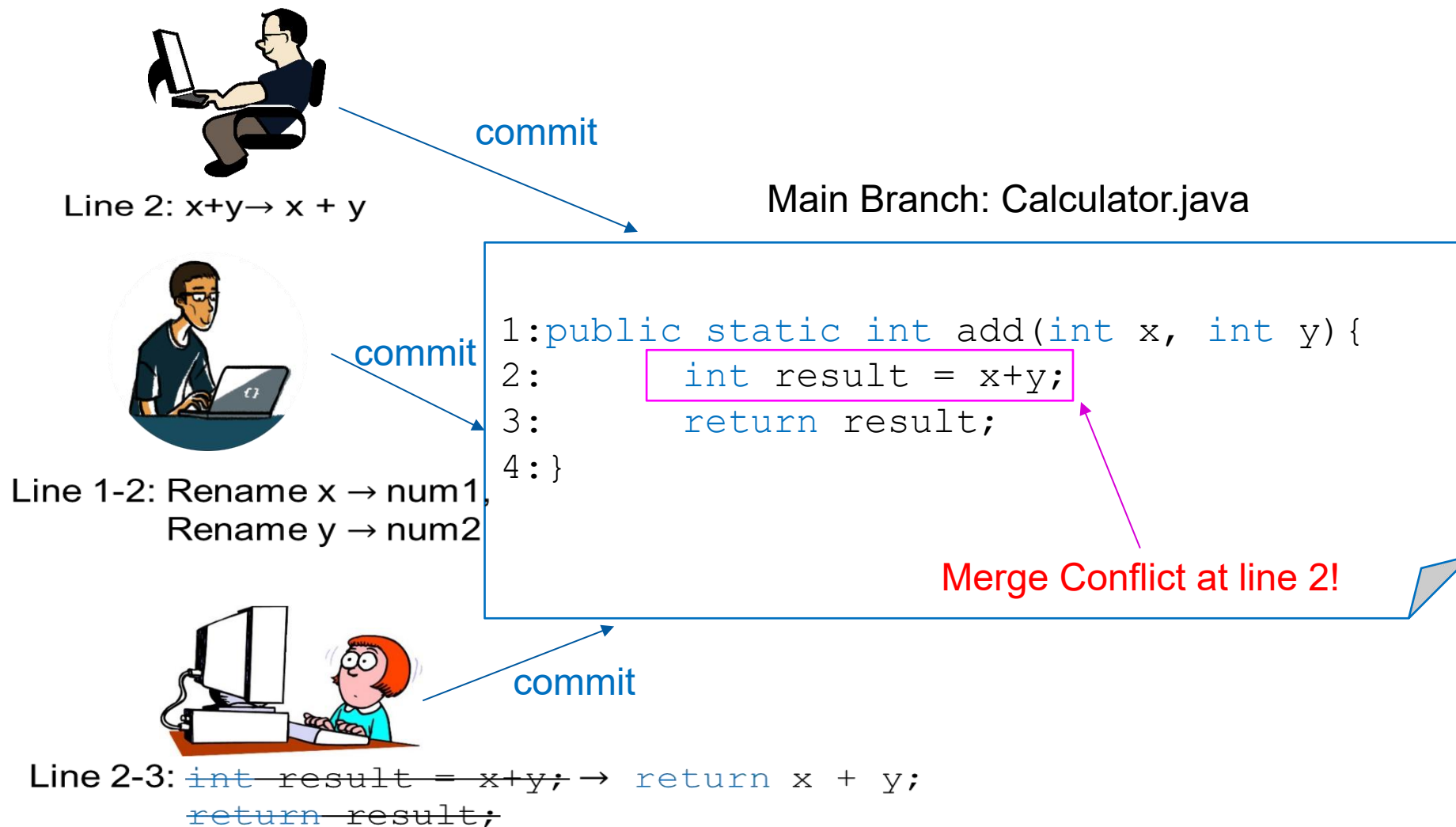
Benefits of Continuous Integration:

- Reduce integration problems
- Allows team to develop cohesive software more rapidly

Martin Fowler

Integration Problem

commit

Line 2: x+y→ x + y

Main Branch: Calculator.java

commit

```
1:public static int add(int x, int y){
2:        int result = x+y;
3:        return result;
4:}
```

Merge Conflict at line 2!

Line 1-2: Rename x → num1,
Rename y → num2

commit

Line 2-3: ~~int result = x+y;~~ → return x + y;
~~return result;~~

Integration Problems

Merge Conflict

- Modifying the same file concurrently

Compile Conflict

- Modified program no longer compile

Test Conflict

- Modified program compile but fails to pass the test

10 Principles of Continuous Integration

- Maintain a code repository – version control
- Automate the build
- Make your build self-testing
- Everyone commits to mainline every day
- Every commit should build mainline on an integration machine
- Keep the build fast
- Test in a clone of the production environment
- Make it easy for anyone to get the latest executable
- Everyone can see what's happening
- Automate deployment

Automate the build

Daily build: Compile working executable on a daily basis

- Develop scripts for compiling & running the system
  - Build Automation Tools:
    - Java: Apache Ant, Maven, Gradle
    - C/C++: make
- Benefits:
  - Allows you to test the quality of your integration
  - Visible Progress
  - Quickly catches/exposes bug that breaks the build

Build from command line

require `build.xml` for building

- One common way of automated build is to allows project to be compiled from the command line.

```xml
<project name="simpleCompile" default="deploy" basedir=".">
  <target name="init">
      <property name="sourceDir" value="src"/ >
      <property name="outputDir" value="classes" />
      <property name="deployJSP" value="/web/deploy/jsp" />
      <property name="deployProperties" value="/web/deploy/conf" />
  </target>
  <target name="clean" depends="init">
      <deltree dir="${outputDir}" />
  </target>
  <target name="prepare" depends="clean">
      <mkdir dir="${outputDir}" />
  </target>
  <target name="compile" depends="prepare">
    <javac srcdir="${sourceDir}" destdir="${outputDir}" />
  </target>
  <target name="deploy" depends="compile,init">
    <copydir src="${jsp}" dest="${deployJSP}"/>
    <copyfile src="server.properties" dest="${deployProperties}"/>
  </target>
</project>
```

`Target:` Build Target
`depends:` Target dependency
`property:` Define a simple
                  name value pair

Make your build self-testing

- **Automated tests:** Tests that can be run from the command line
- Examples:
  - Unit tests
  - Integration tests
  - Smoke test

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)
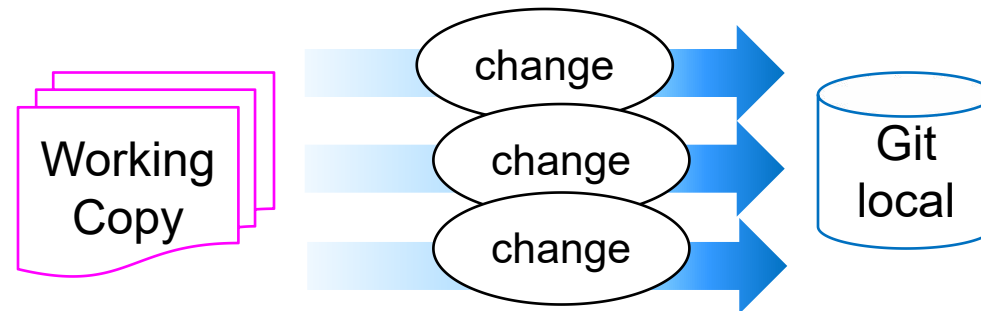 - Expose integration problems earlier
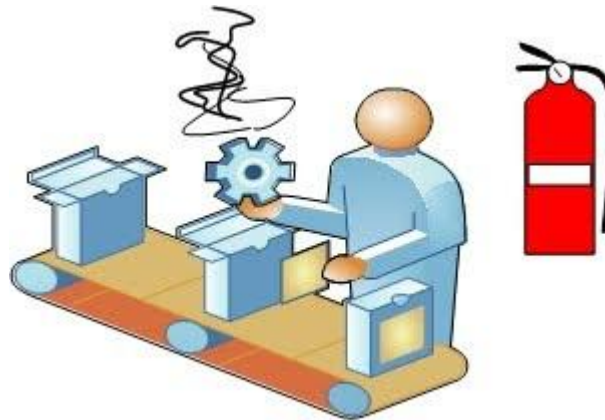
Question:

How do you perform smoke test for a new cup?

Fill it with water. If there is a leaking, then it is a bad cup

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.
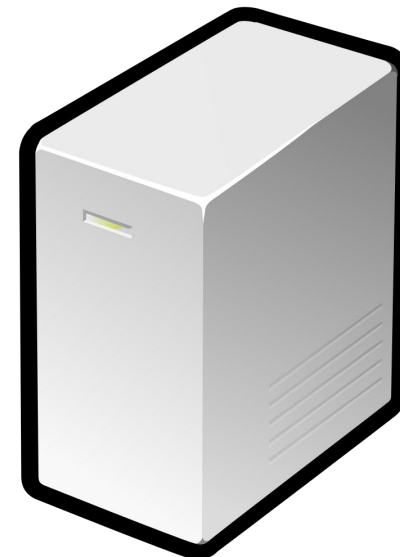
Question:

Does it mean that programmers need to constantly sit and wait for the build and run test? Is it possible to have someone this for me automatically?

Yes, use CI Server

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*

What happen in CI Server?
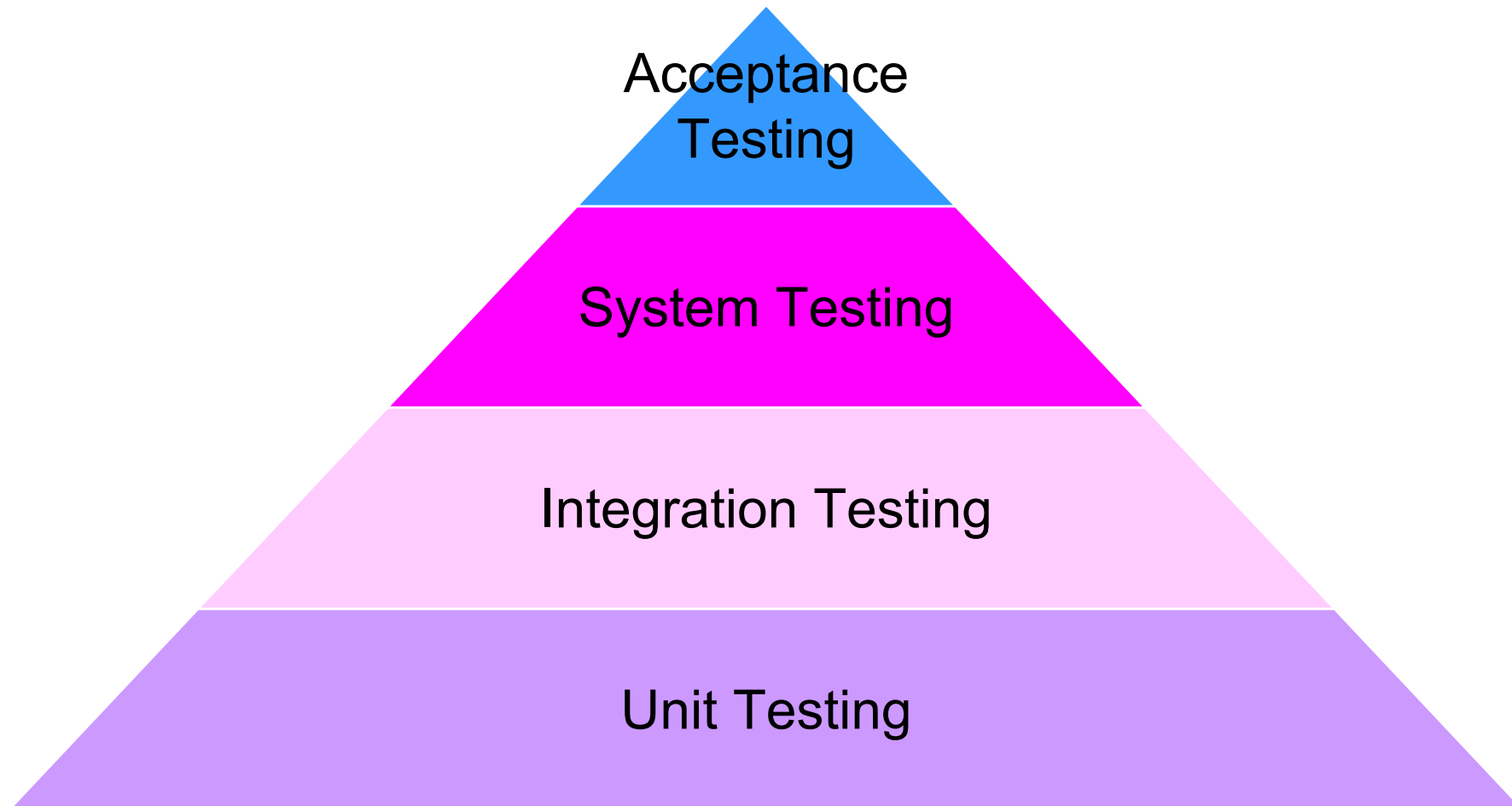

Pic from: https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html

Levels of Software Testing

Acceptance Testing

System Testing

Integration Testing

Unit Testing

Unit Testing

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

```java
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)`
			`}`

```java
assertEquals(expected, div(1, 2));
```
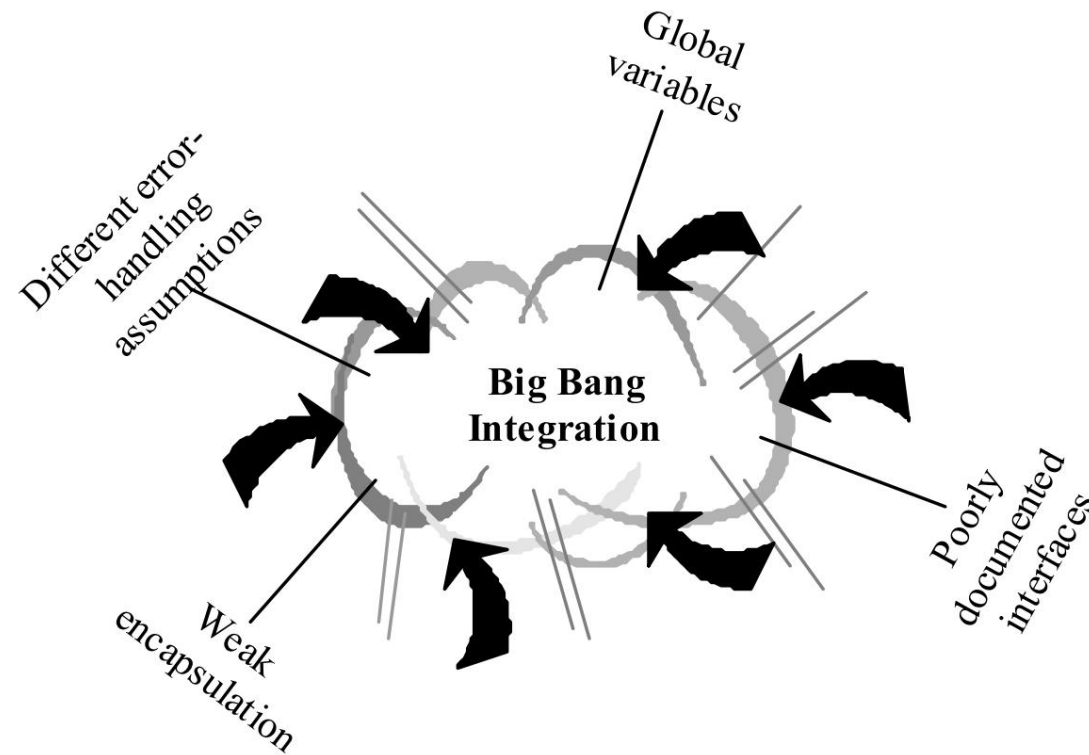
Integration testing

- **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?
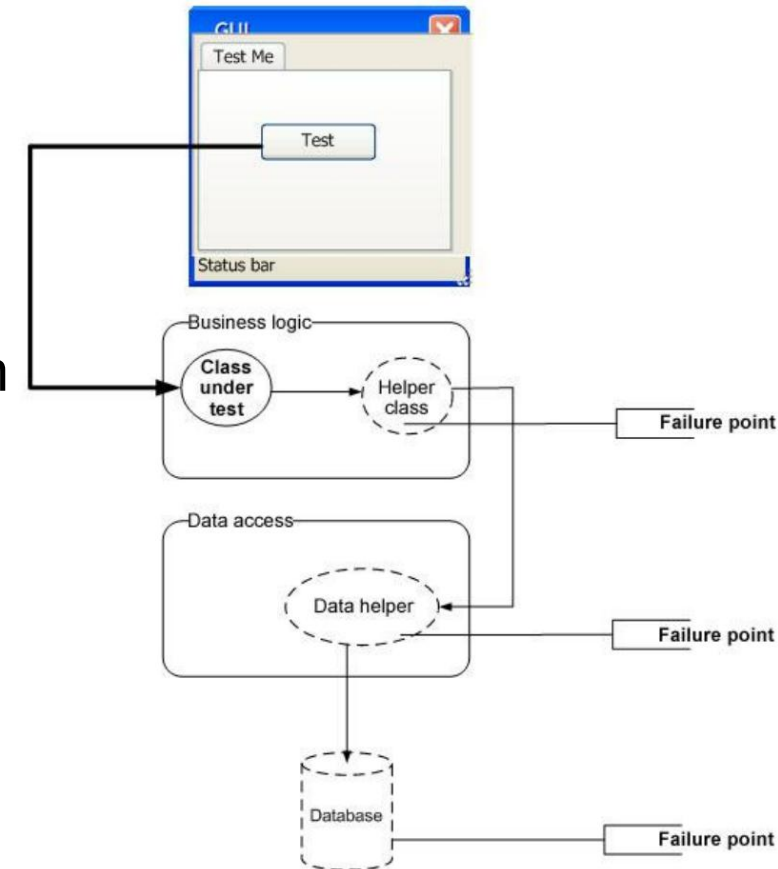
Big-bang Integration Testing

- *All* component are integrated together at *once*
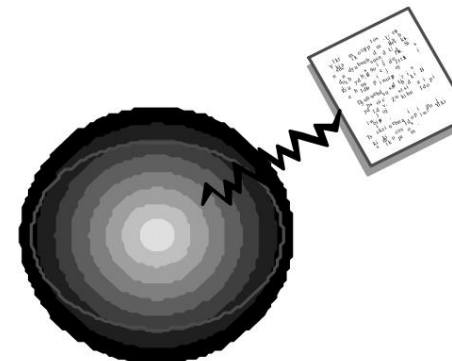
Big-bang Integration Testing

- 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
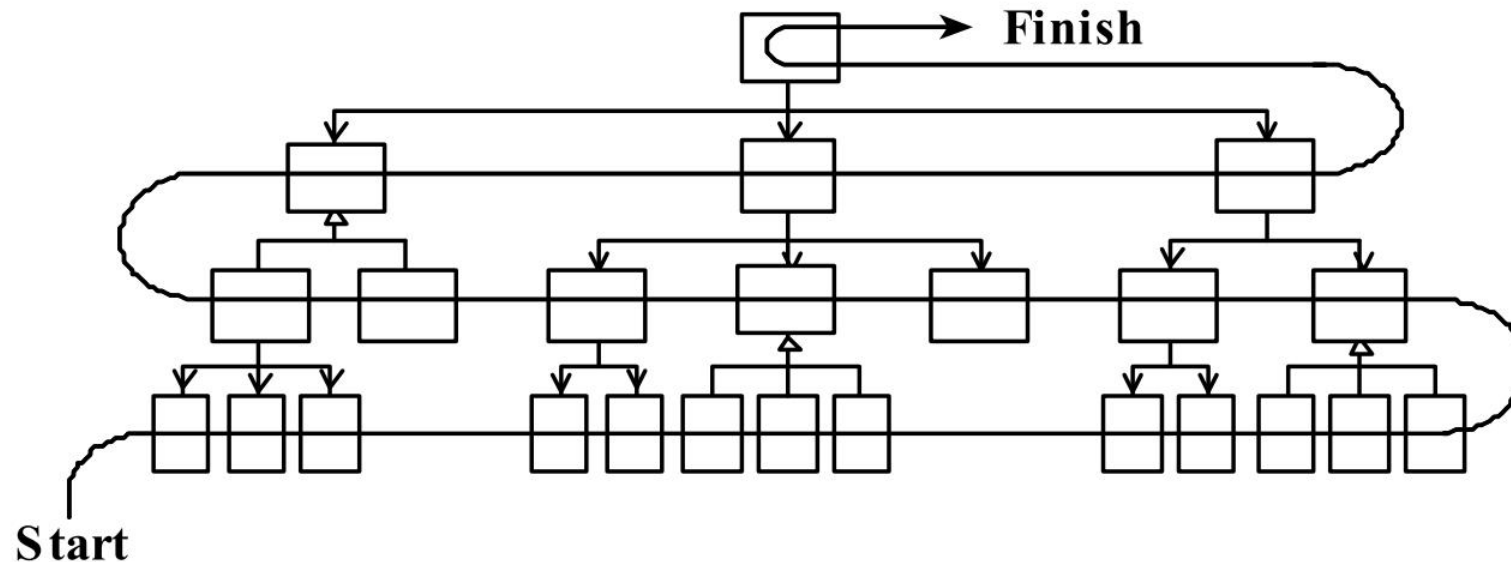
Top-down integration

- 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?)
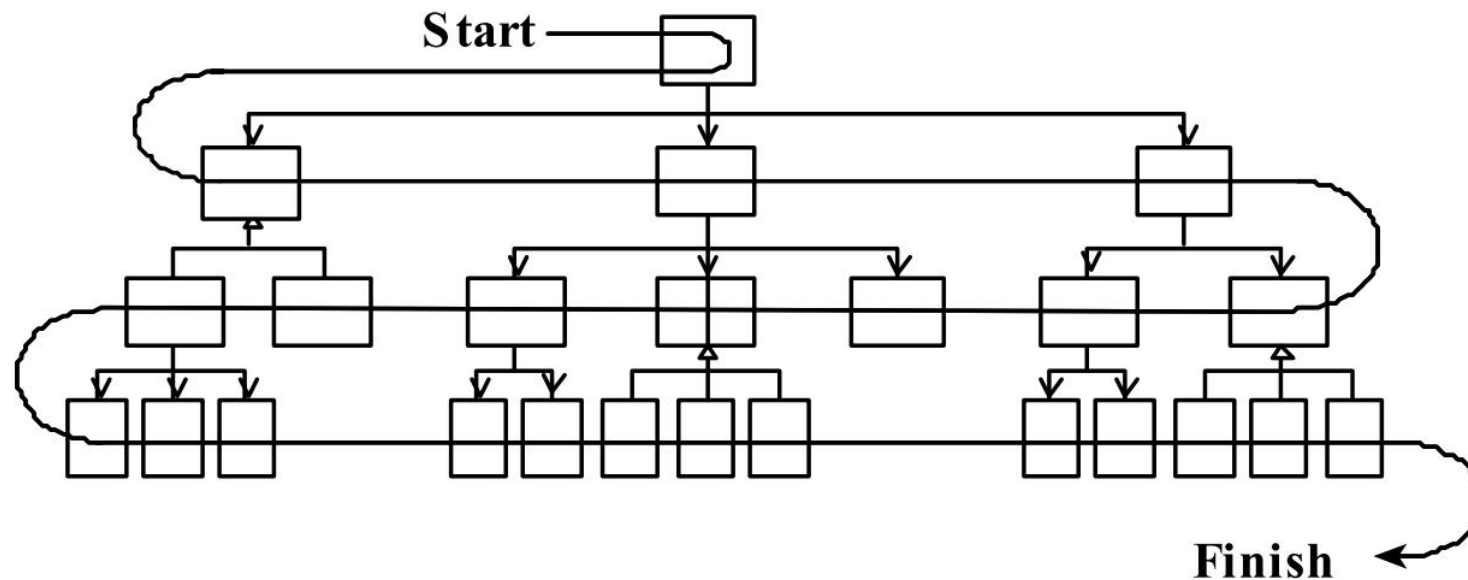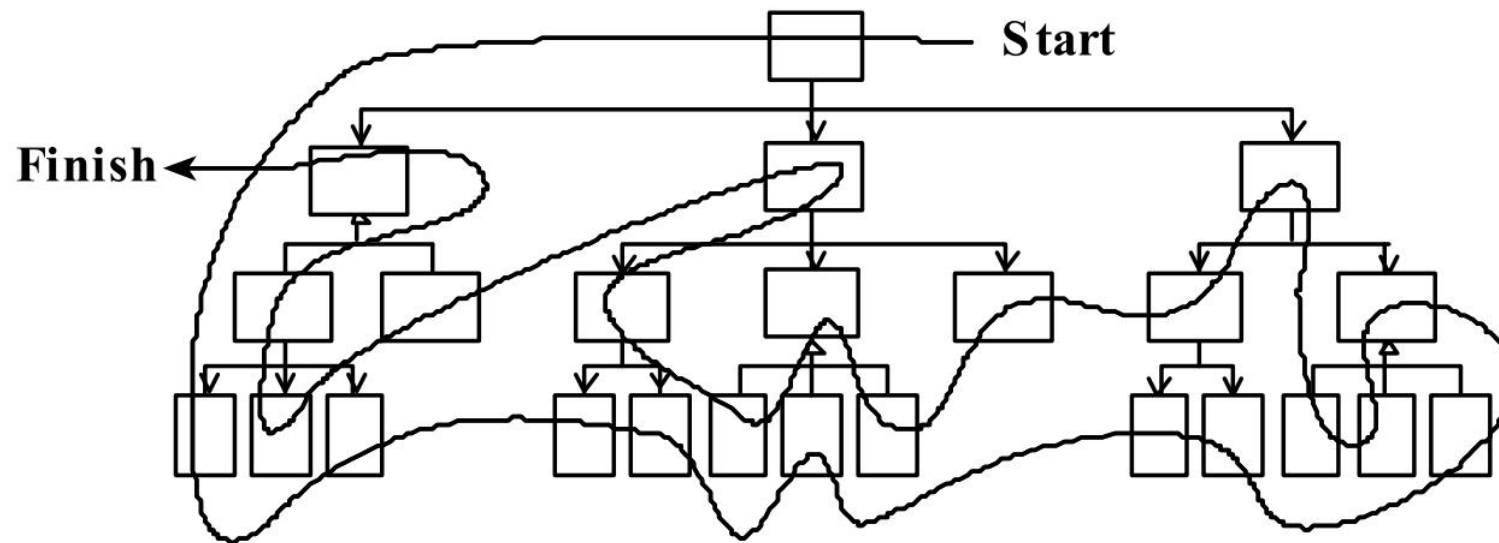
Bottom-up integration

- 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
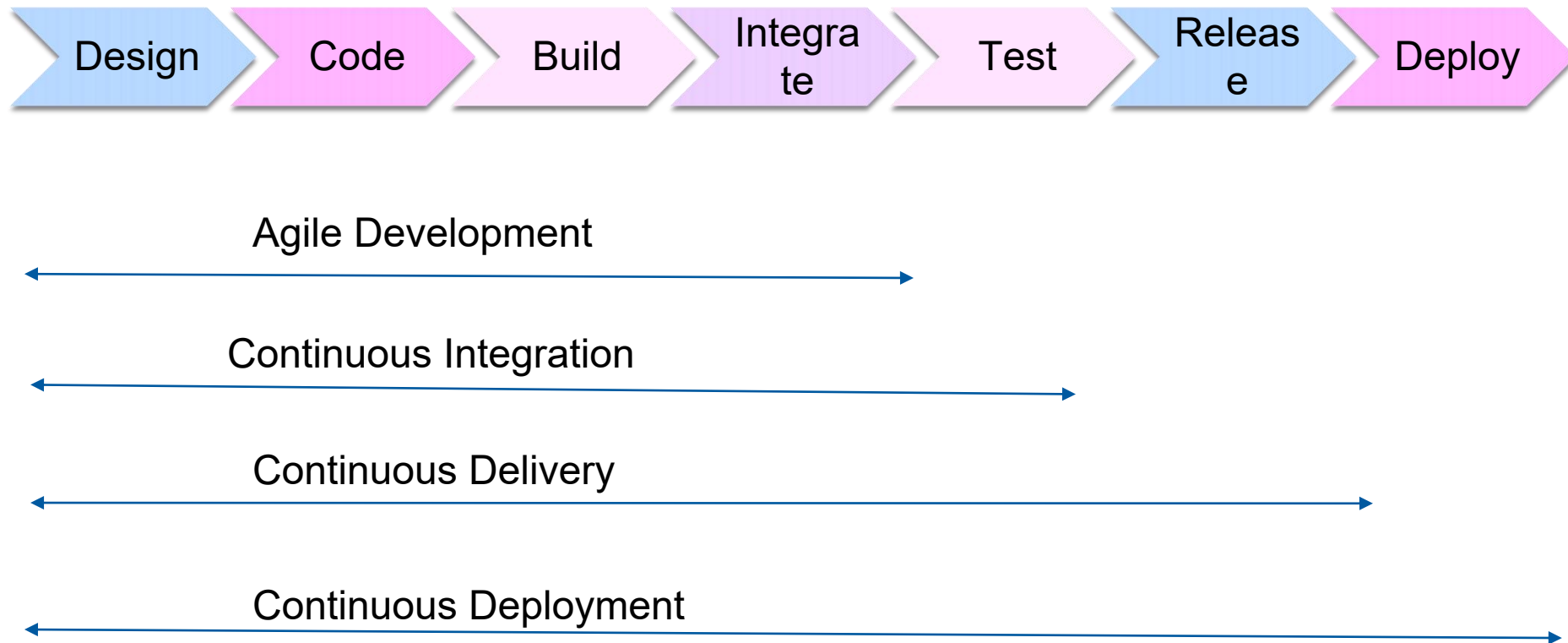 - More practical than top-down or bottom-up?

Stub versus Driver

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

Component: A

Call

Component: B

A

Call

Stub

Driver

Call

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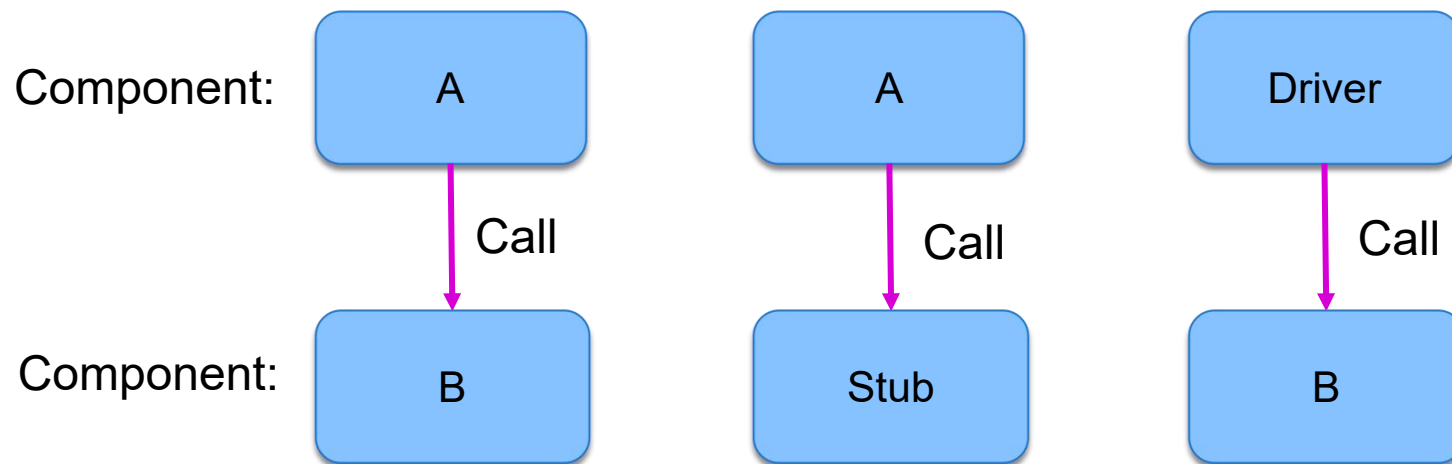
Stub: Dummy function gets called by another function
Driver: Dummy function to call another function

Continuous Integration & Continuous Deployment

Design → Code → Build → Integrate → Test → Release → Deploy

Agile Development
←——————————————————————→

Continuous Integration
←———————————————————————————————→

Continuous Delivery
←——→

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."
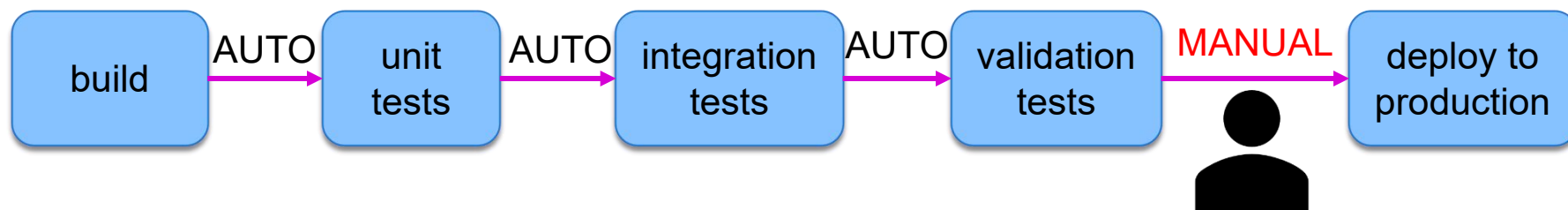
Jez Humble
Author of "Continuous Delivery"

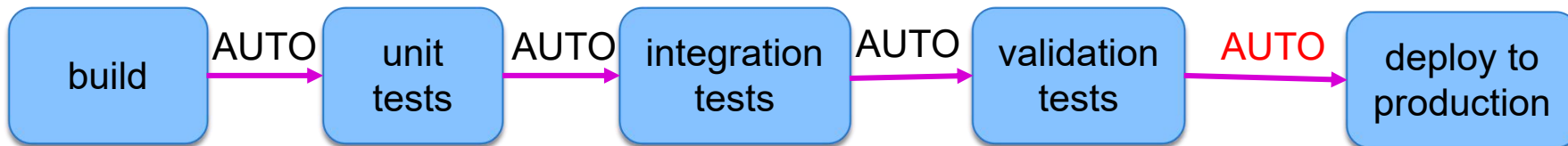# Continuous Integration != Continuous Delivery:
CI != CD

- $Continous\ Delivery\ =\ CI\ +\ automated\ test\ suite$
- 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

build --AUTO--> unit tests --AUTO--> integration tests --AUTO--> validation tests --MANUAL--> deploy to production

Continuous Deployment

build --AUTO--> unit tests --AUTO--> integration tests --AUTO--> validation tests --AUTO--> deploy to production

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
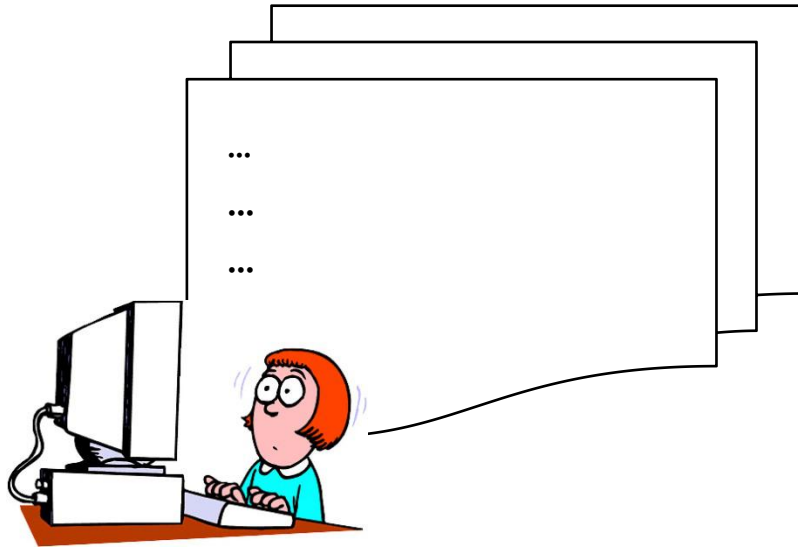 - ➢ Migration of database may not be backward compatible

Safer Strategy: Shut down→Migrate → Deploy

Question:

What is difference between **Continuous Integration** and **Continuous Delivery** ?

Continuous Delivery requires **automated testing** before release.
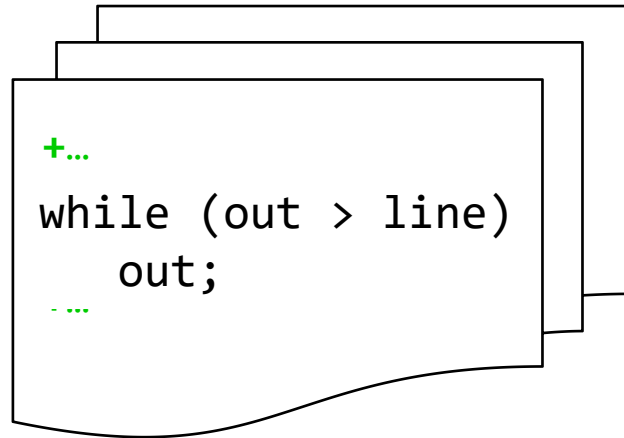
Regression Testing
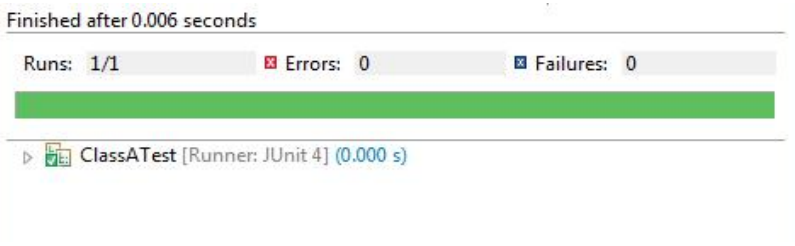
What is Regression?

- 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?

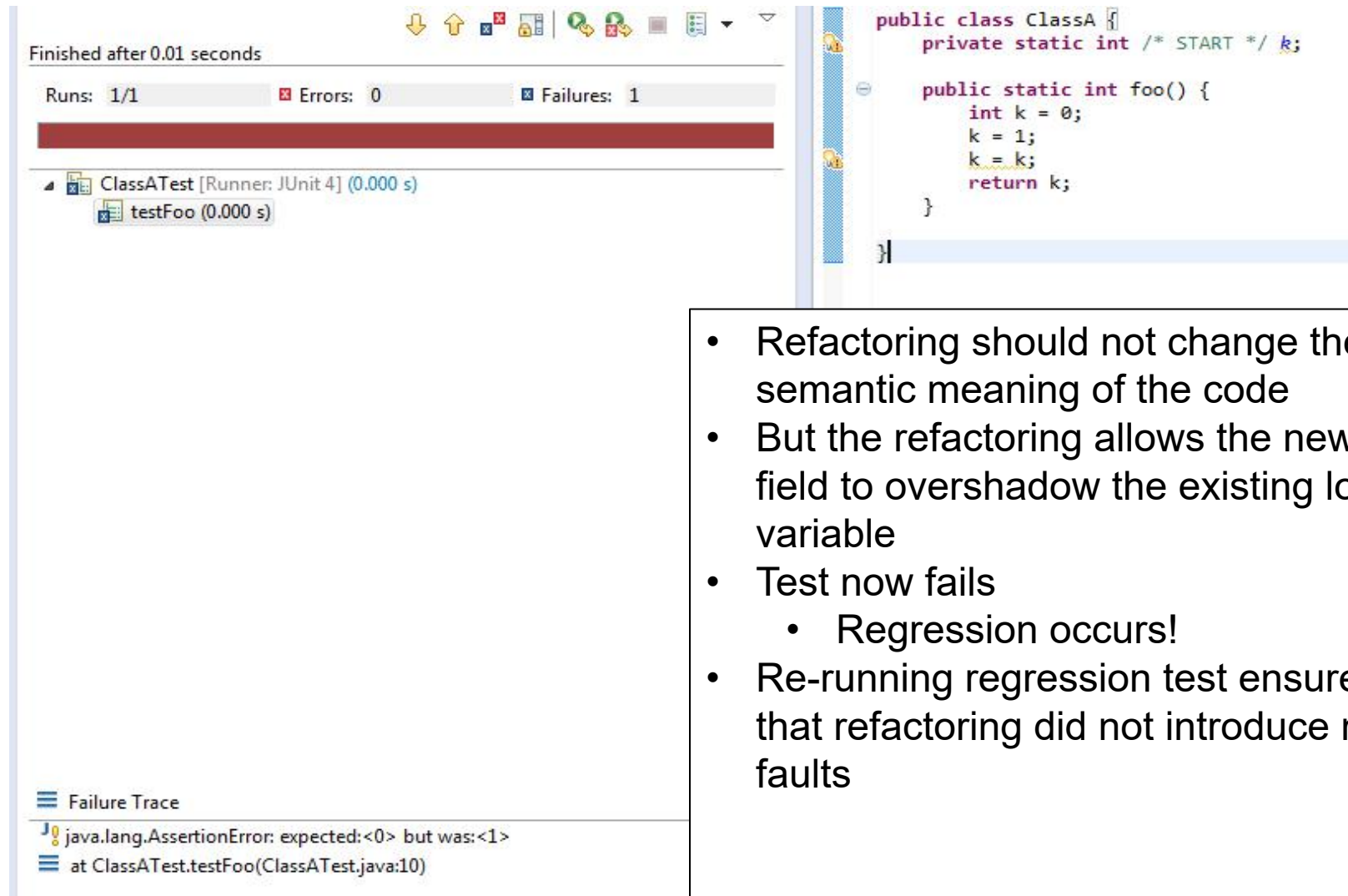| Class	
	``java
	public class ClassA {
	    public static int foo() {
	        int k = 0;
	        int /* START */f1/* end */= 1;
	        f1 = k;
	        return f1;
	    }
	
	}
	``
		• foo() returns 0 before refactoring

Test	
	``java
	⊕ 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	Finished after 0.006 seconds Runs: 1/1 ☒ Errors: 0 ☒ Failures: 0 ▷ 🔲 ClassATest [Runner: JUnit 4] (0.000 s)	• All test passes!

Perform Refactoring:
Convert Local Variable to Field
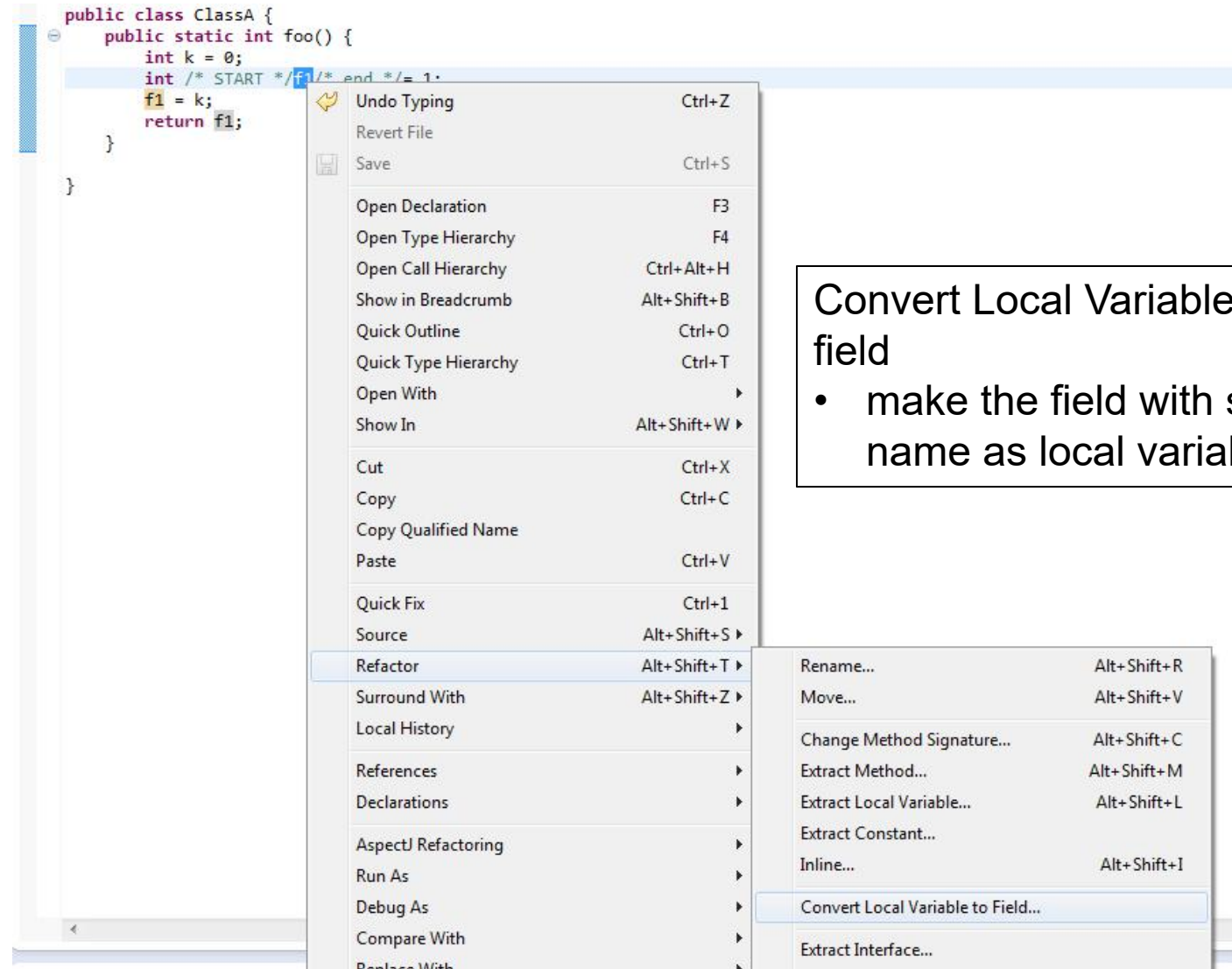
Convert Local Variable 'f1' to field
- make the field with same name as local variable 'k'

After refactoring

Finished after 0.01 seconds

Runs: 1/1   ⊠ Errors: 0   ⊠ Failures: 1

▲ ⊠ ClassATest [Runner: JUnit 4] (0.000 s)
   ⊠ testFoo (0.000 s)
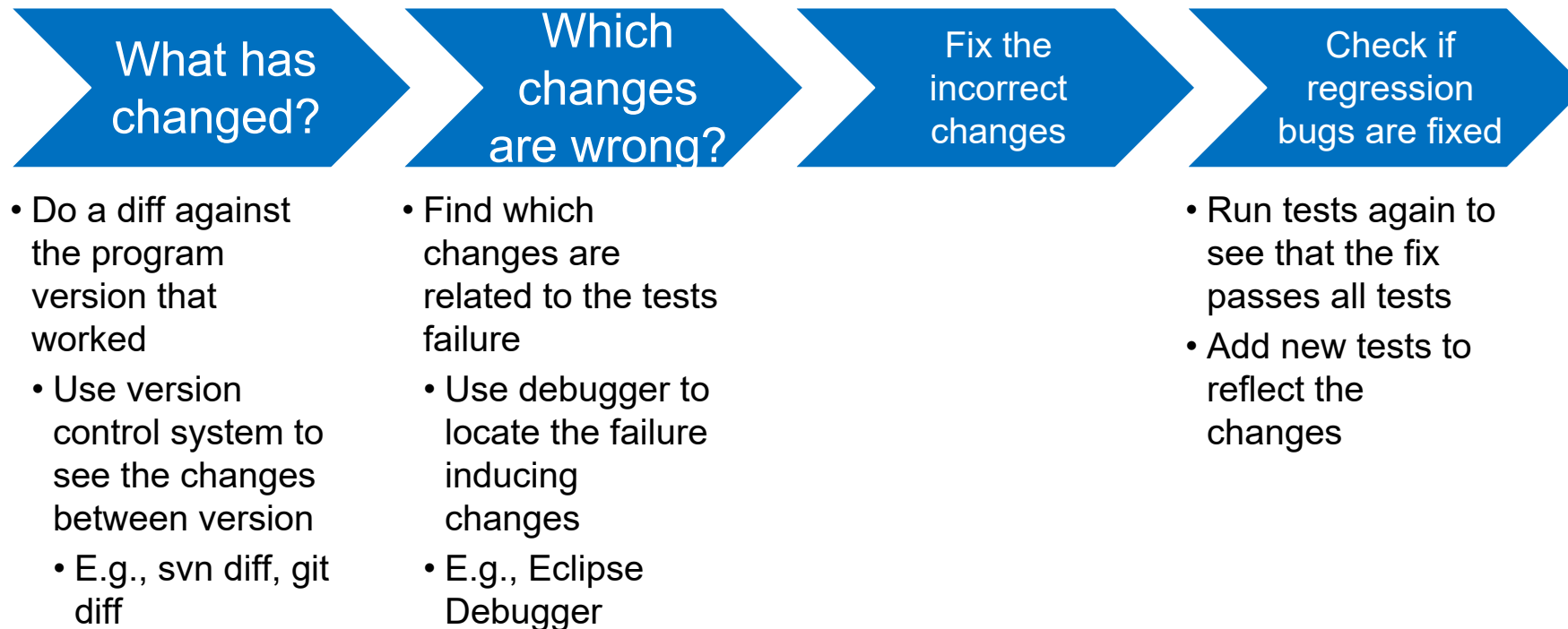
```java
public class ClassA {
    private static int /* START */ k;

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

≡ Failure Trace
⌡ java.lang.AssertionError: expected:<0> but was:<1>
≡ at ClassATest.testFoo(ClassATest.java:10)

- 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?	Which changes are wrong?	Fix the incorrect changes	Check if regression bugs are fixed

- 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

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

- 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
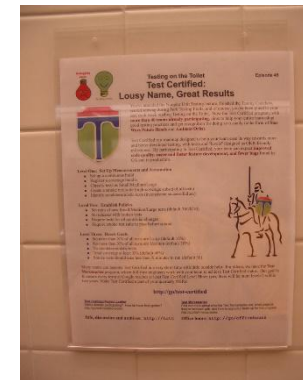
Example: Testing in the toilet

Can you tell if this test is correct?

208: @Test public void testIncrement_existingKey() {

209:    assertEquals(9, tally.get("key1"));

210: }

It's impossible to know without seeing how the tally object is set up:
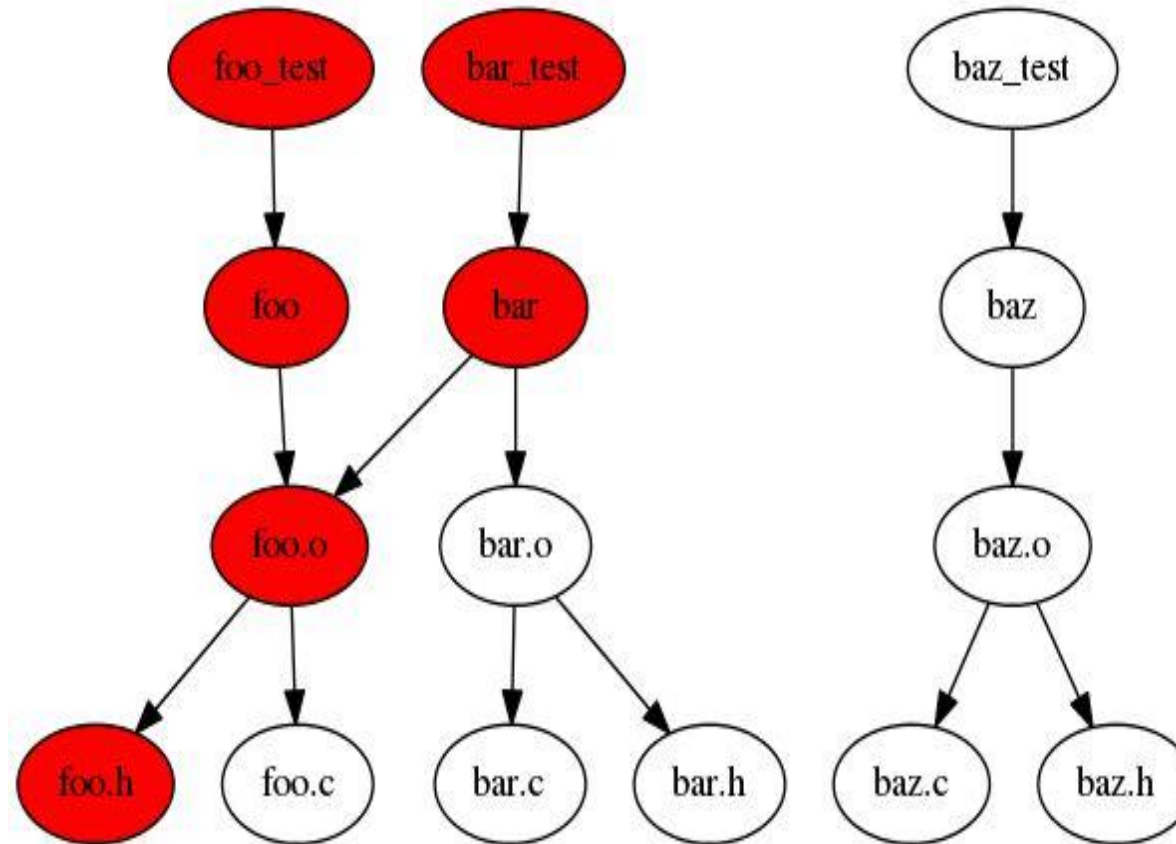
1:    private final Tally tally = new Tally();

2:    @Before public void setUp() {

3:       tally.increment("key1", 8);

4:       tally.increment("key2", 100);

5:       tally.increment("key1", 0);

6:       tally.increment("key1", 1);

7:    }

// 200 lines away

208: @Test public void testIncrement_existingKey() {

209:    assertEquals(9, tally.get("key1"));

210: }

Keep Cause and Effect Clear!
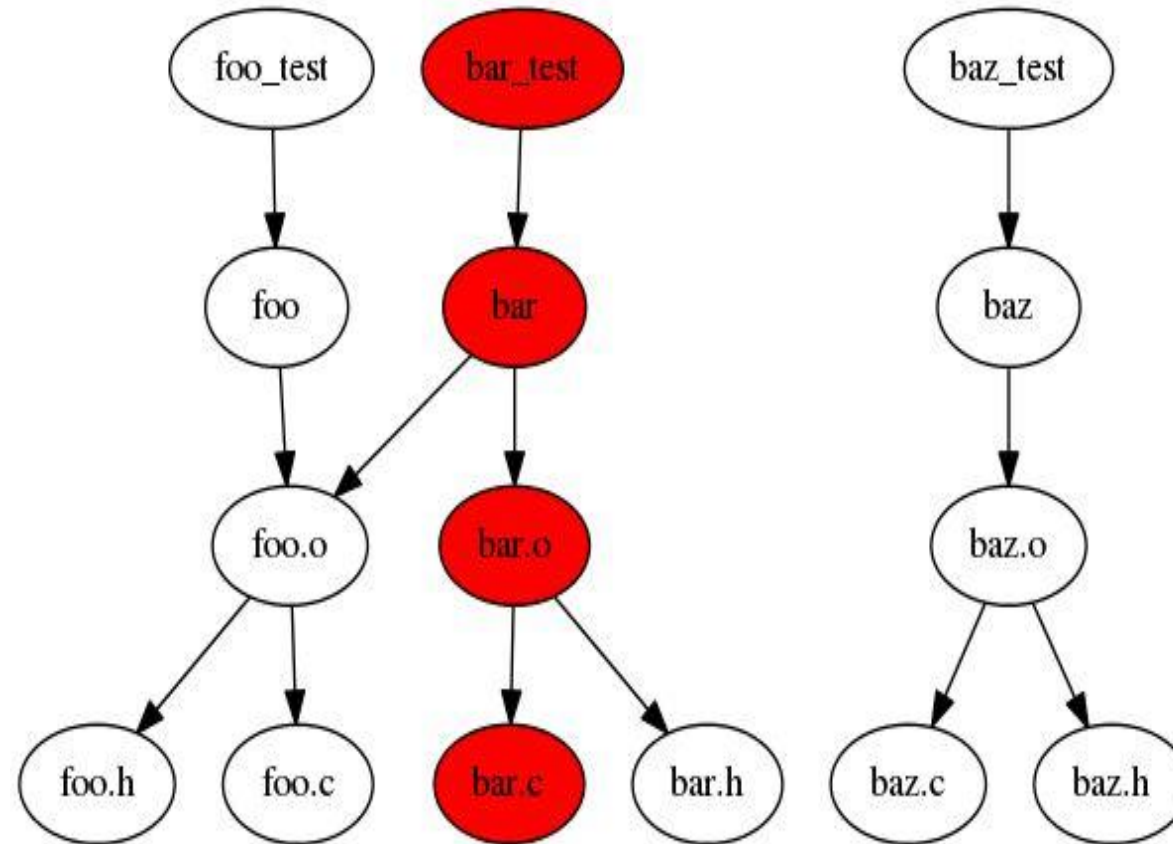
Regression Test Selection (RTS)
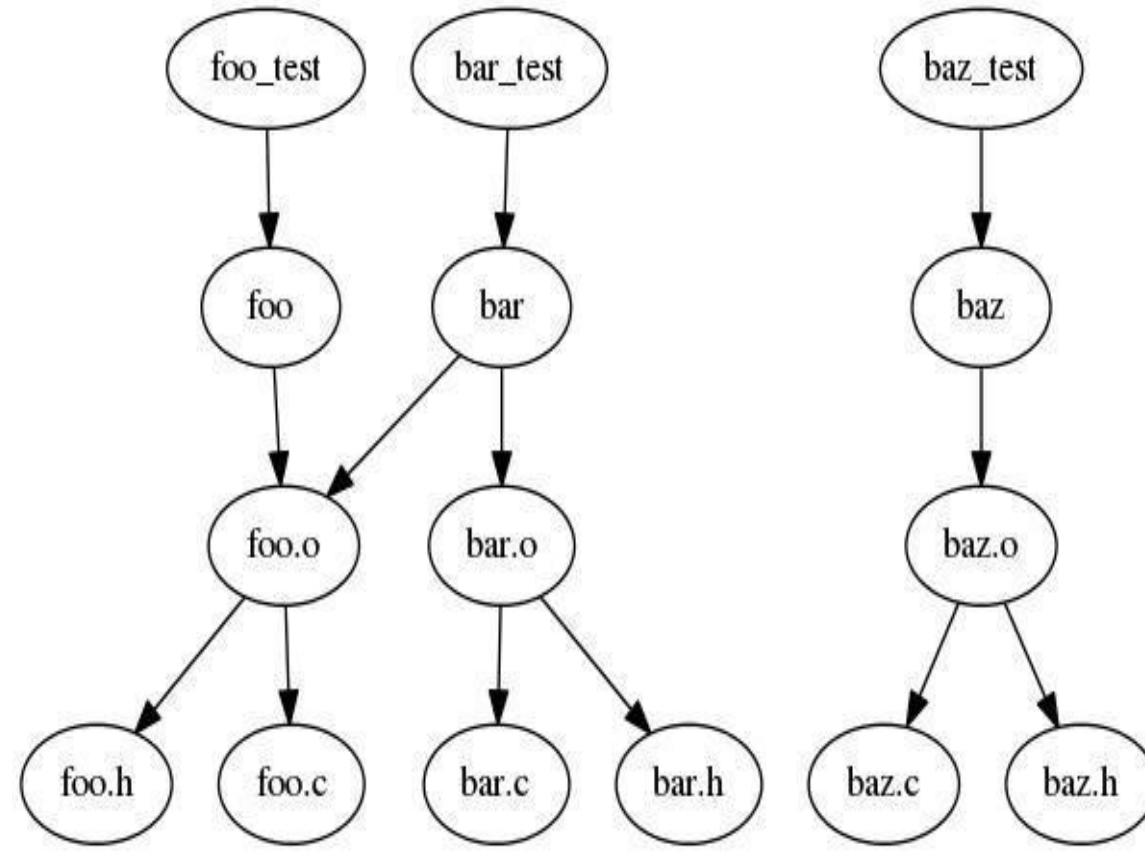
Regression Test Selection (RTS)

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
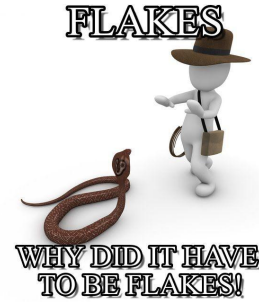
- Analysis of a large sample of tests (1 month) showed:
  - 84% of transitions from Pass -> Fail are from "flaky" tests
  - Only 1.23% of tests ever found a breakage
  - Frequently changed files more likely to cause a breakage
  - 3 or more developers changing a file is more likely to cause a breakage
  - 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
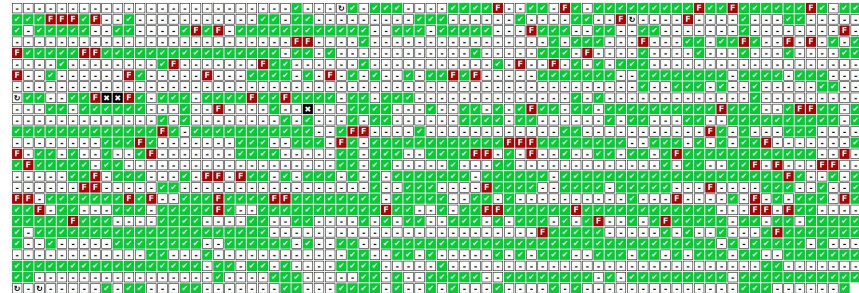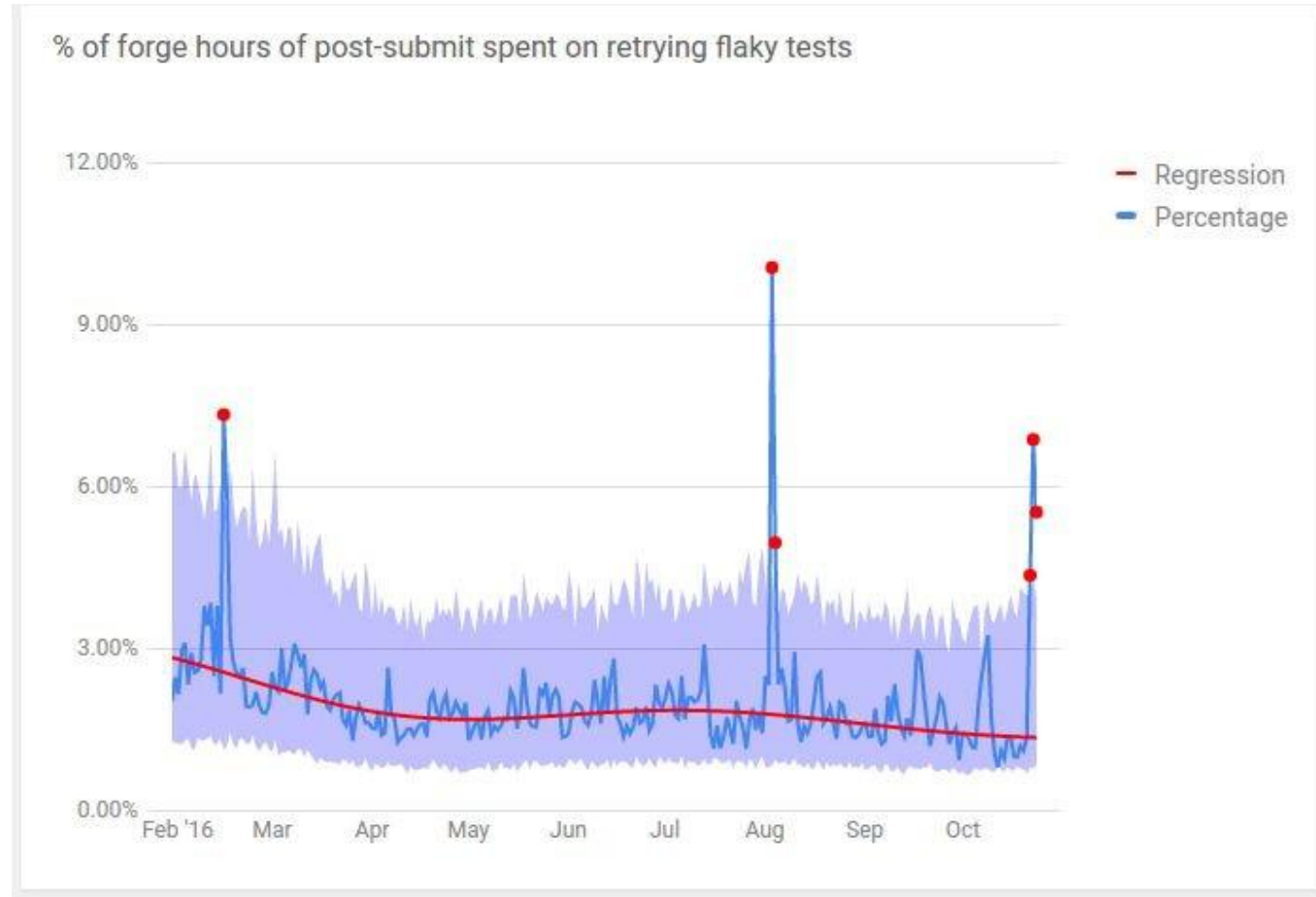
Flaky test impact on project health

- 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

```
1 @Test
2 public void testRsReportsWrongServerName() throws Exception {
3 MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
4 MiniHBaseClusterRegionServer firstServer =
5 (MiniHBaseClusterRegionServer)cluster.getRegionServer(0);
6 HServerInfo hsi = firstServer.getServerInfo();
7 firstServer.setHServerInfo(...);
8
9 // Sleep while the region server pings back
10 Thread.sleep(2000);
11 assertTrue(firstServer.isOnline());
12 assertEquals(2,cluster.getLiveRegionServerThreads().size());
13 ... // similarly for secondServer
14 }
```

- 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
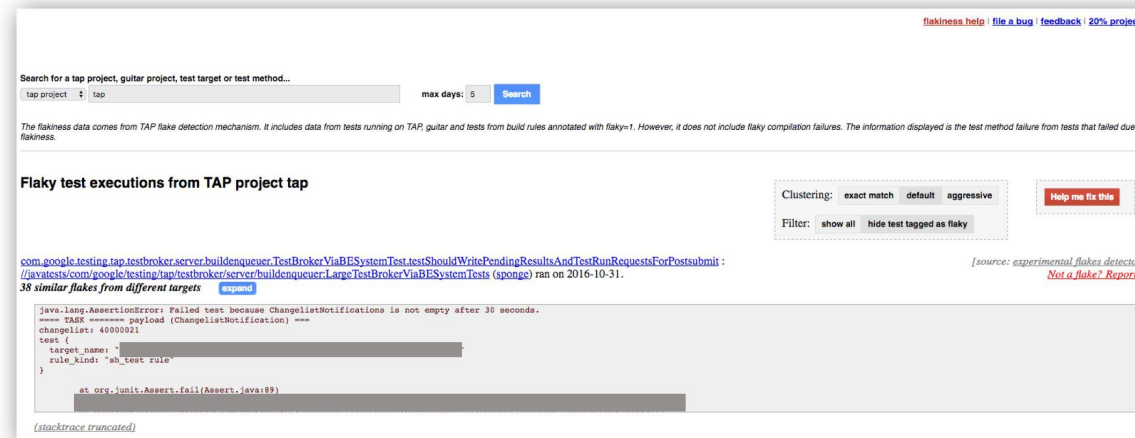- Observed insertion rate is about the same as fix rate

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

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