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

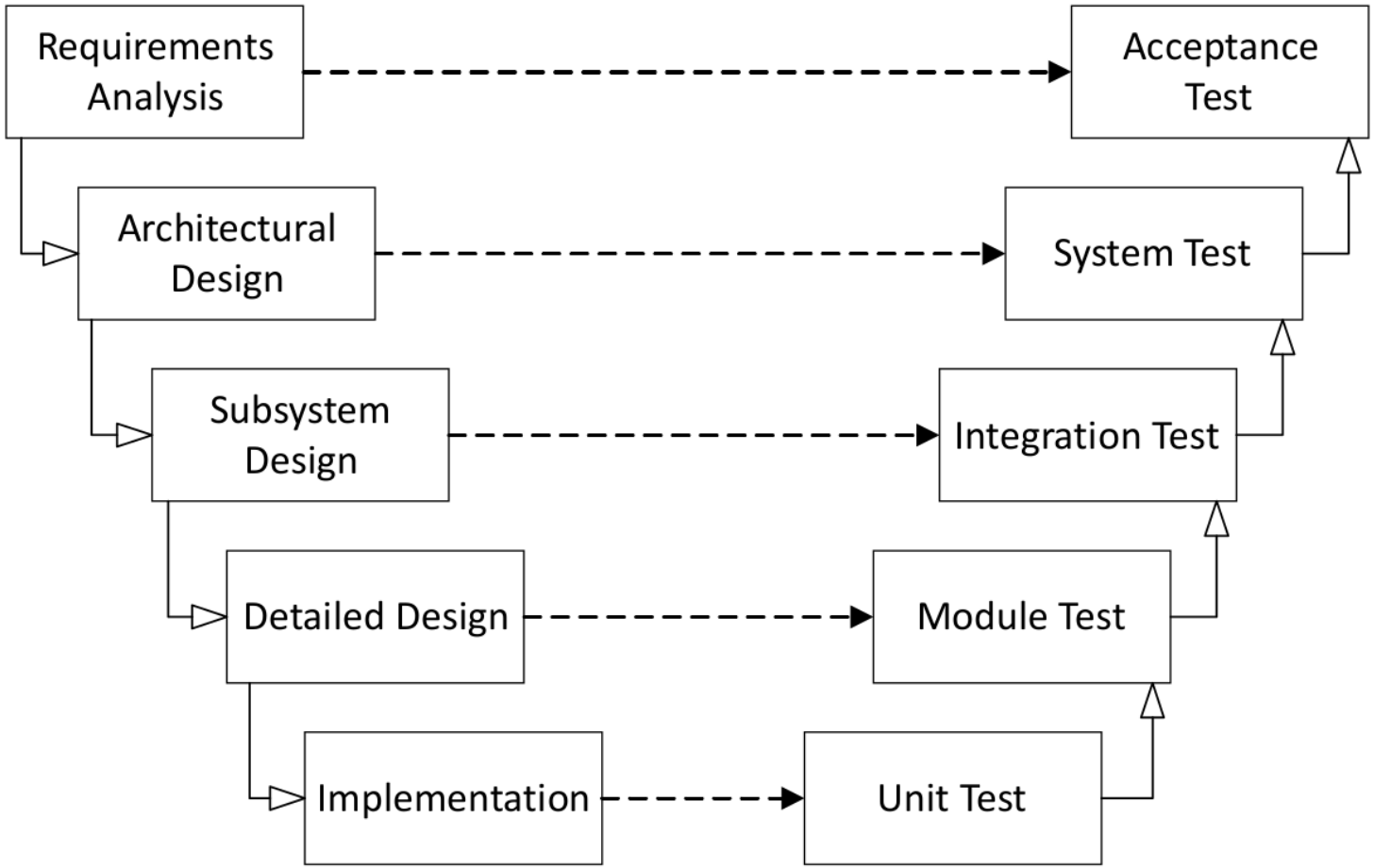
```
public static int countZeroes(int[] x) {
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    if (x[i] == 0) count++;
  }
  return count;
}
```

- 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}`

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

Testing in an agile setting

- **recall**: goal is to have a running product after each sprint
- one instance of V-model for each feature (roughly)
- "write a little, test a little, write a little, test a little"
- testing makes (safe) refactoring possible
- advanced: test-driven development $\rightarrow$ more on that later

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, using a testing framework (e.g., JUnit):

```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 code as to achieve a good coverage of possible behaviors (according to some coverage criteria)
- 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

2 unit tests. 0 integration tests

5:26 PM · Jan 14, 2016 · Twitter Web Client

11.9K Retweets **72** Quote Tweets **8,629** Likes

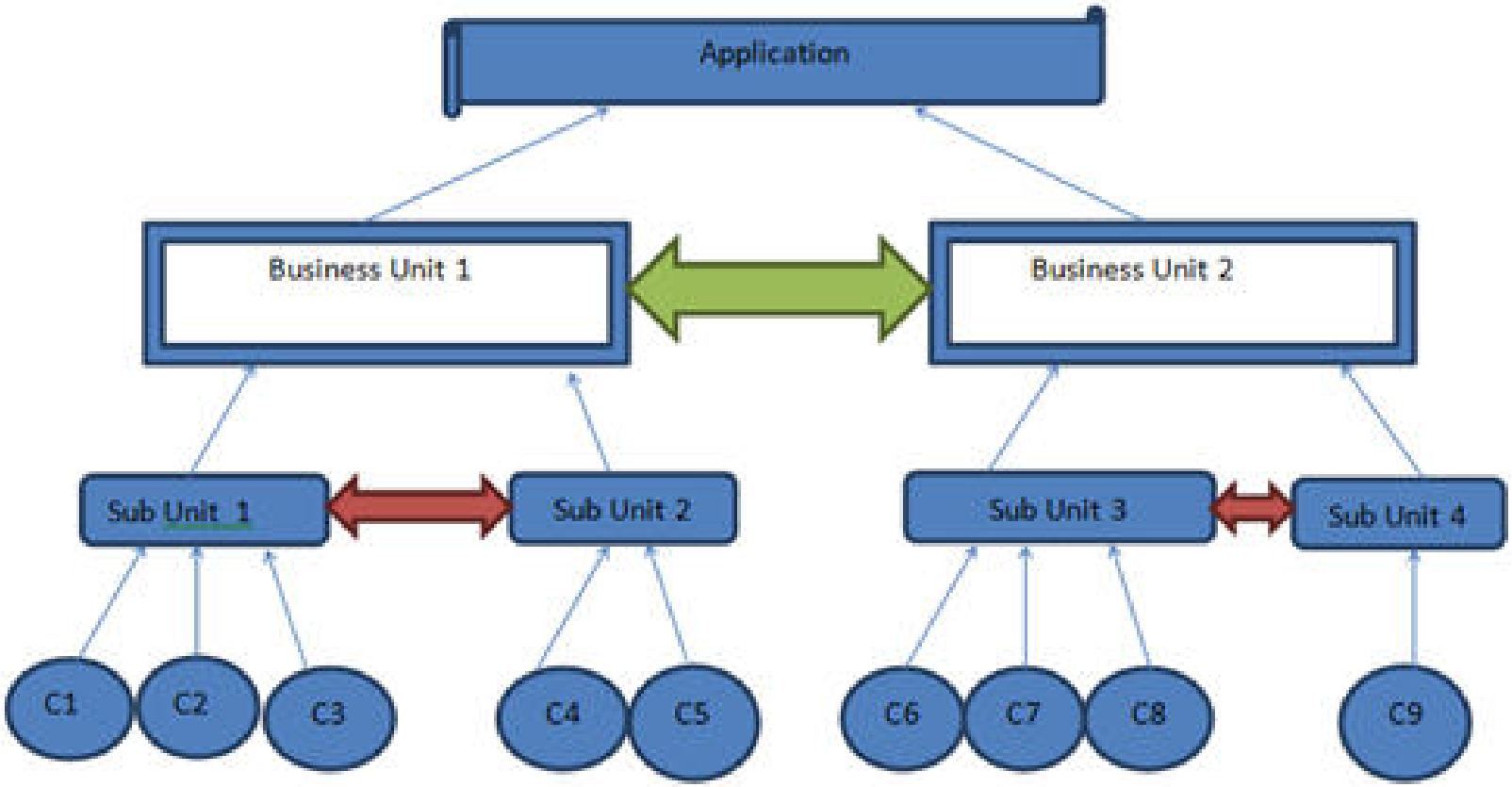
Source: https://twitter.com/ThePracticalDev/status/687672086152753152

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
- Stubs
- Spies
- Mocks

Mocking, stubbing, etc.

- Fake objects – working objects, but which take shortcuts
- Stubs
- Spies
- Mocks

Mocking, stubbing, etc.

- Fake objects – working objects, but which take shortcuts
- Stubs – objects providing fixed answers

- Spies

- Mocks

Mocking, stubbing, etc.

- Fake objects – working objects, but which take shortcuts
- Stubs – objects providing fixed answers
  `int get_random_number() { return 42; }`
- Spies

- Mocks

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

- Mocks

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

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 $\Longleftarrow$ Customer

- 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

Mocking, stubbing, etc.

Cook $\Longleftarrow$ Test 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

Mocking, stubbing, etc.

Cook $\Longleftarrow$ Waiter $\Longleftarrow$ Test 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

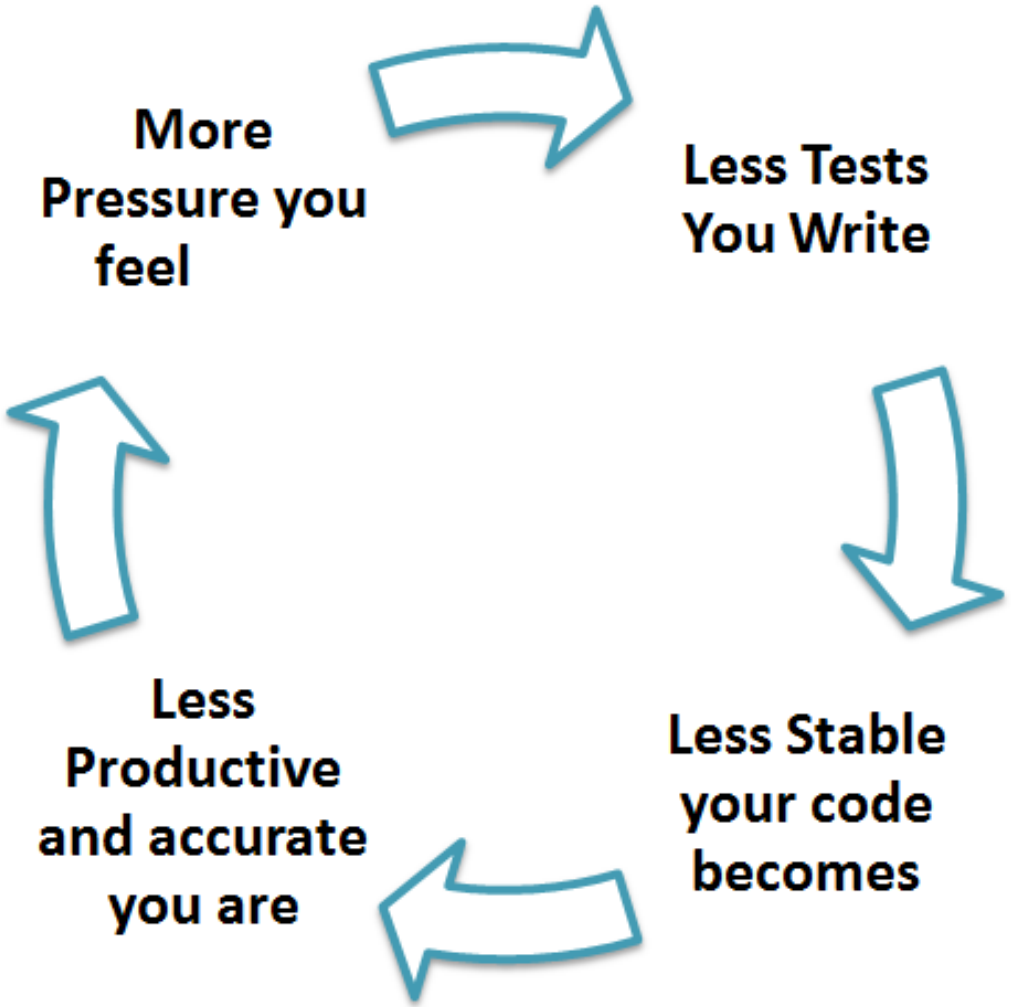
Unit testing advice

- 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

Coverage Criteria

Code coverage

- Exhausting testing of application: impossible
- Two perspectives on coverage
  - Code coverage: all parts of code covered?
  - Input coverage: all (classes of) input data covered?

Basic Code Coverage Criteria

- function coverage: has every function (or subroutine) been called?
- statement coverage: has every statement been executed?
- branch coverage: has every branch of control statements been executed?
- condition coverage: has every boolean clause evaluated to true and false?

Basic Code Coverage Criteria – challenge

```
int f(int x, int y) {
  int z = 0;
  if ((x > 0) && (y > 0)) {
    z = x;
  }
  return z;
}
```

- function coverage:
- statement coverage:
- branch coverage:
- condition coverage:

Basic Code Coverage Criteria – challenge

```
int f(int x, int y) {
  int z = 0;
  if ((x > 0) && (y > 0)) {
    z = x;
  }
  return z;
}
```

- function coverage: `f(0,0)`
- statement coverage:
- branch coverage:
- condition coverage:

Basic Code Coverage Criteria – challenge

```
int f(int x, int y) {
  int z = 0;
  if ((x > 0) && (y > 0)) {
    z = x;
  }
  return z;
}
```

- function coverage: `f(0,0)`
- statement coverage: `f(1,1)`
- branch coverage:
- condition coverage:

Basic Code Coverage Criteria – challenge

```
int f(int x, int y) {
  int z = 0;
  if ((x > 0) && (y > 0)) {
    z = x;
  }
  return z;
}
```

- function coverage: `f(0,0)`
- statement coverage: `f(1,1)`
- branch coverage: `f(1,1)`; `f(1,0)`
- condition coverage:

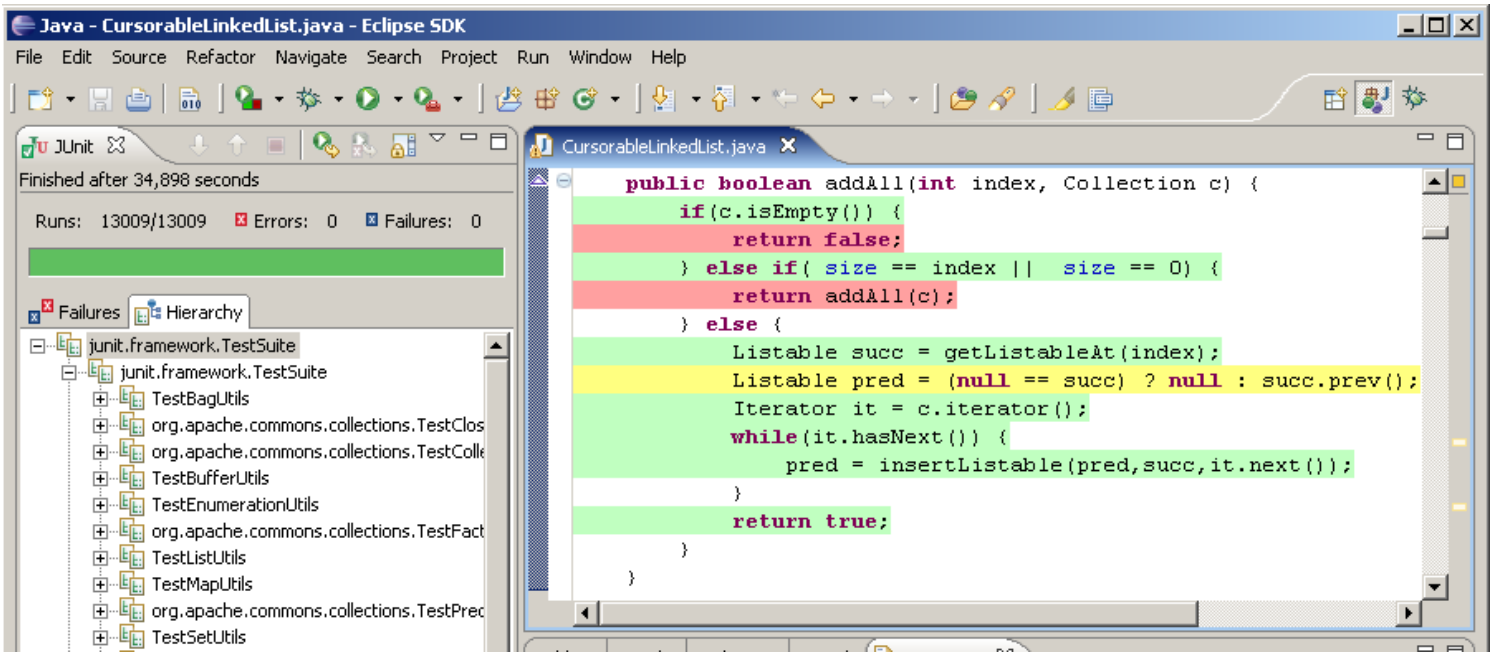
Basic Code Coverage Criteria – challenge

```
int f(int x, int y) {
  int z = 0;
  if ((x > 0) && (y > 0)) {
    z = x;
  }
  return z;
}
```

- function coverage: `f(0,0)`
- statement coverage: `f(1,1)`
- branch coverage: `f(1,1)`; `f(1,0)`
- condition coverage: `f(1,0)`; `f(0,1)`

Tool Support for Basic Coverage Criteria

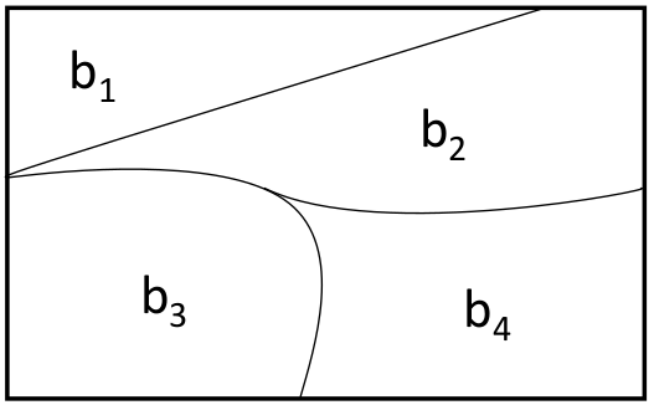
- Easy to automate. Coverage tools available for various languages (e.g., Java: EclEmma)
- Tool executes an instrumented test run
- Gives percentages and visualizes covered and non-covered lines



Input Partitioning

Basic idea:

- divide the set of possible inputs into equivalence classes
- test one input from each class



- assumption: all values in a block are equally useful for testing

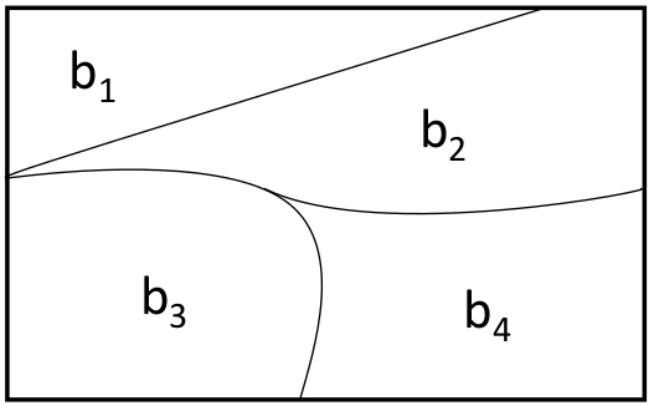
An example

- function: $\texttt{abs}(x)$
- input domain: $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$
- partitions: $\ldots, -3, -2, -1$, 0, $1, 2, 3, \ldots$

Input space partitioning

1. identify the component
   - whole program
   - module
   - class
   - function
2. identify the inputs
   - function/method parameters
   - file contents
   - global variables
   - object state
   - user provided inputs
3. develop an input domain model
   - a way of *describing* the possible inputs
   - partitioned by characteristics

Requirements on partitioning



- partitions must cover the whole input space
- partitions must be disjoint

How to partition?

- semantic partitioning: identify characteristics of the data that correspond to the intended functionality
  - requires domain knowledge, generally not automated
  - advantage: includes semantic information
- boundary-value analysis: make separate classes for boundary values

Semantic partitioning: a classic example

- **Command**: FIND
- **Syntax**: FIND ⟨pattern⟩ ⟨file⟩
- **Function**: The FIND command is used to locate one or more instances of a given pattern in a text file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the number of times the pattern occurs on it.
  The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes ("). To include a quotation mark in the pattern, two quotes in a row ("") must be used.

A classic example

Step 1: analyse the specification

- What is the component?
- What are the parameters?
- What are the characteristics?

A classic example

Step 1: analyse the specification

- What is the component?
- What are the parameters?
- What are the characteristics?

A classic example

Step 1: analyse the specification

- What is the component?
- What are the parameters?
- What are the characteristics?

A classic example

Step 1: analyse the specification

- What is the component?
- What are the parameters?
- What are the characteristics?

A classic example

Step 1: analyse the specification

- What is the component?
  - the FIND program
- What are the parameters?
- What are the characteristics?

A classic example

Step 1: analyse the specification

- What is the component?
  - the FIND program
- What are the parameters?
  - pattern
  - input file
- What are the characteristics?

A classic example

Step 1: analyse the specification

- What is the component?
  - the FIND program
- What are the parameters?
  - pattern
  - input file
- What are the characteristics?
  - pattern size
  - quoting
  - embedded quotes
  - number of pattern occurrences in file
  - number of occurrences on a particular line

A classic example

Step 2: partition the input space

- select one region per characteristic at a time
- combine into test frames (test case plans)
- example:
  - pattern size: empty
  - quoting: pattern is quoted
  - embedded blanks: several embedded blanks
  - embedded quotes: no embedded quotes
  - file name: good file name
  - number of occurrences of pattern in file: none

A classic example

Step 2: partition the input space

- select one region per characteristic at a time
- combine into test frames (test case plans)
- example:
  - pattern size: empty
  - quoting: pattern is quoted
  - embedded blanks: several embedded blanks
  - embedded quotes: no embedded quotes
  - file name: good file name
  - number of occurrences of pattern in file: none

A classic example

Step 2: partition the input space

- select one region per characteristic at a time
- combine into test frames (test case plans)
- example:
  - pattern size: empty
  - quoting: pattern is quoted
  - embedded blanks: several embedded blanks
  - embedded quotes: no embedded quotes
  - file name: good file name
  - number of occurrences of pattern in file: none

A classic example

Step 2: partition the input space

- select one region per characteristic at a time
- combine into test frames (test case plans)
- example:
  - pattern size: empty
  - quoting: pattern is quoted
  - embedded blanks: several embedded blanks
  - embedded quotes: no embedded quotes
  - file name: good file name
  - number of occurrences of pattern in file: none

A classic example

Step 3: identify constraints among the characteristics and blocks

- pattern size:
  - empty [property Empty]
  - single character [property NonEmpty]
  - many characters [property NonEmpty]
  - longer than any line in the file [property NonEmpty]
- quoting:
  - pattern is quoted [property Quoted]
  - pattern is not quoted [if NonEmpty]
  - pattern is improperly quoted [if NonEmpty]

A classic example

Step 4: create tests

- select values that satisfy the selected blocks for each frame
- eliminate tests that cover redundant scenarios

Step 5: run or automate your test cases!

Combination strategies

- All Combinations Coverage (ACoC): what the name says
- Each Choice Coverage (ECC): a value from each block for each characteristic must be used in at least one test
- Pair-Wise Coverage (PWC): a value from each block for each characteristic must be combined with a value from every other block for every other characteristic
- T-Wise Coverage (TWC): like PWC, but with groups of more than two characteristics combined

Combination strategies

Consider a system with the following parameters and values:

- parameter A has values A1 and A2
- parameter B has values B1 and B2
- parameter C has values C1, C2 and C3

- All Combinations Coverage: 12 tests
- Each Choice Coverage: 3 tests
- Pair-Wise Coverage: 6 tests
- T-Wise Coverage: 12 tests

Why pairwise?

- many faults are caused by the interaction between two parameters
- it's simply not practical to cover all interactions between parameters

Boundary-value analysis

- motivation: programmers often make mistakes in values at or near the boundaries
- for example: $x < 0$ instead of $x <= 0$
- solution: always include tests for values at or near the boundaries
  - example: `itemCode` $\in \{99, \ldots, 999\}$
  - include tests for: $98, 99, 100, 998, 999, 1000$

Testability

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.

Typical testability flaws and their remedies

- Constructor Does Real Work
- Digging Into Collaborators
- Brittle Global State & Singletons
- Class Does Too Much

Typical testability flaws and their remedies

- Constructor Does Real Work
- Digging Into Collaborators
- Brittle Global State & Singletons
- Class Does Too Much

Before: Hard to Test

```
// Basic new operators called directly in
//   the class' constructor. (Forever
//   preventing a seam to create different
//   kitchen and bedroom collaborators).
class House {
  Kitchen kitchen = new Kitchen();
  Bedroom bedroom;

  House() {
    bedroom = new Bedroom();
  }

  // ...
}
```

```
// An attempted test that becomes pretty hard

class HouseTest extends TestCase {
  public void testThisIsReallyHard() {
    House house = new House();
    // Darn! I'm stuck with those Kitchen and
    //   Bedroom objects created in the
    //   constructor.

    // ...
  }
}
```

After: Testable and Flexible Design

```
class House {
  Kitchen kitchen;
  Bedroom bedroom;

  // Have Guice create the objects
  //   and pass them in
  @Inject
  House(Kitchen k, Bedroom b) {
    kitchen = k;
    bedroom = b;
  }
  // ...
}
```

```
// New and Improved is trivially testable, with any
//   test-double objects as collaborators.

class HouseTest extends TestCase {
  public void testThisIsEasyAndFlexible() {
    Kitchen dummyKitchen = new DummyKitchen();
    Bedroom dummyBedroom = new DummyBedroom();

    House house =
        new House(dummyKitchen, dummyBedroom);

    // Awesome, I can use test doubles that
    //   are lighter weight.

    // ...
  }
}
```

Typical testability flaws and their remedies

- Constructor Does Real Work
- Digging Into Collaborators
- Brittle Global State & Singletons
- Class Does Too Much

Before: Hard to Test

```
// This is a service object that works with a value
// object (the User and amount).

class SalesTaxCalculator {
  TaxTable taxTable;

  SalesTaxCalculator(TaxTable taxTable) {
    this.taxTable = taxTable;
  }

  float computeSalesTax(User user, Invoice invoice) {
    // note that "user" is never used directly

    Address address = user.getAddress();
    float amount = invoice.getSubTotal();
    return amount * taxTable.getTaxRate(address);
  }
}
```

```
// Testing exposes the problem by the amount
// of work necessary to build the object graph, and // test
the small behavior you are interested in.

class SalesTaxCalculatorTest extends TestCase {

  SalesTaxCalculator calc =
      new SalesTaxCalculator(new TaxTable());
  // So much work wiring together all the
  // objects needed
  Address address =
      new Address("1600 Amphitheatre Parkway...");
  User user = new User(address);
  Invoice invoice =
      new Invoice(1, new ProductX(95.00));
  // ...
  assertEquals(
      0.09, calc.computeSalesTax(user, invoice), 0.05);
}
```

After: Testable and Flexible Design

```
// Reworked, it only asks for the specific
// objects that it needs to collaborate with.

class SalesTaxCalculator {
  TaxTable taxTable;

  SalesTaxCalculator(TaxTable taxTable) {
    this.taxTable = taxTable;
  }

  // Note that we no longer use User, nor do we
  // dig inside the address. (Note: We would
  // use a Money, BigDecimal, etc. in reality).
  float computeSalesTax(Address address,
        float amount) {
    return amount * taxTable.getTaxRate(address);
  }
}
```

```
// The new API is clearer in what collaborators
// it needs.

class SalesTaxCalculatorTest extends TestCase {

  SalesTaxCalculator calc =
      new SalesTaxCalculator(new TaxTable());
    // Only wire together the objects that
    // are needed

    Address address =
        new Address("1600 Amphitheatre Parkway...");
    // ...
    assertEquals(
        0.09,
        calc.computeSalesTax(address, 95.00),
        0.05);
  }

}
```

Typical testability flaws and their remedies

- Constructor Does Real Work
- Digging Into Collaborators
- Brittle Global State & Singletons
- Class Does Too Much

Before: Hard to Test

```
// Awkward and brittle tests, obfuscated by Flags'
//   boilerplate setup and cleanup.

class NetworkLoadCalculatorTest extends TestCase {

  public void testMaximumAlgorithmReturnsHighestLoad() {
    Flags.disableStateCheckingForTest();

ConfigFlags.FLAG_loadAlgorithm.setForTest("maximum");

    NetworkLoadCalculator calc =
        new NetworkLoadCalculator();
    calc.setLoadSources(10, 5, 0);
    assertEquals(10, calc.calculateTotalLoad());

    // Don't forget to clean up after
    // yourself following every test
    // (this could go in tearDown).
    ConfigFlags.FLAG_loadAlgorithm.resetForTest();
    Flags.enableStateCheckingForTest();
  }
}

// Elsewhere... the NetworkLoadCalculator's methods
class NetworkLoadCalculator {

  // ...

  int calculateTotalLoad() {
    // ... somewhere read the flags' global state
    String algorithm =
        ConfigFlags.FLAG_loadAlgorithm.get();
    // ...
  }
}
```

After: Testable and Flexible Design

```
// The new test is easier to understand and less
//   likely to break other tests.

class NetworkLoadCalculatorTest {

  public void testMaximumAlgorithmReturnsHighestLoad()
{
    NetworkLoadCalculator calc =
        new NetworkLoadCalculator("maximum");
    calc.setLoadSources(10, 5, 0);
    assertEquals(10, calc.calculateTotalLoad());
  }
}

  // Replace the global dependency on the Flags with
  // the Guice FlagBinder that gives named annotations
  // to flags automatically. String Flag_xxx is bound
  // to String.class annotated with @Named("xxx").
  // (All flag types are bound, not just String.)
  // In your Module:

     new FlagBinder(binder()).bind(ConfigFlags.class);

// Replace all the old calls where you read Flags with
//   injected values.
class NetworkLoadCalculator {

  String loadAlgorithm;

  // Pass in flag value into the constructor
  NetworkLoadCalculator(
      @Named("loadAlgorithm") String loadAlgorithm) {
    //... use the String value however you want,
    //and for tests, construct different
    //NetworkLoadCalculator objects with other values.
    this.loadAlgorithm = loadAlgorithm;
  }

  // ...
}
```

Typical testability flaws and their remedies

- Constructor Does Real Work
- Digging Into Collaborators
- Brittle Global State & Singletons
- Class Does Too Much

Continuous Integration

Motivation

Have you ever:

Motivation

Have you ever:

- Broken everyone's code with a push?

Motivation

Motivation

Have you ever:

- Broken everyone's code with a push?
  - Due to laxness?

Motivation

Motivation

Have you ever:

- Broken everyone's code with a push?
  - Due to laxness?
  - Due to cached files?
  - Due to different versions of the compiler or operating system?
  - Broken something deeper and failed to notice it quickly?

Motivation

Have you ever:

- Broken everyone's code with a push?
  - Due to laxness?
  - Due to cached files?
  - Due to different versions of the compiler or operating system?
  - Broken something deeper and failed to notice it quickly?
- Decided to postpone integrating a branch because it takes so much time?

Motivation

Have you ever:

- Broken everyone's code with a push?
  - Due to laxness?
  - Due to cached files?
  - Due to different versions of the compiler or operating system?
  - Broken something deeper and failed to notice it quickly?
- Decided to postpone integrating a branch because it takes so much time?
- Spent a lot of time getting a branch working with the rest of the code again?

Motivation

Continuous integration

- merge in small changes frequently
- have a dedicated server to automate building and testing
- receive quick feedback on a broken build
- fix broken builds rather than pushing further changes

Continuous integration

- merge in small changes frequently
- have a dedicated server to automate building and testing
- receive quick feedback on a broken build
- fix broken builds rather than pushing further changes

We have bought for you: GitHub Actions
Instruction lecture from our CTO Joren will be made available!

Motivation

Have you ever:

- Broken everyone's code with a push?
  - Due to laxness?
  - Due to cached files?
  - Due to different versions of the compiler or operating system?
  - Broken something deeper and failed to notice it quickly?
- Decided to postpone integrating a branch because it takes so much time?
- Spent a lot of time getting a branch working with the rest of the code again?
- Spent ages at release time figuring out dependencies?

Motivation

Have you ever:

- Broken everyone's code with a push?
  - Due to laxness?
  - Due to cached files?
  - Due to different versions of the compiler or operating system?
  - Broken something deeper and failed to notice it quickly?
- Decided to postpone integrating a branch because it takes so much time?
- Spent a lot of time getting a branch working with the rest of the code again?
- Spent ages at release time figuring out dependencies?

Motivation

Have you ever:

- Broken everyone's code with a push?
  - Due to laxness?
  - Due to cached files?
  - Due to different versions of the compiler or operating system?
  - Broken something deeper and failed to notice it quickly?
- Decided to postpone integrating a branch because it takes so much time?
- Spent a lot of time getting a branch working with the rest of the code again?
- Spent ages at release time figuring out dependencies?

Motivation

Have you ever:

- Broken everyone's code with a push?
  - Due to laxness?
  - Due to cached files?
  - Due to different versions of the compiler or operating system?
  - Broken something deeper and failed to notice it quickly?
- Decided to postpone integrating a branch because it takes so much time?
- Spent a lot of time getting a branch working with the rest of the code again?
- Spent ages at release time figuring out dependencies?

Motivation

Have you ever:

- Broken everyone's code with a push?
  - Due to laxness?
  - Due to cached files?
  - Due to different versions of the compiler or operating system?
  - Broken something deeper and failed to notice it quickly?
- Decided to postpone integrating a branch because it takes so much time?
- Spent a lot of time getting a branch working with the rest of the code again?
- Spent ages at release time figuring out dependencies?

Motivation

Have you ever:

- Broken everyone's code with a push?
  - Due to laxness?
  - Due to cached files?
  - Due to different versions of the compiler or operating system?
  - Broken something deeper and failed to notice it quickly?
- Decided to postpone integrating a branch because it takes so much time?
- Spent a lot of time getting a branch working with the rest of the code again?
- Spent ages at release time figuring out dependencies?

Motivation

Have you ever:

- Broken everyone's code with a push?
  - Due to laxness?
  - Due to cached files?
  - Due to different versions of the compiler or operating system?
  - Broken something deeper and failed to notice it quickly?
- Decided to postpone integrating a branch because it takes so much time?
- Spent a lot of time getting a branch working with the rest of the code again?
- Spent ages at release time figuring out dependencies?

Motivation

Have you ever:

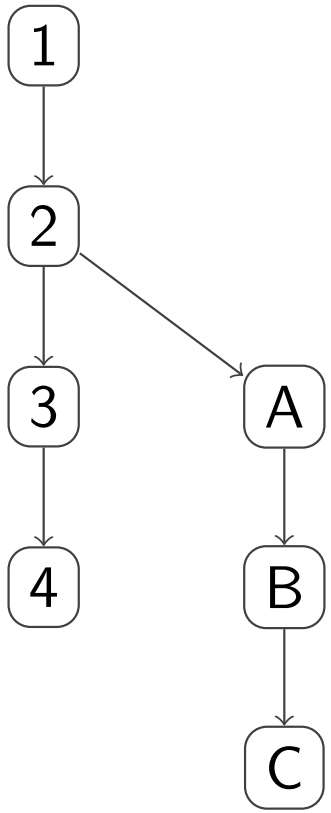
- Broken everyone's code with a push?
  - Due to laxness?
  - Due to cached files?
  - Due to different versions of the compiler or operating system?
  - Broken something deeper and failed to notice it quickly?
- Decided to postpone integrating a branch because it takes so much time?
- Spent a lot of time getting a branch working with the rest of the code again?
- Spent ages at release time figuring out dependencies?

Branches and CI

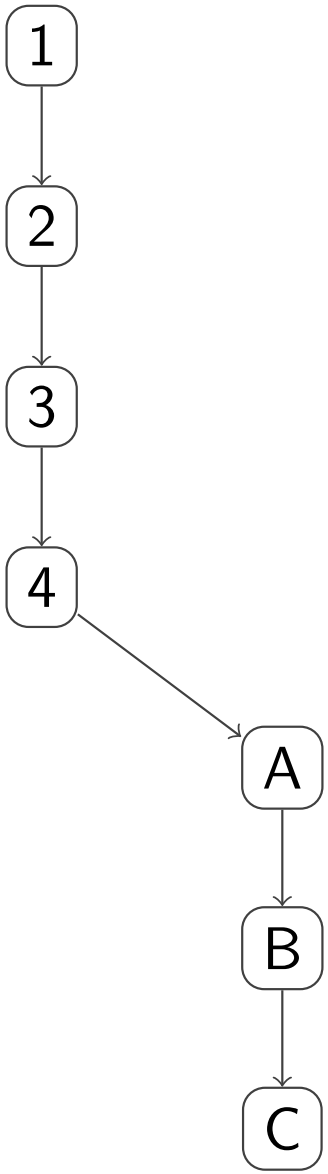
Branches and CI

- GitHub Actions works on branches!
- Keep up-to-date using `git rebase`

Git rebase



Git rebase



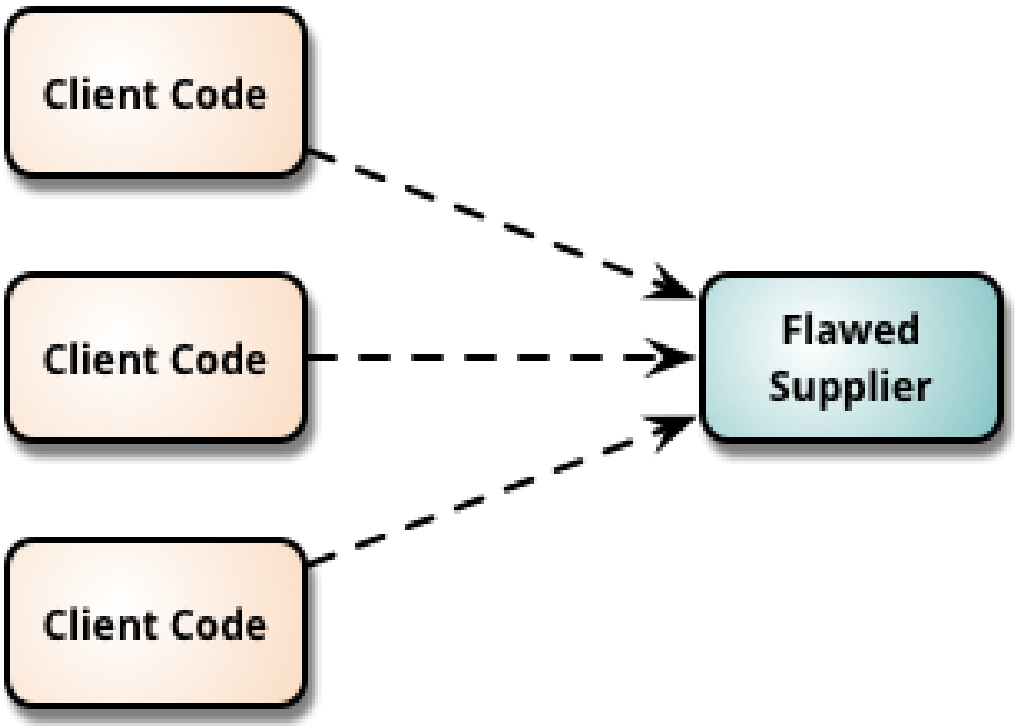
Branches and CI

- Travis CI works on branches!
- Use `.travis.yml` to indicate which branches to do (only or except).
- Keep up-to-date using `git rebase`

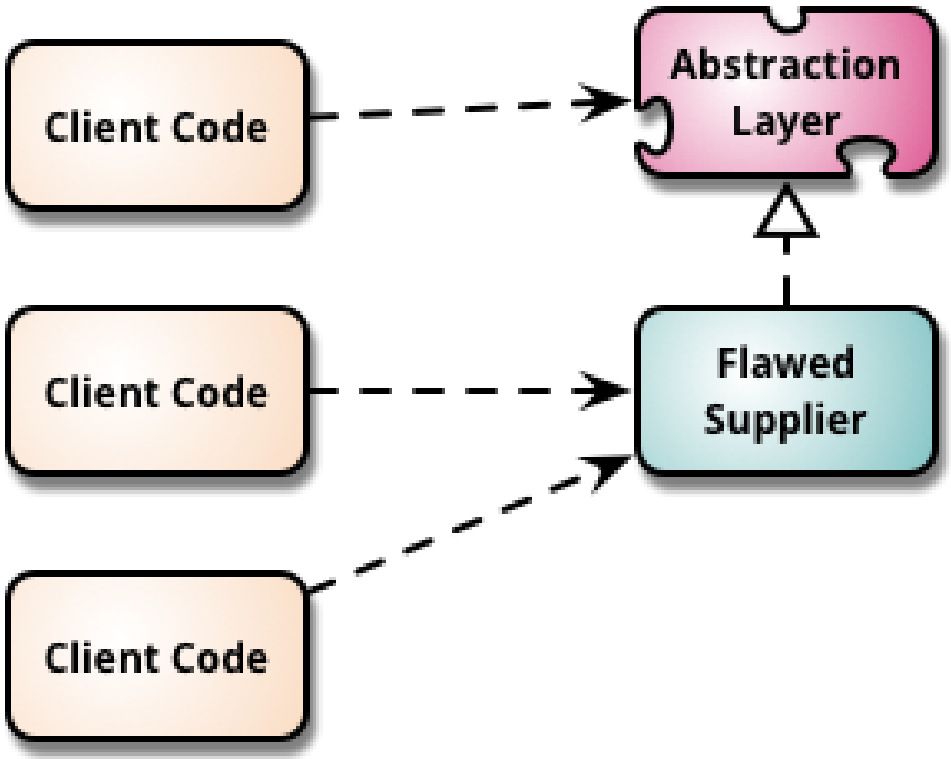
Branches and CI

- Travis CI works on branches!
- Use `.travis.yml` to indicate which branches to do (only or except).
- Keep up-to-date using `git rebase`
- Even in branches, keep everything working.

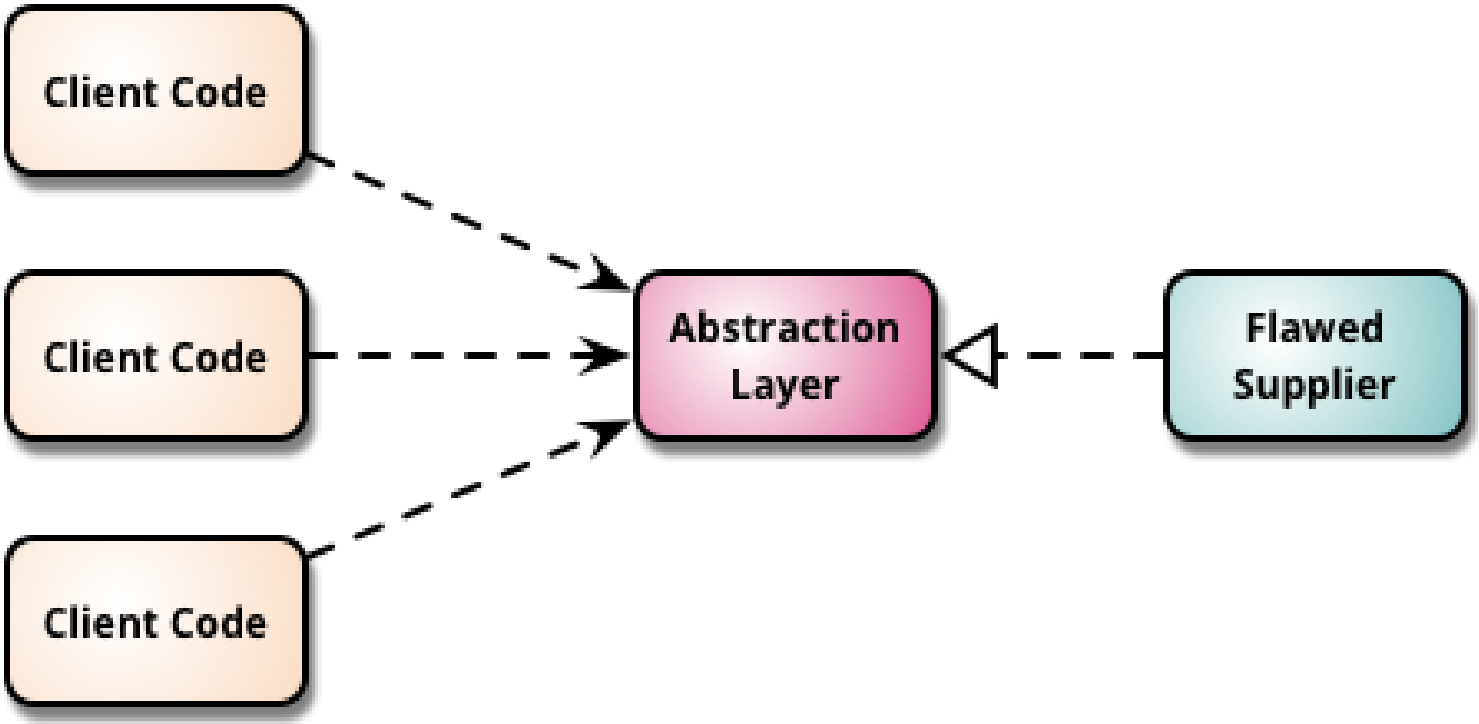
Branch by Abstraction



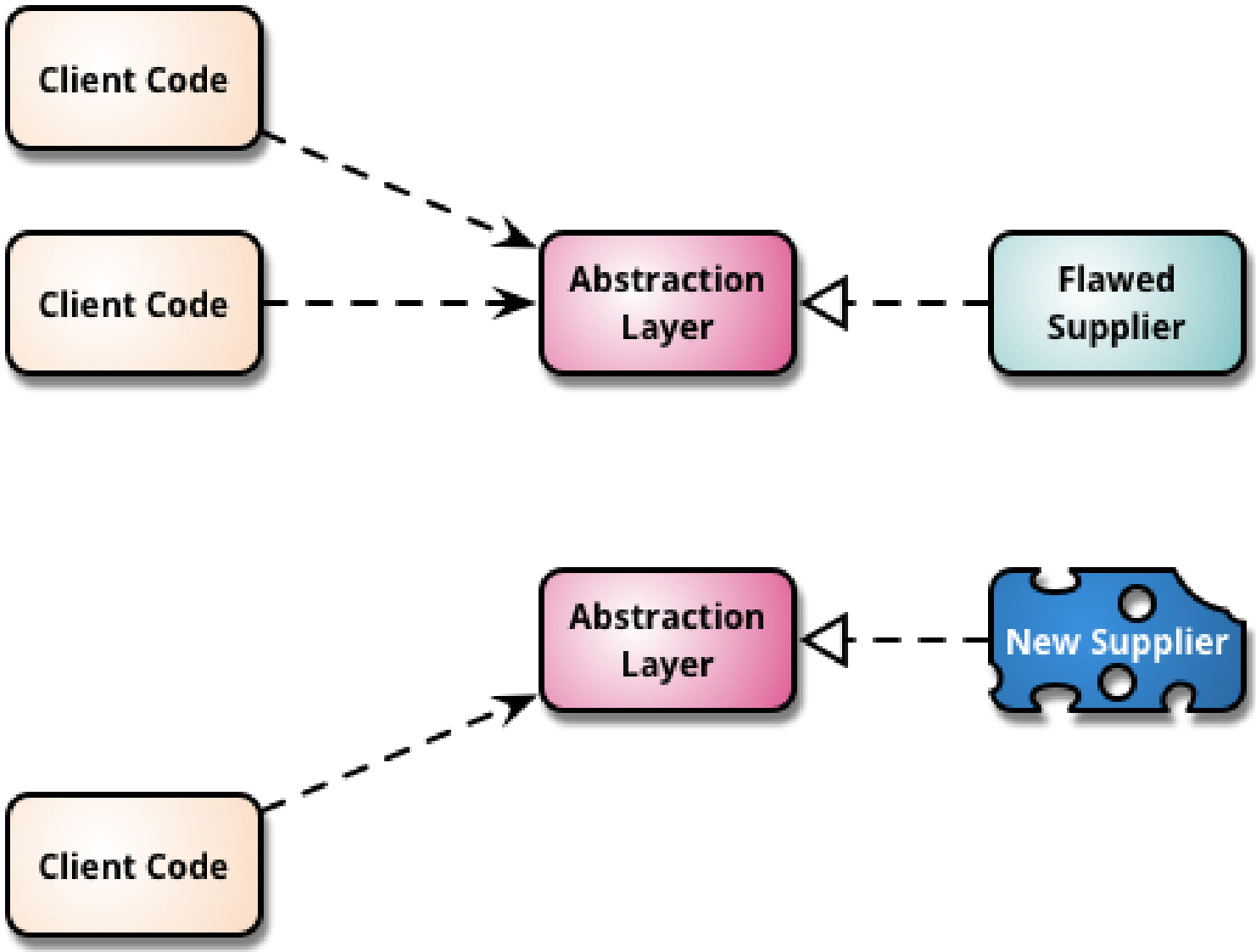
Branch by Abstraction



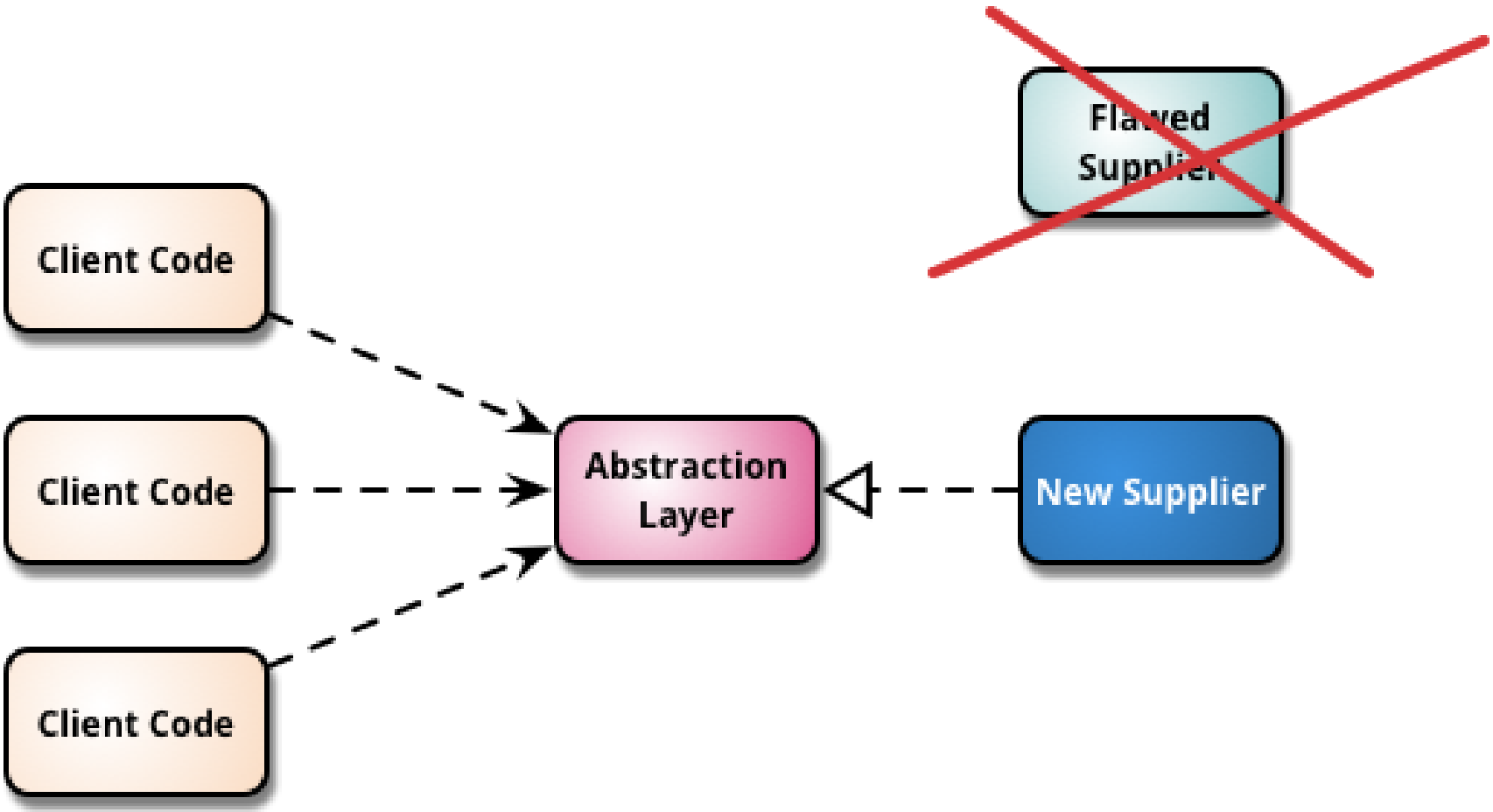
Branch by Abstraction



Branch by Abstraction



Branch by Abstraction



Best principles

- Push regularly.
- Integrate quickly.
- Create a comprehensive automated test suite.
- Keep the build and test process short.
- Still test locally before pushing!
- Don't push further commits on a broken build.
- Discuss rules on responsibility.
- Keep everything in version control (except binaries and passwords).

Continuous delivery

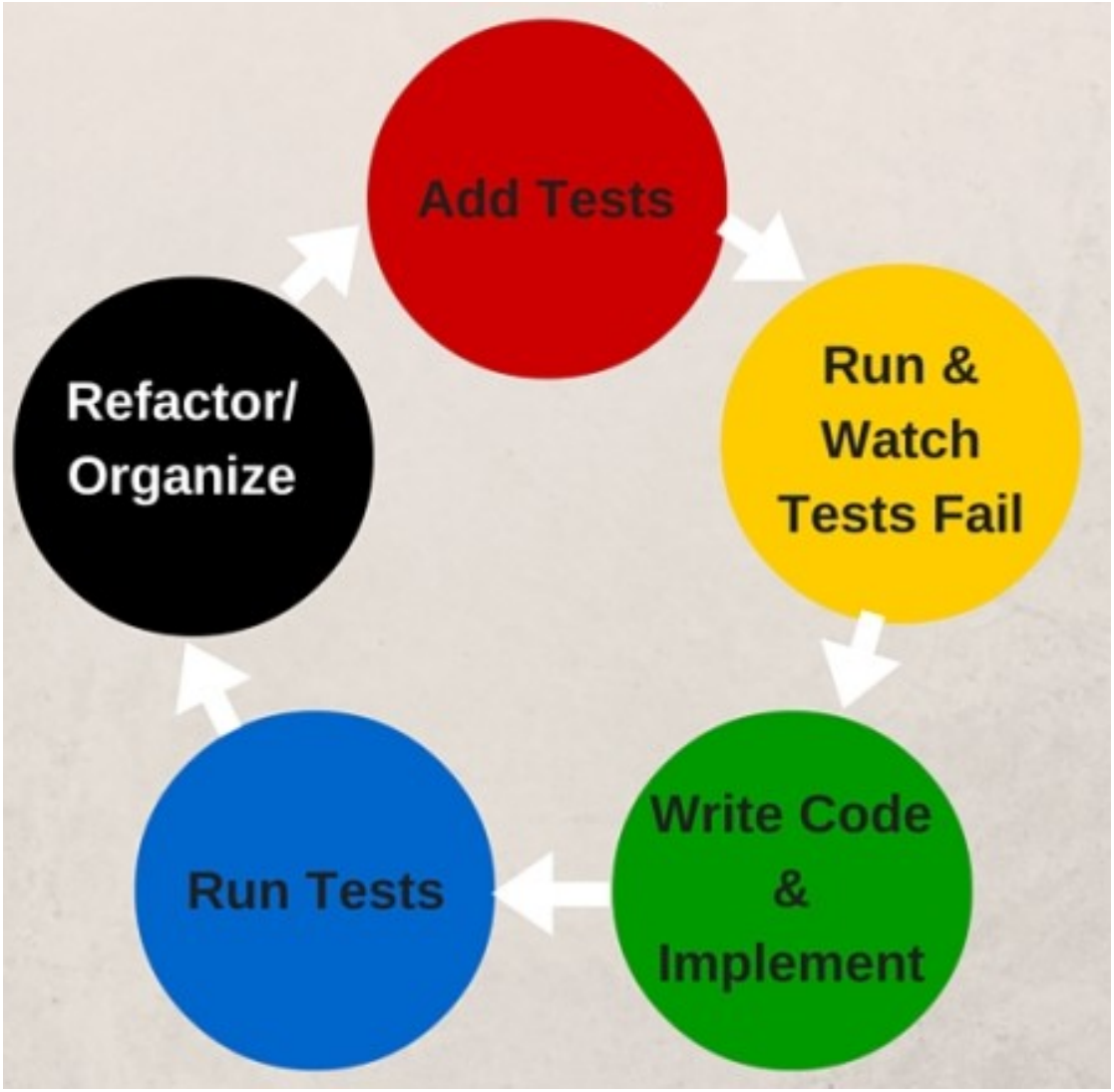
- You already have automatic builds... why not automatic deployment?
- Alternatively: semi-automatic deployment with humans pushing just a few buttons.
- Benefits:
  - low cycle time due to ease of releases
  - high realiability through removing human error
  - easy to roll back smaller changes which cause problems

Concrete benefits of continuous integration

- detecting defects as early as possible
- reducing problems caused by configuration or environment
- every push leads to potentially releasable software
- makes (semi-)automatic deployment easy
- deployment flexibility

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)

How effective are my tests? Mutation testing!

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!

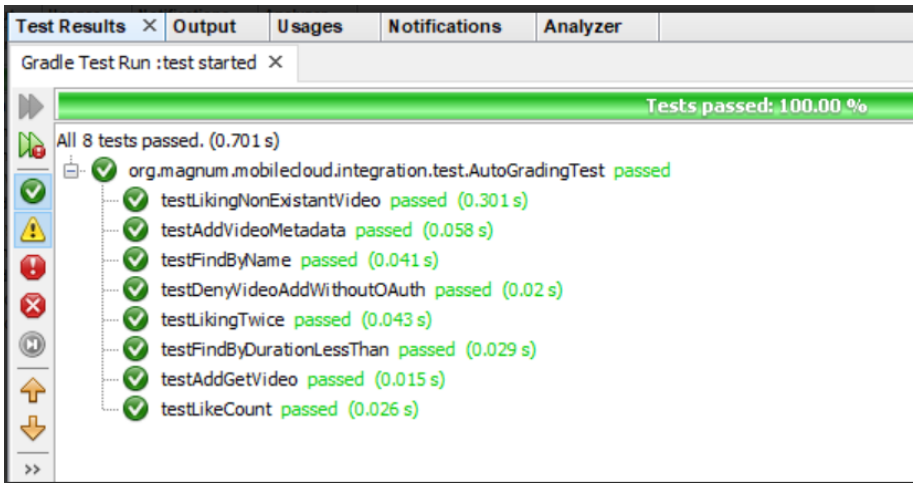
Tasks in testing:

- test design
- test automation
- test execution
- test evaluation

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

Testing: a destructive activity?

- yes: the goal is to uncover flaws
- no: systematically make the program better and better



The fun part: turning "red" tests "green"!

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.

Reminder: Assignment 1

Reflection on Code Quality

Tell us, in 300 - 600 words, what you (personally!) have done to improve the code quality in your product.

1. List design principles and software patterns you (personally!) used and explain how you applied them (with examples from your code).
2. Relate your actions to coupling and cohesion.
3. Explain how the quality of your code helped increase testability. (Hint: Do not discuss how you *tested* your code, but what made your code *testable*.)
4. Discuss premature generalization: Was this ever an issue for you and if yes, how?

(Or if you have not *improved* it because it was built from scratch and/or already very good, describe what you have done to maintain the high quality.)

Sources

Guide to writing testable code:
https://p.rogram.me/resource/attachment/Guide-Writing_Testable_Code.pdf