Week:
10 of 12

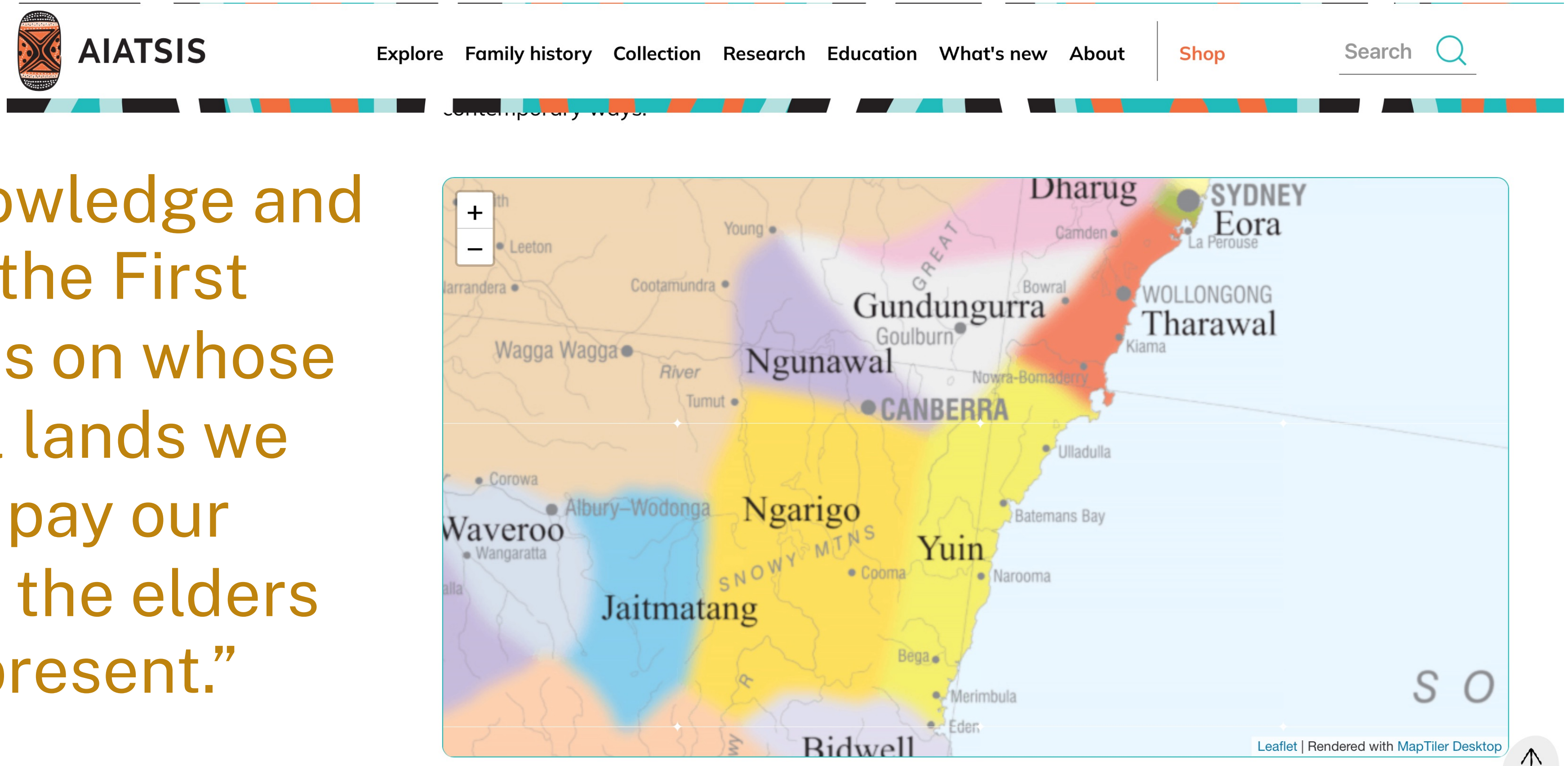
COMP 2120 / COMP 6120

TESTING

A/Prof Alex Potanin

ANU Acknowledgment of Country

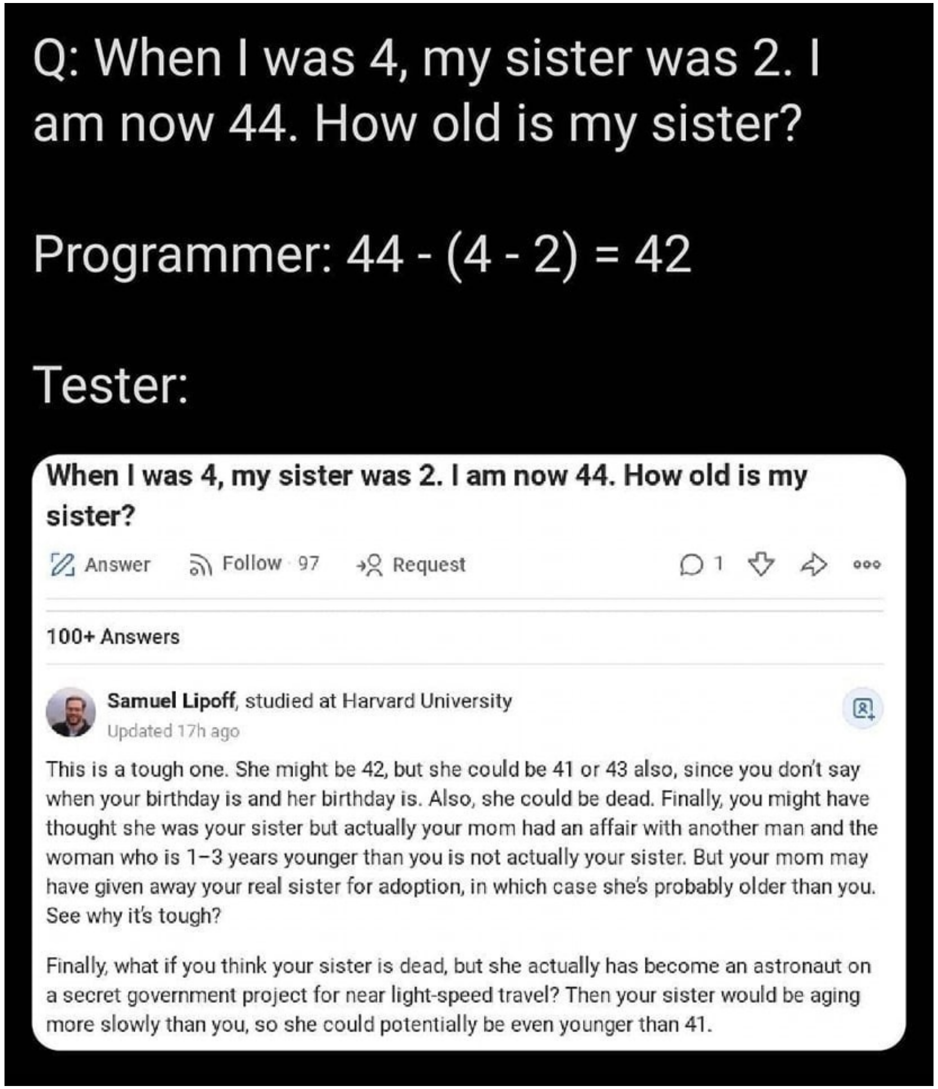
"We acknowledge and celebrate the First Australians on whose traditional lands we meet, and pay our respect to the elders past and present."



https://aiatsis.gov.au/explore/map-indigenous-austra

Today

- Types of Testing
- Test Automation
- Limitations of Testing
- Fuzzing
- Performance Testing
- Chaos Engineering
- A/B Testing



Q: When I was 4, my sister was 2. I am now 44. How old is my sister?

Programmer: 44 - (4 - 2) = 42

Tester:

When I was 4, my sister was 2. I am now 44. How old is my sister?

100+ Answers

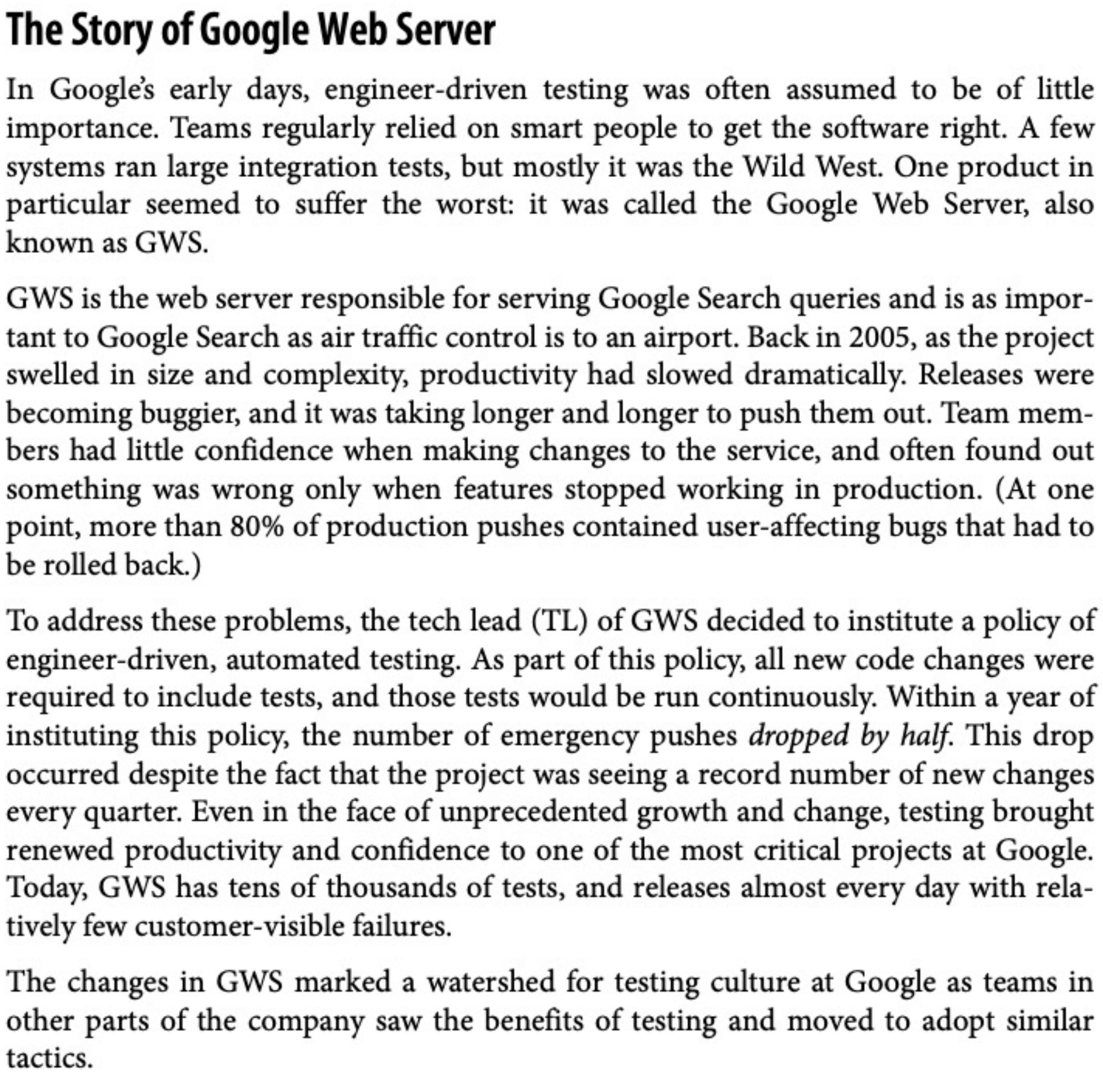
Samuel Lipoff, studied at Harvard University

Updated 17h ago

This is a tough one. She might be 42, but she could be 41 or 43 also, since you don't say when your birthday is and her birthday is. Also, she could be dead. Finally, you might have thought she was your sister but actually your mom had an affair with another man and the woman who is 1–3 years younger than you is not actually your sister. But your mom may have given away your real sister for adoption, in which case she's probably older than you. See why it's tough?

Finally, what if you think your sister is dead, but she actually has become an astronaut on a secret government project for near light-speed travel? Then your sister would be aging more slowly than you, so she could potentially be even younger than 41.

Types of Testing



The Story of Google Web Server

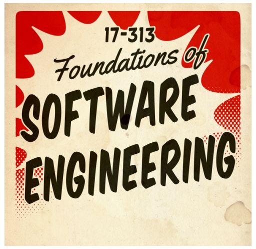
In Google's early days, engineer-driven testing was often assumed to be of little importance. Teams regularly relied on smart people to get the software right. A few systems ran large integration tests, but mostly it was the Wild West. One product in particular seemed to suffer the worst: it was called the Google Web Server, also known as GWS.

GWS is the web server responsible for serving Google Search queries and is as important to Google Search as air traffic control is to an airport. Back in 2005, as the project swelled in size and complexity, productivity had slowed dramatically. Releases were becoming buggier, and it was taking longer and longer to push them out. Team members had little confidence when making changes to the service, and often found out something was wrong only when features stopped working in production. (At one point, more than 80% of production pushes contained user-affecting bugs that had to be rolled back.)

To address these problems, the tech lead (TL) of GWS decided to institute a policy of engineer-driven, automated testing. As part of this policy, all new code changes were required to include tests, and those tests would be run continuously. Within a year of instituting this policy, the number of emergency pushes *dropped by half*. This drop occurred despite the fact that the project was seeing a record number of new changes every quarter. Even in the face of unprecedented growth and change, testing brought renewed productivity and confidence to one of the most critical projects at Google. Today, GWS has tens of thousands of tests, and releases almost every day with relatively few customer-visible failures.

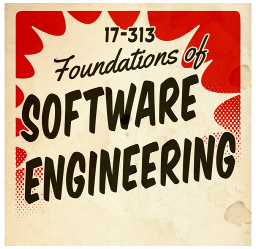
The changes in GWS marked a watershed for testing culture at Google as teams in other parts of the company saw the benefits of testing and moved to adopt similar tactics.

What is Testing???



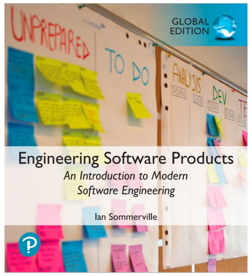
- What is testing?
  - Execution of code on sample inputs in a controlled environment
- Principle goals:
  - Validation: program meets requirements, including quality attributes.
  - Defect testing: reveal failures.
- Other goals:
  - Reveal bugs (main goal)
  - Assess quality (hard to quantify)
  - Clarify the specification, documentation
  - Verify contracts

What is Testing???



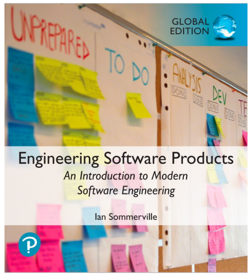
- What can we test for? (Software quality attributes)
  - What can we not test for?
- Why should we test? What does testing achieve?
  - What does testing not achieve?
- When should we test?
  - And where should we run the tests?
- What should we test?
  - What CAN we test?
- How should we test?
  - How many ways can you test the sort() function?
- How good are our tests?
  - How to measure test quality?

Software testing


GLOBAL EDITION
Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville

- Software testing is a process in which you execute your program using data that simulates user inputs.
- You observe its behaviour to see whether or not your program is doing what it is supposed to do.
  - Tests pass if the behaviour is what you expect. Tests fail if the behaviour differs from that expected.
  - If your program does what you expect, this shows that for the inputs used, the program behaves correctly.
- If these inputs are representative of a larger set of inputs, you can infer that the program will behave correctly for all members of this larger input set.

Program bugs


Engineering Software Products

An Introduction to Modern Software Engineering

Ian Sommerville

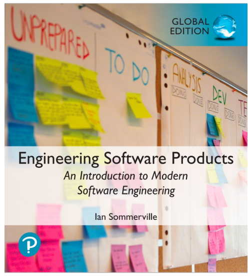
GLOBAL EDITION

- If the behaviour of the program does not match the behaviour that you expect, then this means that there are bugs in your program that need to be fixed.
- There are two causes of program bugs:
  - *Programming errors* You have accidentally included faults in your program code. For example, a common programming error is an 'off-by-1' error where you make a mistake with the upper bound of a sequence and fail to process the last element in that sequence.
  - *Understanding errors* You have misunderstood or have been unaware of some of the details of what the program is supposed to do. For example, if your program processes data from a file, you may not be aware that some of this data is in the wrong format, so your program doesn't include code to handle this.

Test Strategies

- Enumerate various strategies for picking test cases, such as:
  - Specification-based testing
  - Boundary-value testing
  - Structural testing
  - Property testing
  - Regression testing
  - Differential testing
  - Property-based testing
  - Mutation testing
- https://testing.googleblog.com/search/label/TotT

Types of testing



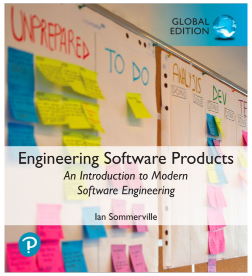
- ***Functional testing***
  Test the functionality of the overall system. The goals of functional testing are to discover as many bugs as possible in the implementation of the system and to provide convincing evidence that the system is fit for its intended purpose.

- ***User testing***
  Test that the software product is useful to and usable by end-users. You need to show that the features of the system help users do what they want to do with the software. You should also show that users understand how to access the software's features and can use these features effectively.

- ***Performance and load testing***
  Test that the software works quickly and can handle the expected load placed on the system by its users. You need to show that the response and processing time of your system is acceptable to end-users. You also need to demonstrate that your system can handle different loads and scales gracefully as the load on the software increases.

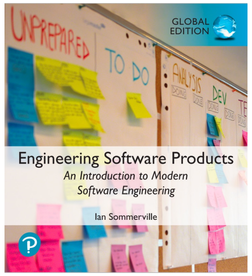
- ***Security testing***
  Test that the software maintains its integrity and can protect user information from theft and damage.

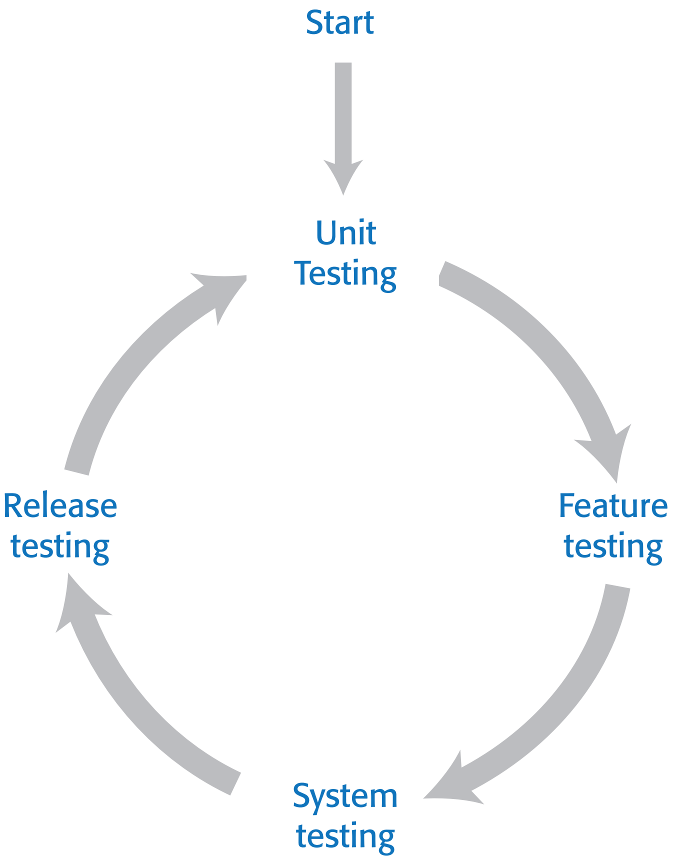
Functional testing



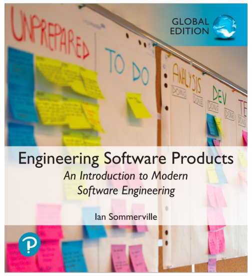
- Functional testing involves developing a large set of program tests so that, ideally, all of a program’s code is executed at least once.
- The number of tests needed obviously depends on the size and the functionality of the application.
- For a business-focused web application, you may have to develop thousands of tests to convince yourself that your product is ready for release to customers.
- Functional testing is a staged activity in which you initially test individual units of code. You integrate code units with other units to create larger units then do more testing.
- The process continues until you have created a complete system ready for release.

Functional testing


GLOBAL EDITION
Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville

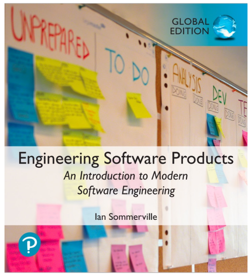


Functional testing processes


Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville
GLOBAL EDITION

- **Unit testing**
  The aim of unit testing is to test program units in isolation. Tests should be designed to execute all of the code in a unit at least once. Individual code units are tested by the programmer as they are developed.
- **Feature testing**
  Code units are integrated to create features. Feature tests should test all aspects of a feature. All of the programmers who contribute code units to a feature should be involved in its testing.
- **System testing**
  Code units are integrated to create a working (perhaps incomplete) version of a system. The aim of system testing is to check that there are no unexpected interactions between the features in the system. System testing may also involve checking the responsiveness, reliability and security of the system. In large companies, a dedicated testing team may be responsible for system testing. In small companies, this is impractical, so product developers are also involved in system testing.
- **Release testing**
  The system is packaged for release to customers and the release is tested to check that it operates as expected. The software may be released as a cloud service or as a download to be installed on a customer's computer or mobile device. If DevOps is used, then the development team are responsible for release testing otherwise a separate team has that responsibility.

Unit testing


GLOBAL EDITION
Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville

- As you develop a code unit, you should also develop tests for that code.

- A code unit is anything that has a clearly defined responsibility. It is usually a function or class method but could be a module that includes a small number of other functions.

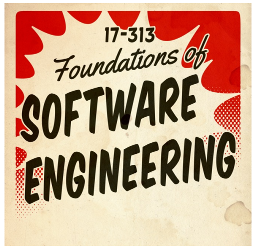
- Unit testing is based on a simple general principle:

  - If a program unit behaves as expected for a set of inputs that have some shared characteristics, it will behave in the same way for a larger set whose members share these characteristics.

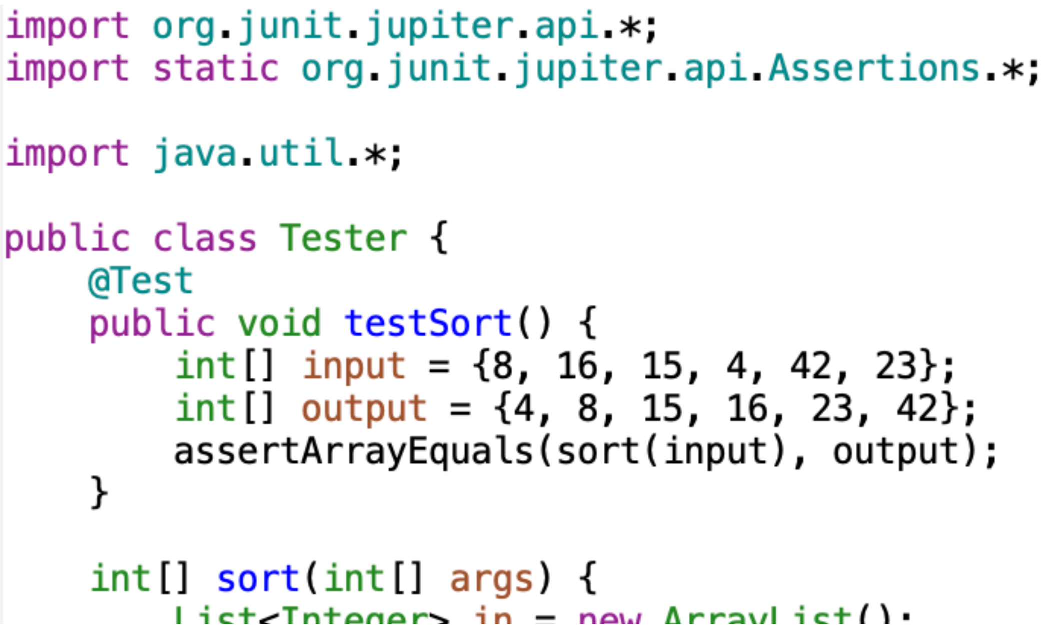
- To test a program efficiently, you should identify sets of inputs (equivalence partitions) that will be treated in the same way in your code.

- The equivalence partitions that you identify should not just include those containing inputs that produce the correct values. You should also identify ‘incorrectness partitions’ where the inputs are deliberately incorrect.

JUnit

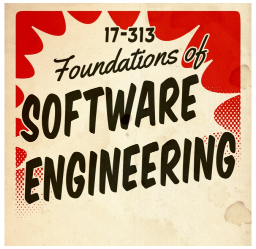


- Popular unit-testing framework for Java
- Easy to use
- Tool support available (Maven, Gradle, etc.)
- Can be used as design mechanis

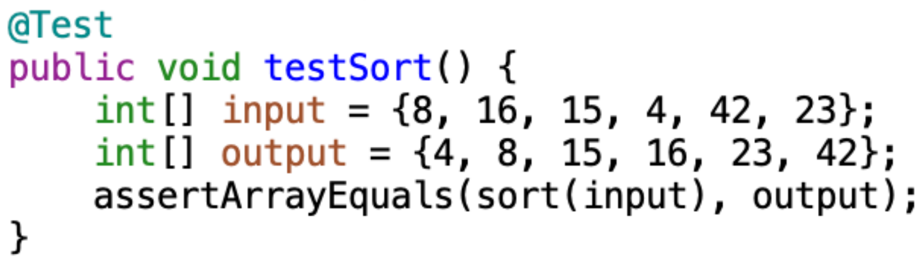


```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

import java.util.*;

public class Tester {
    @Test
    public void testSort() {
        int[] input = {8, 16, 15, 4, 42, 23};
        int[] output = {4, 8, 15, 16, 23, 42};
        assertArrayEquals(sort(input), output);
    }

    int[] sort(int[] args) {
        List<Integer> in = new ArrayList();
```

Basic Elements of a Test



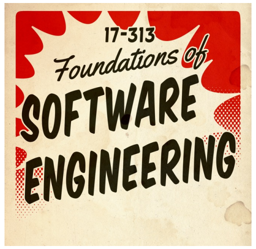
- Tests usually need an *input* and *expected output.*

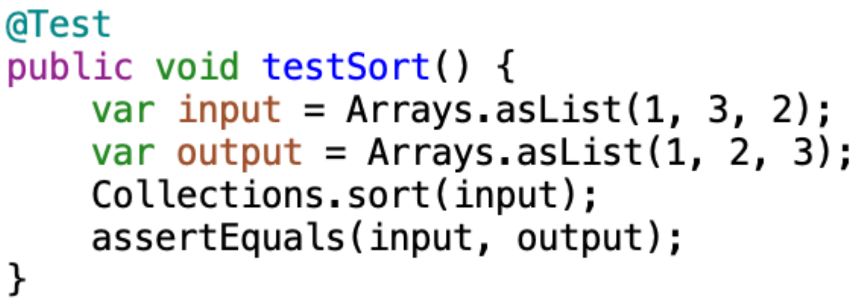


```
@Test
public void testSort() {
    int[] input = {8, 16, 15, 4, 42, 23};
    int[] output = {4, 8, 15, 16, 23, 42};
    assertArrayEquals(sort(input), output);
}
```

- More generally, a *test environment, a test harness*, and a *test oracle*
  - Environment: Resources needed to execute a family of tests
  - Harness: Triggers execution of a test case (aka *entry point*)
  - Oracle: A mechanism for determining whether a test was successful

Basic Unit Test for Sort





```
@Test
public void testSort() {
    var input = Arrays.asList(1, 3, 2);
    var output = Arrays.asList(1, 2, 3);
    Collections.sort(input);
    assertEquals(input, output);
}
```

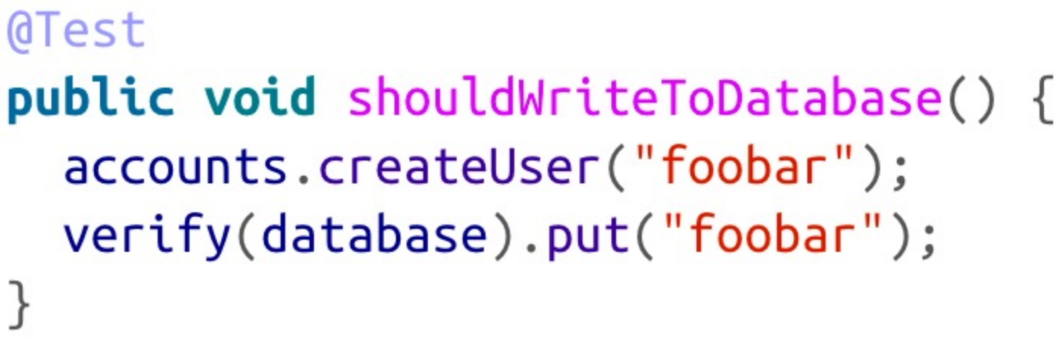
- What are some interesting values to test?
  - List tuples <input, output, reason>

Unit Tests (from SE @ Google Chapter 12)

- What are *unit tests*?
  - Small tests are fast and deterministic, allowing developers to run them frequently as part of their workflow and get immediate feedback
  - Easy to write at the same time as the code they're testing, allowing engineers to focus their tests on the code they're working on without having to set up and understand a larger system
  - Promote high levels of test coverage because they are quick and easy to write
  - Easy to understand what's wrong when they fail because each test is conceptually simple and focused on a particular part of the system
  - Serve as documentation and examples, showing engineers how to use the part of the system being tested and how that system is intended to work
- Should be 80% unit tests (maintainable!) and 20% broader-scoped

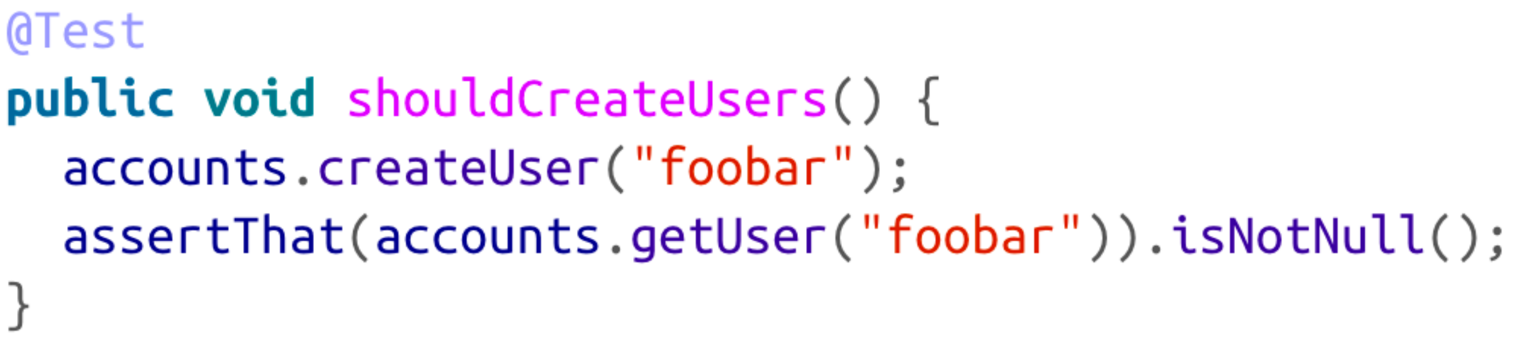
Avoiding *Brittle* Tests

- Strive for **unchanging** tests
- Test via public **API’s**
- Test **state**, not Interactions



```java
@Test
public void shouldWriteToDatabase() {
  accounts.createUser("foobar");
  verify(database).put("foobar");
}
```

versus

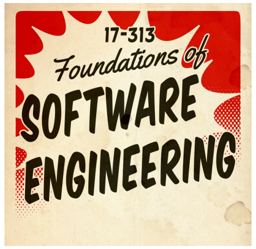


```java
@Test
public void shouldCreateUsers() {
  accounts.createUser("foobar");
  assertThat(accounts.getUser("foobar")).isNotNull();
}
```

Writing *Clear* Tests

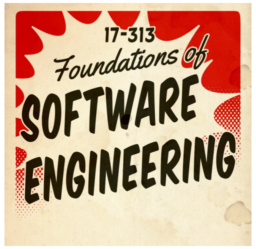
- Make your tests **complete** and **concise**
  - test’s body should contain all of the information needed to understand it without containing any irrelevant or distracting information
- Test **behaviours**, not methods
  - Behaviours can often be expressed using the words “given,” “when,” and “then”: “Given that a bank account is empty, when attempting to withdraw money from it, then the transaction is rejected.”
- **Don’t put logic** in tests
- Write **clear** failure messages

Black-box & Specification-Based Testing



- Test cases are often designed based on behavioral equivalence classes.
  - *Assumption*: if test passes for one value => test will pass for all values in the equivalence class.
- Systematic tests can be drawn from specification.
  - For example: A year is a *leap year* if:
    - the year is divisible by 4;
    - and the year is not divisible by 100;
      - except when the year is divisible by 400
  - Tests:
    - assert isLeapYear(1945) == false
    - assert isLeapYear(1944) == true
    - assert isLeapYear(1900) == false
    - assert isLeapYear(2000) == true

Boundary-Value Testing

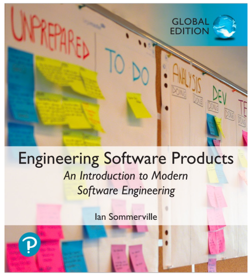


- *Aim*: Test for cases that are at the “boundary” of **equivalence classes** in the specification.
  - Small change in input moves it from one class to another.
  - Example: Testing a function *divide*(int a, int b)
    - One boundary may be at `a == b`

- *Edge case:* One of many parameters are at the boundary
  - E.g. for *divide*: a=0, b=42 or a=42, b = 0
  - E.g. for *sort*: list contains duplicates, list is empty

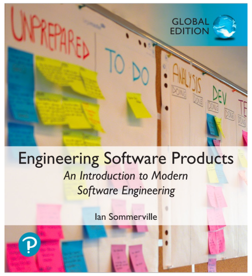
- *Corner* case: Combination of parameters are at the boundary
  - E.g. for *divide*: a=0, b=0

Unit testing guidelines (1)



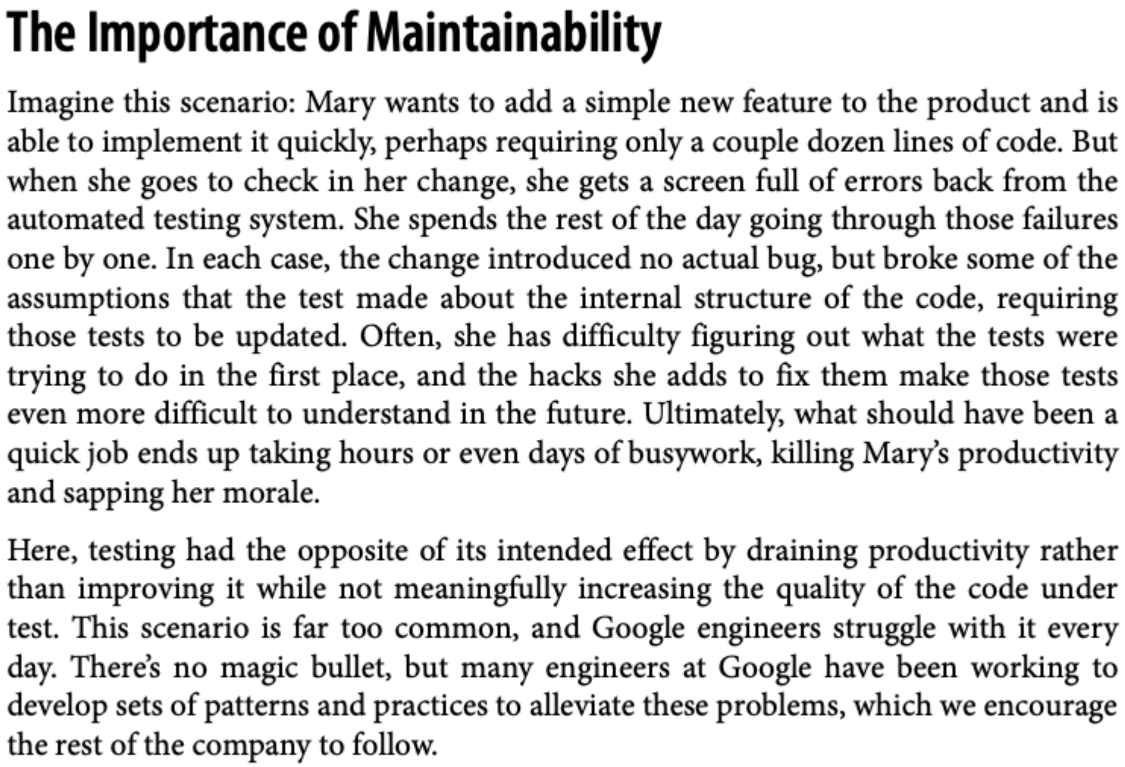
- ***Test edge cases***
  If your partition has upper and lower bounds (e.g. length of strings, numbers, etc.) choose inputs at the edges of the range.
- ***Force errors***
  Choose test inputs that force the system to generate all error messages. Choose test inputs that should generate invalid outputs.
- ***Fill buffers***
  Choose test inputs that cause all input buffers to overflow.
- ***Repeat yourself***
  Repeat the same test input or series of inputs several times.

Unit testing guidelines (2)



- ***Overflow and underflow***
  If your program does numeric calculations, choose test inputs that cause it to calculate very large or very small numbers.
- ***Don’t forget null and zero***
  If your program uses pointers or strings, always test with null pointers and strings. If you use sequences, test with an empty sequence. For numeric inputs, always test with zero.
- ***Keep count***
  When dealing with lists and list transformation, keep count of the number of elements in each list and check that these are consistent after each transformation.
- ***One is different***
  If your program deals with sequences, always test with sequences that have a single value.

But, remember...

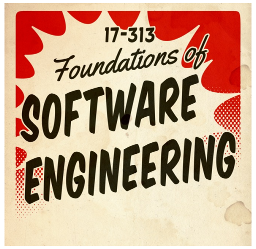


The Importance of Maintainability

Imagine this scenario: Mary wants to add a simple new feature to the product and is able to implement it quickly, perhaps requiring only a couple dozen lines of code. But when she goes to check in her change, she gets a screen full of errors back from the automated testing system. She spends the rest of the day going through those failures one by one. In each case, the change introduced no actual bug, but broke some of the assumptions that the test made about the internal structure of the code, requiring those tests to be updated. Often, she has difficulty figuring out what the tests were trying to do in the first place, and the hacks she adds to fix them make those tests even more difficult to understand in the future. Ultimately, what should have been a quick job ends up taking hours or even days of busywork, killing Mary's productivity and sapping her morale.

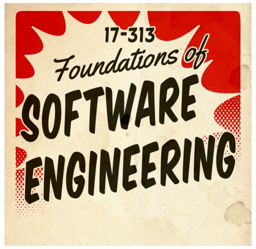
Here, testing had the opposite of its intended effect by draining productivity rather than improving it while not meaningfully increasing the quality of the code under test. This scenario is far too common, and Google engineers struggle with it every day. There's no magic bullet, but many engineers at Google have been working to develop sets of patterns and practices to alleviate these problems, which we encourage the rest of the company to follow.

Test Design principles



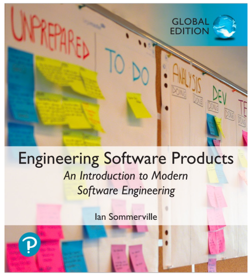
- Use public APIs only
- Clearly distinguish inputs, configuration, execution, and oracle
- Be simple; avoid complex control flow such as conditionals and loops
- Tests shouldn't need to be frequently changed or refactored
  - Definitely not as frequently as the code being tested changes

Anti-patterns



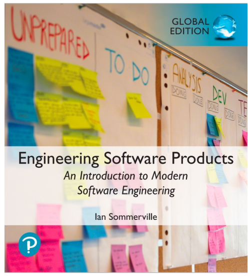
- Snoopy oracles
  - Relying on implementation state instead of observable behavior
  - E.g. Checking variables or fields instead of return values
- Brittle tests
  - Overfitting to special-case behavior instead of general principle
  - E.g. hard-coding message strings instead of behavior
- Slow tests
  - Self-explanatory (beware of heavy environments, I/O, and `sleep()`)
- Flaky tests
  - Tests that pass or fail nondeterministically
  - Often because of reliance on random inputs, timing (e.g. `sleep(1000)`), availability of external services (e.g. fetching data over the network in a unit test), or dependency on order of test execution (e.g. previous test sets up global variables in certain way)

Feature testing



- Features have to be tested to show that the functionality is implemented as expected and that the functionality meets the real needs of users.
  - For example, if your product has a feature that allows users to login using their Google account, then you have to check that this registers the user correctly and informs them of what information will be shared with Google.
  - You may want to check that it gives users the option to sign up for email information about your product.
- Normally, a feature that does several things is implemented by multiple, interacting, program units.
- These units may be implemented by different developers and all of these developers should be involved in the feature testing process.

Types of feature test


GLOBAL EDITION
Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville

- Interaction tests
  - These test the interactions between the units that implement the feature. The developers of the units that are combined to make up the feature may have different understandings of what is required of that feature.
  - These misunderstandings will not show up in unit tests but may only come to light when the units are integrated.
  - The integration may also reveal bugs in program units, which were not exposed by unit testing.
- Usefulness tests
  - These test that the feature implements what users are likely to want.
  - For example, the developers of a login with Google feature may have implemented an opt-out default on registration so that users receive all emails from a company. They must expressly choose what type of emails that they don't want.
  - What might be preferred is an opt-in default so that users choose what types of email they do want to receive.

User stories for the sign-in with Goog feature

GLOBAL EDITION

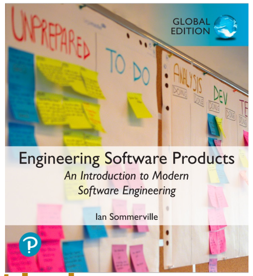
Engineering Software Products

An Introduction to Modern Software Engineering

Ian Sommerville

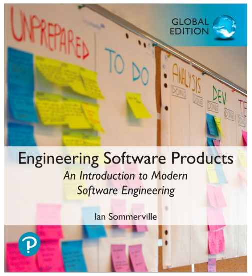
- ***User registration***
  As a user, I want to be able to login without creating a new account so that I don't have to remember another login id and password.
- ***Information sharing***
  As a user, I want to know what information you will share with other companies. I want to be able to cancel my registration if I don't want to share this information.
- ***Email choice***
  As a user, I want to be able to choose the types of email that I'll get from you when I register for an account.

Feature tests for sign-in with Google



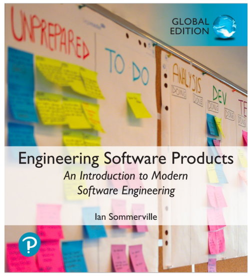
- ***Initial login screen***
  Test that the screen displaying a request for Google account credentials is correctly displayed when a user clicks on the 'Sign-in with Google' link. Test that the login is completed if the user is already logged in to Google.
- ***Incorrect credentials***
  Test that the error message and retry screen is displayed if the user inputs incorrect Google credentials.
- ***Shared information***
  Test that the information shared with Google is displayed, along with a cancel or confirm option. Test that the registration is cancelled if the cancel option is chosen.
- ***Email opt-in***
  Test that the user is offered a menu of options for email information and can choose multiple items to opt-in to emails. Test that the user is not registered for any emails if no options are selected.

System and release testing


GLOBAL EDITION
Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville

- System testing involves testing the system as a whole, rather than the individual system features.
- System testing should focus on four things:
  - Testing to discover if there are unexpected and unwanted interactions between the features in a system.
  - Testing to discover if the system features work together effectively to support what users really want to do with the system.
  - Testing the system to make sure it operates in the expected way in the different environments where it will be used.
  - Testing the responsiveness, throughput, security and other quality attributes of the system.

Scenario-based testing

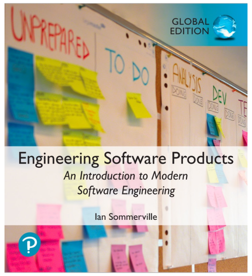

Engineering Software Products

An Introduction to Modern Software Engineering

Ian Sommerville

- The best way to systematically test a system is to start with a set of scenarios that describe possible uses of the system and then work through these scenarios each time a new version of the system is created.
- Using the scenario, you identify a set of end-to-end pathways that users might follow when using the system.
- An end-to-end pathway is a sequence of actions from starting to use the system for the task, through to completion of the task.

Choosing a holiday destination



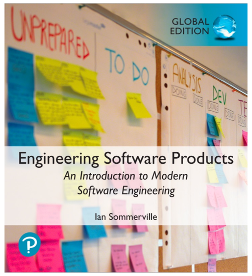
- Andrew and Maria have a two year old son and a four month old daughter. They live in Scotland and they want to have a holiday in the sunshine. However, they are concerned about the hassle of flying with young children. They decide to try a family holiday planner product to help them choose a destination that is easy to get to and that fits in with their childrens' routines.

- Maria navigates to the holiday planner website and selects the 'find a destination' page. This presents a screen with a number of options. She can choose a specific destination or can choose a departure airport and find all destinations that have direct flights from that airport. She can also input the time band that she'd prefer for flights, holiday dates and a maximum cost per person.

- Edinburgh is their closest departure airport. She chooses 'find direct flights'. The system then presents a list of countries that have direct flights from Edinburgh and the days when these flights operate. She selects France, Italy, Portugal and Spain and requests further information about these flights. She then sets a filter to display flights that leave on a Saturday or Sunday after 7.30am and arrive before 6pm.

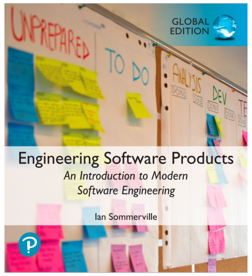
- She also sets the maximum acceptable cost for a flight. The list of flights is pruned according to the filter and is redisplayed. Maria then clicks on the flight she wants. This opens a tab in her browser showing a booking form for this flight on the airline's website.

End-to-end pathways


Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville
GLOBAL EDITION

1. User inputs departure airport and chooses to see only direct flights. User quits.
2. User inputs departure airport and chooses to see all flights. User quits.
3. User chooses destination country and chooses to see all flights. User quits.
4. User inputs departure airport and chooses to see direct flights. User sets filter specifying departure times and prices. User quits.
5. User inputs departure airport and chooses to see direct flights. User sets filter specifying departure times and prices. User selects a displayed flight and clicks through to airline website. User returns to holiday planner after booking flight.

Release testing



- Release testing is a type of system testing where a system that's intended for release to customers is tested.

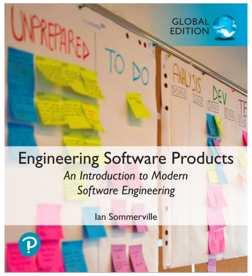
- The fundamental differences between release testing and system testing are:

    - Release testing tests the system in its real operational environment rather than in a test environment. Problems commonly arise with real user data, which is sometimes more complex and less reliable than test data.

    - The aim of release testing is to decide if the system is good enough to release, not to detect bugs in the system. Therefore, some tests that 'fail' may be ignored if these have minimal consequences for most users.

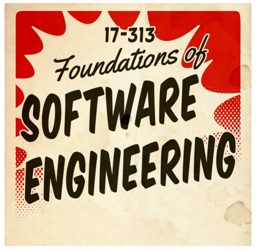
- Preparing a system for release involves packaging that system for deployment (e.g. in a container if it is a cloud service) and installing software and libraries that are used by your product. You must define configuration parameters such as the name of a root directory, the database size limit per user and so on.

Risk analysis


GLOBAL EDITION
Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville

- Once you have identified security risks, you then analyze them to assess how they might arise. For example, for the first risk two slides earlier (unauthorized attacker) there are several possibilities:
  - The user has set weak passwords that can be guessed by an attacker.
  - The system's password file has been stolen and passwords discovered by attacker.
  - The user has not set up two-factor authentication.
  - An attacker has discovered credentials of a legitimate user through social engineering techniques.
- You can then develop tests to check some of these possibilities.
  - For example, you might run a test to check that the code that allows users to set their passwords always checks the strength of passwords.

White-box or Structural Testing

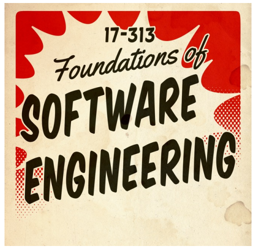


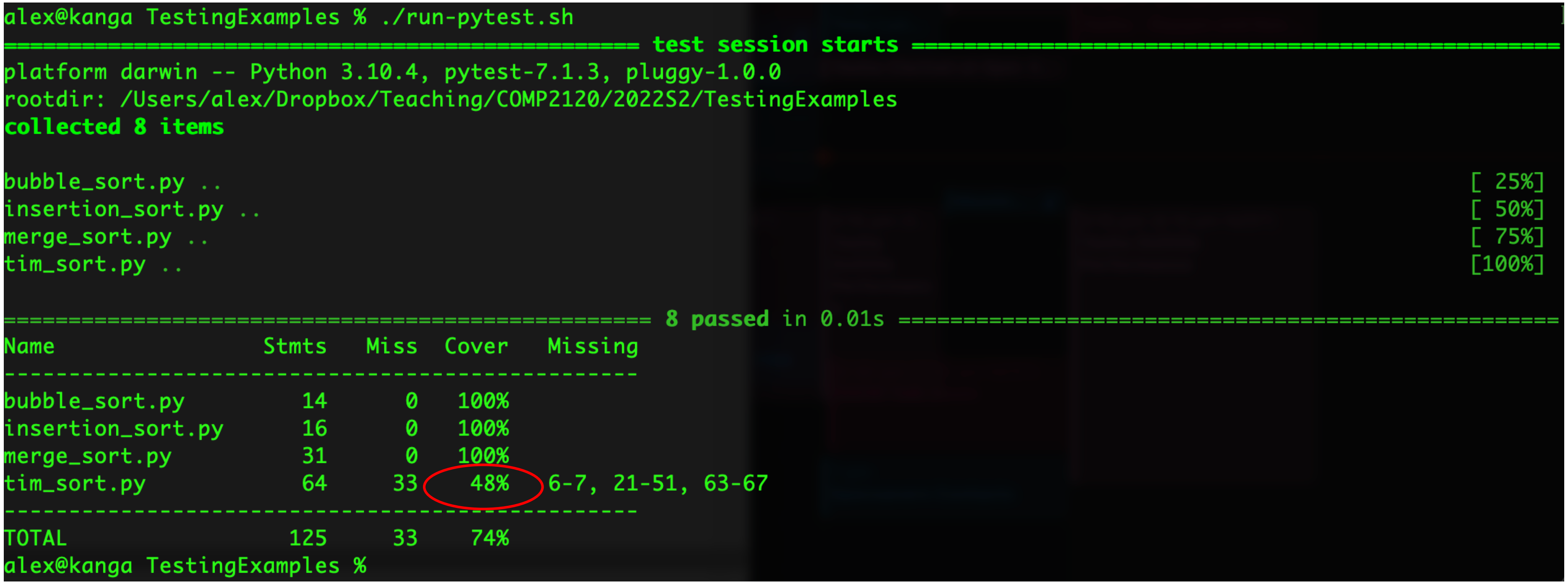
- *Aim*: Test for cases that exercise various program elements (e.g. functions, lines, statements, branches)

- *Key idea*: If you don't execute some code, you can't find bugs in that code. So, let's execute all the code.

- Which one do you think is harder: black-box boundary-value testing or white-box structural testing?

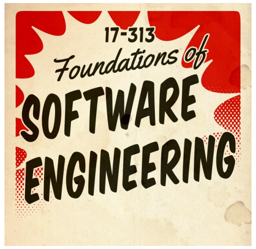
Coverage of the Basic Unit Test





```
alex@kanga TestingExamples % ./run-pytest.sh
================================================== test session starts ==================================================
platform darwin -- Python 3.10.4, pytest-7.1.3, pluggy-1.0.0
rootdir: /Users/alex/Dropbox/Teaching/COMP2120/2022S2/TestingExamples
collected 8 items

bubble_sort.py ..                                                                                                 [ 25%]
insertion_sort.py ..                                                                                              [ 50%]
merge_sort.py ..                                                                                                  [ 75%]
tim_sort.py ..                                                                                                    [100%]

=================================================== 8 passed in 0.01s ===================================================
Name                Stmts   Miss  Cover   Missing
-------------------------------------------------
bubble_sort.py         14      0   100%
insertion_sort.py      16      0   100%
merge_sort.py          31      0   100%
tim_sort.py            64     33    48%   6-7, 21-51, 63-67
-------------------------------------------------
TOTAL                 125     33    74%
alex@kanga TestingExamples %
```

But the basic unit test worked well for Merge and otherSort....

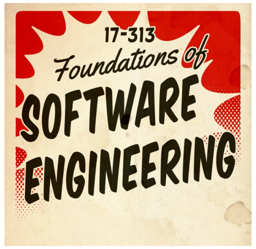


Coverage != Completeness

Mutation Testing

- *Key idea*: Inject bugs in the program by *mutating* the source code.
- *Ideally*: at least one test should fail on the mutated program (= catch bug).
  - If this happens, the mutant is said to be “killed”.
  - If all tests continue to pass under the mutated program, then the mutant is said to “survive”.
  - Mutation score = (mutants killed) / (total mutants). This is a better predictor of bug-finding capability than coverage.
- *Competent programmer assumption*: programs are mostly correct, except for very small errors.
  - Shows that tests are falsifiable at the boundary of implementation (as opposed to boundary of specification).

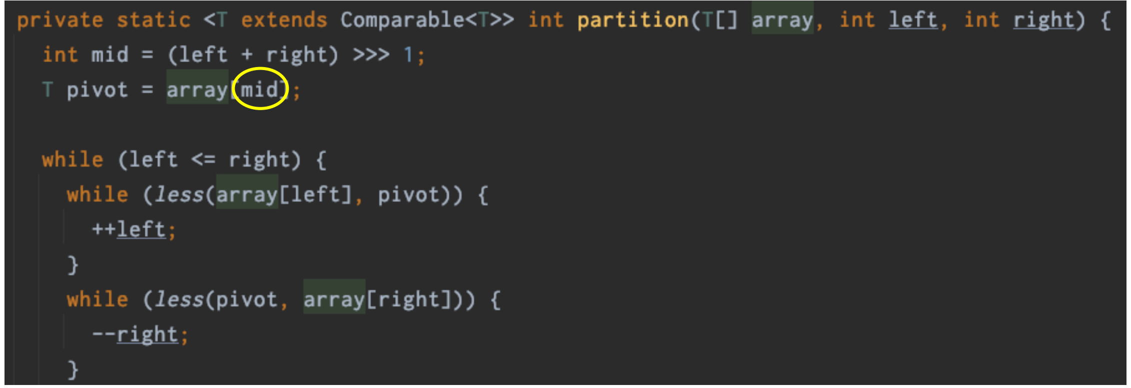
Mutation Testing



- Sample mutations include:
  - Change ‘a + b’ to ‘a – b’
  - Change ‘if (a > b)’ to ‘if (a >= b)’ or ‘if(b > a)’
  - Change ‘i++’ to ‘i—’
  - Replace integer variables with 0
  - Change ‘return x’ to ‘return True’ (or some other constant)
  - Delete lines containing void method calls (e.g. ‘x.setFoo(1)’)
  - … and many more
- Over time, standard list of mutators curated by researchers
- Pitest is a popular mutation testing tool for Java (pitest.org)

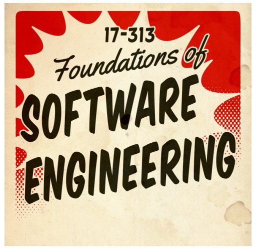
Mutation Testing

- Nice idea but has several limitations:
  1. *Equivalent* mutations: Modifications that do not affect program semantics (e.g. affecting the pivot in Quicksort).
  2. Needs a pretty *complete* test oracle: Otherwise, some genuine bugs may never be caught. We'll come back to this point later.
  3. Expensive to run. N mutants require N test executions. Program testing costs scale quadratically (because N also grows with size).



```
private static <T extends Comparable<T>> int partition(T[] array, int left, int right) {
  int mid = (left + right) >>> 1;
  T pivot = array[mid];

  while (left <= right) {
    while (less(array[left], pivot)) {
      ++left;
    }
    while (less(pivot, array[right])) {
      --right;
    }
```

Test Oracles



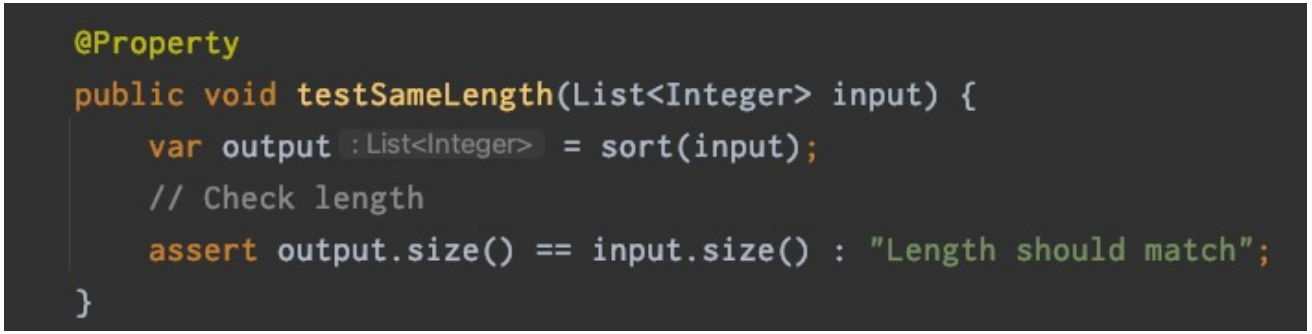
- Obvious in some applications (e.g. “sort()”) but more challenging in others (e.g. “encrypt()” or UI-based tests)

- Lack of good oracles can limit the scalability of testing. Easy to generate lots of input data, but not easy to validate if output (or other program behavior) is correct.

- Fortunately, we have some tricks.

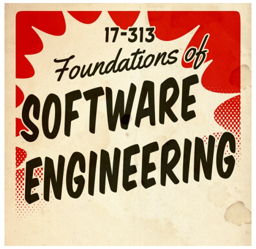
Property-Based Testing

- Intends to validate invariants that are always true of a computed result.
  - E.g. if testing a list-reversing function called `rev`, then we have the invariant: `rev(rev(list)).equals(list)`
- Key idea: Can now easily scale testing to very large data sets, either hand-written or automatically generated, without the need for hard-coding expected outputs completely.



```
@Property
public void testSameLength(List<Integer> input) {
    var output : List<Integer> = sort(input);
    // Check length
    assert output.size() == input.size() : "Length should match";
}
```

Differential Testing



- If you have two implementations of the same specification, then their output should match on all inputs.
  - E.g. `timSort(x).equals(quickSort(x))` → should always be true
  - Special case of a property test, with a free oracle.
- If a differential test fails, at least one of the two implementations is wrong.
  - But which one?
  - If you have N > 2 implementations, run them all and compare. Majority wins (the odd one out is buggy).
- Differential testing works well when testing programs that implement standard specifications such as compilers, browsers, SQL engines, XML/JSON parsers, media players, etc.
  - Not feasible in general

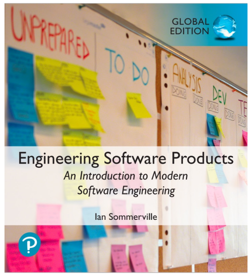
Regression Testing

- Differential testing through time (or *versions*, say V1 and V2).

- Assuming V1 and V2 don't add a new feature or fix a known bug, then f(x) in V1 should give the same result as f(x) in V2.

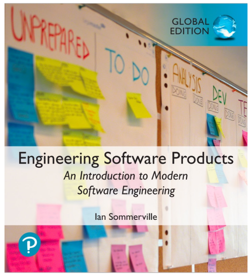
- *Key Idea*: Assume the current version is correct. Run program on current version and log output. Compare all future versions to that output.

Security testing


GLOBAL EDITION
Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville

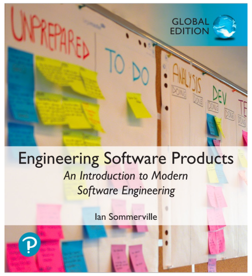
- Security testing aims to find vulnerabilities that may be exploited by an attacker and to provide convincing evidence that the system is sufficiently secure.
- The tests should demonstrate that the system can resist attacks on its availability, attacks that try to inject malware and attacks that try to corrupt or steal users' data and identity.
- Comprehensive security testing requires specialist knowledge of software vulnerabilities and approaches to testing that can find these vulnerabilities.

Risk-based security testing


GLOBAL EDITION
Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville

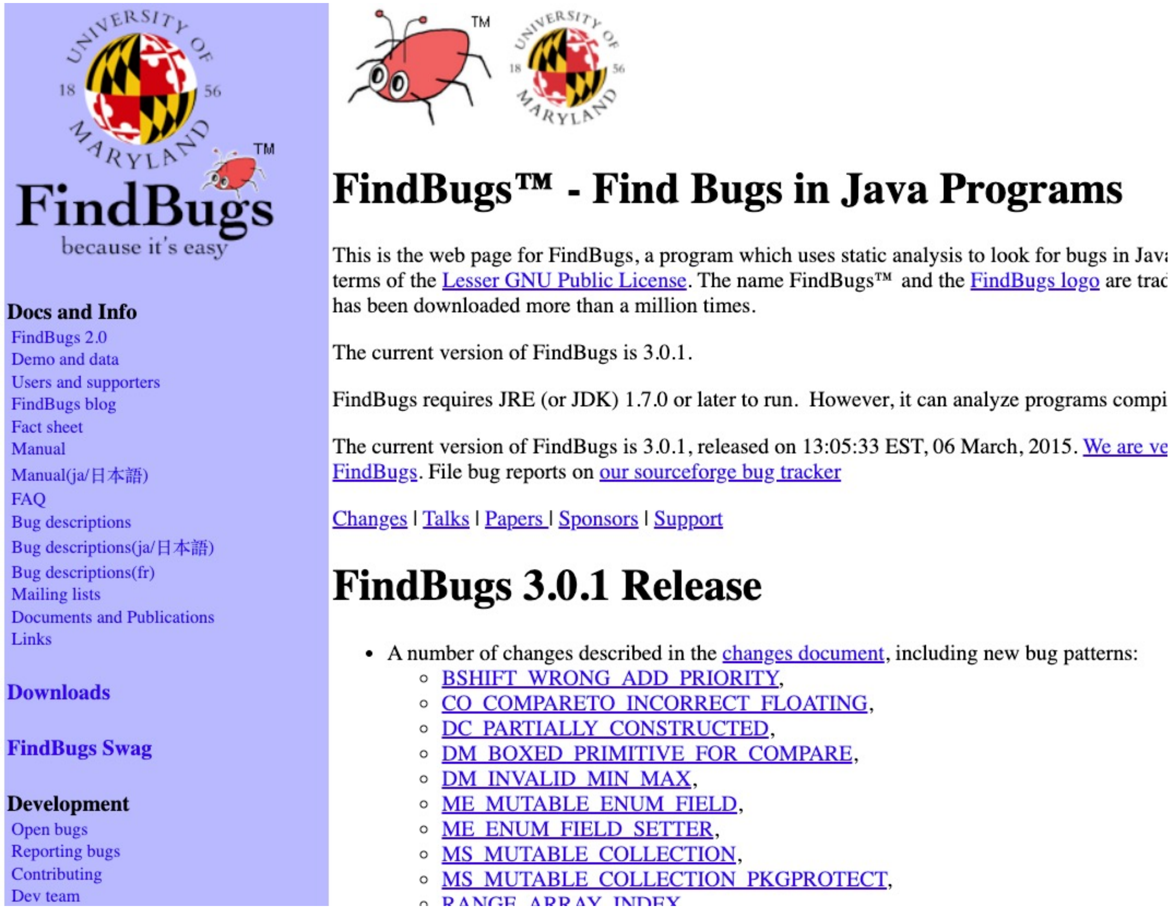
- A risk-based approach to security testing involves identifying common risks and developing tests to demonstrate that the system protects itself from these risks.
- You may also use automated tools that scan your system to check for known vulnerabilities, such as unused HTTP ports being left open.
- Based on the risks that have been identified, you then design tests and checks to see if the system is vulnerable.
- It may be possible to construct automated tests for some of these checks, but others inevitably involve manual checking of the system's behaviour and its files.

Examples of security risks


GLOBAL EDITION
Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville

- Unauthorized attacker gains access to a system using authorized credentials
- Authorized individual accesses resources that are forbidden to them
- Authentication system fails to detect unauthorized attacker
- Attacker gains access to database using SQL poisoning attack
- Improper management of HTTP session
- HTTP session cookies revealed to attacker
- Confidential data are unencrypted
- Encryption keys are leaked to potential attackers

FindBugs (2006!)



Docs and Info
- FindBugs 2.0
- Demo and data
- Users and supporters
- FindBugs blog
- Fact sheet
- Manual
- Manual(ja/日本語)
- FAQ
- Bug descriptions
- Bug descriptions(ja/日本語)
- Bug descriptions(fr)
- Mailing lists
- Documents and Publications
- Links

Downloads

FindBugs Swag

Development
- Open bugs
- Reporting bugs
- Contributing
- Dev team

FindBugs™ - Find Bugs in Java Programs

This is the web page for FindBugs, a program which uses static analysis to look for bugs in Jav terms of the Lesser GNU Public License. The name FindBugs™ and the FindBugs logo are trac has been downloaded more than a million times.

The current version of FindBugs is 3.0.1.

FindBugs requires JRE (or JDK) 1.7.0 or later to run. However, it can analyze programs compi

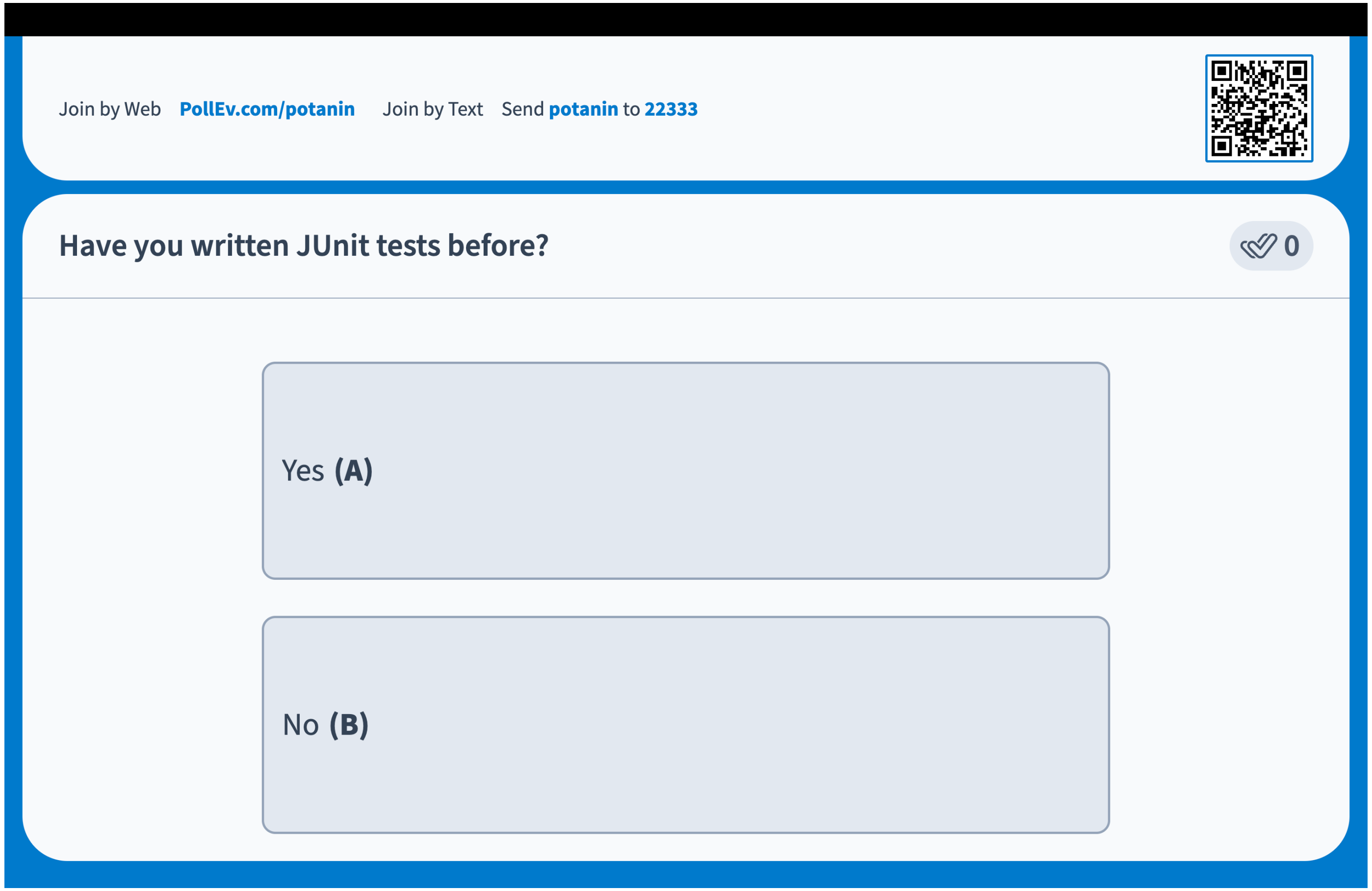
The current version of FindBugs is 3.0.1, released on 13:05:33 EST, 06 March, 2015. We are ve FindBugs. File bug reports on our sourceforge bug tracker

Changes | Talks | Papers | Sponsors | Support

FindBugs 3.0.1 Release

- A number of changes described in the changes document, including new bug patterns:
  - BSHIFT_WRONG_ADD_PRIORITY,
  - CO_COMPARETO_INCORRECT_FLOATING,
  - DC_PARTIALLY_CONSTRUCTED,
  - DM_BOXED_PRIMITIVE_FOR_COMPARE,
  - DM_INVALID_MIN_MAX,
  - ME_MUTABLE_ENUM_FIELD,
  - ME_ENUM_FIELD_SETTER,
  - MS_MUTABLE_COLLECTION,
  - MS_MUTABLE_COLLECTION_PKGPROTECT,
  - RANGE_ARRAY_INDEX,

Poll Everywhere Time!

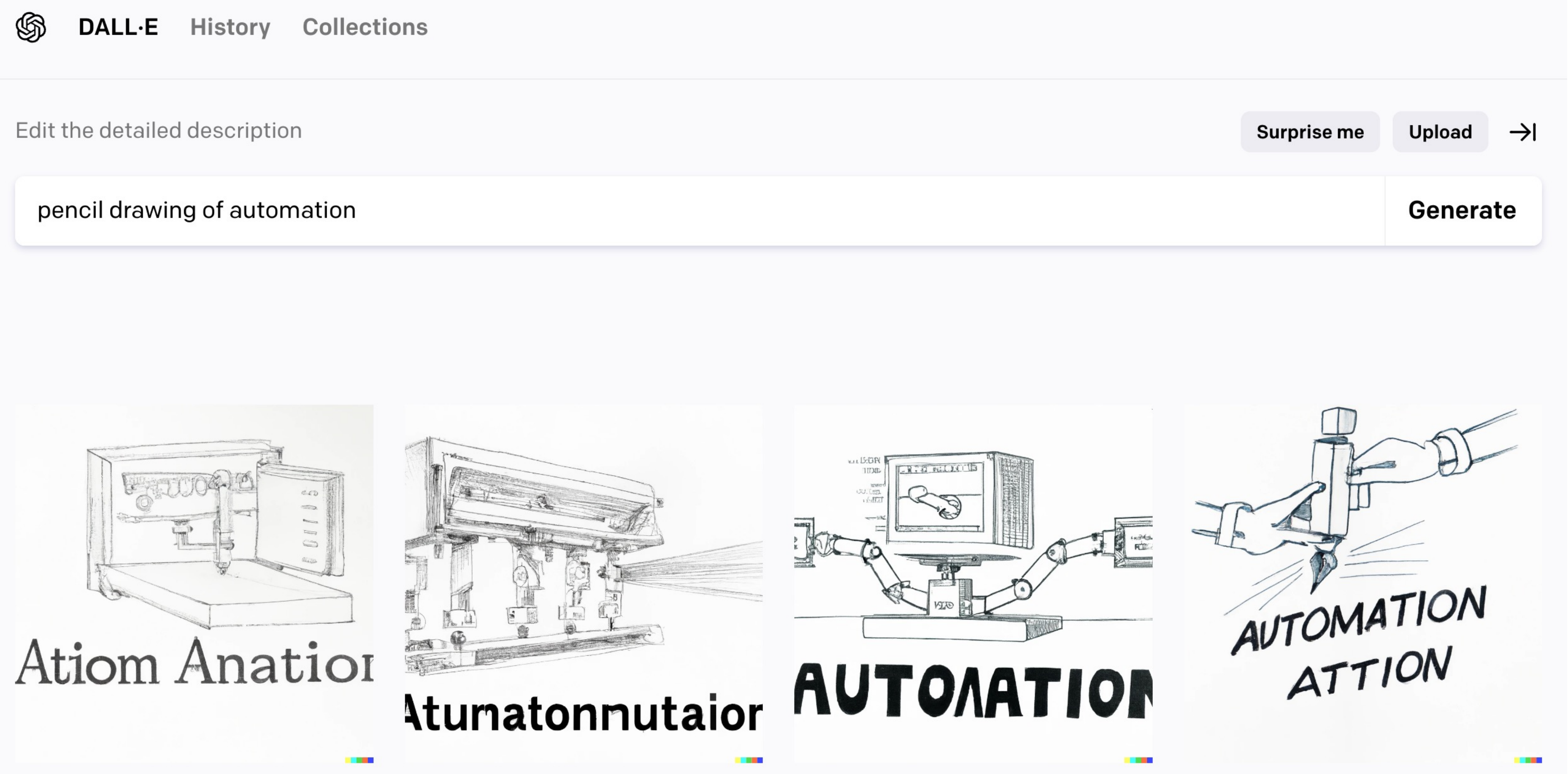


Join by Web **PollEv.com/potanin** Join by Text Send **potanin** to **22333**

Have you written JUnit tests before? ✓ 0

Yes **(A)**

No **(B)**

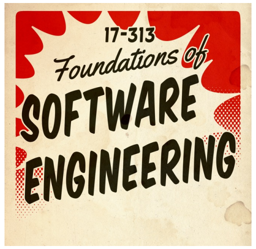


Edit the detailed description

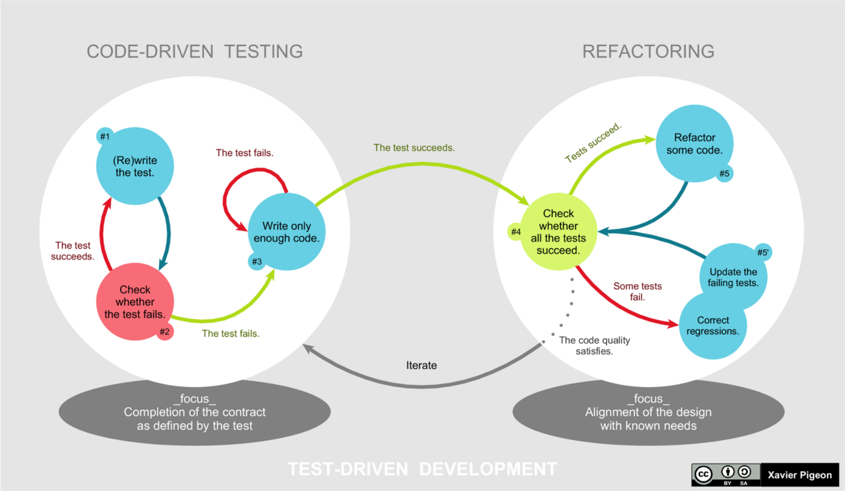
pencil drawing of automation

Test Automation

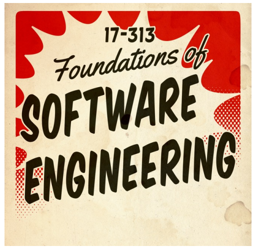
Test Driven Development (TDD)



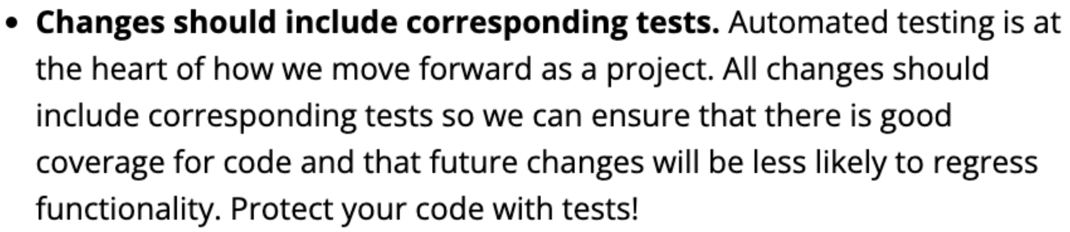
- Tests first!
- Popular agile technique
- Write tests as specifications before code
- Never write code without a failing test
- Claims:
  - Design approach toward testable design
  - Think about interfaces first
  - Avoid unneeded code
  - Higher product quality
  - Higher test suite quality
  - Higher overall productivity



Common bar for contributions

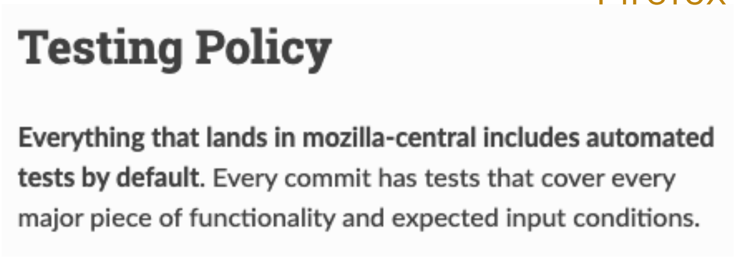


Chromium



- **Changes should include corresponding tests.** Automated testing is at the heart of how we move forward as a project. All changes should include corresponding tests so we can ensure that there is good coverage for code and that future changes will be less likely to regress functionality. Protect your code with tests!

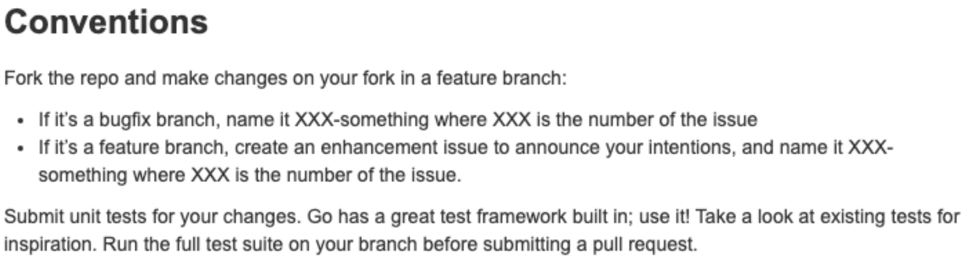
Firefox



Testing Policy

Everything that lands in mozilla-central includes automated tests by default. Every commit has tests that cover every major piece of functionality and expected input conditions.

Docker



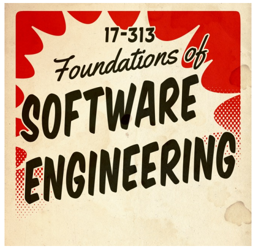
Conventions

Fork the repo and make changes on your fork in a feature branch:

- If it's a bugfix branch, name it XXX-something where XXX is the number of the issue
- If it's a feature branch, create an enhancement issue to announce your intentions, and name it XXX-something where XXX is the number of the issue.

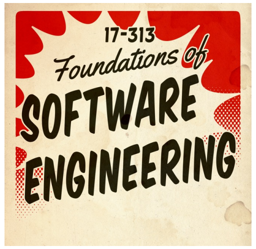
Submit unit tests for your changes. Go has a great test framework built in; use it! Take a look at existing tests for inspiration. Run the full test suite on your branch before submitting a pull request.

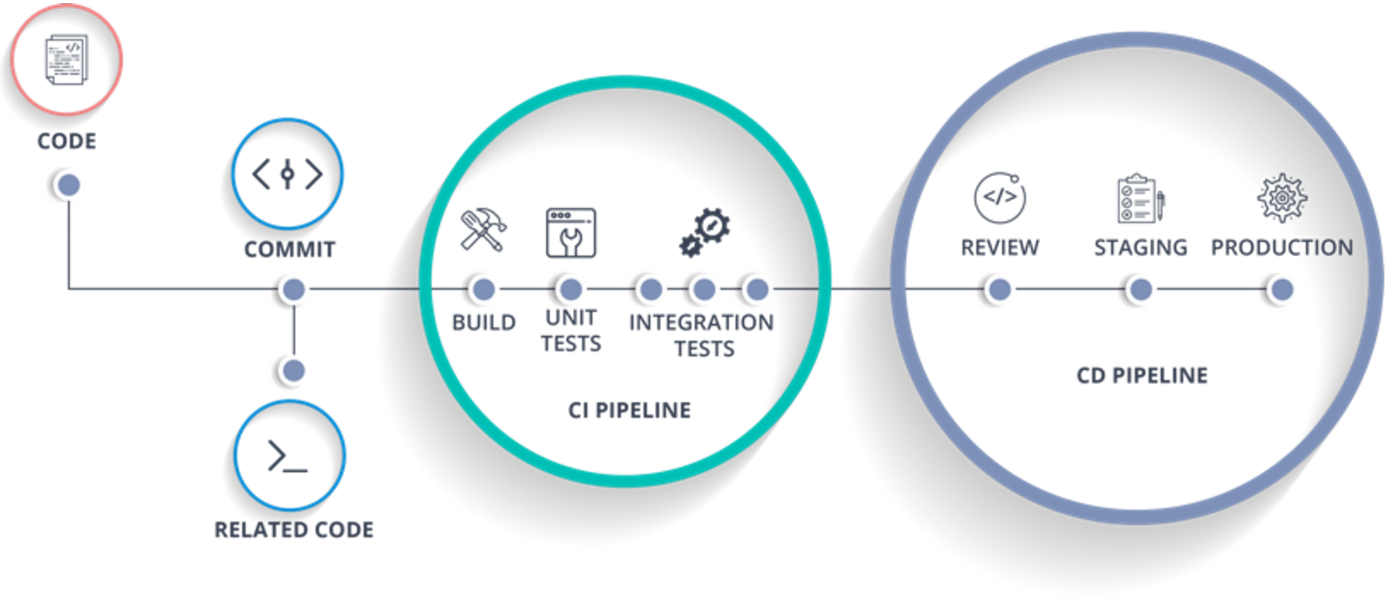
Regression testing



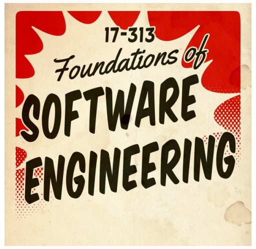
- Usual model:
  - Introduce regression tests for bug fixes, etc.
  - Compare results as code evolves
    - Code1 + TestSet → TestResults1
    - Code2 + TestSet → TestResults2
  - As code evolves, compare TestResults1 with TestResults2, etc.
- Benefits:
  - Ensure bug fixes remain in place and bugs do not reappear.
  - Reduces reliance on specifications, as <TestSet,TestResults1> acts as one.

Continuous Integration



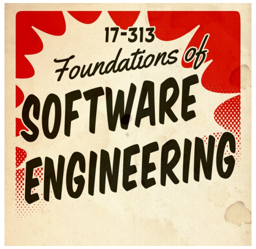


Testing Levels



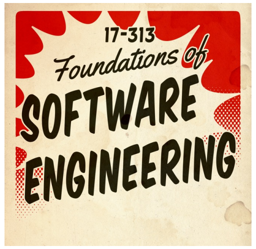
- **Unit testing**
- **Integration testing**
- **System testing**

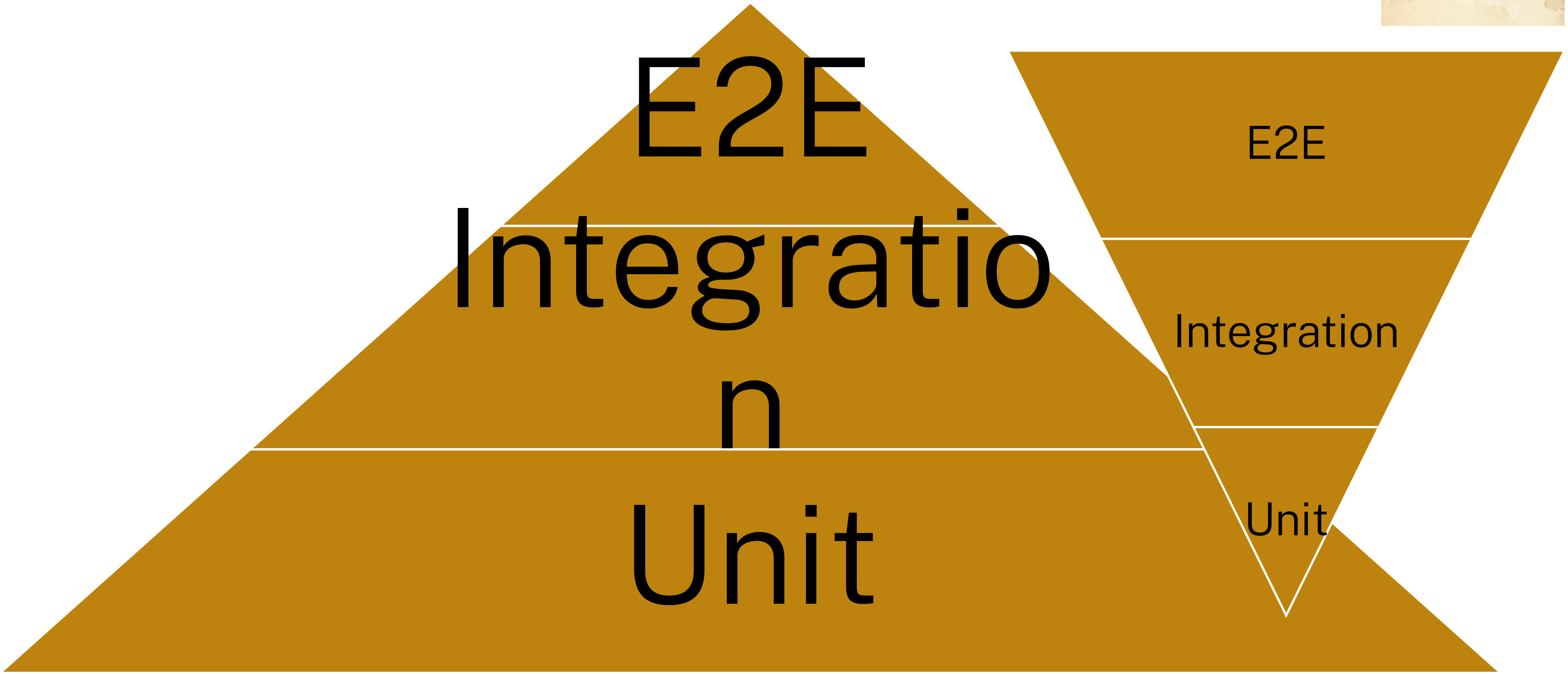
Testing Levels



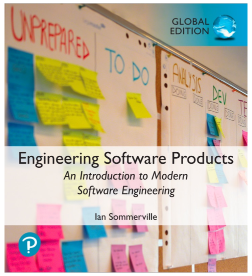
- **Unit testing**
  - Code level, E.g. is a function implemented correctly?
  - Does not require setting up a complex environment
- **Integration testing**
  - Do components interact correctly? E.g. a feature that cuts across client and server.
  - Usually requires some environment setup, but can abstract/mock out other components that are not being tested (e.g. network)
- **System testing**
  - Validating the whole system end-to-end (E2E)
  - Requires complete deployment in a staging area, but fake data
- **Testing in production**
  - Real data but more risks

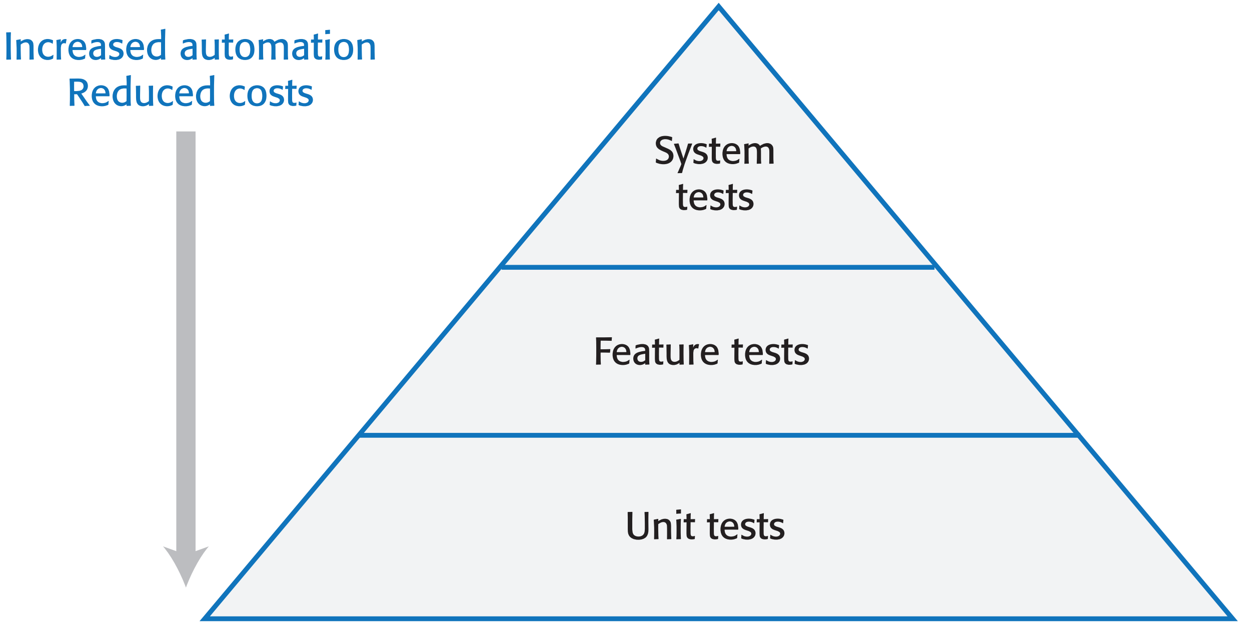
What’s a good distribution of test levels?



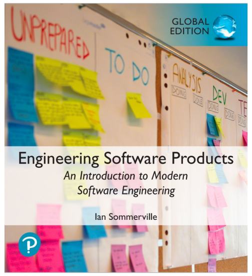


The test pyramid





Automated tests

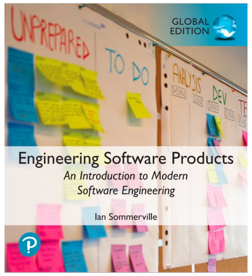

Engineering Software Products

An Introduction to Modern Software Engineering

Ian Sommerville

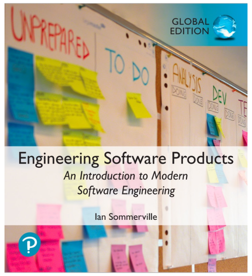
- It is good practice to structure automated tests into three parts:
  - Arrange You set up the system to run the test. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed.
  - Action You call the unit that is being tested with the test parameters.
  - Assert You make an assertion about what should hold if the unit being tested has executed successfully. In program on the previous slide, we use assertEquals, which checks if its parameters are equal.
- If you use equivalence partitions to identify test inputs, you should have several automated tests based on correct and incorrect inputs from each partition.

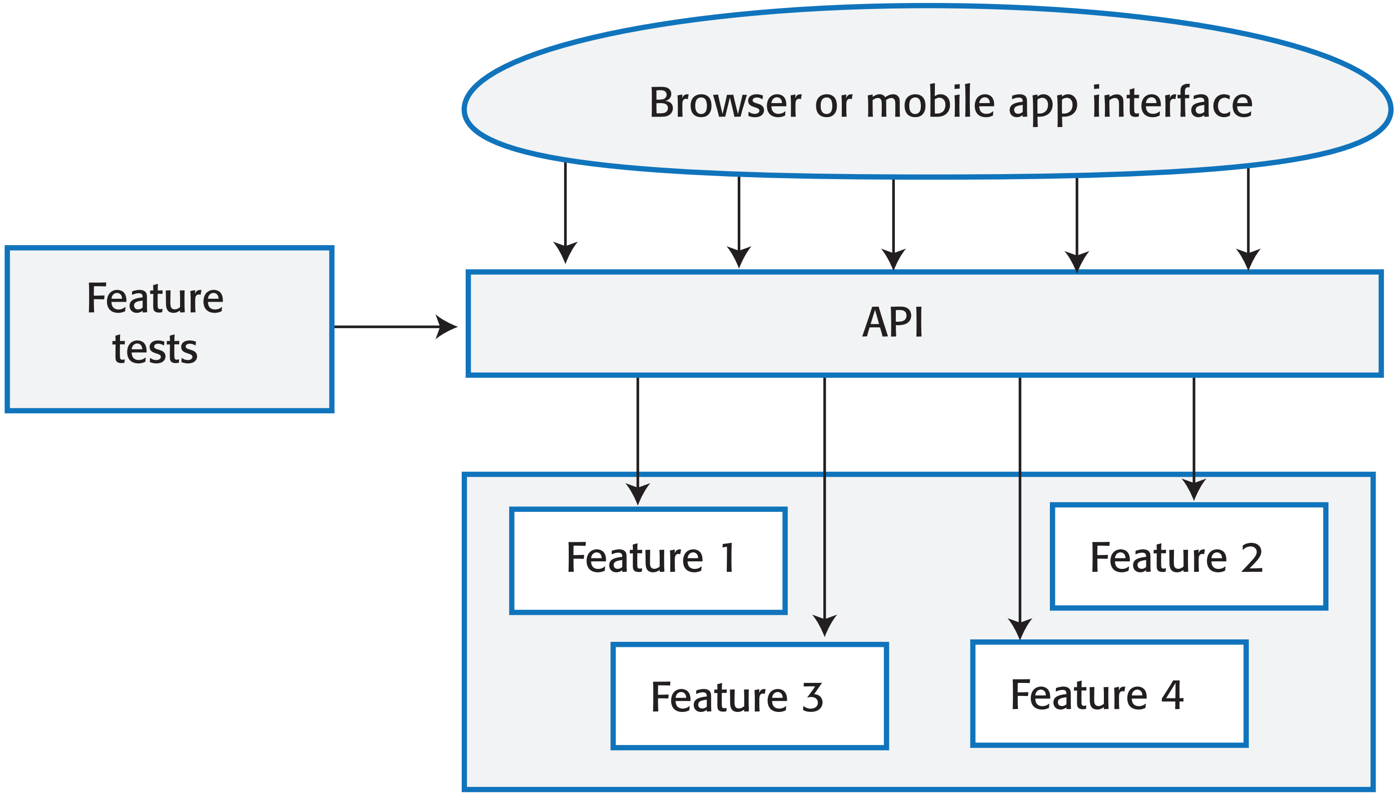
Automated feature testing


GLOBAL EDITION
Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville

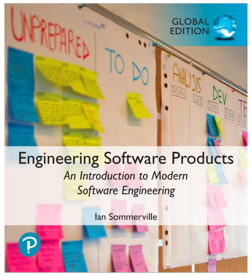
- Generally, users access features through the product’s graphical user interface (GUI).
- However, GUI-based testing is expensive to automate so it is best to design your product so that its features can be directly accessed through an API and not just from the user interface.
- The feature tests can then access features directly through the API without the need for direct user interaction through the system’s GUI.
- Accessing features through an API has the additional benefit that it is possible to re-implement the GUI without changing the functional components of the software.

Feature editing through an API



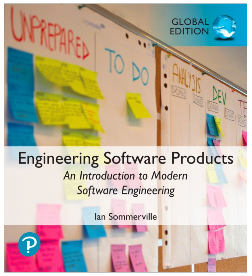


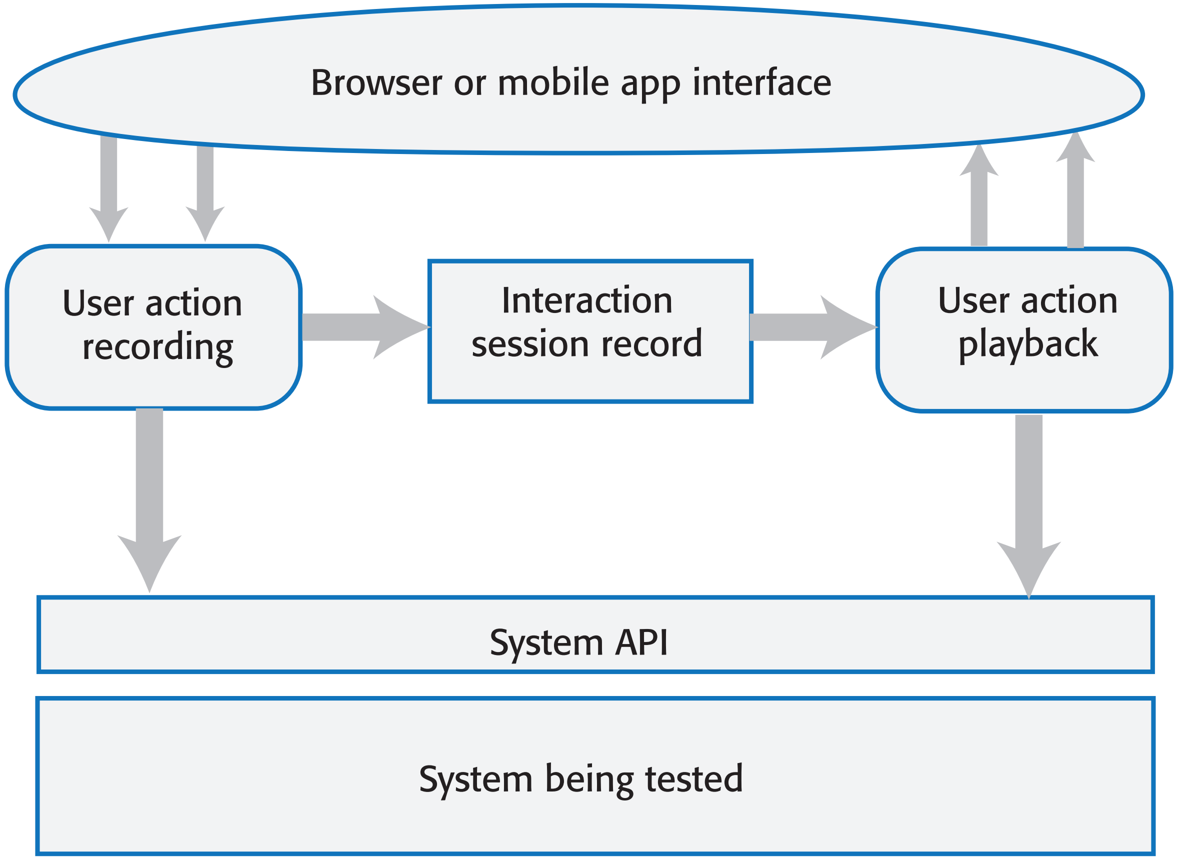
System testing



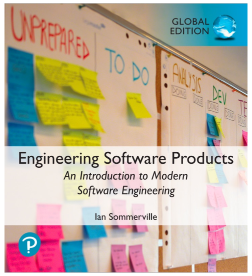
- System testing, which should follow feature testing, involves testing the system as a surrogate user.
- As a system tester, you go through a process of selecting items from menus, making screen selections, inputting information from the keyboard and so on.
- You are looking for interactions between features that cause problems, sequences of actions that lead to system crashes and so on.
- Manual system testing, when testers have to repeat sequences of actions, is boring and error-prone. In some cases, the timing of actions is important and is practically impossible to repeat consistently.
    - To avoid these problems, testing tools have been developed that can record a series of actions and automatically replay these when a system is retested

Interaction recording and playback



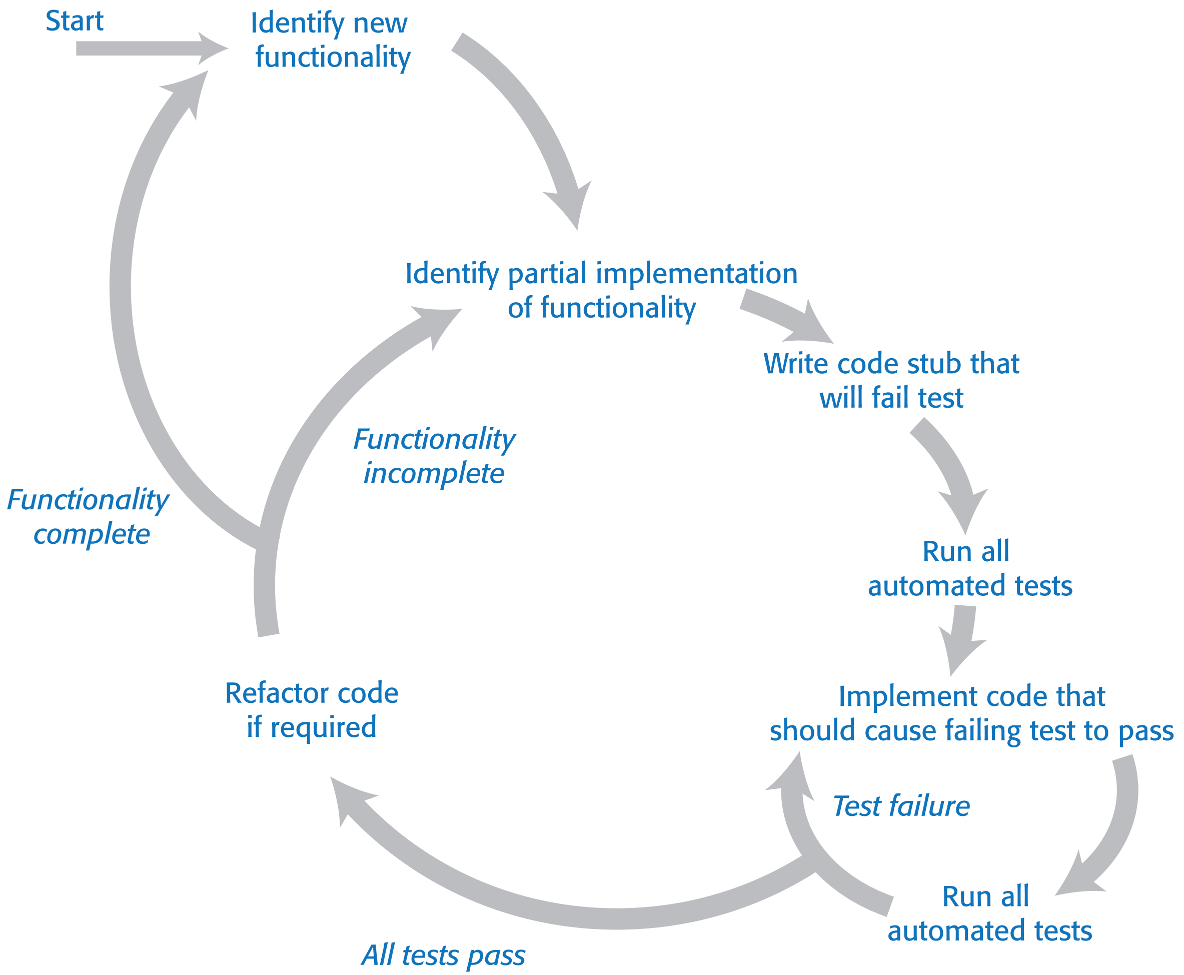


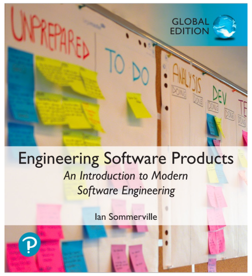
Test-driven development



- Test-driven development (TDD) is an approach to program development that is based around the general idea that you should write an executable test or tests for code that you are writing before you write the code.
- It was introduced by early users of the Extreme Programming agile method, but it can be used with any incremental development approach.
- Test-driven development works best for the development of individual program units and it is more difficult to apply to system testing.
- Even the strongest advocates of TDD accept that it is challenging to use this approach when you are developing and testing systems with graphical user interfaces.

Test-driven development

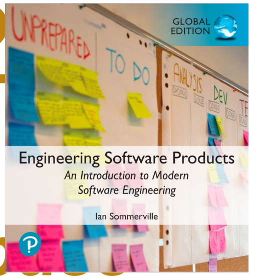

Start
Identify new functionality
Identify partial implementation of functionality
Write code stub that will fail test
Run all automated tests
Implement code that should cause failing test to pass
Test failure
Run all automated tests
All tests pass
Refactor code if required
Functionality incomplete
Functionality complete


Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville
GLOBAL EDITION

Stages of test-driven development (1)

- ***Identify partial implementation***
  Break down the implementation of the functionality required into smaller mini-units. Choose one of these mini-units for implementation.
- ***Write mini-unit tests***
  Write one or more automated tests for the mini-unit that you have chosen for implementation. The mini-unit should pass these tests if it is properly implemented.
- **Write a code stub that will fail test**
  Write incomplete code that will be called to implement the mini-unit. You know this will fail.
- ***Run all existing automated tests***
  All previous tests should pass. The test for the incomplete code should fail.

Stages of test-driven development (2


UNPREPARED
TO DO
ANALYSIS
DEV
GLOBAL EDITION
Engineering Software Products
An Introduction to Modern Software Engineering
Ian Sommerville
P

- ***Implement code that should cause the failing test to pass***
  Write code to implement the mini-unit, which should cause it to operate correctly
- ***Rerun all automated tests***
  If any tests fail, your code is probably incorrect. Keep working on it until all tests pass.
- ***Refactor code if necessary***
  If all tests pass, you can move on to implementing the next mini-unit. If you see ways of improving your code, you should do this before the next stage of implementation.

Benefits of test-driven development

GLOBAL EDITION

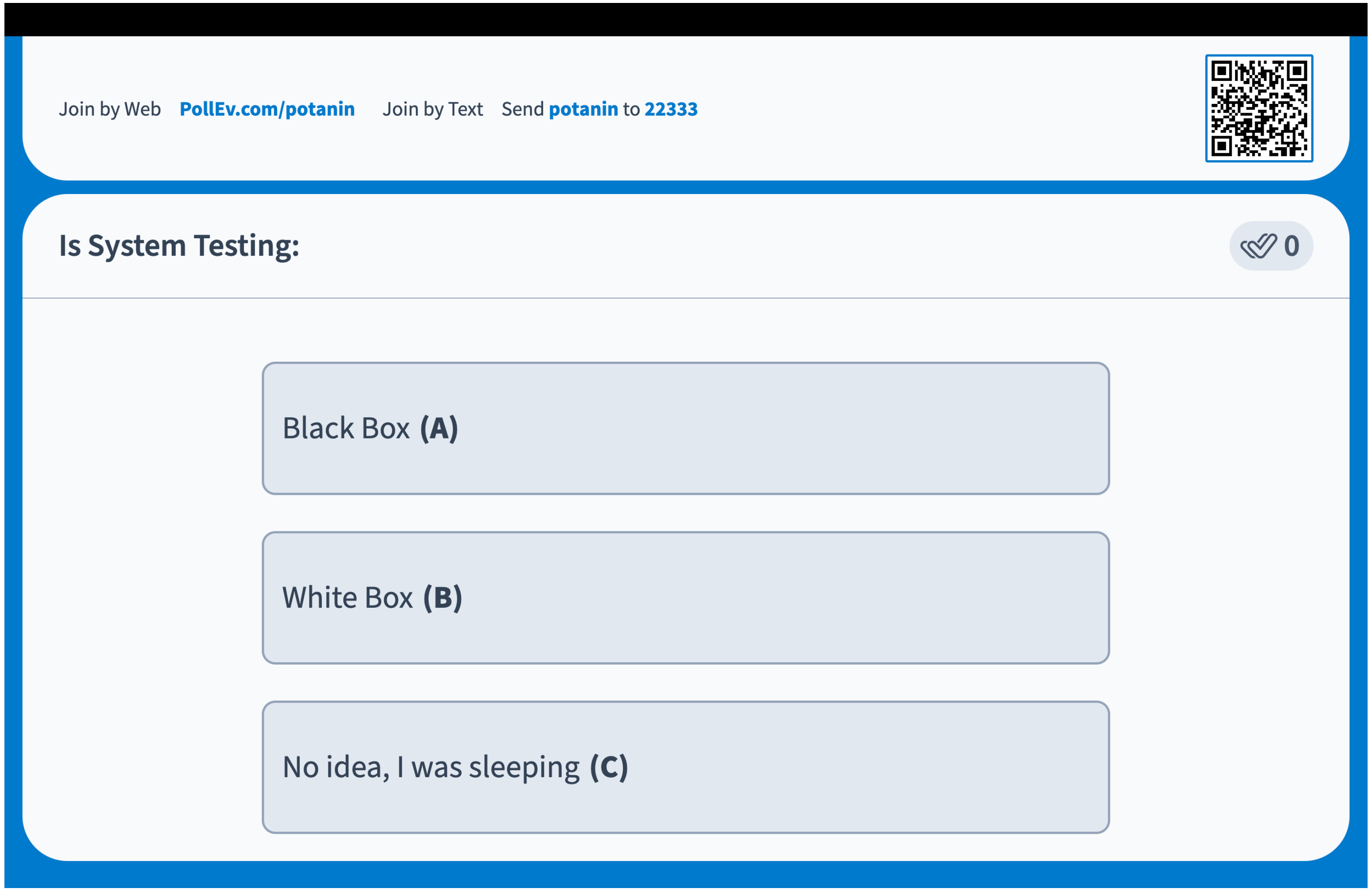
Engineering Software Products

An Introduction to Modern Software Engineering

Ian Sommerville

- It is a systematic approach to testing in which tests are clearly linked to secti the program code.
    - This means you can be confident that your tests cover all of the code that has been developed and that there are no untested code sections in the delivered code. In my view, this is the most significant benefit of TDD.
- The tests act as a written specification for the program code. In principle at least, it should be possible to understand what the program does by reading the tests.
- Debugging is simplified because, when a program failure is observed, you can immediately link this to the last increment of code that you added to the system.
- It is argued that TDD leads to simpler code as programmers only write code that’s necessary to pass tests. They don’t over-engineer their code with complex features that aren’t needed.

Poll Everywhere Time!



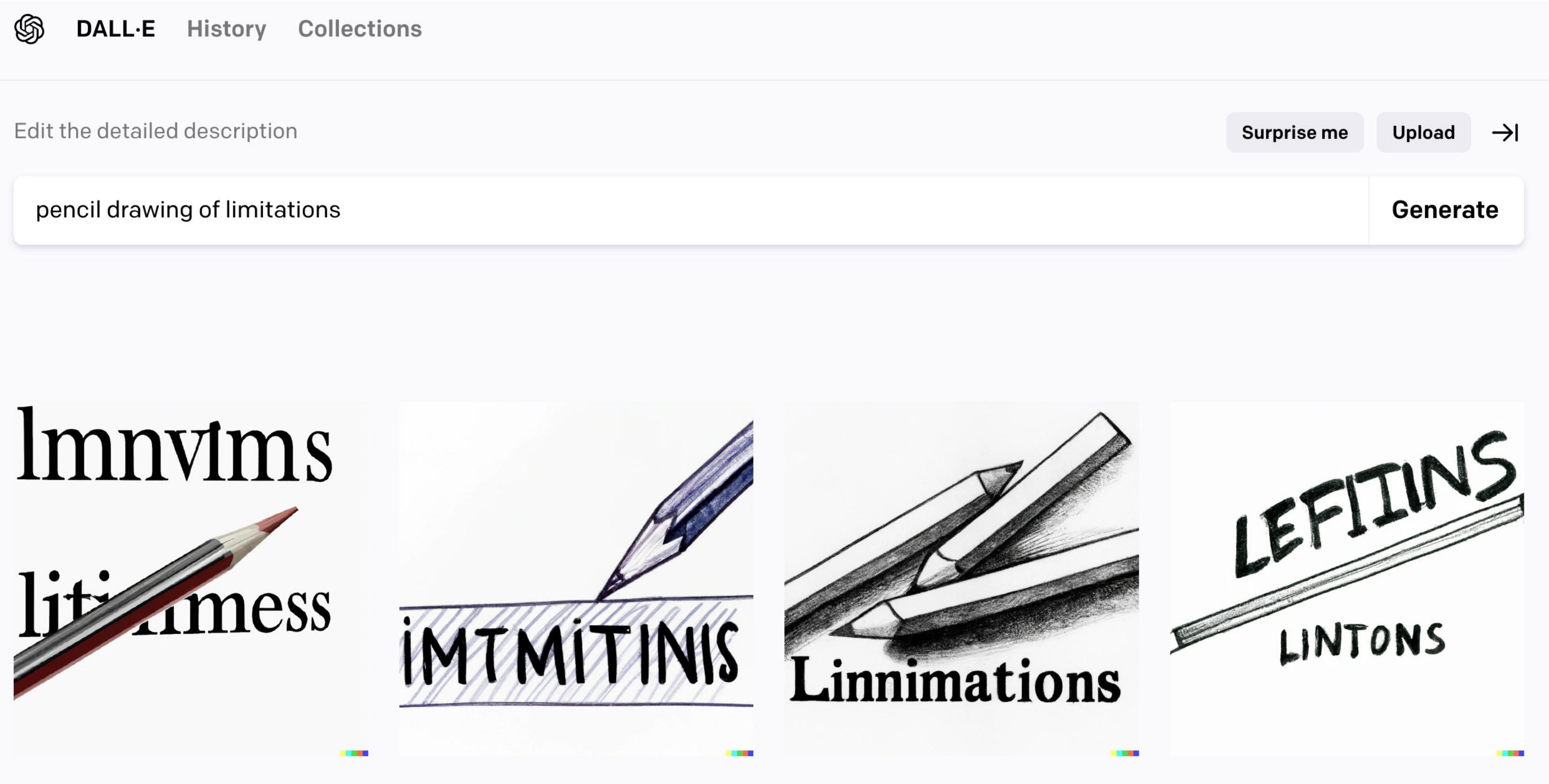
Join by Web **PollEv.com/potanin** Join by Text Send **potanin** to **22333**

Is System Testing:

Black Box **(A)**

White Box **(B)**

No idea, I was sleeping **(C)**



Edit the detailed description

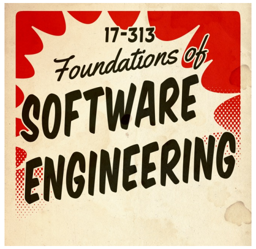
pencil drawing of limitations

Limitations of Testing

Why Should We Test?

- What does testing help us achieve?

Things we might try to test



- **Program/system functionality:**
  - Execution space (white box).
  - Input or requirements space (black box).
- The expected user experience (usability).
  - GUI testing, A/B testing
- The expected performance envelope (performance, reliability, robustness, integration).
  - Security, robustness, fuzz, and infrastructure testing.
  - Performance and reliability: soak and stress testing.
  - Integration and reliability: API/protocol testing

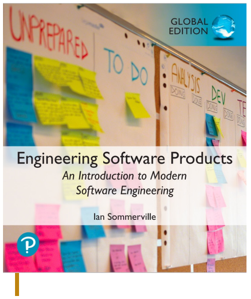
Software Errors

- Functional errors
- Performance errors
- Deadlock
- Race conditions
- Boundary errors
- Buffer overflow
- Integration errors
- Usability errors
- Robustness errors
- Load errors
- Design defects
- Versioning and configuration errors
- Hardware errors
- State management errors
- Metadata errors
- Error-handling errors
- User interface errors
- API usage errors
- …

Value of Testing

- [Low bar] Ensure that our software meets requirements, is correct, etc.
- Preventing bugs or quality degradations from being accidentally introduced in the future
- Helps uncover unexpected behaviors that can't be identified by reading source code
- Increased confidence in changes ("will I break the internet with this commit?")
- Bridges the gap between a declarative view of the system (i.e., requirements) and an imperative view (i.e., implementation) by means of redundancy.
- Tests are executable documentation; increases code maintainability
- Forces writing testable code <-> checks software design

Sommerville’s reasons for not using TDD



- ***TDD discourages radical program change***
  I found that I was reluctant to make refactoring decisions that I knew would cause many tests to fail. I tended to avoid radical program change for this reason.

- ***I focused on the tests rather than the problem I was trying to solve***
  A basic principle of TDD is that your design should be driven by the tests you have written. I found that I was unconsciously redefining the problem I was trying to solve to make it easier to write tests. This meant that I sometimes didn’t implement important checks, because it was difficult to write tests in advance of their implementation.

- ***I spent too much time thinking about implementation details rather than the programming problem***
  Sometimes when programming, it is best to step back and look at the program as a whole rather than focusing on implementation details. TDD encourages a focus on details that might cause tests to pass or fail and discourages large-scale program revisions.

- ***It is hard to write ‘bad data’ tests***
  Many problems involving dealing with messy and incomplete data. It is practically impossible to anticipate all of the data problems that might arise and write tests for these in advance. You might argue that you should simply reject bad data but this is sometimes impractical.

Limitations of Testing

- What Can We Run Automated Tests For?
  - Correctness
  - Performance
  - Reliability
  - Security?
  - Maintainability?
- What Can We NOT (Easily) Test For?
  - Can’t prove correctness or other quality attributes over all cases:
    - can only measure a sample
  - Only as good as the tests you write – will come back to Oracle Problem
  - Testing does not validate specifications / requirements

Limitations of Testing

"Testing shows the presence, not the absence of bugs.’

-Edsger W. Dijkstra

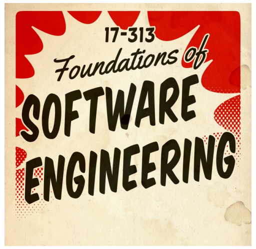
- Testing doesn’t really give any formal assurances
- Writing tests is hard, time consuming
- Knowing if your tests are good enough is not obvious
- Executing tests can be expensive, especially as software complexity and configuration space grows
  - Full test suite for a single large app can take several days to run
- Halting Problem

When Should We Test?

- At Release Time?
- After Every Code Change?
- After Writing New Code?
- Before Writing New Code?
- Before or After Merging?
- In Production?

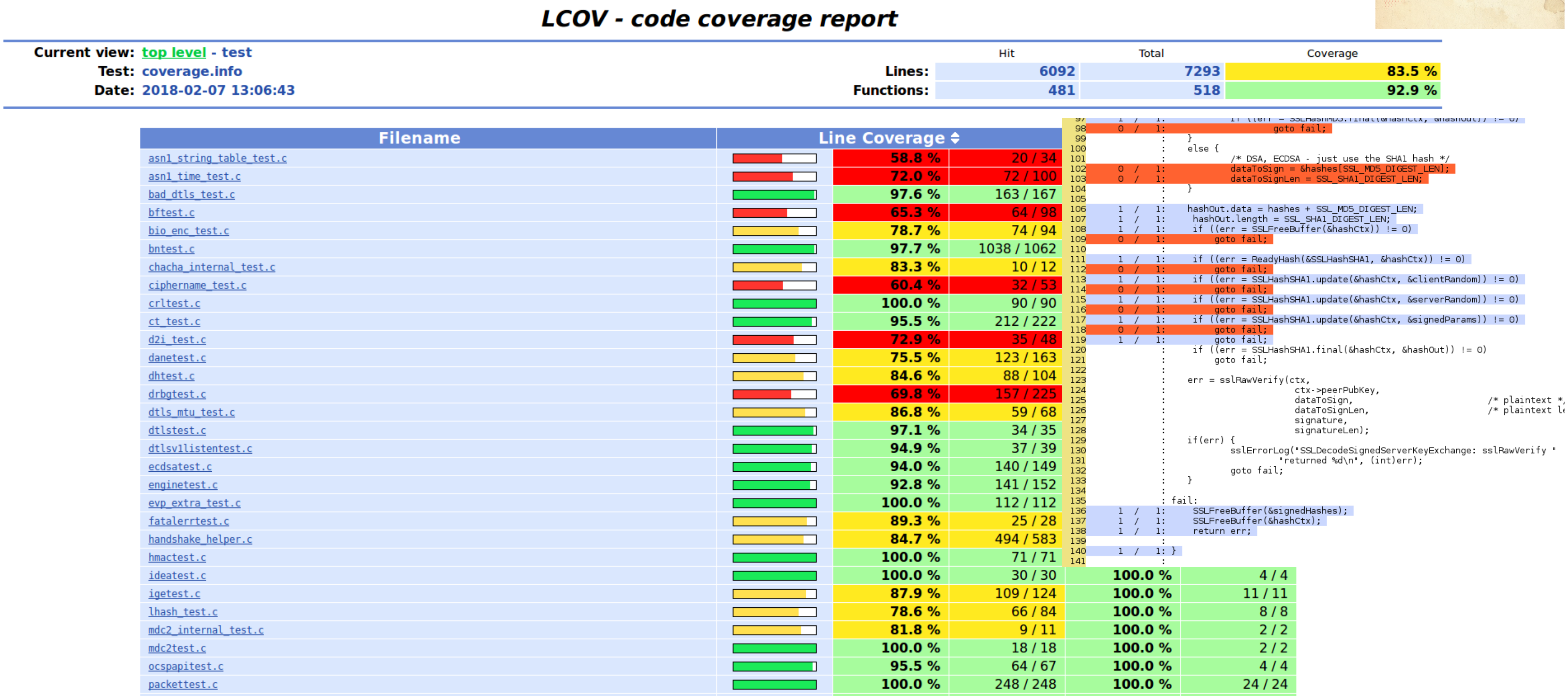
YOU SCHEDULED THE END OF THE TEST PHASE AFTER THE START OF THE PRODUCTION PHASE.
WE'RE FEELING CONFIDENT.
IT'S TOO BAD THAT BEING SMART DOESN'T COME WITH SOME SORT OF GOOD FEELING LIKE THAT.
Dilbert.com DilbertCartoonist@gmail.com
5-17-11 ©2011 Scott Adams, Inc. Dist. by Universal Uclick

Code Coverage



- Line coverage
- Statement coverage
- Branch coverage
- Instruction coverage
- Basic-block coverage
- Edge coverage
- Path coverage
- …

Code Coverage



LCOV - code coverage report

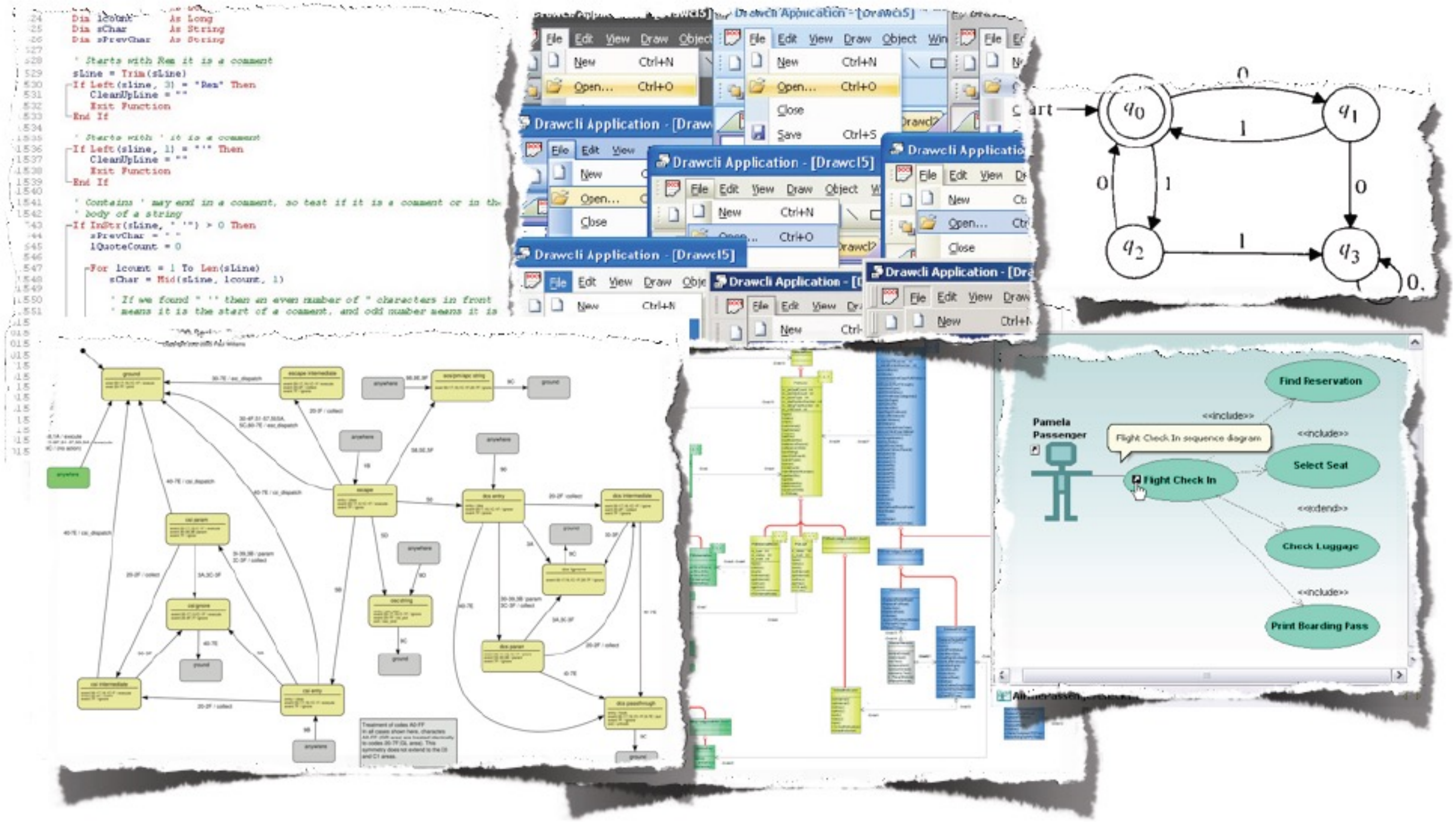
Current view: top level - test
Test: coverage.info
Date: 2018-02-07 13:06:43

	Hit	Total	Coverage
Lines:	6092	7293	83.5 %
Functions:	481	518	92.9 %

Filename	Line Coverage ⇕			
asn1_string_table_test.c	58.8 %	20 / 34		
asn1_time_test.c	72.0 %	72 / 100		
bad_dtls_test.c	97.6 %	163 / 167		
bftest.c	65.3 %	64 / 98		
bio_enc_test.c	78.7 %	74 / 94		
bntest.c	97.7 %	1038 / 1062		
chacha_internal_test.c	83.3 %	10 / 12		
ciphername_test.c	60.4 %	32 / 53		
crltest.c	100.0 %	90 / 90		
ct_test.c	95.5 %	212 / 222		
d2i_test.c	72.9 %	35 / 48		
danetest.c	75.5 %	123 / 163		
dhtest.c	84.6 %	88 / 104		
drbgtest.c	69.8 %	157 / 225		
dtls_mtu_test.c	86.8 %	59 / 68		
dtlstest.c	97.1 %	34 / 35		
dtlsv1listentest.c	94.9 %	37 / 39		
ecdsatest.c	94.0 %	140 / 149		
enginetest.c	92.8 %	141 / 152		
evp_extra_test.c	100.0 %	112 / 112		
fatalerrtest.c	89.3 %	25 / 28		
handshake_helper.c	84.7 %	494 / 583		
hmactest.c	100.0 %	71 / 71		
ideatest.c	100.0 %	30 / 30	100.0 %	4 / 4
igetest.c	87.9 %	109 / 124	100.0 %	11 / 11
lhash_test.c	78.6 %	66 / 84	100.0 %	8 / 8
mdc2_internal_test.c	81.8 %	9 / 11	100.0 %	2 / 2
mdc2test.c	100.0 %	18 / 18	100.0 %	2 / 2
ocspapitest.c	95.5 %	64 / 67	100.0 %	4 / 4
packettest.c	100.0 %	248 / 248	100.0 %	24 / 24

```
  97      1  /   1:             if ((err = SSLHashMD5.final(&hashCtx, &hashOut)) != 0)
  98      0  /   1:                     goto fail;
  99                :     }
 100                :     else {
 101                :             /* DSA, ECDSA - just use the SHA1 hash */
 102      0  /   1:             dataToSign = &hashes[SSL_MD5_DIGEST_LEN];
 103      0  /   1:             dataToSignLen = SSL_SHA1_DIGEST_LEN;
 104                :     }
 105                :
 106      1  /   1:     hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
 107      1  /   1:      hashOut.length = SSL_SHA1_DIGEST_LEN;
 108      1  /   1:      if ((err = SSLFreeBuffer(&hashCtx)) != 0)
 109      0  /   1:          goto fail;
 110                :
 111      1  /   1:      if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
 112      0  /   1:          goto fail;
 113      1  /   1:      if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
 114      0  /   1:          goto fail;
 115      1  /   1:      if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
 116      0  /   1:          goto fail;
 117      1  /   1:      if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
 118      0  /   1:          goto fail;
 119      1  /   1:          goto fail;
 120                :      if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
 121                :          goto fail;
 122                :
 123                :     err = sslRawVerify(ctx,
 124                :                        ctx->peerPubKey,
 125                :                        dataToSign,                    /* plaintext *
 126                :                        dataToSignLen,                 /* plaintext le
 127                :                        signature,
 128                :                        signatureLen);
 129                :     if(err) {
 130                :             sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
 131                :                      "returned %d\n", (int)err);
 132                :             goto fail;
 133                :     }
 134                :
 135                : fail:
 136      1  /   1:      SSLFreeBuffer(&signedHashes);
 137      1  /   1:      SSLFreeBuffer(&hashCtx);
 138      1  /   1:      return err;
 139                :
 140      1  /   1: }
 141                :
```

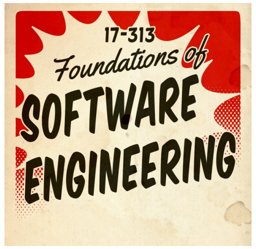
We can measure coverage on almost anything



```
Dim lcount        As Long
Dim sChar         As String
Dim sPrevChar     As String

' Starts with Rem it is a comment
sLine = Trim(sLine)
If Left(sline, 3) = "Rem" Then
    CleanUpLine = ""
    Exit Function
End If

' Starts with ' it is a comment
If Left(sline, 1) = "'" Then
    CleanUpLine = ""
    Exit Function
End If

' Contains ' may end in a comment, so test if it is a comment or in th
' body of a string
If InStr(sLine, " '") > 0 Then
    sPrevChar = " "
    lQuoteCount = 0

    For lcount = 1 To Len(sLine)
        sChar = Mid(sLine, lcount, 1)

        ' If we found " '' then an even number of " characters in front
        ' means it is the start of a comment, and odd number means it is
```

A. Zeller, Testing and Debugging Advanced course, 2010

Beware of coverage chasing



- Recall: issues with metrics and incentives
- Also: Numbers can be deceptive
  - 100% coverage != exhaustively tested
- “Coverage is not strongly correlated with suite effectiveness”
  - Based on empirical study on GitHub projects [Inozemtseva and Holmes, ICSE’14]
- Still, it’s a good low bar
  - Code that is not executed has definitely not been tested

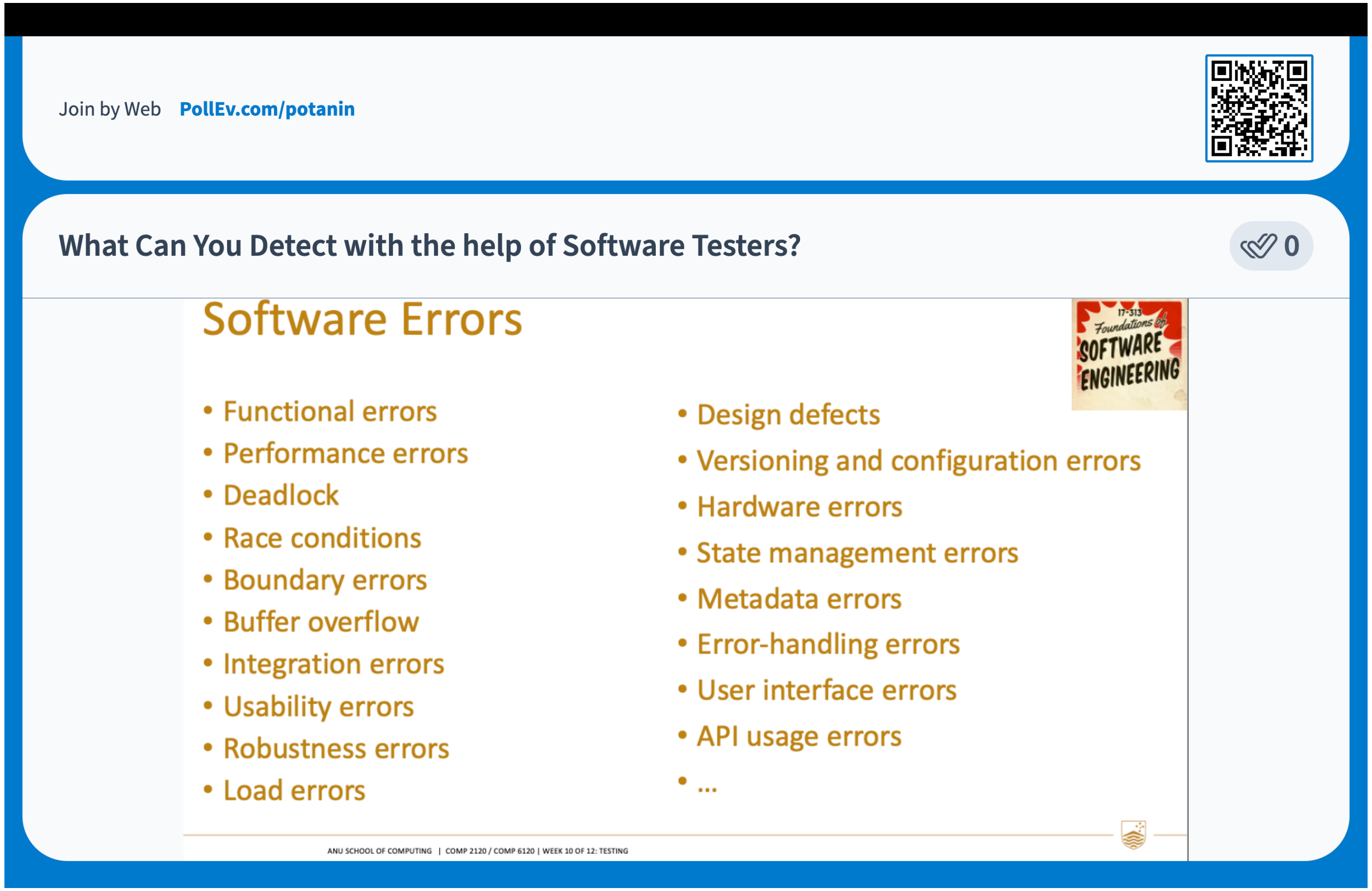
Coverage of what?

- Distinguish code being tested and code being executed
- Library code >>>> Application code
  - Can selectively measure coverage
- All application code >>> code being tested
  - Not always easy to do this within an application

Coverage != Outcome

- What's better, tests that always pass or tests that always fail?
- Tests should ideally be *falsifiable.* Boundary determines specification
- Ideally:
  - Correct implementations should pass all tests
  - Buggy code should fail at least one test
  - Intuition behind *mutation testing*
- What if tests have bugs?
  - Pass on buggy code or fail on correct code
- Even worse: flaky tests
  - Pass or fail on the same test case nondeterministically
- What's the worst type of test?

Poll Everywhere Time!



Join by Web **PollEv.com/potanin**

What Can You Detect with the help of Software Testers?

Software Errors

- Functional errors
- Performance errors
- Deadlock
- Race conditions
- Boundary errors
- Buffer overflow
- Integration errors
- Usability errors
- Robustness errors
- Load errors
- Design defects
- Versioning and configuration errors
- Hardware errors
- State management errors
- Metadata errors
- Error-handling errors
- User interface errors
- API usage errors
- ...

ANU SCHOOL OF COMPUTING | COMP 2120 / COMP 6120 | WEEK 10 OF 12: TESTING

Prompt

Draw fuzzing tester

Fuzzing

Puzzle:
Find `x` such that `p1(x)` returns `True`

```
def p1(x):
  if x * x - 10 == 15:
    return True
  return False
```

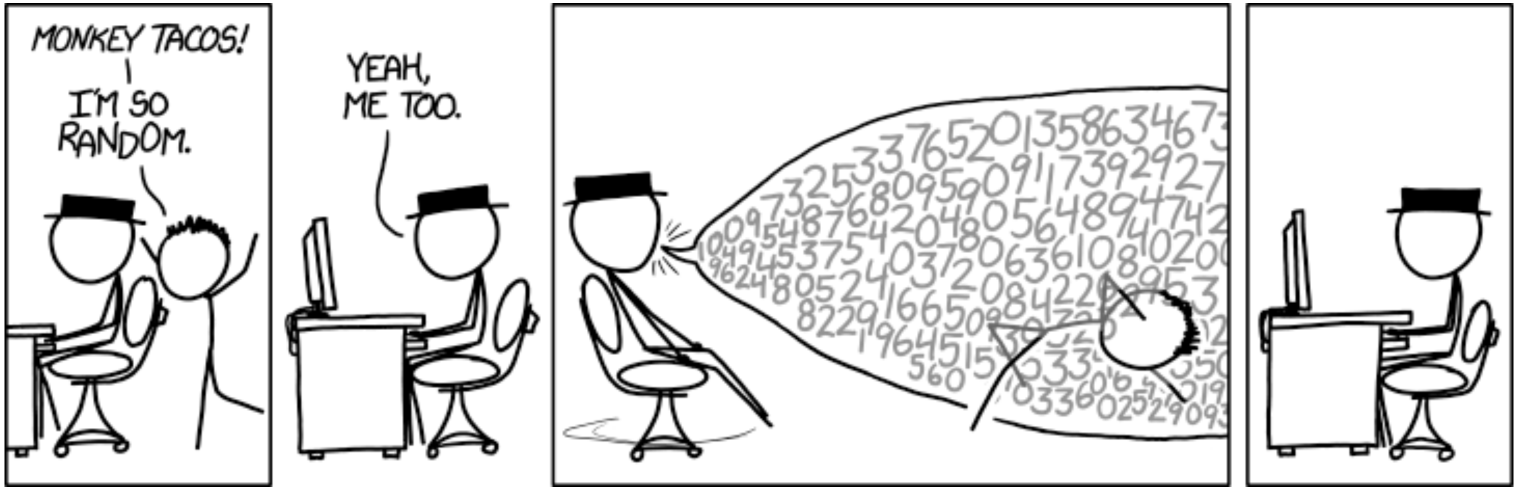
Puzzle:
Find x such that p2(x) returns True

```
def p2(x):
  if x > 0 and x < 1000:
    if ((x - 32) * 5/9 == 100):
      return True
  return False
```

Puzzle:
Find x such that p3(x) returns True

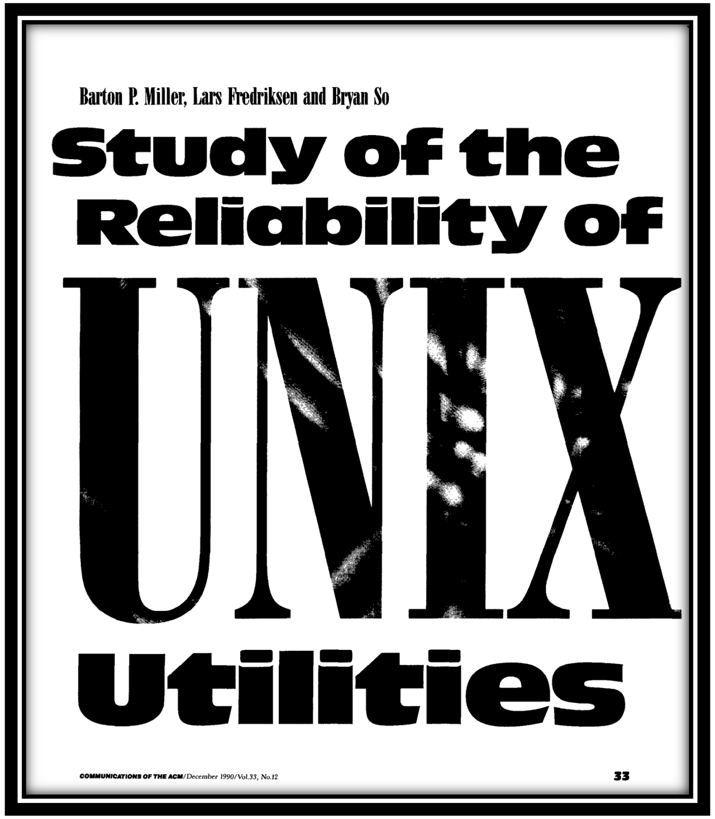
```
def p3(x):
  if x > 3 and x < 100:
    z = x - 2
    c = 0
    while z >= 2:
      if z ** (x - 1) % x == 1:
        c = c + 1
      z = z - 1
    if c == x - 3:
      return True
  return False
```

Fuzz Testing



Original: https://xkcd.com/1210 CC-BY-NC 2.5

Fuzz Testing

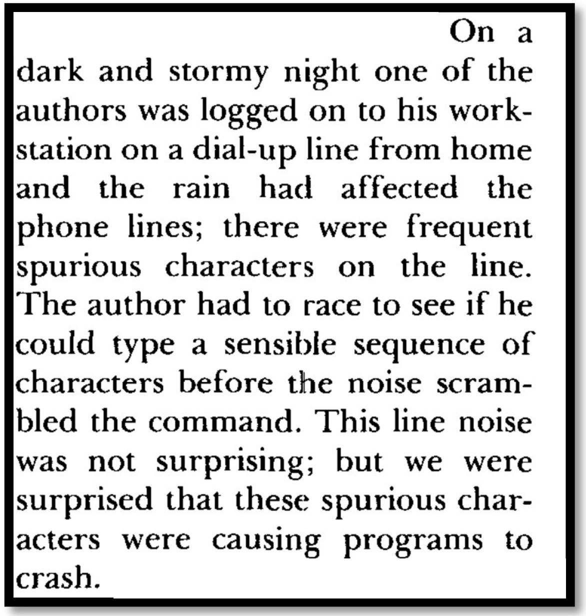


Barton P. Miller, Lars Fredriksen and Bryan So

Study of the Reliability of UNIX Utilities

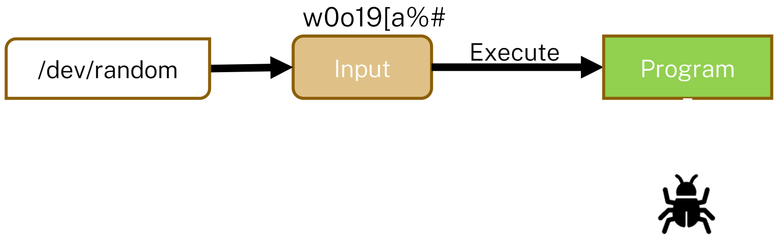
COMMUNICATIONS OF THE ACM/December 1990/Vol.33, No.12 33

Communications of the ACM (1990)



“On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. The author had to race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash.”

Fuzz Testing



1990 study found crashes in:
adb, as, bc, cb, col, diction, emacs, eqn, ftp, indent, lex, look, m4, make, nroff, plot, prolog, ptx, refer!, spell, style, tsort, uniq, vgrind, vi

Common Fuzzer-Found Bugs in C/C++

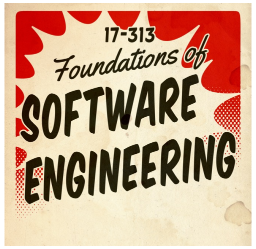
Causes: incorrect arg validation, incorrect type casting, executing untrusted code, etc.

Effects: buffer-overflows, memory leak, division-by-zero, use-after-free, assertion violation, etc. (“crash”)

Impact: security, reliability, performance, correctness

How to identify these bugs in languages like C/C++?

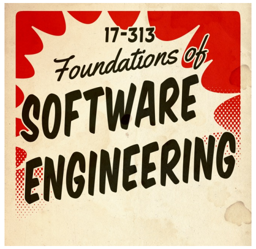
Automatic Oracles: Sanitizers



- Address Sanitizer (ASAN)
- LeakSanitizer (comes with ASAN)
- Thread Sanitizer (TSAN)
- Undefined-behavior Sanitizer (UBSAN)

https://github.com/google/sanitizers

AddressSanitizer



Compile with `clang –fsanitize=address`

```
int get_element(int* a, int i) {
   return a[i];
}
```

Is it null?

```
int get_element(int* a, int i) {
   if (a == NULL) abort();
   return a[i];
}
```

Is the access out of bounds?

```
int get_element(int* a, int i) {
   if (a == NULL) abort();
   region = get_allocation(a);
   if (in_heap(region)) {
     low, high = get_bounds(region);
     if ((a + i) < low || (a +i) > high) {
       abort();
     }
   }
   return a[i];
}
```

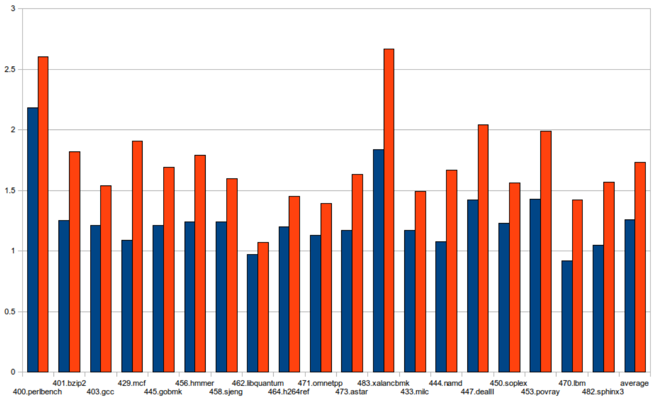
Is this a reference to a stack-allocated variable after return?

```
int get_element(int* a, int i) {
   if (a == NULL) abort();
   region = get_allocation(a);
   if (in_stack(region)) {
     if (popped(region)) abort();
     …
   }
   if (in_heap(region)) { ... }
   return a[i];
}
```

AddressSanitizer

Asan is a memory error detector for C/C++. It finds:

- Use after free (dangling pointer dereference)
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

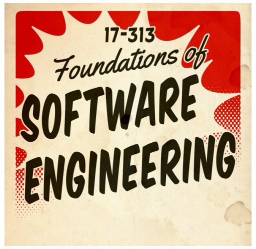


https://github.com/google/sanitizers/wiki/AddressSanitizer

Strengths and Limitations

- **Exercise**: Write down two strengths and two weaknesses of fuzzing. Bonus: Write down one or more assumptions that fuzzing depends on.

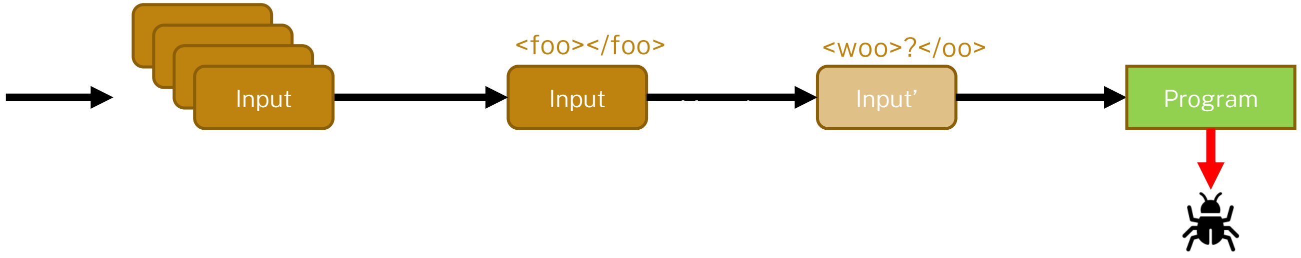
Strengths and Limitations



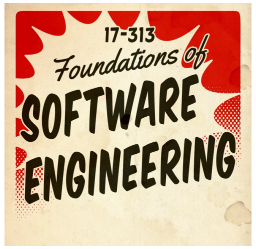
- Strengths:
  - Cheap to generate inputs
  - Easy to debug when a failure is identified

- Limitations:
  - Randomly generated inputs don’t make sense most of the time.
    - E.g. Imagine testing a browser and providing some ”input” HTML randomly: dgsad5135o gsd;gj lsdkg3125j@!T%#(W+123sd asf j
  - Unlikely to exercise interesting behavior in the web browser
  - Can take a long time to find bugs. Not sure when to stop.

Mutation-Based Fuzzing (e.g. Radams

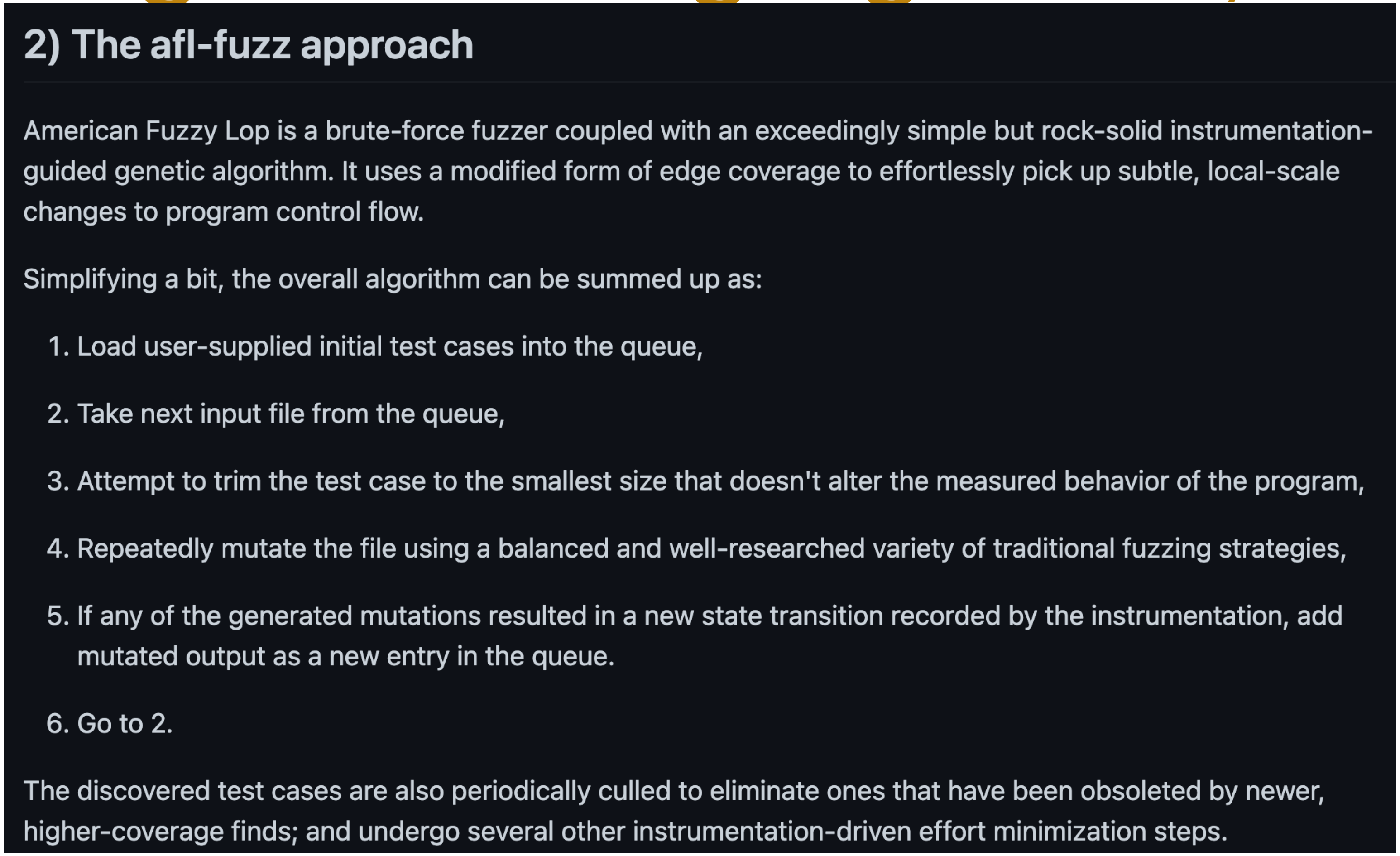


Mutation Heuristics



- Binary input
  - Bit flips, byte flips
  - Change random bytes
  - Insert random byte chunks
  - Delete random byte chunks
  - Set randomly chosen byte chunks to *interesting* values e.g. INT_MAX, INT_MIN, 0, 1, -1, …
  - Other suggestions?
- Text input
  - Insert random symbols or keywords from a dictionary
  - Other suggestions?

American Fuzzy Lop (https://github.com/google/AFL)



2) The afl-fuzz approach

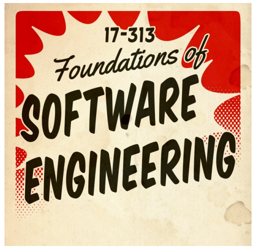
American Fuzzy Lop is a brute-force fuzzer coupled with an exceedingly simple but rock-solid instrumentation-guided genetic algorithm. It uses a modified form of edge coverage to effortlessly pick up subtle, local-scale changes to program control flow.

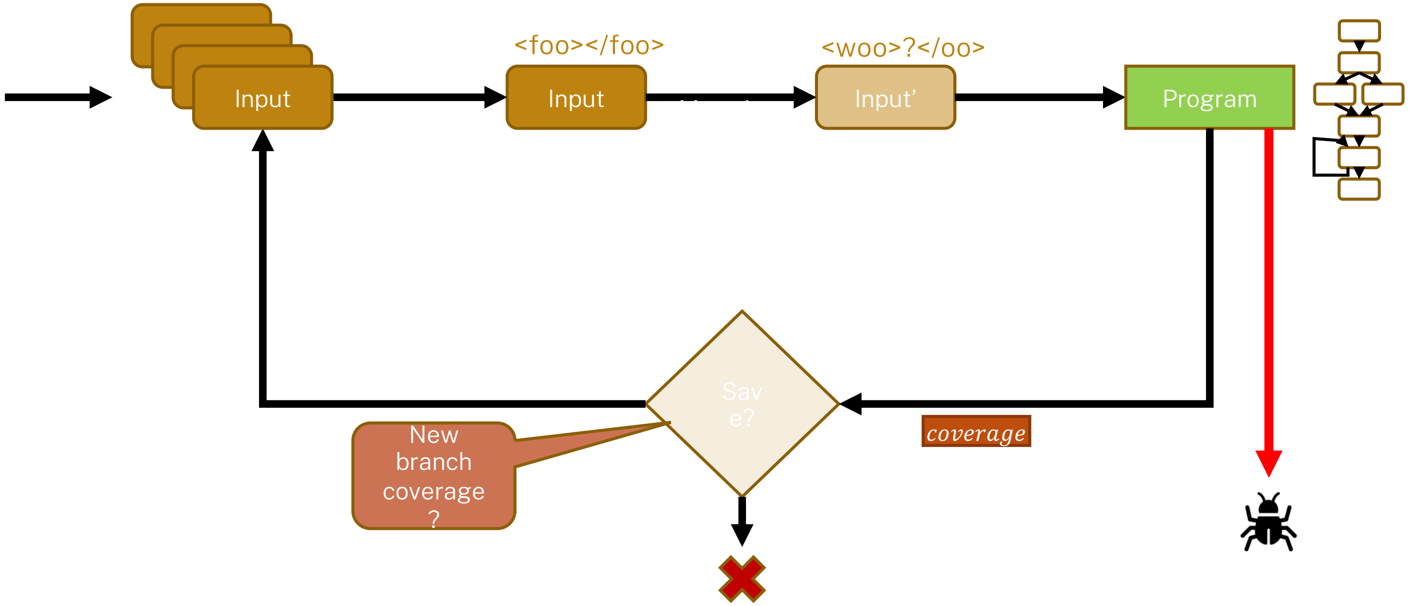
Simplifying a bit, the overall algorithm can be summed up as:

1. Load user-supplied initial test cases into the queue,
2. Take next input file from the queue,
3. Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program,
4. Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies,
5. If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue.
6. Go to 2.

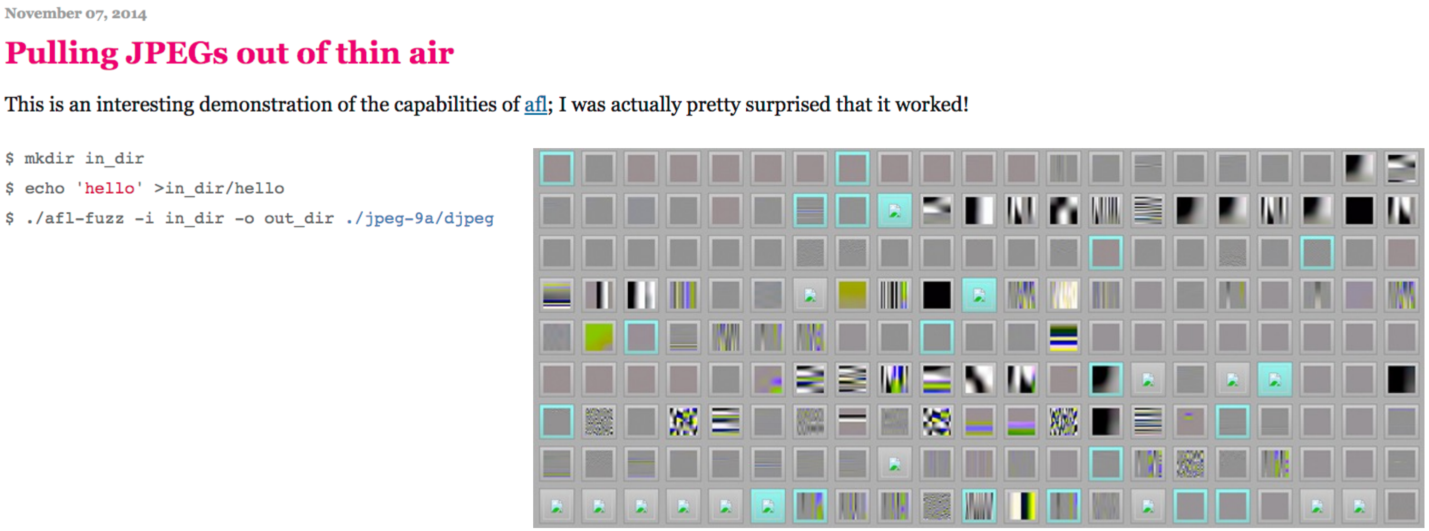
The discovered test cases are also periodically culled to eliminate ones that have been obsoleted by newer, higher-coverage finds; and undergo several other instrumentation-driven effort minimization steps.

Coverage-Guided Fuzzing (e.g. AFL)





Coverage-Guided Fuzzing with AFL



November 07, 2014

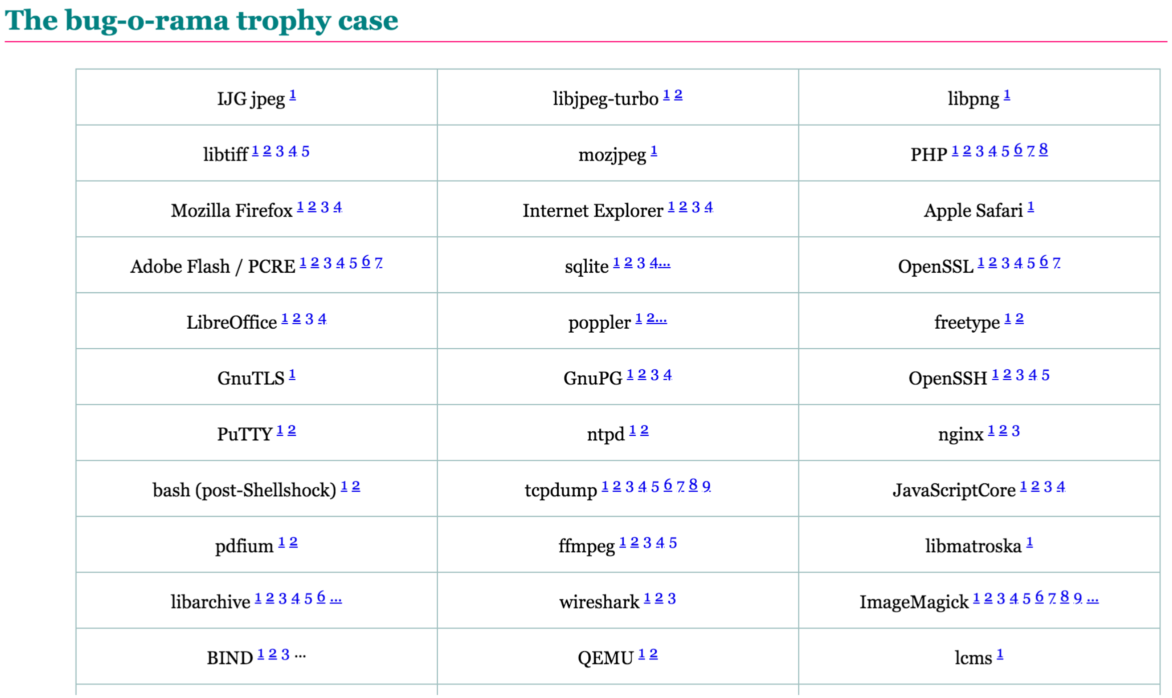
Pulling JPEGs out of thin air

This is an interesting demonstration of the capabilities of afl; I was actually pretty surprised that it worked!

```
$ mkdir in_dir
$ echo 'hello' >in_dir/hello
$ ./afl-fuzz -i in_dir -o out_dir ./jpeg-9a/djpeg
```

http://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html

Coverage-Guided Fuzzing with AFL

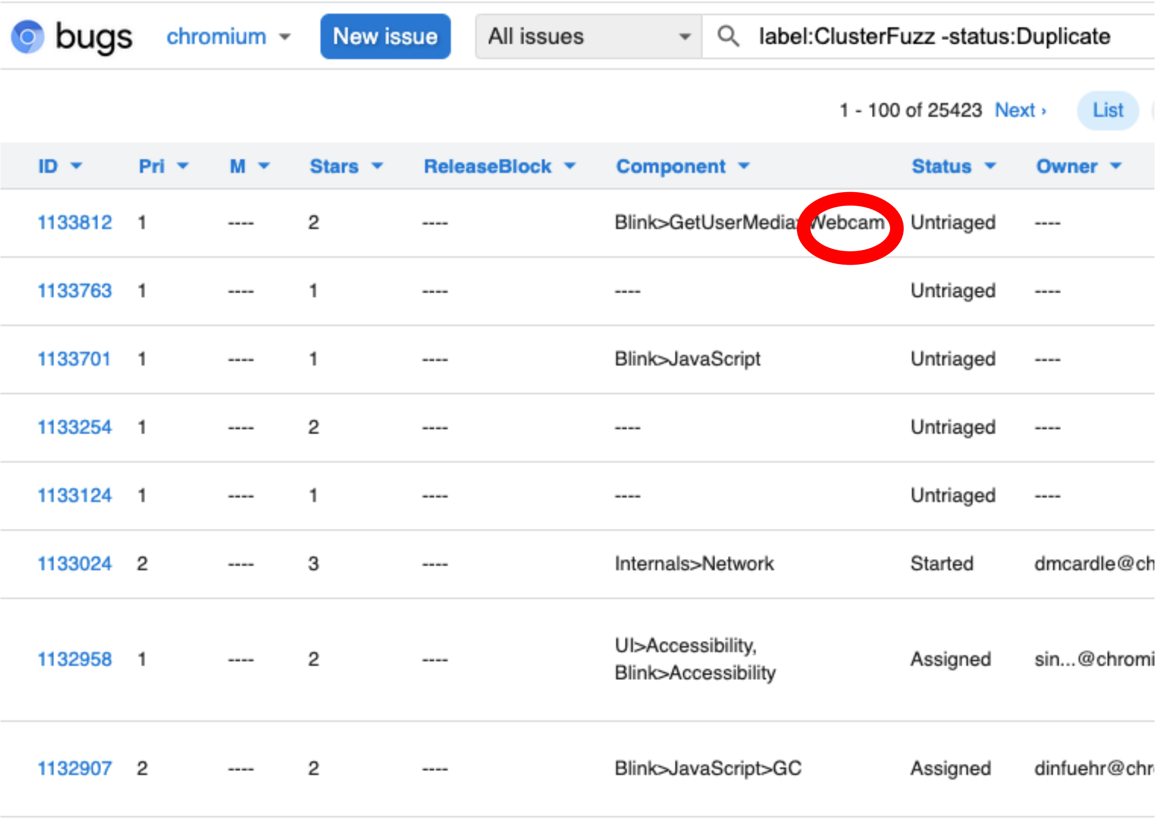


The bug-o-rama trophy case

IJG jpeg 1	libjpeg-turbo 1 2	libpng 1
libtiff 1 2 3 4 5	mozjpeg 1	PHP 1 2 3 4 5 6 7 8
Mozilla Firefox 1 2 3 4	Internet Explorer 1 2 3 4	Apple Safari 1
Adobe Flash / PCRE 1 2 3 4 5 6 7	sqlite 1 2 3 4...	OpenSSL 1 2 3 4 5 6 7
LibreOffice 1 2 3 4	poppler 1 2...	freetype 1 2
GnuTLS 1	GnuPG 1 2 3 4	OpenSSH 1 2 3 4 5
PuTTY 1 2	ntpd 1 2	nginx 1 2 3
bash (post-Shellshock) 1 2	tcpdump 1 2 3 4 5 6 7 8 9	JavaScriptCore 1 2 3 4
pdfium 1 2	ffmpeg 1 2 3 4 5	libmatroska 1
libarchive 1 2 3 4 5 6 ...	wireshark 1 2 3	ImageMagick 1 2 3 4 5 6 7 8 9 ...
BIND 1 2 3 ...	QEMU 1 2	lcms 1

http://lcamtuf.coredump.cx/afl/

ClusterFuzz @ Chromium

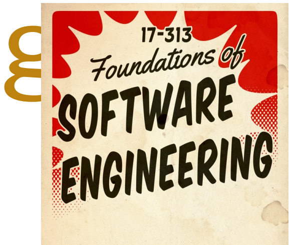


label:ClusterFuzz -status:Duplicate

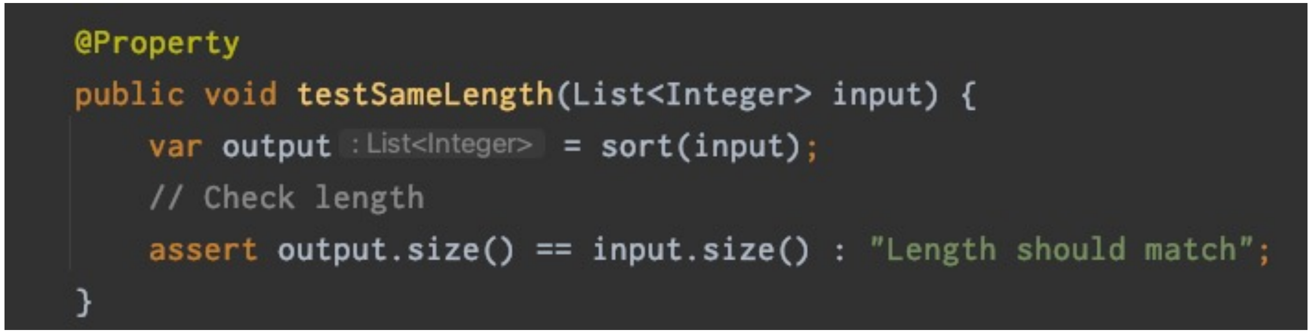
1 - 100 of 25423 Next ›

ID	Pri	M	Stars	ReleaseBlock	Component	Status	Owner
1133812	1	----	2	----	Blink>GetUserMedia>Webcam	Untriaged	----
1133763	1	----	1	----	----	Untriaged	----
1133701	1	----	1	----	Blink>JavaScript	Untriaged	----
1133254	1	----	2	----	----	Untriaged	----
1133124	1	----	1	----	----	Untriaged	----
1133024	2	----	3	----	Internals>Network	Started	dmcardle@ch
1132958	1	----	2	----	UI>Accessibility, Blink>Accessibility	Assigned	sin...@chromi
1132907	2	----	2	----	Blink>JavaScript>GC	Assigned	dinfuehr@chr

Can fuzzing be applied to unit testing



- Where “inputs” are not just strings or binary files?
- Yes! Possible to randomly generate strongly typed values, data structures, API calls, etc.
- Recall: Property-Based Testing



```
@Property
public void testSameLength(List<Integer> input) {
    var output : List<Integer> = sort(input);
    // Check length
    assert output.size() == input.size() : "Length should match";
}
```

Generators

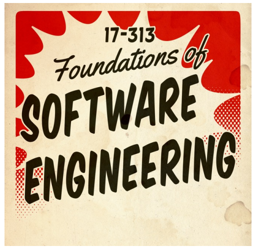
Random List<Integer>

Exercise: Write a generator for
Creating random HashMap<String, Integer>

```
List list = new ArrayList();
while (randomBoolean()) {    // randomly stop/go
  list.append(randomInt());  // random element
}
return list;
```

```
List list = new ArrayList();
int len = randomInt();        // pick a random length
for (int i = 0 to len) {
   list.append(randomInt());  // random element
}
return list;
```

Mutators



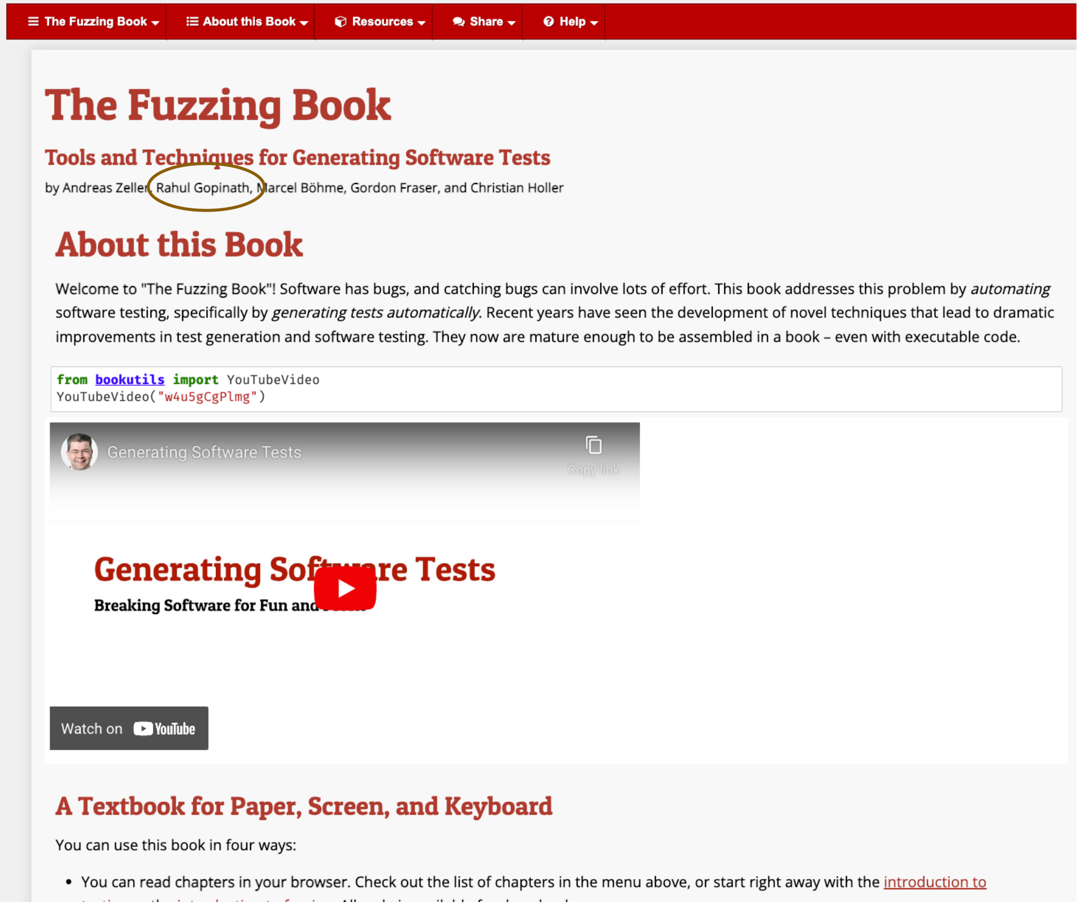
Mutator for **list**: List<Integer>

```
int k = randomInt(0, len(list));
int action = randomChoice(ADD, DELETE, UPDATE);
switch (action) {
  case UPDATE: list.set(k, randomInt()); // update element at k
  case ADD: list.addAt(k, randomInt());  // add random element at k
  case DELETE: list.removeAt(k);         // delete k-th element
}
```

Exercise: Write a mutator
HashMap<String, Integer>

The Fuzzing Book

https://www.fuzzingbook.org



The Fuzzing Book

Tools and Techniques for Generating Software Tests

by Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler

About this Book

Welcome to "The Fuzzing Book"! Software has bugs, and catching bugs can involve lots of effort. This book addresses this problem by *automating* software testing, specifically by *generating tests automatically*. Recent years have seen the development of novel techniques that lead to dramatic improvements in test generation and software testing. They now are mature enough to be assembled in a book – even with executable code.

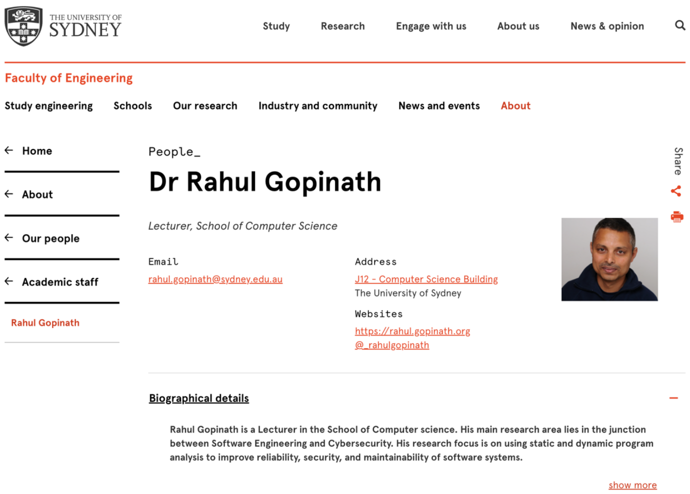
```
from bookutils import YouTubeVideo
YouTubeVideo("w4u5gCgPlmg")
```

Generating Software Tests
Copy link
Generating Software Tests
Breaking Software for Fun and
Watch on YouTube

A Textbook for Paper, Screen, and Keyboard

You can use this book in four ways:

- You can read chapters in your browser. Check out the list of chapters in the menu above, or start right away with the introduction to



Faculty of Engineering

Study engineering | Schools | Our research | Industry and community | News and events | About

← Home

← About

← Our people

← Academic staff

Rahul Gopinath

People_

Dr Rahul Gopinath

Lecturer, School of Computer Science

Email
rahul.gopinath@sydney.edu.au

Address
J12 - Computer Science Building
The University of Sydney

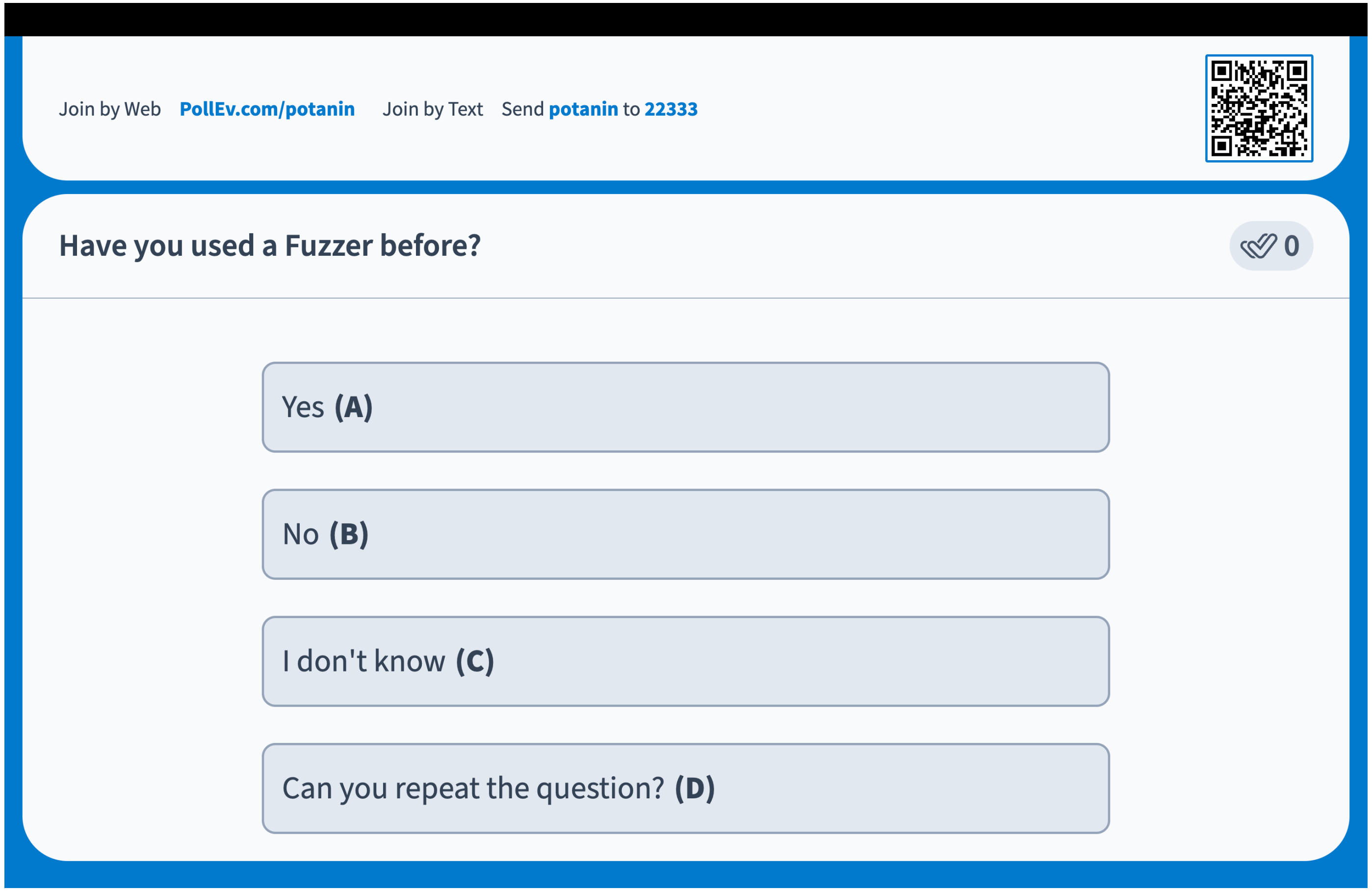
Websites
https://rahul.gopinath.org
@_rahulgopinath

Biographical details

Rahul Gopinath is a Lecturer in the School of Computer science. His main research area lies in the junction between Software Engineering and Cybersecurity. His research focus is on using static and dynamic program analysis to improve reliability, security, and maintainability of software systems.

show more

Poll Everywhere Time!



Join by Web PollEv.com/potanin Join by Text Send potanin to 22333

Have you used a Fuzzer before?

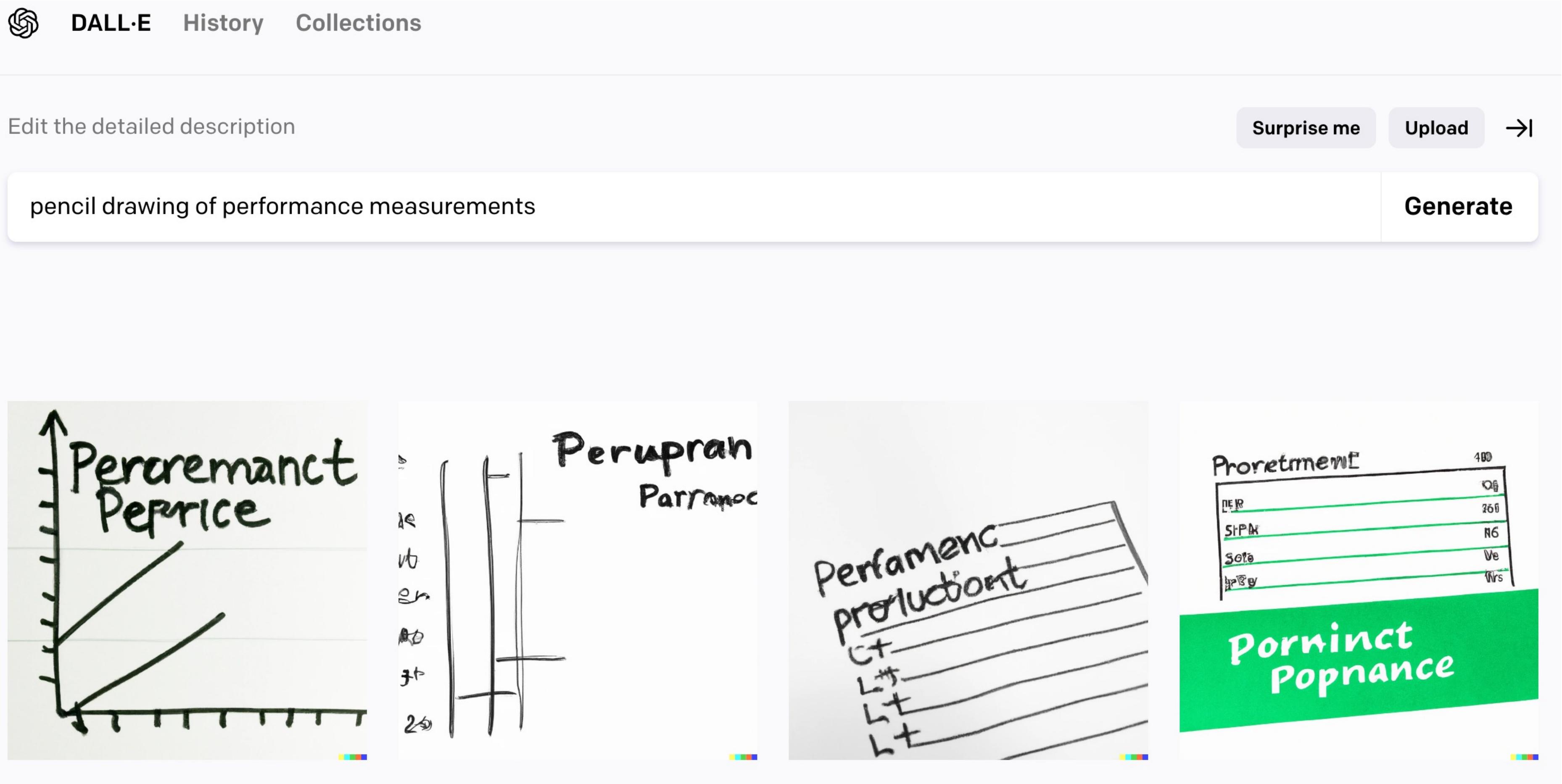
✓ 0

Yes **(A)**

No **(B)**

I don't know **(C)**

Can you repeat the question? **(D)**



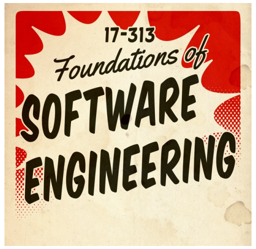
pencil drawing of performance measurements

Performance Testing

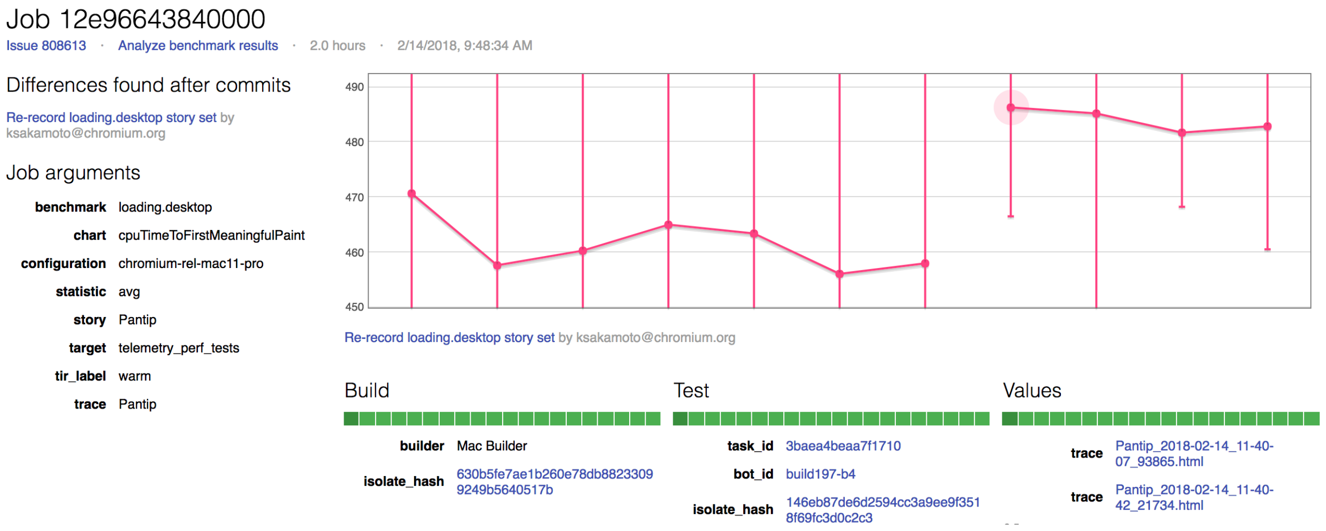
Performance Testing

- Goal: Identify *performance bugs*. What are these?
  - Unexpected bad performance on some subset of inputs
  - Performance degradation over time
  - Difference in performance across versions or platforms
- Not as easy as functional testing. What’s the oracle?
  - Fast = good, slow = bad // but what’s the threshold?
  - How to get reliable measurements?
  - How to debug where the issue lies?

Performance Regression Testing



- Measure execution time of critical components
- Log execution times and compare over time



Job 12e96643840000

Issue 808613 · Analyze benchmark results · 2.0 hours · 2/14/2018, 9:48:34 AM

Differences found after commits

Re-record loading.desktop story set by ksakamoto@chromium.org

Job arguments

benchmark	loading.desktop
chart	cpuTimeToFirstMeaningfulPaint
configuration	chromium-rel-mac11-pro
statistic	avg
story	Pantip
target	telemetry_perf_tests
tir_label	warm
trace	Pantip

Re-record loading.desktop story set by ksakamoto@chromium.org

Build

builder	Mac Builder
isolate_hash	630b5fe7ae1b260e78db8823309 9249b5640517b

Test

task_id	3baea4beaa7f1710
bot_id	build197-b4
isolate_hash	146eb87de6d2594cc3a9ee9f351 8f69fc3d0c2c3

Values

trace	Pantip_2018-02-14_11-40-07_93865.html
trace	Pantip_2018-02-14_11-40-42_21734.html

Source: https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/speed/addressing_performance_regressions.md

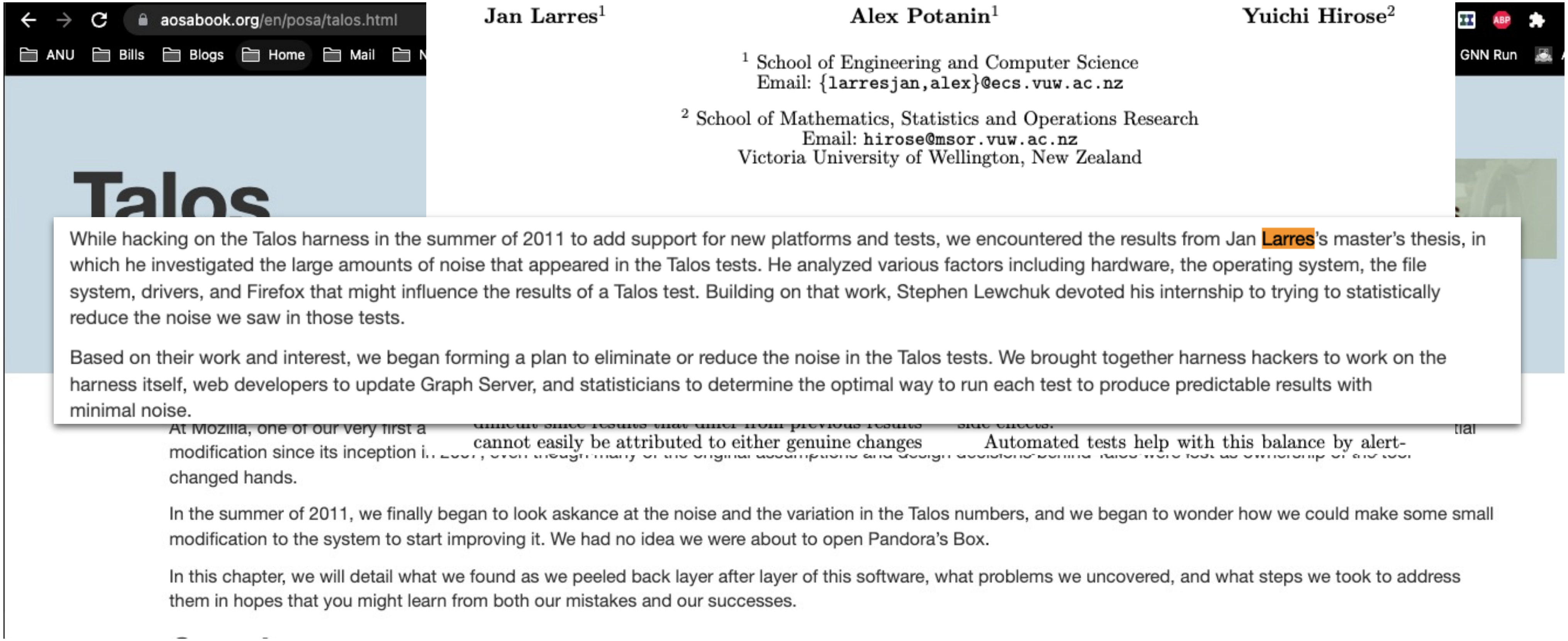
Firefox

A Study of Performance Variations in the Mozilla Firefox Web Browser

Jan Larres[1] Alex Potanin[1] Yuichi Hirose[2]

[1] School of Engineering and Computer Science
Email: {larresjan,alex}@ecs.vuw.ac.nz

[2] School of Mathematics, Statistics and Operations Research
Email: hirose@msor.vuw.ac.nz
Victoria University of Wellington, New Zealand



Talos

While hacking on the Talos harness in the summer of 2011 to add support for new platforms and tests, we encountered the results from Jan Larres's master's thesis, in which he investigated the large amounts of noise that appeared in the Talos tests. He analyzed various factors including hardware, the operating system, the file system, drivers, and Firefox that might influence the results of a Talos test. Building on that work, Stephen Lewchuk devoted his internship to trying to statistically reduce the noise we saw in those tests.

Based on their work and interest, we began forming a plan to eliminate or reduce the noise in the Talos tests. We brought together harness hackers to work on the harness itself, web developers to update Graph Server, and statisticians to determine the optimal way to run each test to produce predictable results with minimal noise.

difficult since results that differ from previous results cannot easily be attributed to either genuine changes side effects. Automated tests help with this balance by alert-

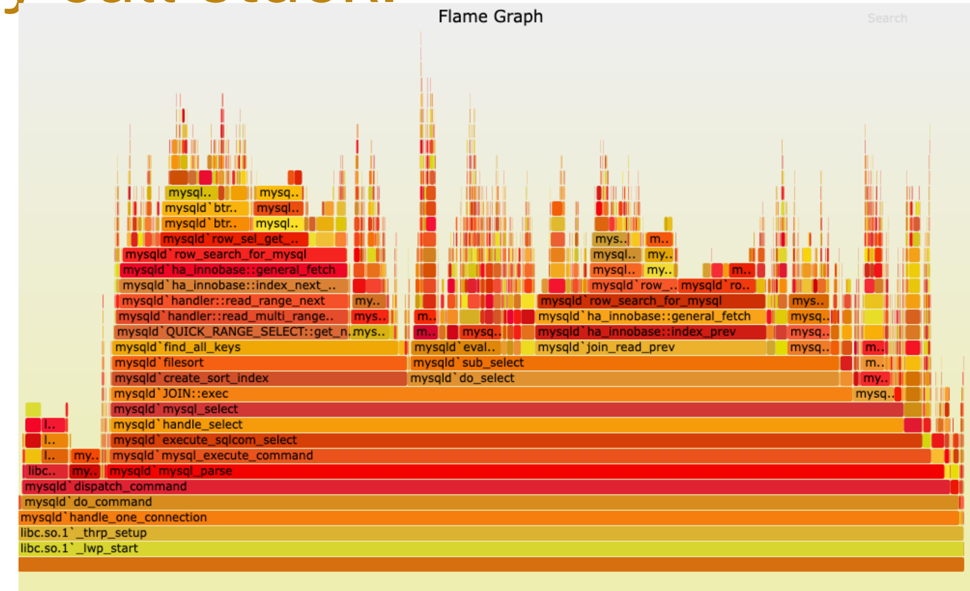
At Mozilla, one of our very first a modification since its inception in 2007, even though many of the original assumptions and design decisions behind Talos were lost as ownership of the tool changed hands.

In the summer of 2011, we finally began to look askance at the noise and the variation in the Talos numbers, and we began to wonder how we could make some small modification to the system to start improving it. We had no idea we were about to open Pandora's Box.

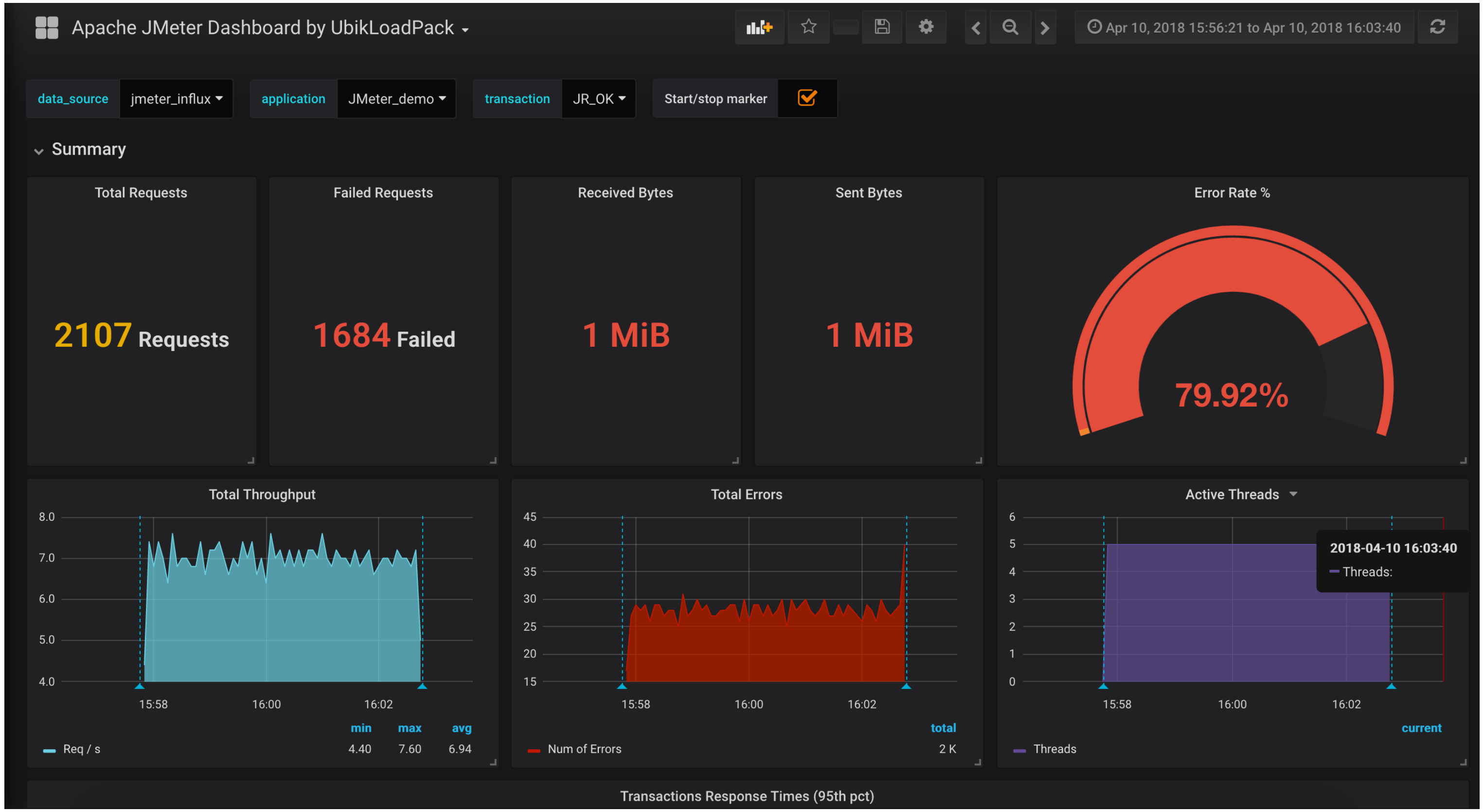
In this chapter, we will detail what we found as we peeled back layer after layer of this software, what problems we uncovered, and what steps we took to address them in hopes that you might learn from both our mistakes and our successes.

Profiling

- Finding bottlenecks in execution time and memory
- Flame graphs are a popular visualization of resource consumption by call stack.



Domain-Specific Perf Testing (e.g. JMeter)



Apache JMeter Dashboard by UbikLoadPack

Apr 10, 2018 15:56:21 to Apr 10, 2018 16:03:40

data_source jmeter_influx | application JMeter_demo | transaction JR_OK | Start/stop marker ☑

Summary

Total Requests: 2107 Requests

Failed Requests: 1684 Failed

Received Bytes: 1 MiB

Sent Bytes: 1 MiB

Error Rate %: 79.92%

Total Throughput — Req / s: min 4.40, max 7.60, avg 6.94

Total Errors — Num of Errors: total 2 K

Active Threads — Threads: current

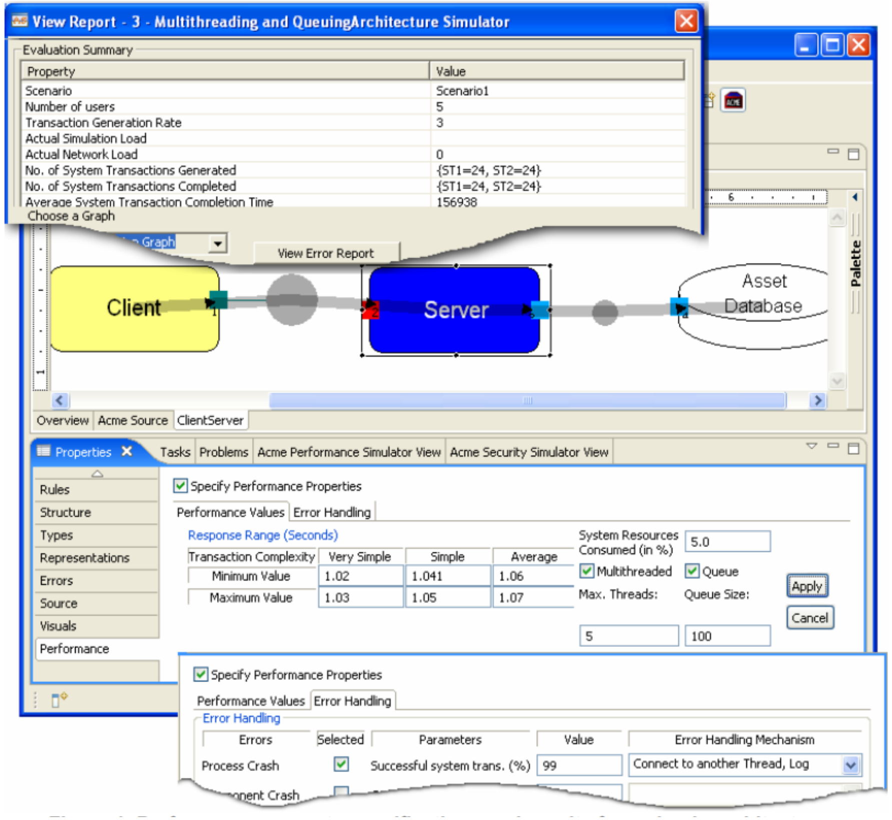
2018-04-10 16:03:40
Threads:

Transactions Response Times (95th pct)

http://jmeter.apache.org

Performance-driven Design

- Modeling and simulation
  - e.g. queuing theory
- Specify load distributions and derive or test configuratio



View Report - 3 - Multithreading and QueuingArchitecture Simulator

Evaluation Summary

Property	Value
Scenario	Scenario1
Number of users	5
Transaction Generation Rate	3
Actual Simulation Load	
Actual Network Load	0
No. of System Transactions Generated	{ST1=24, ST2=24}
No. of System Transactions Completed	{ST1=24, ST2=24}
Average System Transaction Completion Time	156938

Choose a Graph

☑ Specify Performance Properties

Performance Values | Error Handling

Response Range (Seconds)

Transaction Complexity	Very Simple	Simple	Average
Minimum Value	1.02	1.041	1.06
Maximum Value	1.03	1.05	1.07

System Resources Consumed (in %): 5.0

☑ Multithreaded ☑ Queue

Max. Threads: 5

Queue Size: 100

☑ Specify Performance Properties

Performance Values | Error Handling

Error Handling

Errors	Selected	Parameters	Value	Error Handling Mechanism
Process Crash	☑	Successful system trans. (%)	99	Connect to another Thread, Log
…nent Crash	□			

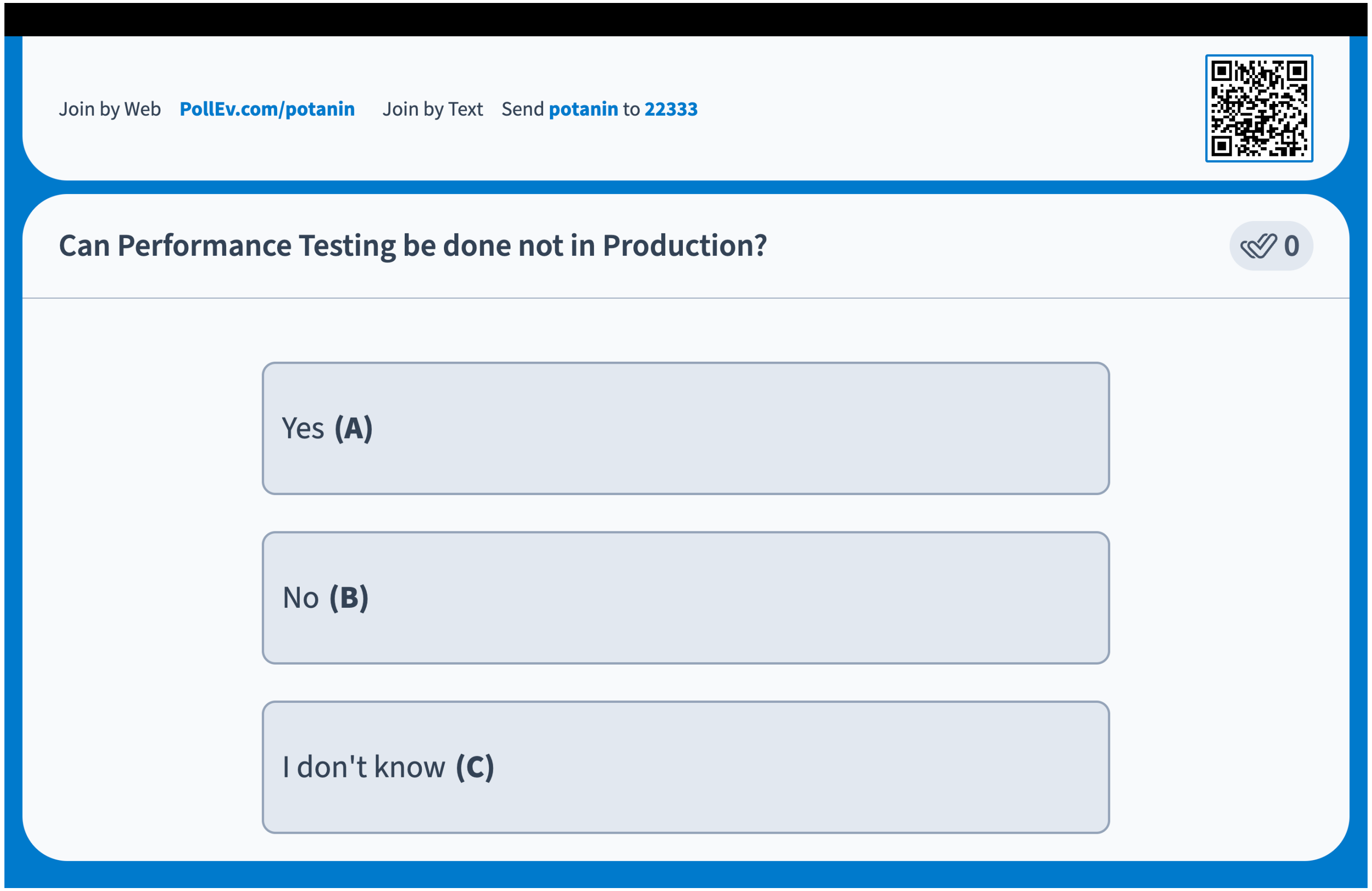
Stress testing

- Robustness testing technique: test beyond the limits of normal operation.
- Can apply at any level of system granularity.
- Stress tests commonly put a greater emphasis on robustness, availability, and error handling under a heavy load, than on what would be considered “correct” behavior under normal circumstances.

Soak testing

- **Problem:** A system may behave exactly as expected under artificially limited execution conditions.
  - E.g., Memory leaks may take longer to lead to failure (also motivates static/dynamic analysis, but we'll talk about that later).
- **Soak testing:** testing a system with a significant load over a significant period of time (*positive*).
- Used to check reaction of a subject under test under a possible simulated environment for a given duration and for a given threshold.

Poll Everywhere Time!



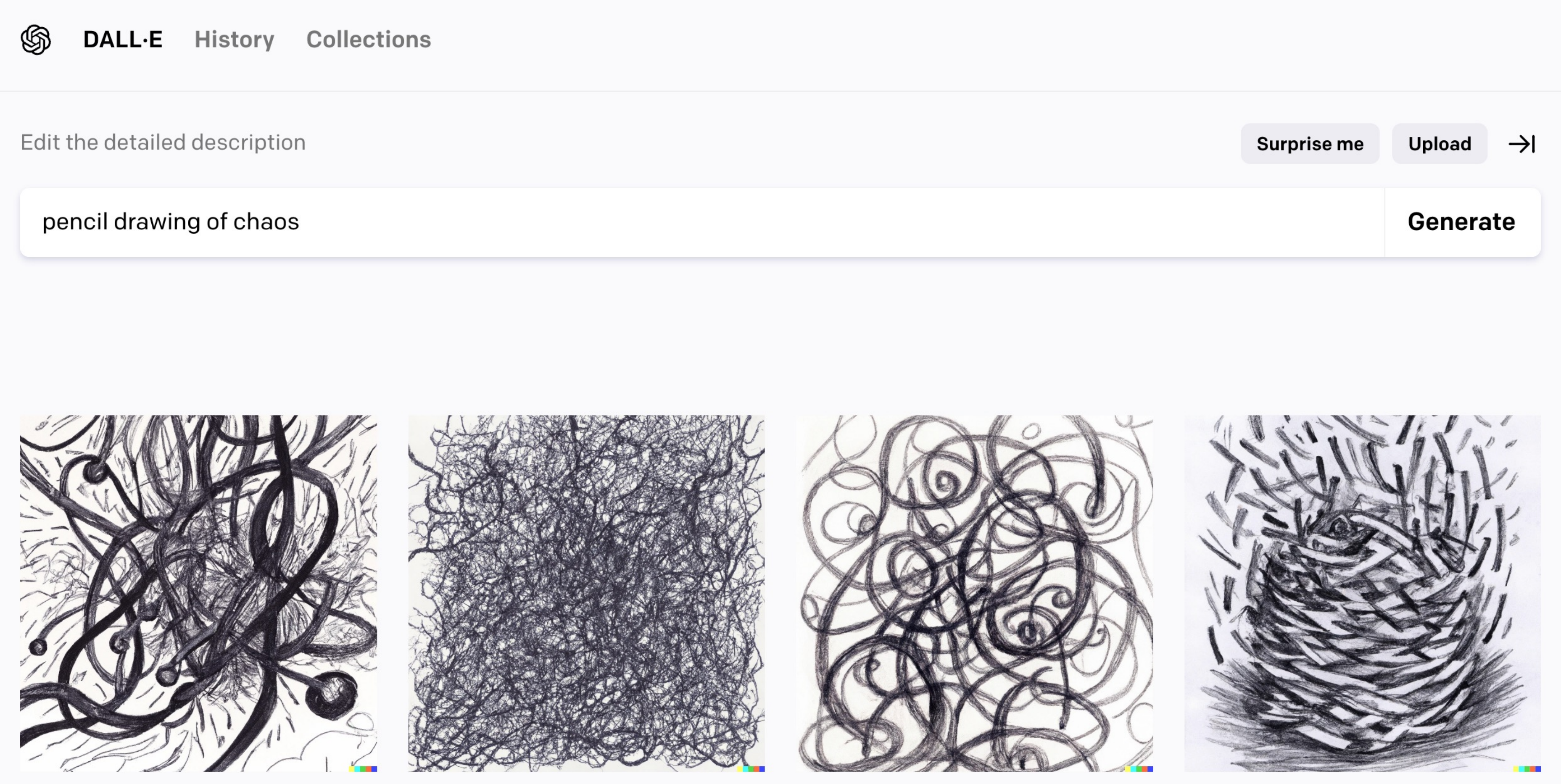
Join by Web **PollEv.com/potanin** Join by Text Send **potanin** to **22333**

Can Performance Testing be done not in Production?

Yes **(A)**

No **(B)**

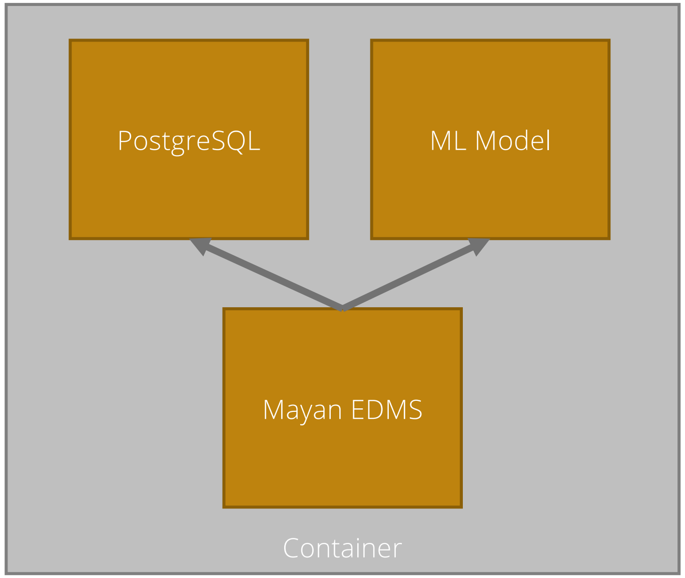
I don't know **(C)**



pencil drawing of chaos

Chaos Engineering

Monolithic Application



What kind of failures can happen here?

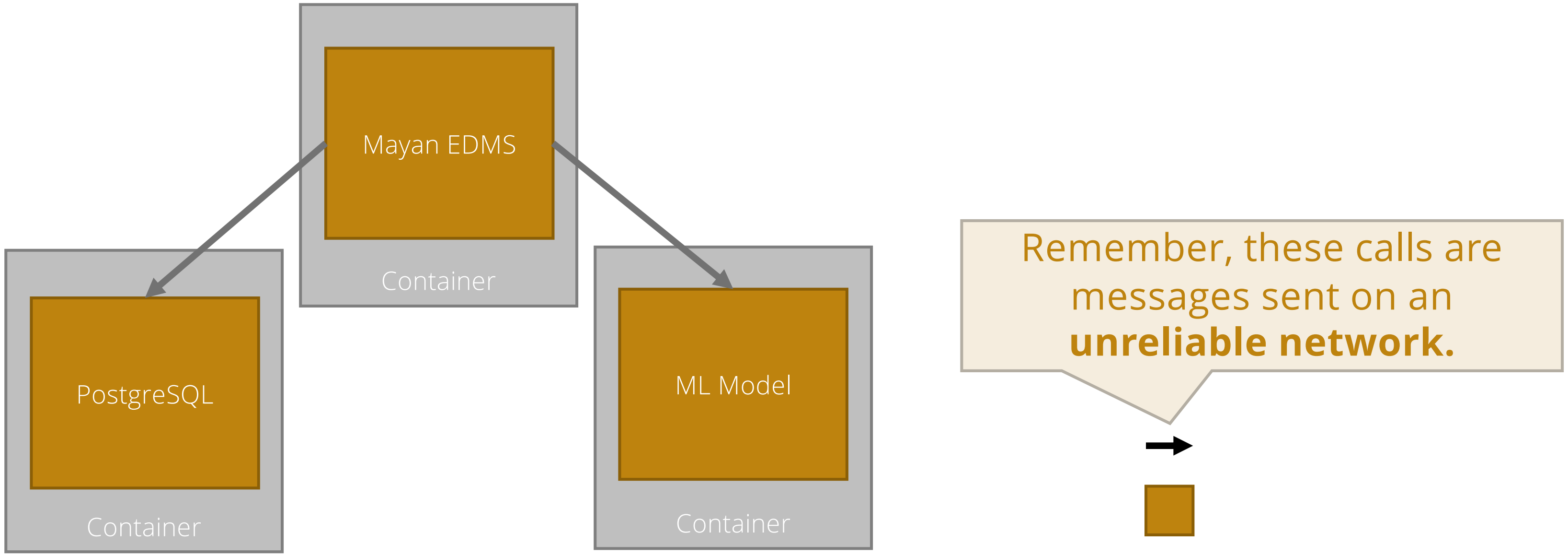
How likely is that error to happen?

How do I fix it?

→

■ Microservice

Microservice Application



Remember, these calls are messages sent on an **unreliable network.**

Failures in Microservice Architecture

1. Network may **be partitioned**

2. Server instance **may be down**

3. Communication between services may **be dela**

4. Server **could be overloaded** and responses del

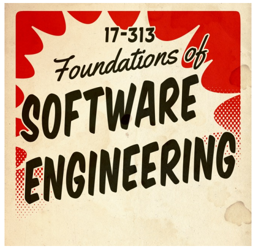
5. Server **could run out of** memory or CPU

All of these issues
can be indistinguishable
from one another!

Making the calls across the network to multiple machines makes the probability that the system is operating under failure **much higher.**

These are the problems of
latency and **partial failure.**

Where Do We Start?

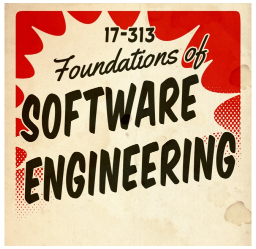


How do we even **begin to test these scenarios?**

Is there any **software** that can be used to test these types of failures?

Let’s look at a **few ways** companies do this.

Game Days



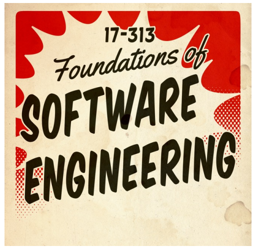
Purposely **injecting failures** into critical systems in order to:

- Identify **flaws** and “latent defects”
- Identify **subtle dependencies** (which may or may not lead to a flaw/defect)
- Prepare a **response** for a disastrous event

Comes from “resilience engineering” typical in high-risk industries

Practiced by Amazon, Google, Microsoft, Etsy, Facebook, Flickr, etc.

Game Days



Our applications are built on and with **“unreliable”** components

Failure is inevitable (fraction of percent; at Google scale, ~multiple times)

Goals:

- **Preemptively trigger** the failure, observe, and fix the error
- Script testing of **previous failures** and ensure system remains resilient
- Build the necessary relationships between teams **before** disaster strikes

Example: Amazon GameDay

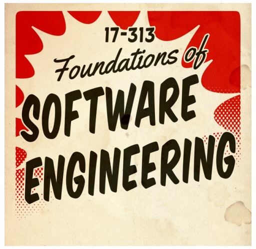
Full data center destruction (Amazon EC2 region)

- No advanced notice of **which** data center will be taken offline
- No notice will be taken offline
- Only adva a GameDay **will be happening**
- **Real failures in the production environment**

Not all failures can be actually performed and must be **simulated!**

Discovered **latent defect** where the monitoring infrastructure responsible for detecting errors and paging employees **was located in the zone of the failure!**

Cornerstones of Resilence



1. Anticipation: know what to expect
2. Monitoring: know what to look for
3. Response: know what to do
4. Learning: know what just happened (e.g, postmortems)

Some Example Google Issues

Terminate network in Sao Paulo for testing:

- Hidden dependency takes down links in Mexico which would have remained undiscovered without testing

Turn off data center to find that machines won’t come back:

- Ran out of DHCP leases (for IP address allocation) when a large number of machines come back online unexpectedly.

Netflix: Cloud Computing

Significant deployment in Amazon Web Services in order to remain
elastic in times of high and low load (first public, 100% w/o content delivery.)

Pushes code into production and modifies runtime configuration
hundreds of times a day

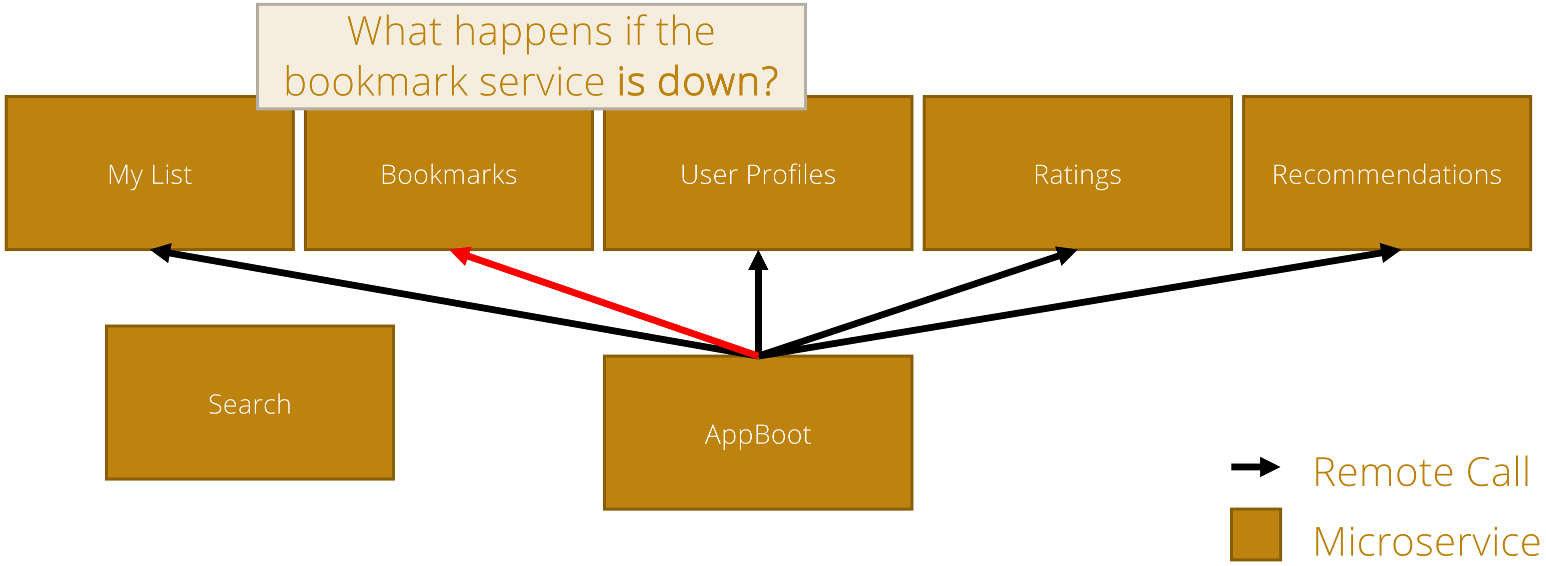
Key metric: **availability**

amazon
webservices
CHAOS
MONKEY

Chaos monkey/Simian army

- A Netflix infrastructure testing system.
- “Malicious” programs randomly trample on components, network, datacenters, AWS instances…
  - Chaos monkey was the first – disables production instances at random.
  - Other monkeys include Latency Monkey, Doctor Monkey, Conformity Monkey, etc… Fuzz testing at the infrastructure level.
  - Force failure of components to make sure that the system architecture is resilient to unplanned/random outages.
- Netflix has open-sourced their chaos monkey code.

Netflix UI: AppBoot



What happens if the bookmark service **is down?**

Graceful Degradation: Anticipating Failure

Allow the system to degrade in a way it’s still usable

Fallbacks:

- Cache miss due to failure of cache;
- Go to the bookmarks service and use value at possible latency penalty

Personalized content, use a reasonable default instead:

- What happens if **recommendations** are unavailable?
- What happens if **bookmarks are unavailable?**

Principles of Chaos Engineering

1. Build a **hypothesis** around steady state behavior

2. Vary **real-world events**
   experimental events, crashes, etc.

3. Run **experiments** in production
   control group vs. experimental group
   draw conclusions, invalidate hypothesis

4. **Automate experiments** to run continuously

Does everything seem to be **working properly?**

Are users **complaining?**

Steady State Behavior

Back to **quality attributes: availability!**

SPS is the primary indicator of the system's overall health.

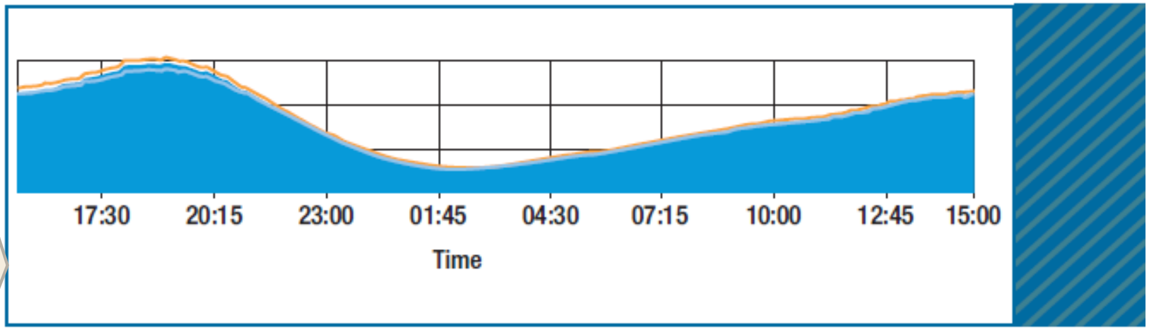
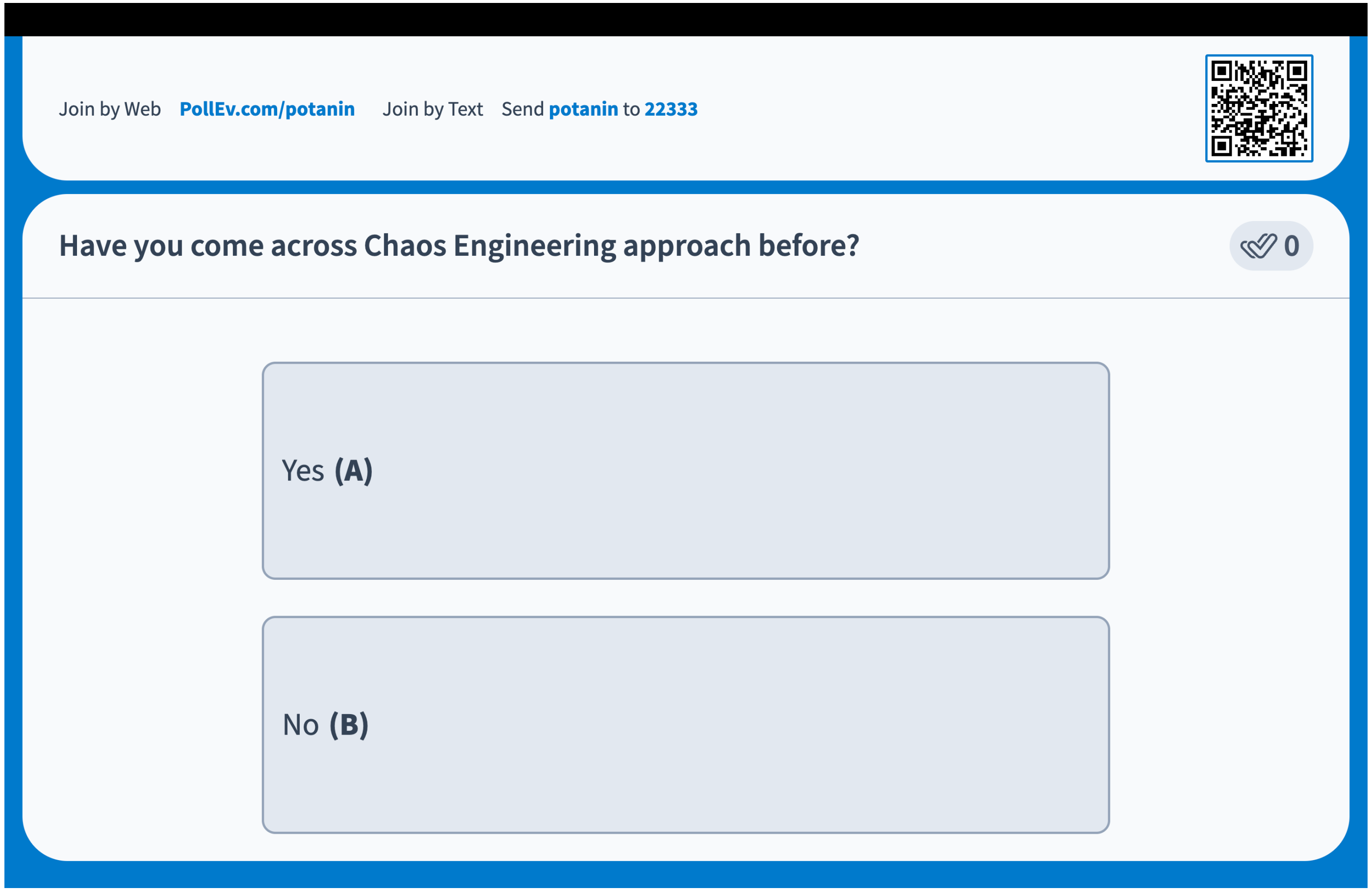


FIGURE 2. A graph of SPS ([stream] starts per second) over a 24-hour period. This metric varies slowly and predictably throughout a day. The orange line shows the trend for the prior week. The *y*-axis isn't labeled because the data is proprietary.

Poll Everywhere Time!

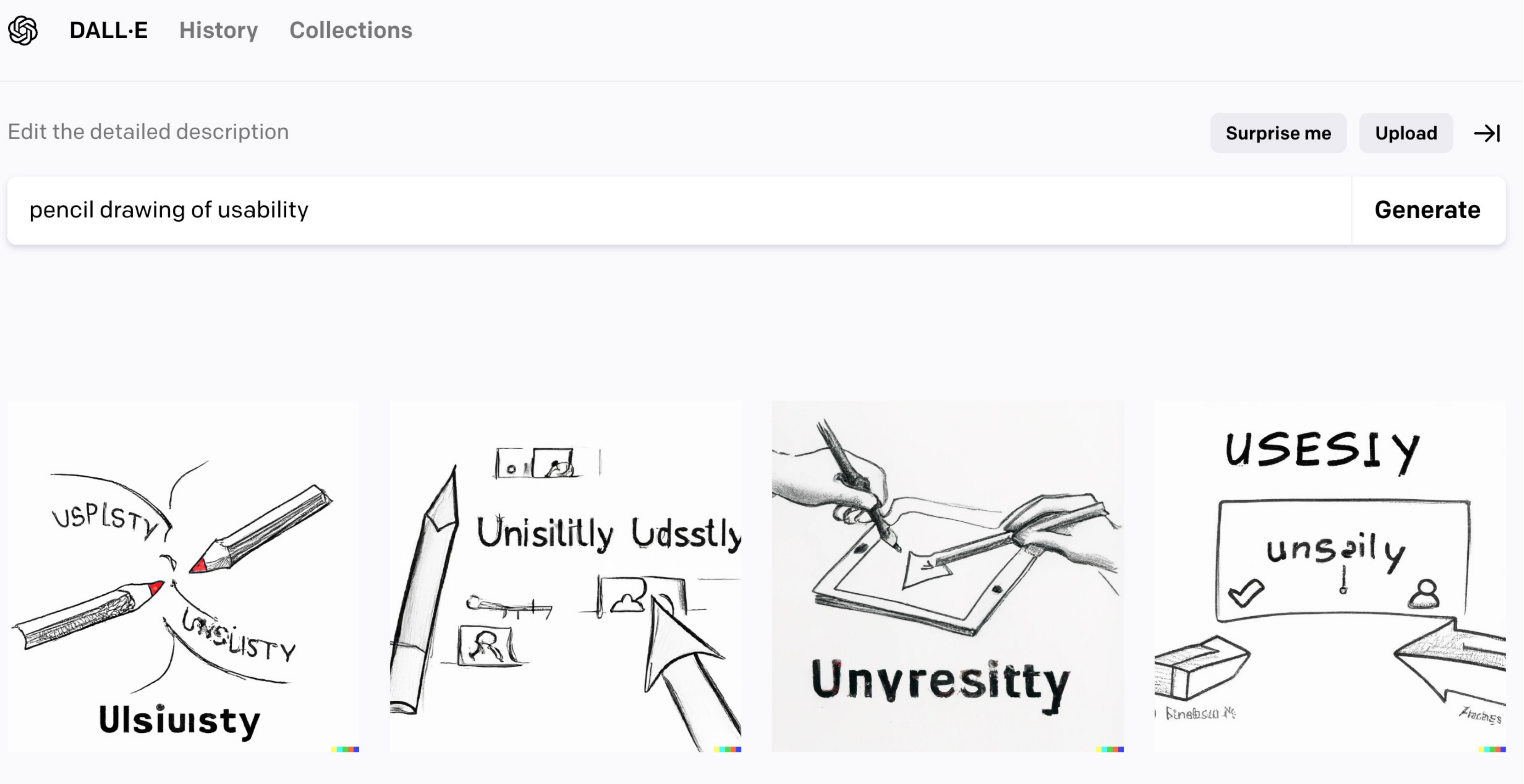


Join by Web **PollEv.com/potanin** Join by Text Send **potanin** to **22333**

Have you come across Chaos Engineering approach before? ✓ 0

Yes **(A)**

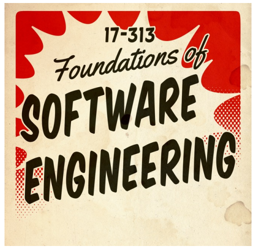
No **(B)**



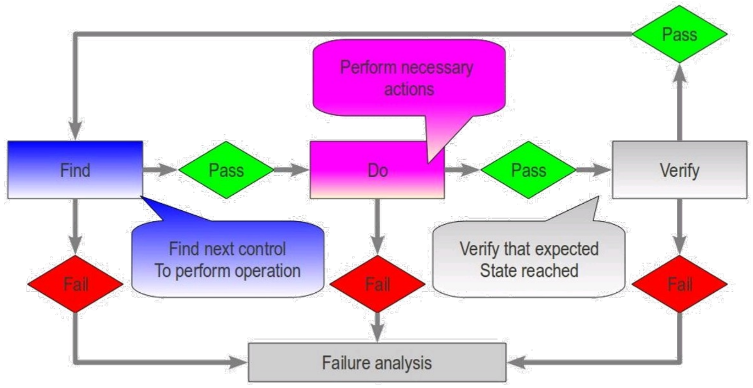
pencil drawing of usability

A/B Testing

Automating GUI/Web Testing



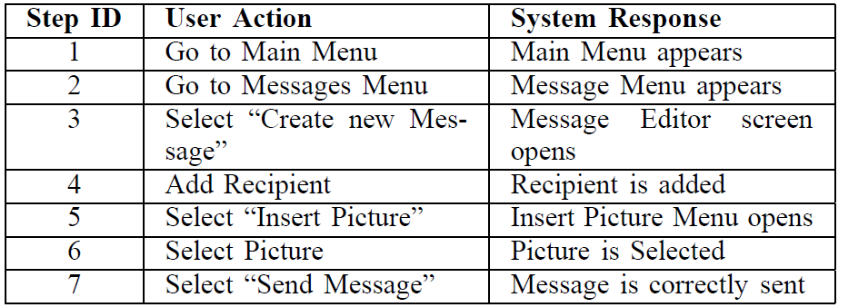
- This is hard
- Capture and Replay Strategy
  - mouse actions
  - system events


Pass
Perform necessary actions
Find
Pass
Do
Pass
Verify
Find next control To perform operation
Verify that expected State reached
Fail
Fail
Fail
Failure analysis

- Test Scripts: (click on button labeled "Start" expect value X in field Y)
- Lots of tools and frameworks
  - e.g. Selenium for browsers
- (Avoid load on GUI testing by separating model from GUI)
- Beyond functional correctness?

Manual Testing

GENERIC TEST CASE: USER SENDS MMS WITH PICTURE ATTACHED.



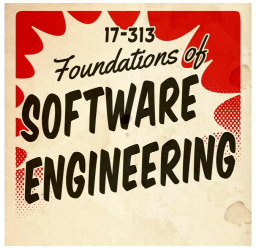
Step ID	User Action	System Response
1	Go to Main Menu	Main Menu appears
2	Go to Messages Menu	Message Menu appears
3	Select "Create new Message"	Message Editor screen opens
4	Add Recipient	Recipient is added
5	Select "Insert Picture"	Insert Picture Menu opens
6	Select Picture	Picture is Selected
7	Select "Send Message"	Message is correctly sent

- Live System?
- Extra Testing System?
- Check output / assertions?
- Effort, Costs?
- Reproducible?
- Higher Quality Feedback to Develope

Usability: A/B testing

- Controlled randomized experiment with two variants, A and B, which are the control and treatment.
- One group of users given A (current system); another random group presented with B; outcomes compared.
- Often used in web or GUI-based applications, especially to test advertising or GUI element placement or design decisions.

Example



- A company sends an advertising email to its customer database, varying the photograph used in the ad...

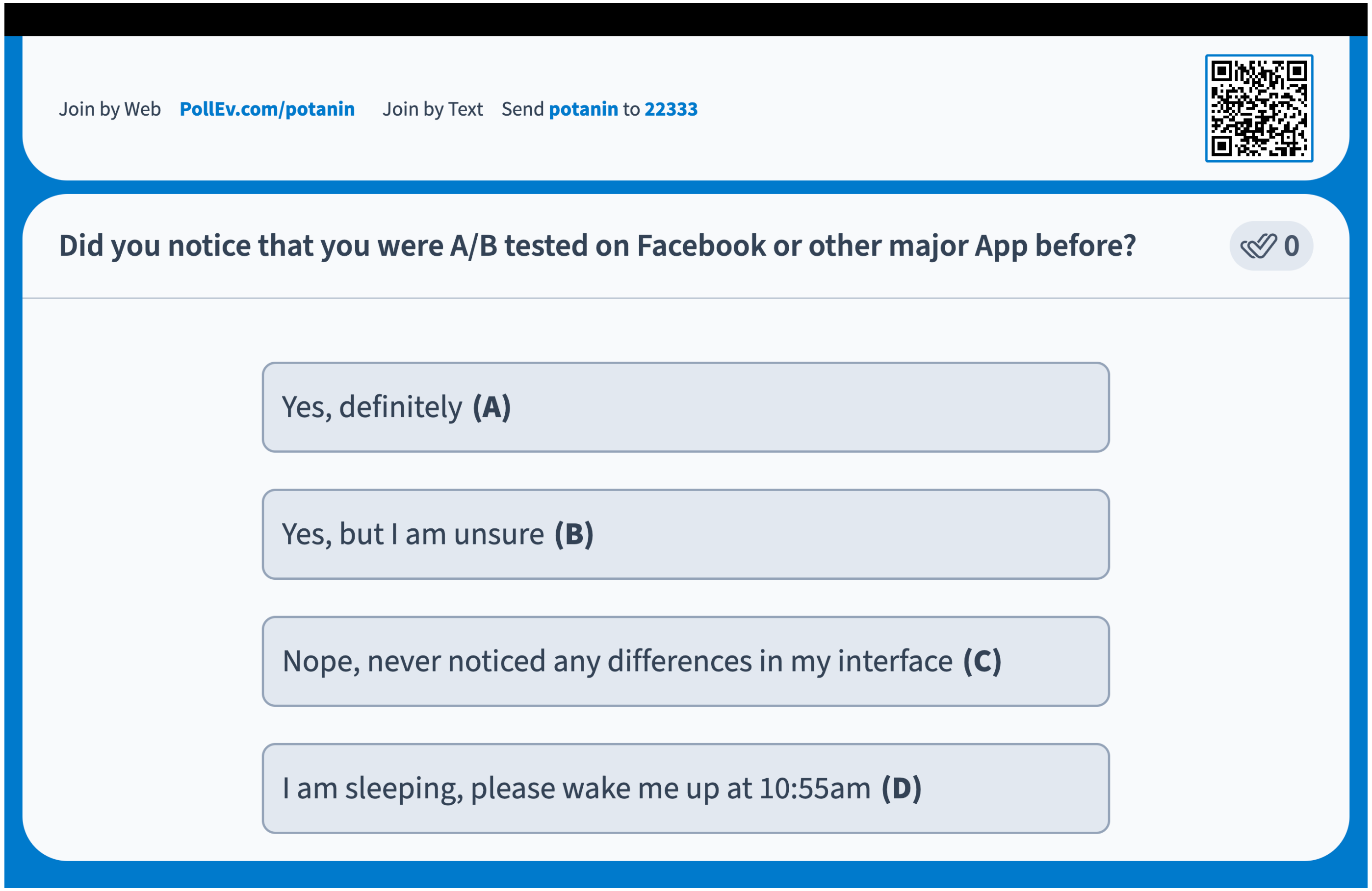
Example: group A (99% of users)

Example: group B (1%)

A/B Testing

- Requires good metrics and statistical tools to identify significant differences.
- E.g. clicks, purchases, video plays
- Must control for confounding factors

Poll Everywhere Time!



Join by Web **PollEv.com/potanin** Join by Text Send **potanin** to **22333**

Did you notice that you were A/B tested on Facebook or other major App before?

- Yes, definitely **(A)**
- Yes, but I am unsure **(B)**
- Nope, never noticed any differences in my interface **(C)**
- I am sleeping, please wake me up at 10:55am **(D)**