Software testing

Programmers! Cast out your guilt! Spend half your time in joyous testing and debugging! Stalk bugs with care, methodology, and reason. Build traps for them. Be more artful than those devious bugs and taste the joys of guiltless programming!

– Boris Beizer

Testing maturity (Beizer 1990)

1. testing is just for debugging
2. testing is to show correctness of the product
3. testing is to show incorrectness of the product
4. testing is to reduce risk
5. testing is a mental discipline to develop better software

Kinds of testing

- Functional
- Non-functional
- Maintenance

Functional testing – terminology

What is it we are looking for?
Bugs, faults, errors, failures?

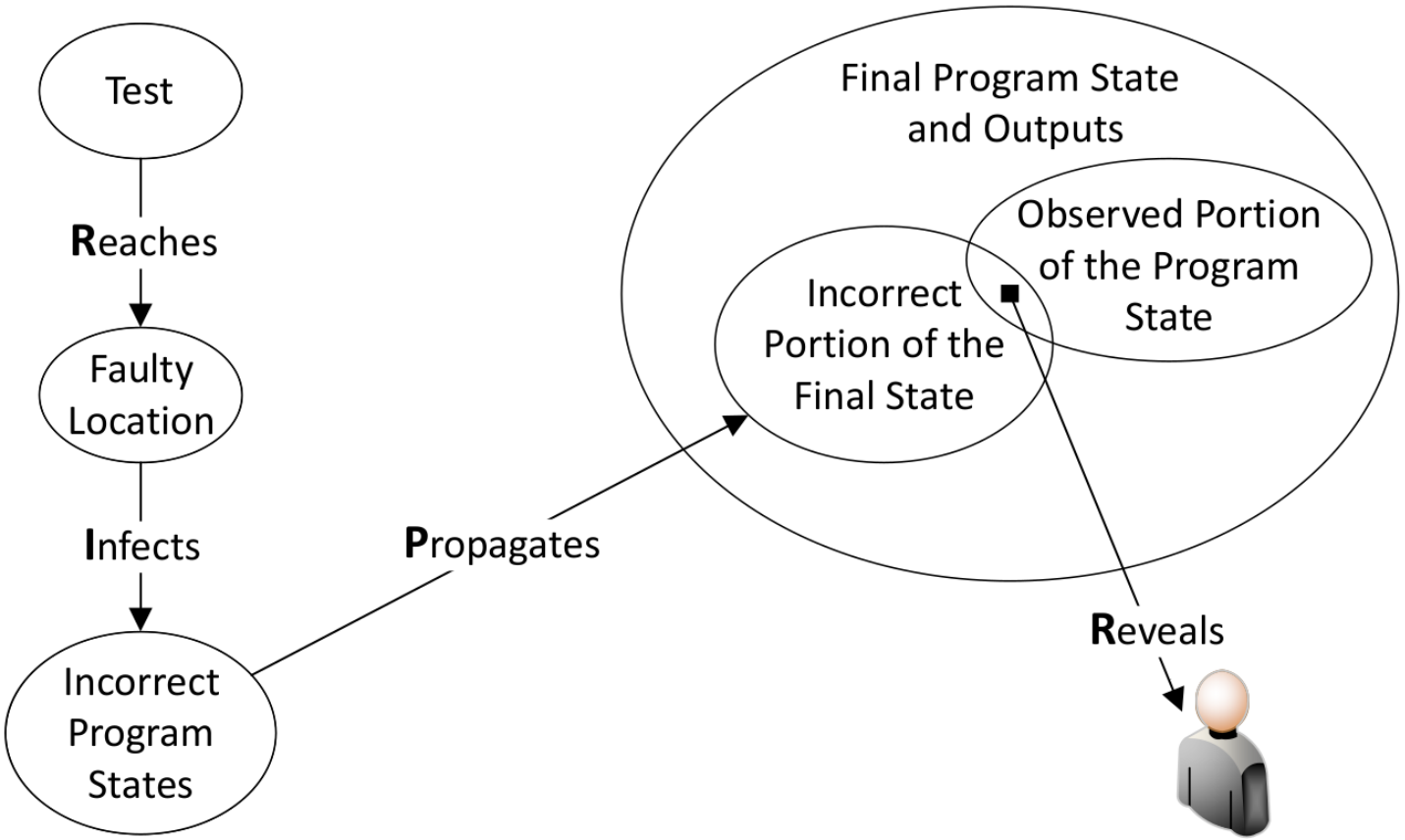
- Fault: a static defect
- Error: an incorrect state: the manifestation of some fault
- Failure: an incorrect behaviour

Terminology

```
public static int numZero(int[] x) {
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) count++;
  }
  return count;
}
```

- Fault: `int i = 1` instead of `int i = 0`
- Error: program state before the first loop check
- Failure: wrong result for input `x = {0, 7, 2}`

The RIPR model



The RIPR model

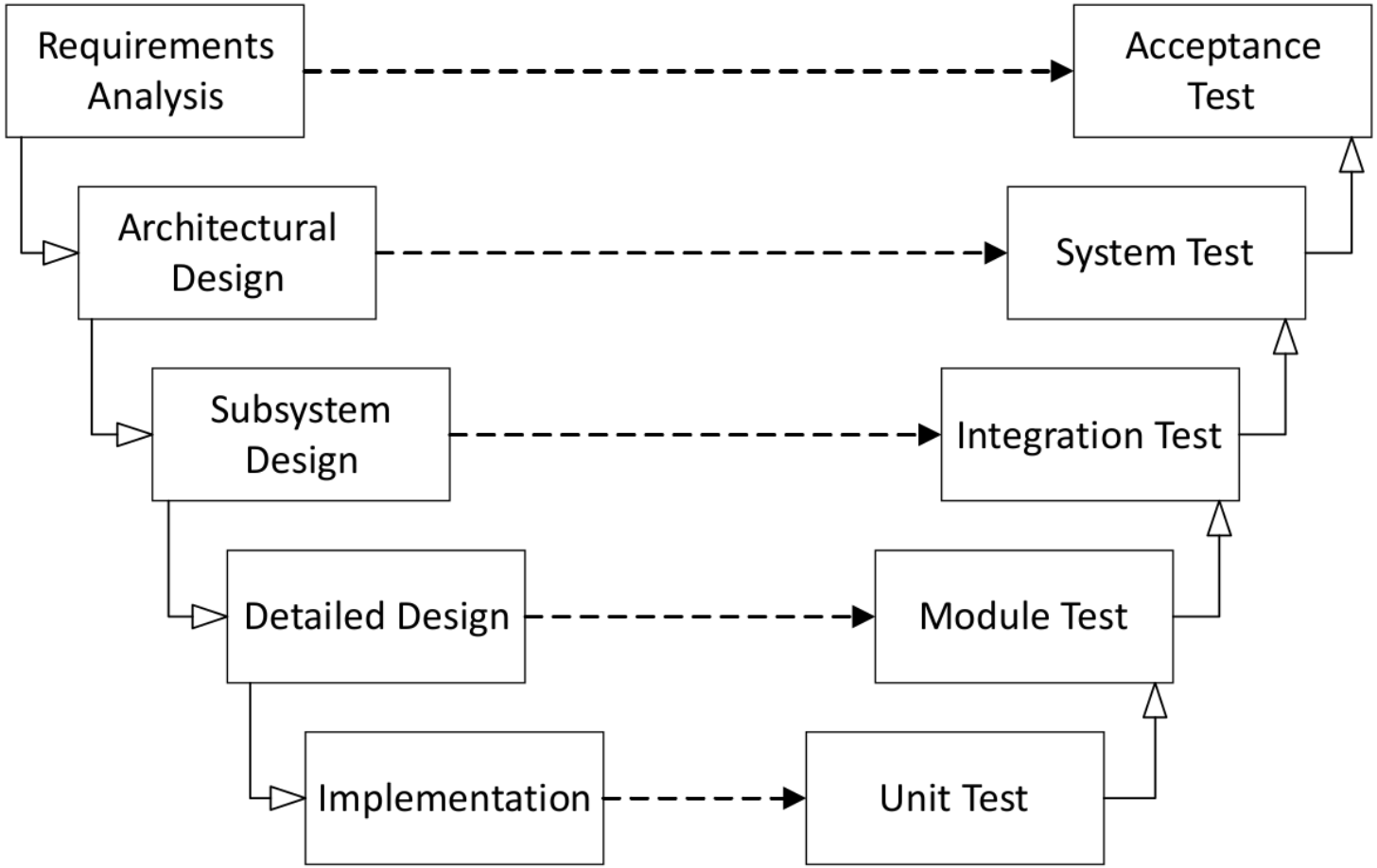
Four conditions necessary for a failure to be observed:

- Reachability: the location(s) in the program that contain the fault must be reached.
- Infection: the state of the program must be incorrect.
- Propagation: the infected state must cause some output or final part of the program to be incorrect.
- Reveal: the tester must observe part of the incorrect portion of the program state.

Traditional testing levels

- unit testing (or intra-method)
- module testing (or inter-method, intra-class)
- integration testing (or inter-class)
- system testing
- acceptance testing

The V-model


Requirements Analysis
Acceptance Test
Architectural Design
System Test
Subsystem Design
Integration Test
Detailed Design
Module Test
Implementation
Unit Test

Unit and module testing

Testing correctness of individual units of code.

Manual:

```
void test_validation() {
  do {
    string num = input("Type account number:  ");
    println("Result:  " + validate_account(num));
  }
}
```

Unit and module testing

Testing correctness of individual units of code.

Automatic:

```java
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class MyTests {

    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero() {
        MyClass tester = new MyClass(); // MyClass is tested

        // assert statements
        assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");
        assertEquals(0, tester.multiply(0, 10), "0 x 10 must be 0");
        assertEquals(0, tester.multiply(0, 0), "0 x 0 must be 0");
    }
}
```

Unit and module testing

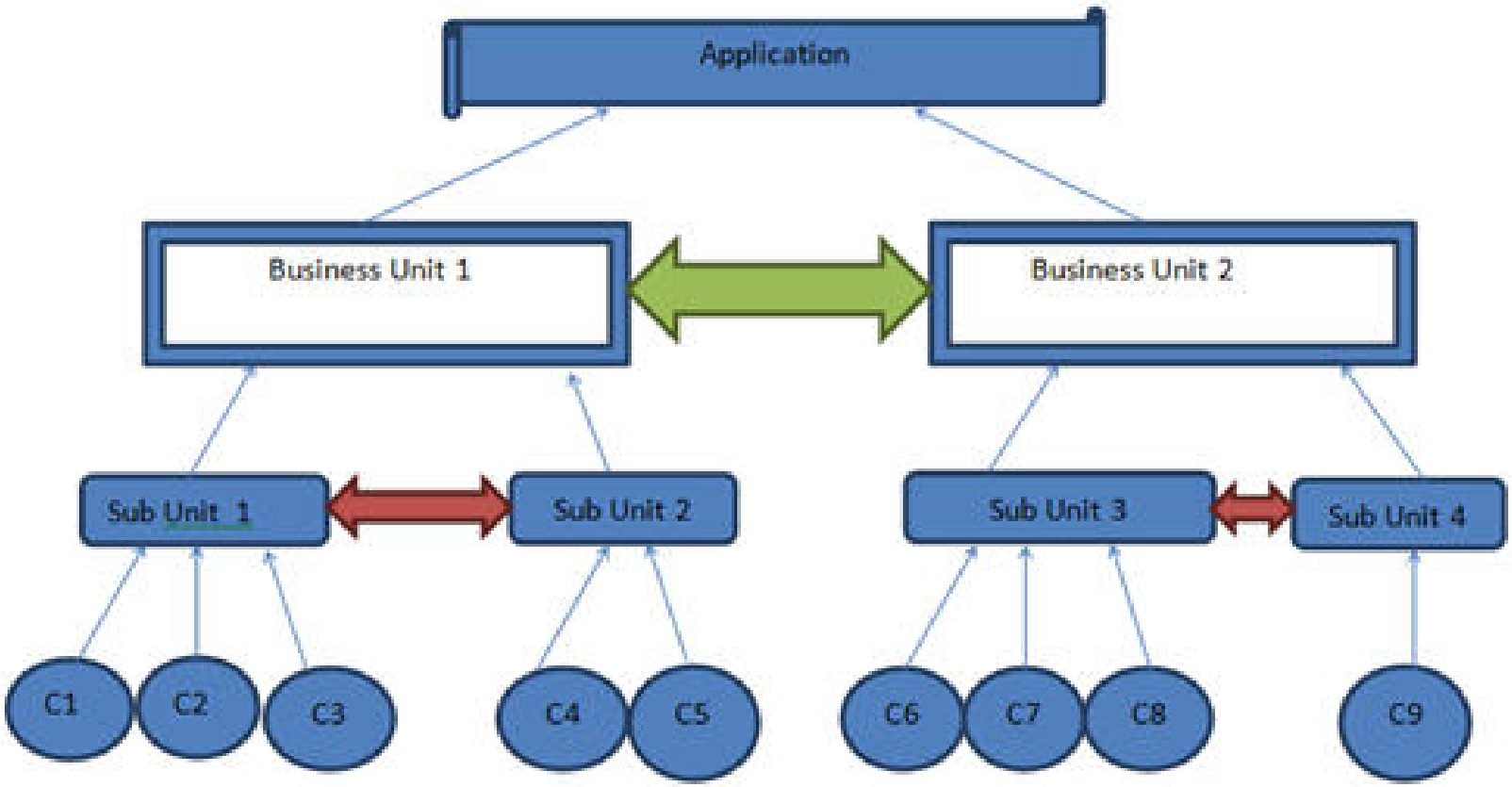
- goal: verify the internal logic of the code by testing every possible branch (aka test coverage)
- static unit testing: code review for all possible behaviours
- dynamic unit testing: program unit is executed and its outcomes observed, or compared to expected outcomes
- can be done both manually and automatically, white-box and black box

Integration testing

Verifying that different software modules work in unity.

- focuses on interactions and data flow between modules
- done before, during and after integration of a new module into the main software package
- input: unit-tested modules or stubs / mock objects.
- modules are put together in an incremental manner
- additional modules should work without disturbing existing functionality
- can be done both manually and automatically, white box and black box (but typically black box)

Integration testing



Kinds of integration testing

- Big Bang – integrate and test all components at once.
- Bottom Up – combine low-level modules first.
- Top Down – combine high-level modules first.
- Sandwich – a combination of the above.

System testing

Testing documented requirements of the fully integrated software.

- black box testing, typically done by a professional testing agent
- includes both functional and non-functional testing
  - Smoke testing – does the absolute core functionality work?
  - Functionality testing – does it do what it should?
  - Robustness – does it recover well from input errors or failures?
  - Stress – what are the limitations / how does it deal with them?
  - Performance – does it respond quickly / use low resources?
  - Scalability – can it be used on a large scale?
  - Stability – does it keep running under full load?
  - Regression – does everything still work after maintenance?

Acceptance testing (or: beta-testing)

Testing usability by actual users.

- should be undertaken by a subject matter expert
- typically done by the customer or end-users

Designing automated tests

Bill Sempf
@sempf

Follow

QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv.

10:56 AM - 23 Sep 2014

Automated unit testing

- automatically test a single unit of code
- tests are designed to be repeatable
- testing outside the usual call chain exposes dependencies
- should contain both positive and negative outcomes
- can be done early in development
- typically done as white-box testing, but also black-box
- to test units, use mock objects, method stubs

Mocking, stubbing, etc.

- Fake objects – working objects, but which take shortcuts
- Stubs – objects providing fixed answers
  `int get_random_number() { return 42; }`
- Spies – stubs that record some information
  `void send_mail() { sent++; }`
- Mocks – objects preprogrammed with expectations

Mocking, stubbing, etc.

Cook $\Longleftarrow$ Waiter $\Longleftarrow$ CustomerTest driver

- Fake cook: supplies frozen dinners with the right name
- Stub cook: always gives a hotdog
- Spy cook: always gives a hotdog, but remembers what was actually asked
- Mock cook: is told by the test driver to expect a hamburger request and given a hamburger to return, and will start screaming if asked for a hot dog

Side benefits of automated unit testing

- forces you to consider edge cases and error handling early on
- pushes you to have a decoupled and cohesive design
- decreases the barrier to refactoring code
- provides a kind of living documentation for the system

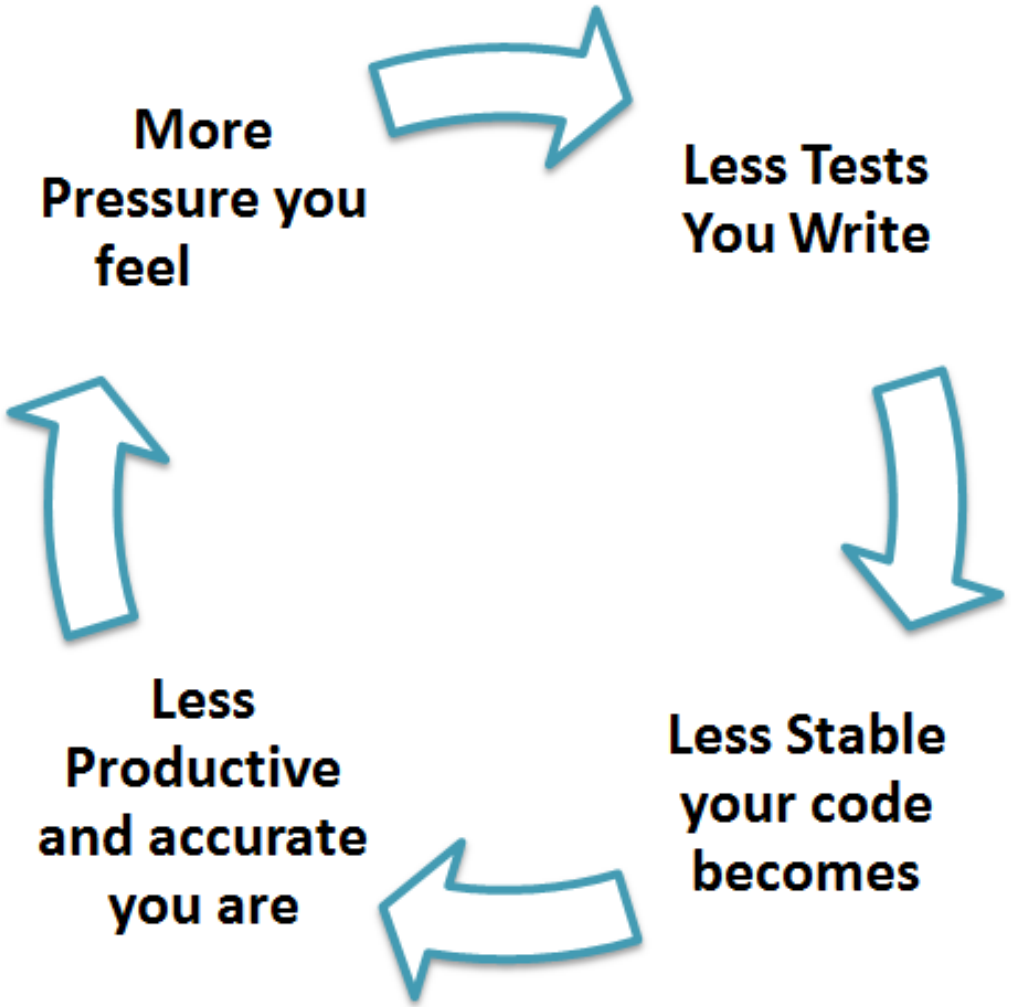
Unit testing advice

- clear descriptive names of test functions
- clear failure description on asserts
  `assertEquals("adding one day to 2050/2/15", expected, actual);`
- when appropriate, use a timeout
  `@Test(timeout = 5000)`
- test one thing at a time per method (preferably: one assert)
  (use `@Before` and `@After` for setup and teardown functions)
- tests should avoid logic (minimise if/else, loops, try/catch)

Unit tests – things to think about

- What is wrong – the thing that is tested, or the test?
- Does not show absence of errors, only particular errors.
- If the same person writes the test and the code, both may have the same problem.
- The longer a unit test exists, the greater the chance that it is not representative.
- Maintain unit tests as a first-class part of the code.
- May seem to take a lot of time, but actually saves time.

Unit tests – things to think about



Manual versus automatic testing

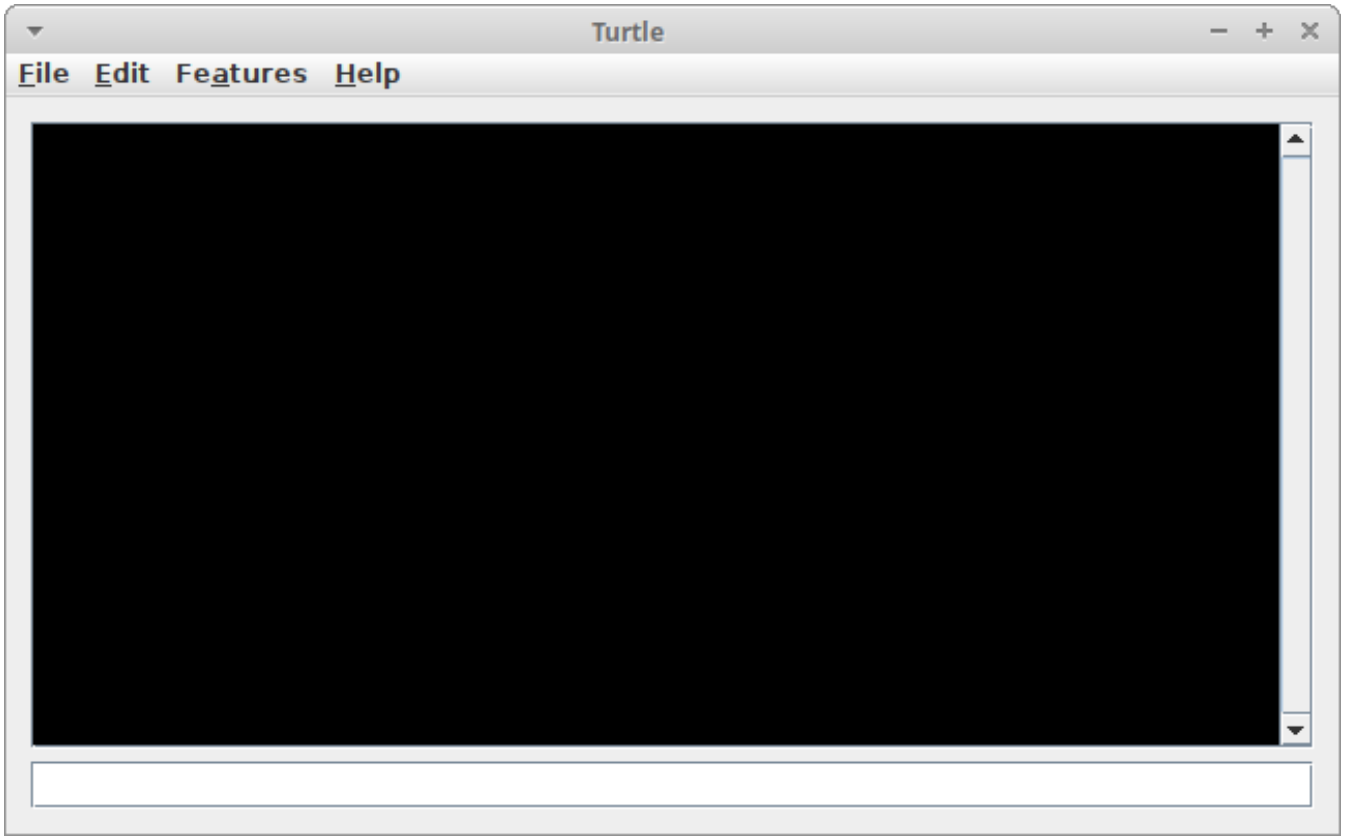
- manual testing is time and cost consuming, especially with repetition
- prone to human error (but: this works both ways)
- manual testing will capture problems that are hard to find automatically
- of course: both needed

Recall: software design principles

- Single Responsibility Principle
- Open-Closed Principle
- Dependency Inversion Principle
- Isolate Third-Party Components

Satisfying these principles makes the code more testable! Both because it allows units to be tested separately (and without unnecessary side effects), and because good use of principles makes it easier to mock certain objects like low-level classes or third-party components.

Class exercise: designing testable code



Class exercise: designing testable code

The input window:

- It should mostly act like a normal GUI component (left and right arrow keys, entering commands, selecting, etc.).
- When return is pressed, the current text should be sent, and selected (for easy deleting).
- The up and down arrow keys browse through the "history" of previously sent lines.
- When new text is sent, it is put at the bottom of the history, and this position is selected.
- When browsing history, any change also causes the history position to go to the bottom.
- (Perhaps some more requirements.)

Class exercise: designing testable code

Naive implementation:

- Put it all in one class "input window".

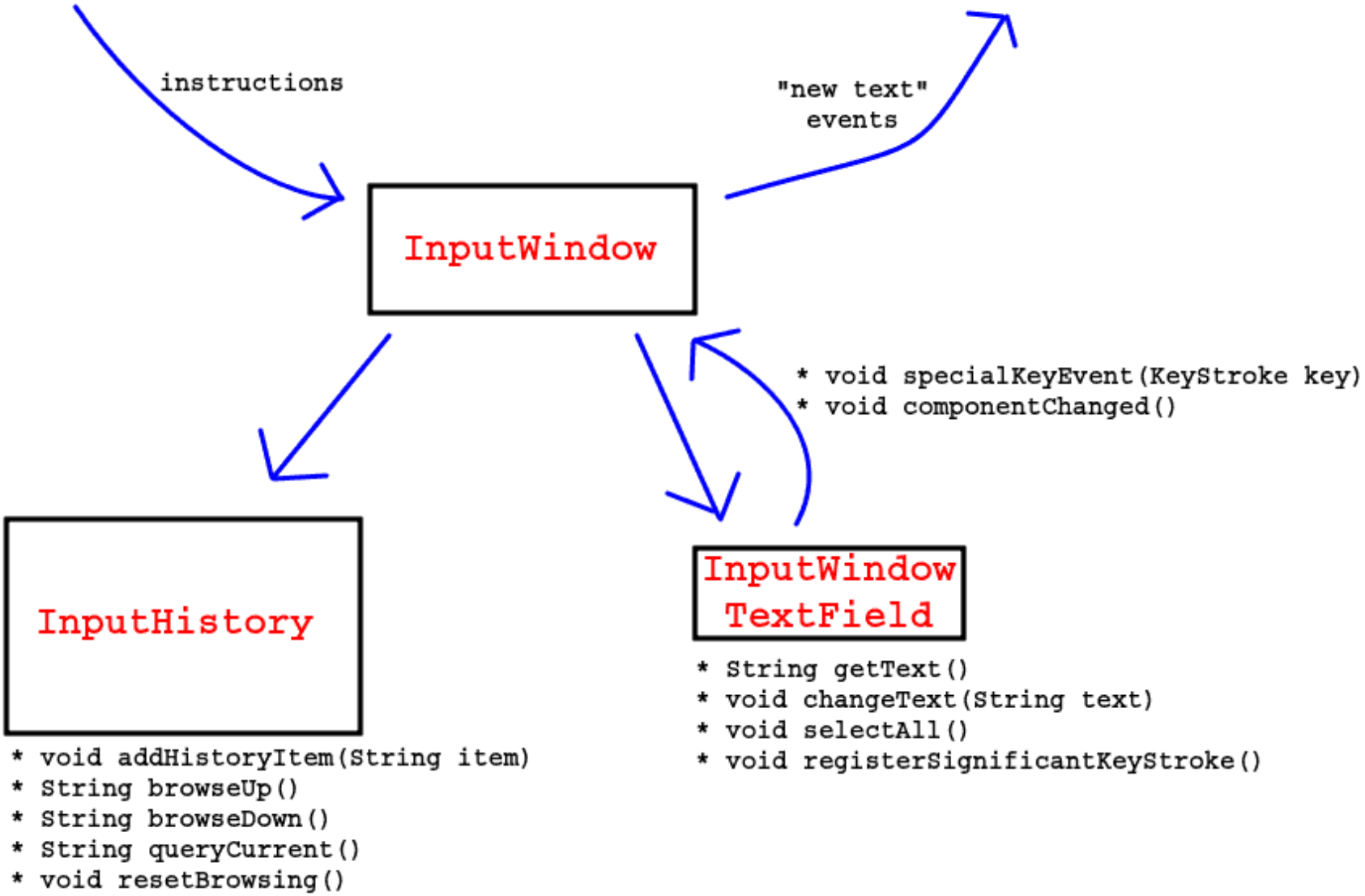
Better implementation:

- Separate the history! This is a component with its own responsibility.

Best implementation:

- Also separate the actual GUI component!

Class exercise: designing testable code



Class exercise: designing testable code

Note:

- The InputHistory is a basic class with a single responsibility and no dependencies. It is easy to test automatically.
- The InputWindow is a manager class. It is the only one of the three classes that interacts with the outside world (e.g., giving events when return is pressed).
- The InputWindow can be tested without the InputHistory and InputWindowTextField by replacing these two by fake, stub or mock objects. For example: call InputWindow.specialKeyEvent to indicate that return was pressed, and test whether it sends the text in the textfield stub on the event bus and passes it into the history spy.

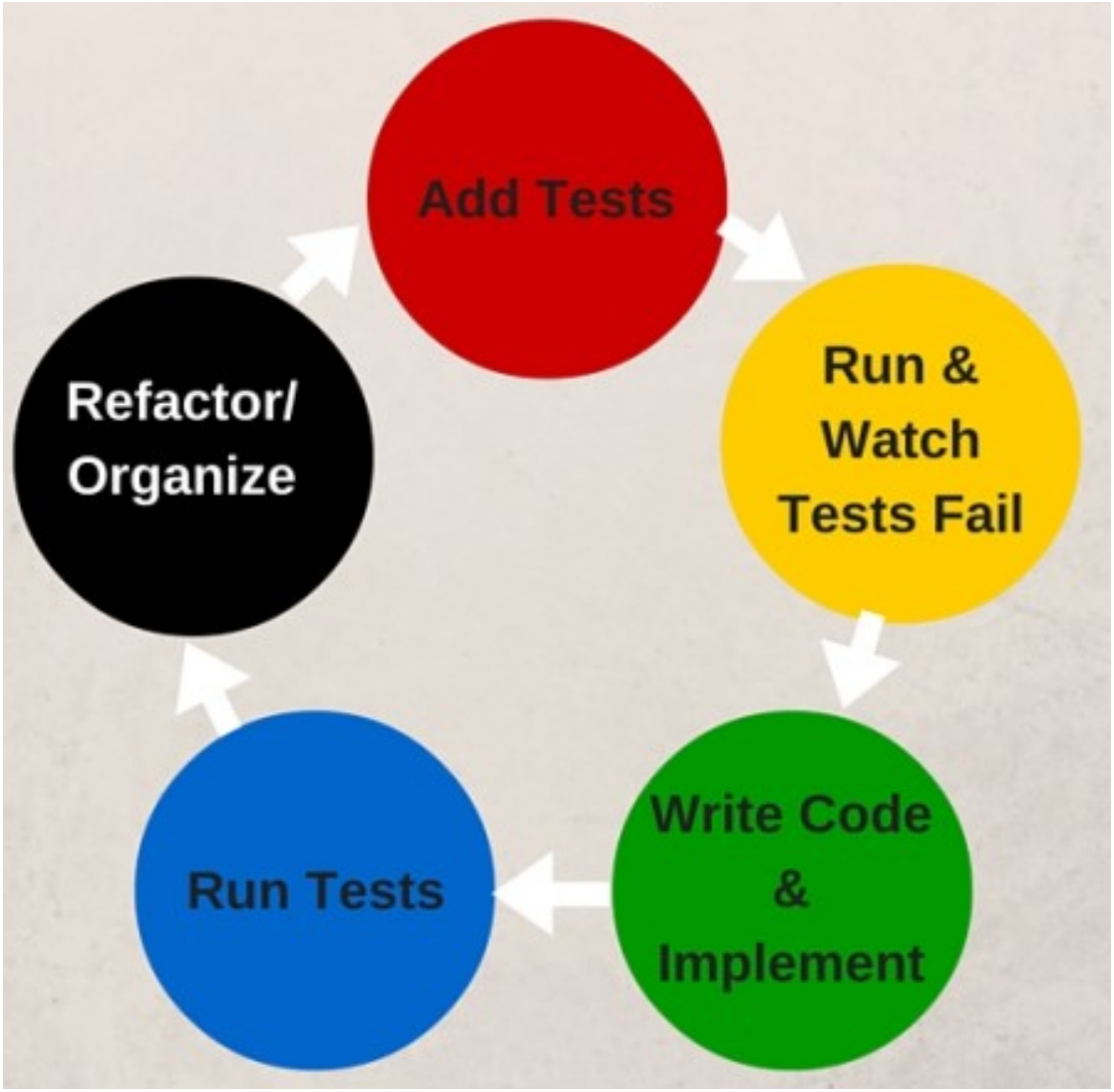
Class exercise: designing testable code

Note:

- The InputWindowTextField is hard to test. Depending on your test framework, this may require manual testing. This can still be done systematically: define manual tests, and agree that they are executed whenever someone changes the component.
- The InputWindowTextField is a very small class, which inherits the relevant GUI component, can be questioned for active text, and passes on requested key events. Because it is so small, it will rarely need changing, and only minimal testing.
- The InputWindowTextField does not know about the InputHistory: it is simply given an object to which it must pass special key events and componentChanged() notifications. This makes it easier to systematically test it in isolation.

Some Considerations

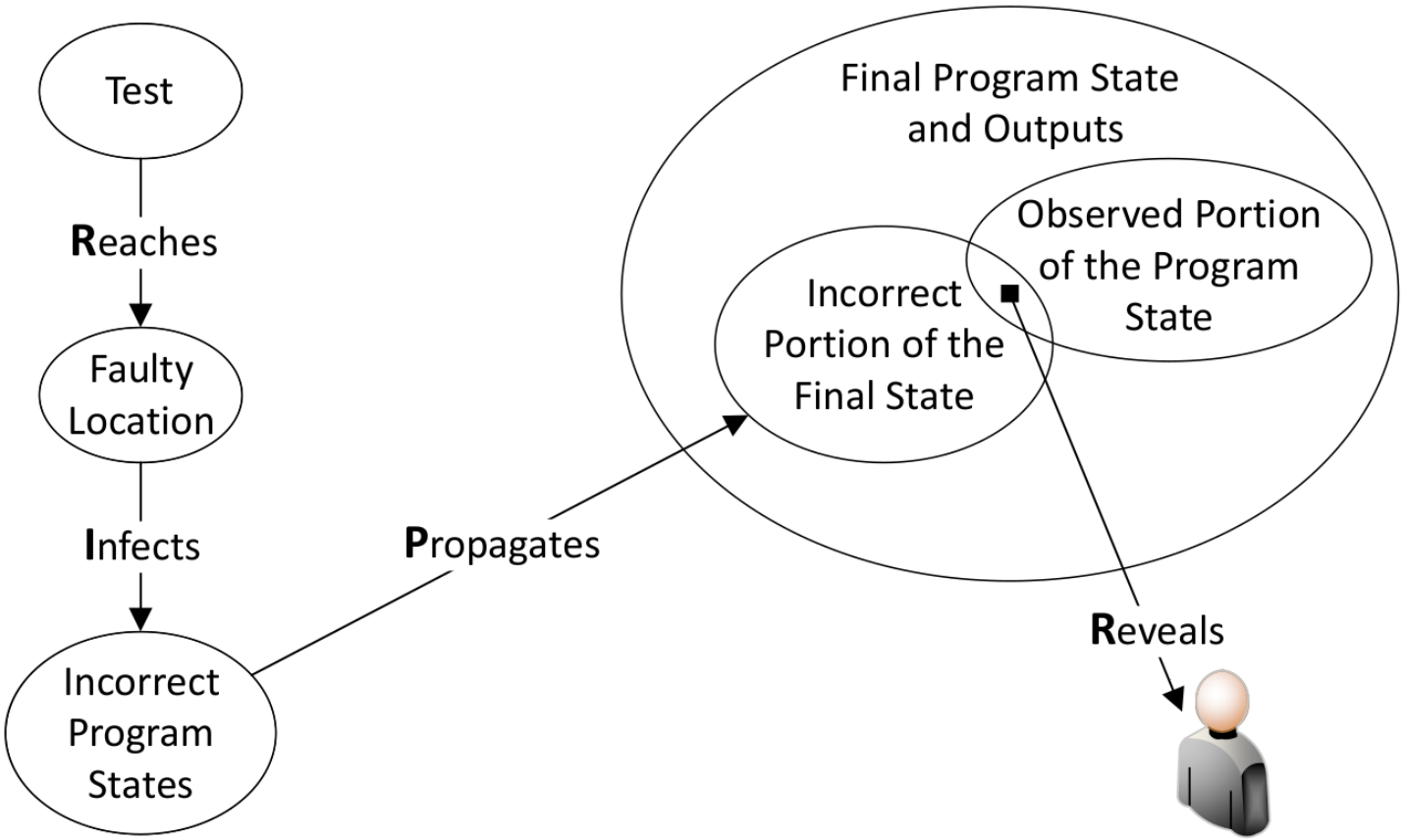
Test-Driven Development



Principles of TDD

- always write the test before the code
- write the simplest code that works
- not just unit tests; also have tests based on user stories (ATDD)
- works well with specification by example
- minimise code in hard-to-test modules
- best used from the start
  (but boy scout rule)

Recall: the RIPR model



Mutation testing: considering the **P** in RIPR

Basic idea:

- A program is changed at exactly one place, in a small way.
- The test set is run to see whether it outlaws this mutant.
- If so, the mutant is called killed.
- The aim is to kill as many mutants as possible.
- The killing score is perceived as a goodness measure for the test set.

Erroneous perception

"I'll just find the bugs by running the client program."

Too often:

- testing is seen as a novice's job
- testing is assigned to the least experienced team member
- testing is done as an afterthought (or not at all)

Definition of done: includes testing!

Tasks in testing:

- test design
  - criteria-based
  - human-based
- test automation
- test execution
- test evaluation

Each type of activity requires different skills, background, knowledge, education and training!

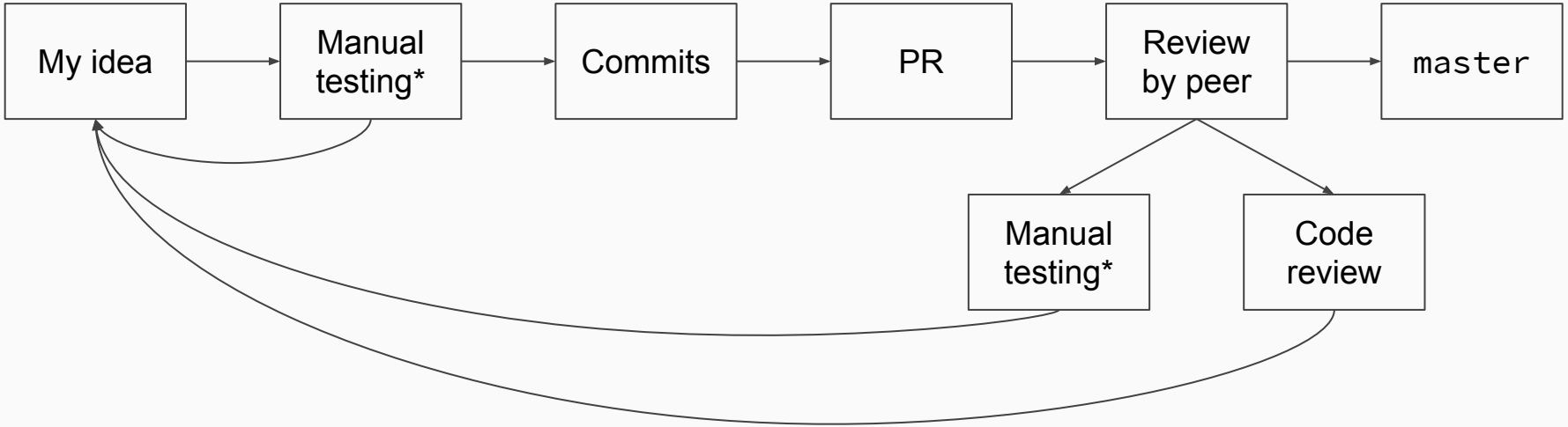
Principles of testing

- Exhaustive testing is not possible.
  (So you'll have to do risk assessment and try to find a representative sample of test cases.)
- Defect clustering: 80% of the problems in 20% of the modules.
- Pesticide paradox: review test cases occasionally.
- Testing shows *presence*, not *absence* of defects.
- Absence of error does not imply usability!
- Test early, test often.

Continuous Integration

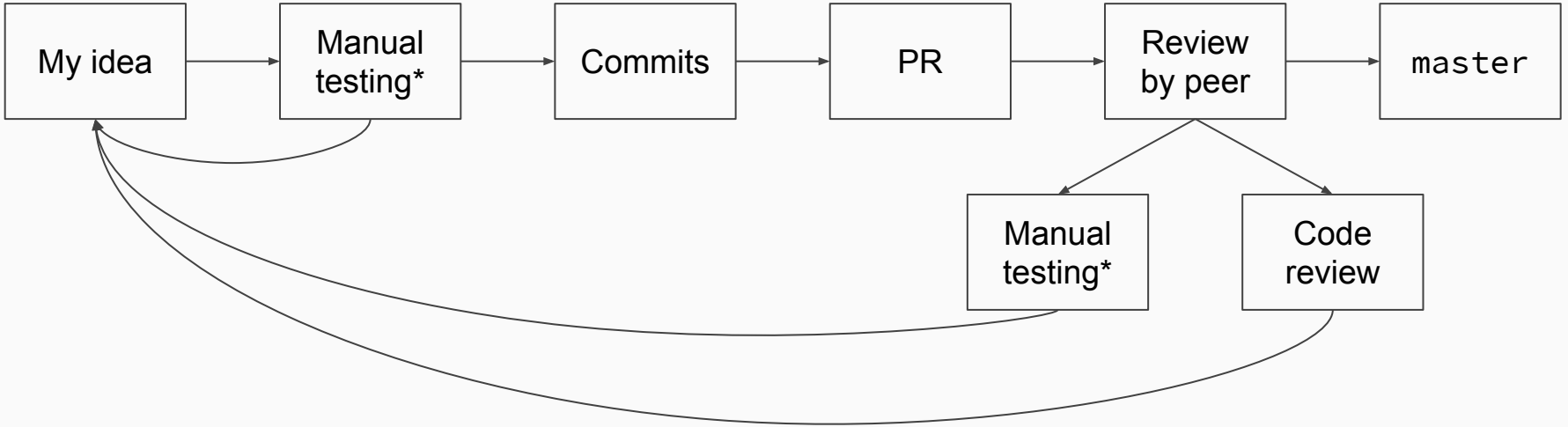
By Joren Vrancken

Manual workflow



*manually testing changes and manually running unit/integration/etc tests.

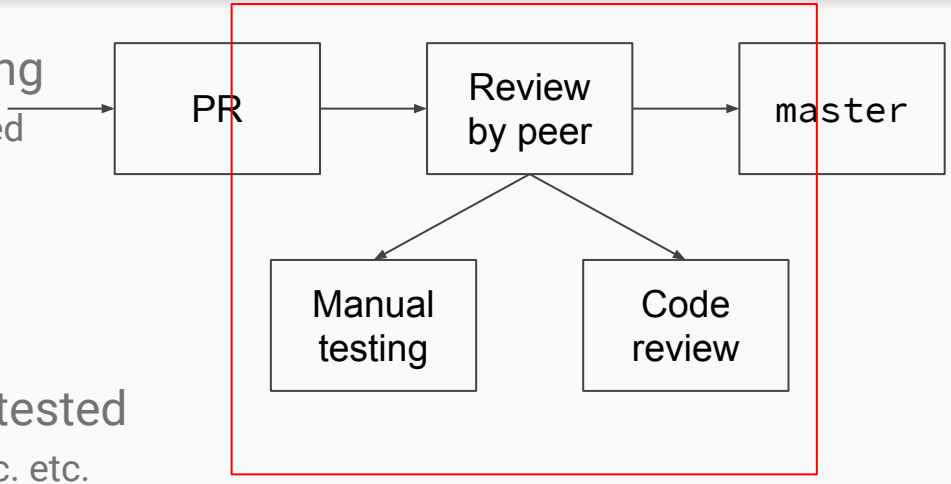
What Can Go Wrong?



*manually testing changes and manually running unit/integration/etc tests.

The human error in manual workflow

- Every change needs manual checking
  - Codestyle mistakes are easily overlooked
- No consistency in what is tested
  - Easy to forget something
  - Different views between reviewers
  - “This is so simple, I don’t have to test it”
- No consistency in how a change is tested
  - Different operating systems, laptops, etc. etc.



Continuous Integration

Automated tasks on your codebase that run under certain conditions.

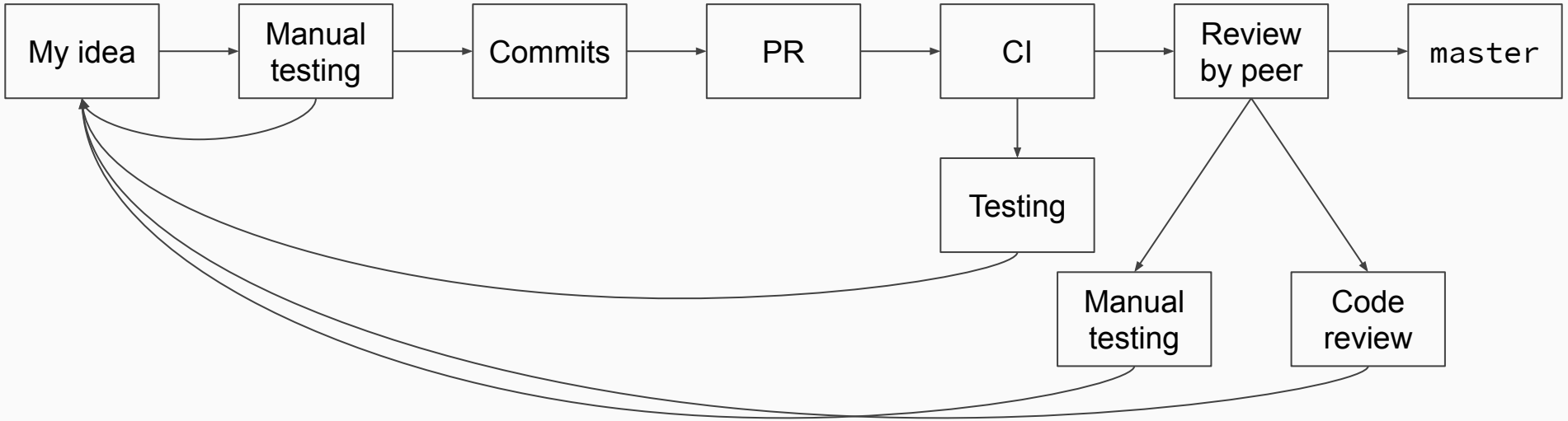
Goals of CI

- Automate running the boring/tedious/time consuming parts of a review
- Consistency in what is tested
- Consistency in test environment
  - Similar to the production environment

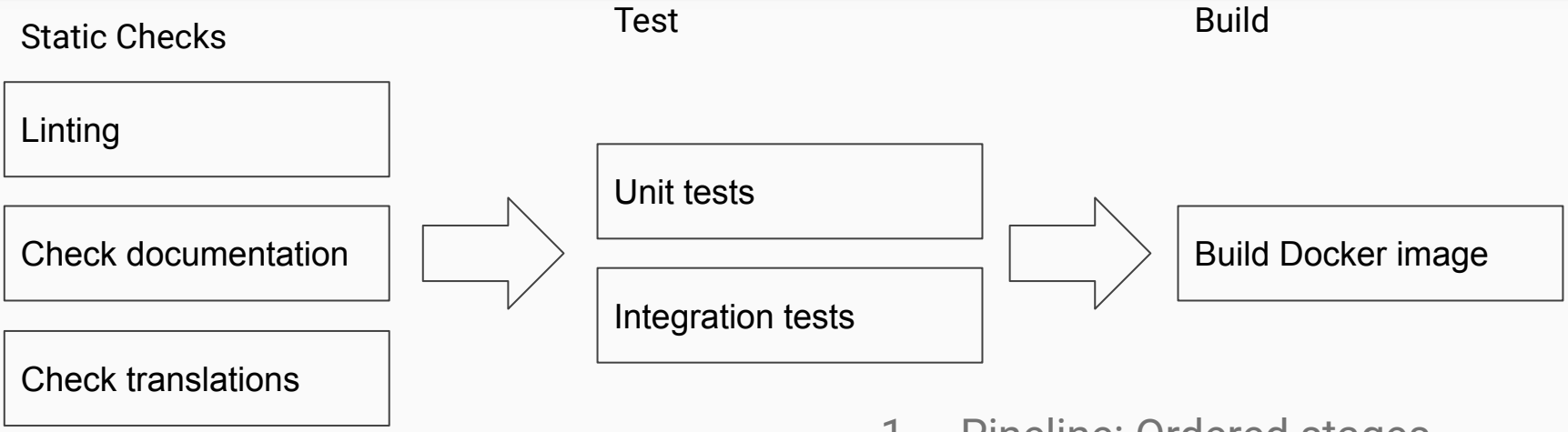
What to automate using CI?

- Running integration/unit tests
- Linting
- Checking whether documentation is up to date
- Checking for visual changes
- Whatever you want

Continuous Integration Workflow



CI Pipeline



1. Pineline: Ordered stages
2. Stage: One or more jobs
3. Jobs: Ordered steps to reach a goal

Continuous Deployment

- Deploy software using CI
- Automate building
- Automate deployment
  - Under specific conditions (e.g. only on the `master` branch)

GitHub Actions

- GitHubs own built-in CI platform
- Released last November and perceived as a development “game changer”
- More than CI
  - Distributable
  - Define steps using Unix commands, Javascript or Docker images
  - Execute jobs on events (e.g. starring, first contributor)

My first GitHub Actions Workflow

Demonstration

GitHub Actions Documentation

https://help.github.com/en/actions