Principles of Software Construction: Objects, Design, and Concurrency

Testing

toad

Spring 2014

Christian Kästner Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

Learning Goals

- Understand the nature of testing
- Select test cases
- Write practical unit tests with JUnit
- Automate test execution
- Write tests with stubs
- Decide when to stop testing, interpret coverage criteria

Formal Verification

- Proving the correctness of an implementation with respect to a formal specification, using formal methods of mathematics.
- Formally prove that all possible executions of an implementation fulfill the specification
- Manual effort; partial automation; not automatically decidable

Testing

- Executing the program with selected inputs in a controlled environment

- Goals:
  - Reveal bugs (main goal)
  - Assess quality (hard to quantify)
  - Clarify the specification, documentation
  - Verify contracts

"Testing shows the presence, not the absence of bugs

Edsger W. Dijkstra 1969

What to test?

- Functional correctness of a method (e.g., computations, contracts)
- Functional correctness of a class (e.g., class invariants)
- Behavior of a class in a subsystem/multiple subsystems/the entire system
- Behavior when interacting with the world
  - Interacting with files, networks, sensors, …
  - Erroneous states
  - Nondeterminism, Parallelism
  - Interaction with users

Testing Decisions

Who tests?

- Developers
- Other Developers
- Separate Quality Assurance Team
- Customers

When to test?

- Before development
- During development
- After milestones
- Before shipping

(More in 15-313)

From problem to idea to correct program

- "While the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962."
  — Donald E. Knuth, Stanford

- "Given ample time, only about 10% of professional programmers were able to get this small program right"
  — Jon Bentley, AT&T Bell Labs

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 / assertations?
- Effort, Costs?
- Reproducable?

Automate Testing

- Execute a program with specific inputs, check output for expected values
- Easier to test small pieces than testing user interactions
- Set up testing infrastructure
- Execute tests regularly

Example

```
/**
 * computes the sum of the first len values of the array
 *
 * @param array array of integers of at least length len
 * @param len number of elements to sum up
 * @return sum of the array values
 */
int total(int array[], int len);
```

Black box testing

Example

```
/**
 * computes the sum of the first len values of the array
 *
 * @param array array of integers of at least length len
 * @param len number of elements to sum up
 * @return sum of the array values
 */
int total(int array[], int len);
```

- Test empty array
- Test array of length 1 and 2
- Test negative numbers
- Test invalid length (negative or longer than array.length)
- Test null as array
- Test with a very long array

Black box testing

Selecting Test Cases: Common Strategies

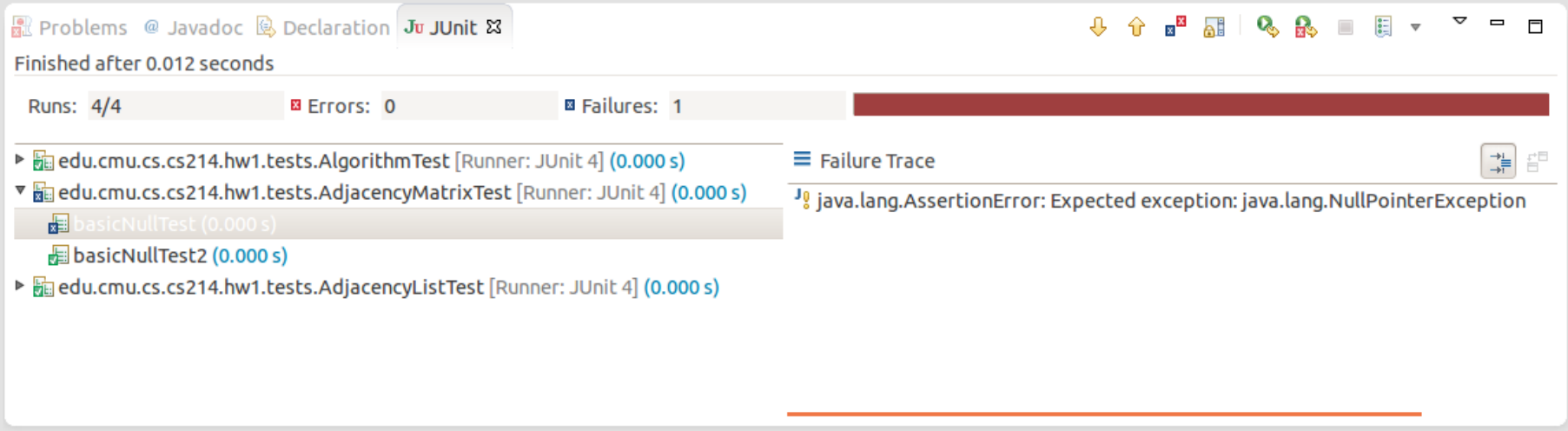
- Read specification

- Write tests for representative case
  - Small instances are usually sufficient
- Write tests for invalid cases
- Write tests to check boundary conditions
- Are there difficult cases? (error guessing)
  - Stress tests? Complex algorithms?
- Think like a user, not like a programmer
  - The tester's goal is to find bugs!

- Specification covered?
- Feel confident? Time/money left?

JUnit

- Popular unit-testing framework for Java
- Easy to use
- Tool support available
- Can be used as design mechanism



JUnit

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
        @Test
        public void testSanityTest(){
                Graph g1 = new AdjacencyListGraph(10)
                Vertex s1 = new Vertex("A");
                Vertex s2 = new Vertex("B");
                assertEquals(true, g1.addVertex(s1));
                assertEquals(true, g1.addVertex(s2));
                assertEquals(true, g1.addEdge(s1, s2));
                assertEquals(s2, g1.getNeighbors(s1)[0]);
        }

        @Test
        public void test….

        private int helperMethod…
}
```

Set up tests

Check expected results

Unit Tests

- Unit tests for small units: functions, classes, subsystems
  - Smallest testable part of a system
  - Test parts before assembling them
  - Intended to catch local bugs
- Typically written by developers
- Many small, fast-running, independent tests
- Little dependencies on other system parts or environment
- Insufficient but a good starting point, extra benefits:
  - Documentation (executable specification)
  - Design mechanism (design for testability)

assert, Assert

- assert is a native Java statement throwing an AssertionError exception when failing
  - **assert** expression: "Error Message";
- org.junit.Assert is a library that provides many more specific methods
  - static void **assertTrue**(java.lang.String message, boolean condition)
    // Asserts that a condition is true.
  - static void **assertEquals**(java.lang.String message, long expected, long actual);
    // Asserts that two longs are equal.
  - static void **assertEquals**(double expected, double actual, double delta);
    // Asserts that two doubles are equal to within a positive delta
  - static void **assertNotNull**(java.lang.Object object)
    // Asserts that an object isn't null.
  - static void **fail**(java.lang.String message)
    //Fails a test with the given message.

JUnit Conventions

- TestCase collects multiple tests (in one class)
- TestSuite collects test cases (typically package)
- Tests should run fast
- Tests should be independent

- Tests are methods without parameter and return value
- AssertError signals failed test (unchecked exception)

- Test Runner knows how to run JUnit tests
  - (uses reflection to find all methods with @Test annotat.)

Common Setup

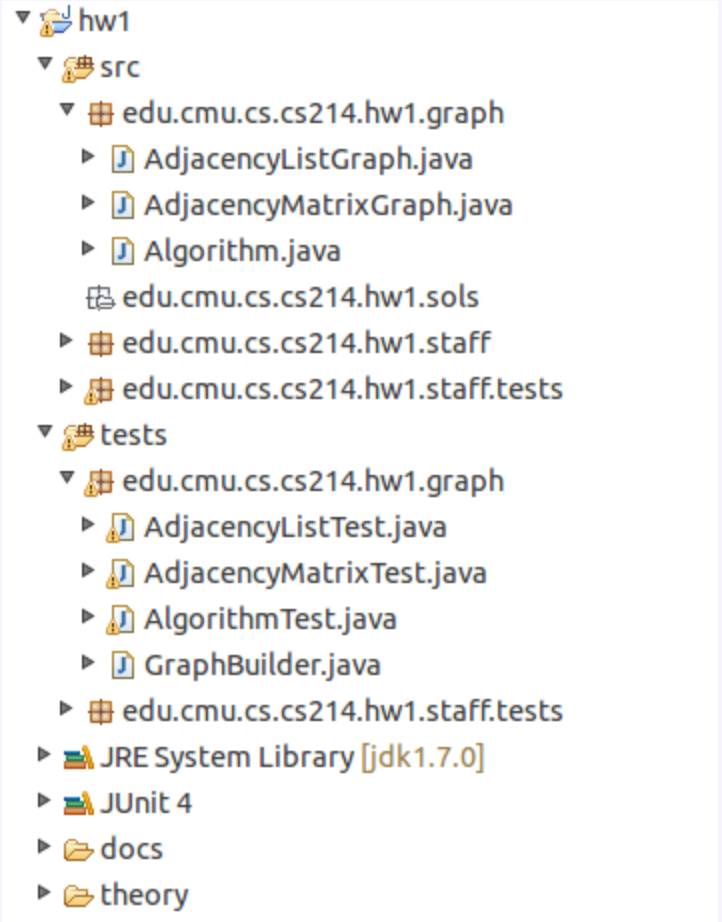
```
import org.junit.*;
import org.junit.Before;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
        Graph g;

        @Before
        public void setUp() throws Exception {
                graph = createTestGraph();

        @Test
        public void testSanityTest(){
                Vertex s1 = new Vertex("A");
                Vertex s2 = new Vertex("B");
                assertEquals(3, g.getDistance(s1, s2));
        }
```

Checking for presence of an exception

```
import org.junit.*;
import static org.junit.Assert.fail;

public class Tests {

	@Test
	public void testSanityTest(){
		try {
			openNonexistingFile();
			fail("Expected exception");
		} catch(IOException e) { }
	}

	@Test(expected = IOException.class)
	public void testSanityTestAlternative() {
		openNonexistingFile();
	}
}
```

Test organization

- Conventions (not requirements)
- Have a test class ATest for each class A
- Have a source directory and a test directory
  - Store ATest and A in the same package
  - Tests can access members with default (package) visibility
- Alternatively store exceptions in the source directory but in a separate package



Exercise (on paper!)

- Test a priority queue for Strings

```
public interface Queue {
        void add(String s);
        String getFirstAlphabetically();
}
```

- Write various kinds of test cases

Testable Code

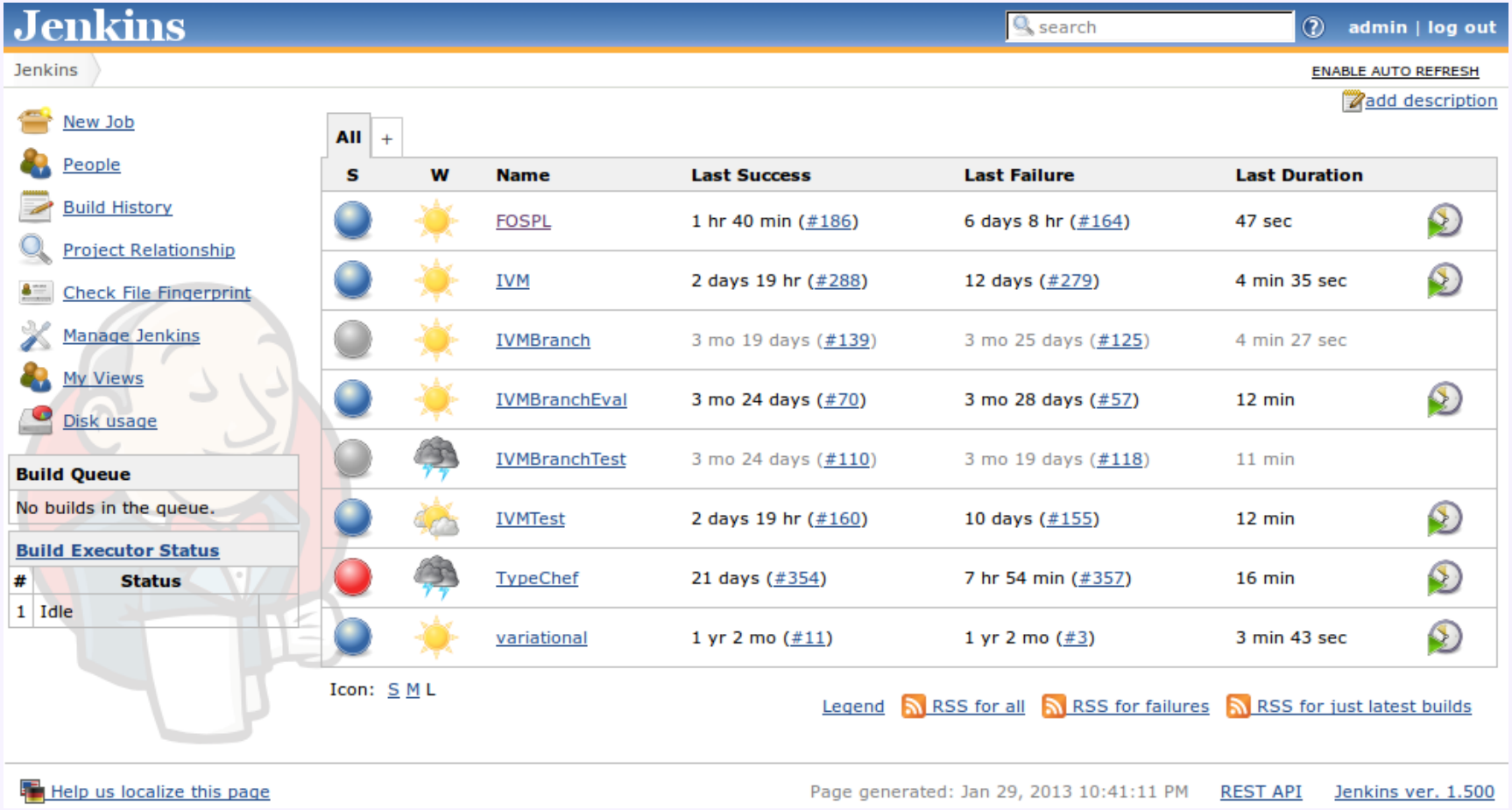
- Think about testing when writing code
- Unit testing encourages to write testable code
- Separate parts of the code to make them independently testable
- Abstract functionality behind interface, make it replaceable

- Test-Driven Development
  - A design and development method in which you write tests before you write the code!

Run tests frequently

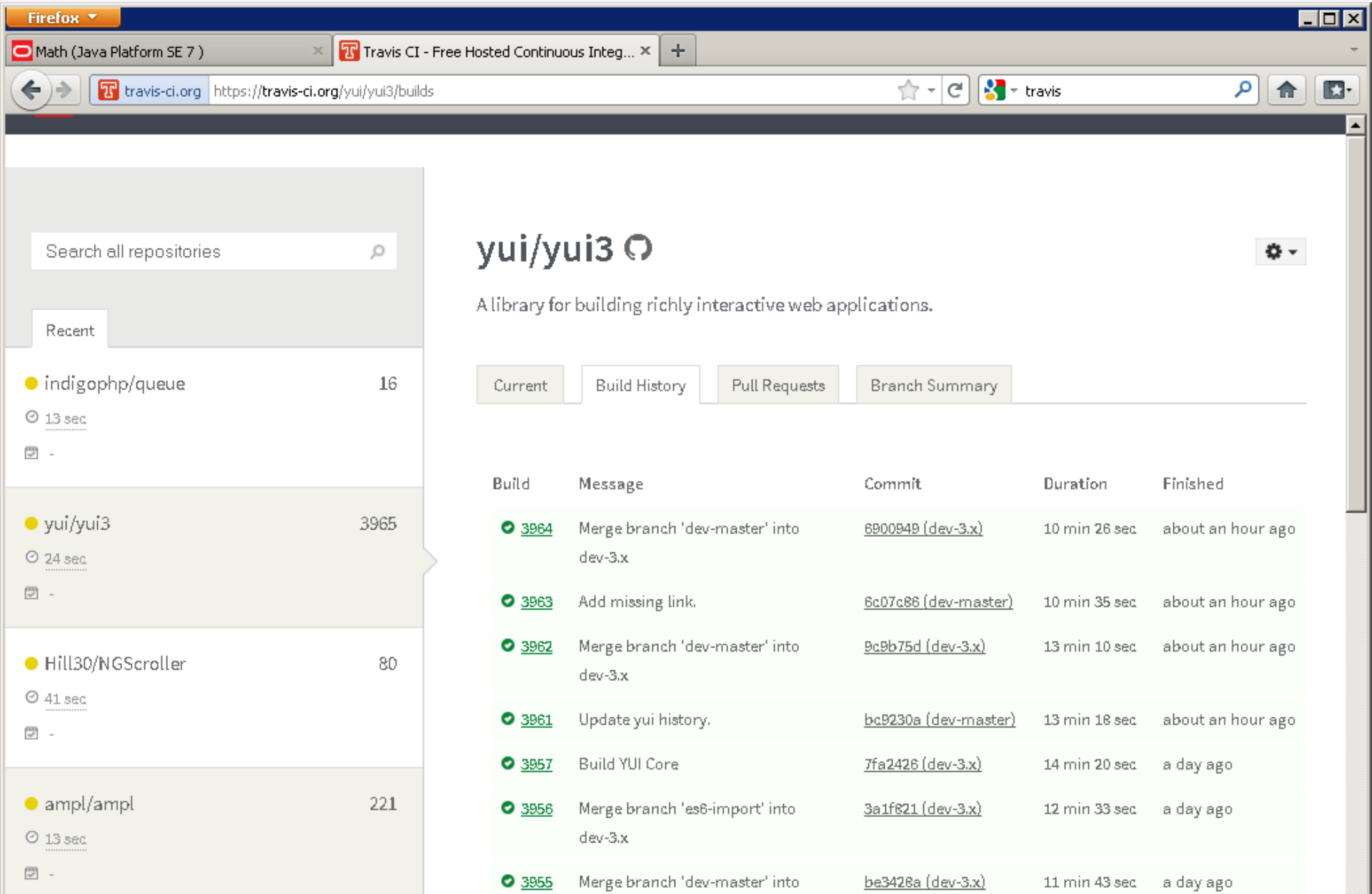
- You should only commit code that is passing all tests
- Run tests before every commit
- Run tests before trying to understand other developers' code
- If entire test suite becomes too large and slow for rapid feedback, run local tests ("smoke tests", e.g. all tests in package) frequently, run all tests nightly
  - Medium sized projects easily have 1000s of test cases and run for minutes
- Continuous integration servers help to scale testing

Continuous Integration



See also travis-ci.org

Travis CI

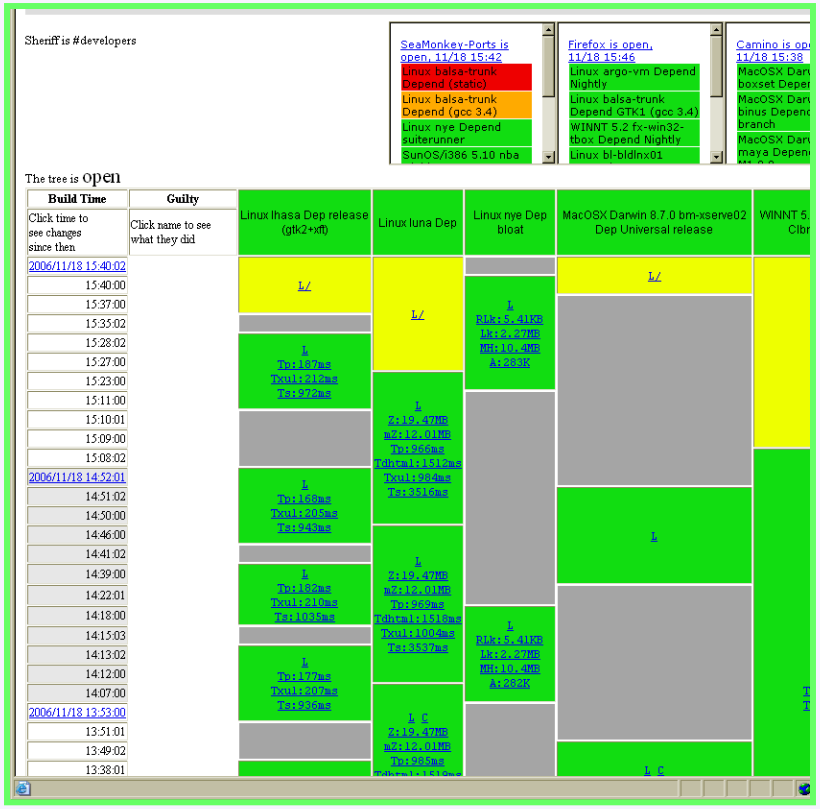

Firefox
Math (Java Platform SE 7)
Travis CI - Free Hosted Continuous Integ...
travis-ci.org
https://travis-ci.org/yui/yui3/builds
travis
Search all repositories
Recent
indigophp/queue 16
13 sec
-
yui/yui3 3965
24 sec
-
Hill30/NGScroller 80
41 sec
-
ampl/ampl 221
13 sec
-
yui/yui3
A library for building richly interactive web applications.
Current
Build History
Pull Requests
Branch Summary
Build
Message
Commit
Duration
Finished
3964 Merge branch 'dev-master' into dev-3.x 6900949 (dev-3.x) 10 min 26 sec about an hour ago
3963 Add missing link. 6c07c86 (dev-master) 10 min 35 sec about an hour ago
3962 Merge branch 'dev-master' into dev-3.x 9c9b75d (dev-3.x) 13 min 10 sec about an hour ago
3961 Update yui history. bc9230a (dev-master) 13 min 18 sec about an hour ago
3957 Build YUI Core 7fa2426 (dev-3.x) 14 min 20 sec a day ago
3956 Merge branch 'es6-import' into dev-3.x 3a1f821 (dev-3.x) 12 min 33 sec a day ago
3955 Merge branch 'dev-master' into be3428a (dev-3.x) 11 min 43 sec a day ago

Automating Test Execution

```
ckaestne@kastner-desktop:~/work/TypeChef/TypeChef$ sbt "project FeatureExprLib" test
Detected sbt version 0.12.2
[info] Loading global plugins from /usr0/home/ckaestne/.sbt/plugins
[info] Loading project definition from /usr0/home/ckaestne/work/TypeChef/TypeChef/project/project
[info] Loading project definition from /usr0/home/ckaestne/work/TypeChef/TypeChef/project
[info] Set current project to TypeChef (in build file:/usr0/home/ckaestne/work/TypeChef/TypeChef/)
[info] Set current project to FeatureExprLib (in build file:/usr0/home/ckaestne/work/TypeChef/TypeChef/)
[info] Compiling 10 Scala sources to /usr0/home/ckaestne/work/TypeChef/TypeChef/FeatureExprLib/target/scala-2.10/test
-classes...
[info] + FeatureExpr.parse(print(x))==x: OK, passed 100 tests.
[info] + FeatureExpr.and1: OK, passed 100 tests.
[info] + FeatureExpr.and0: OK, passed 100 tests.
[info] + FeatureExpr.andSelf: OK, passed 100 tests.
[info] + FeatureExpr.or1: OK, passed 100 tests.
[info] + FeatureExpr.or0: OK, passed 100 tests.
[info] + FeatureExpr.orSelf: OK, passed 100 tests.
[info] + FeatureExpr.a eq a: OK, passed 100 tests.
[info] + FeatureExpr.a equals a: OK, passed 100 tests.
[info] + FeatureExpr.a equivalent a: OK, passed 100 tests.
[info] + FeatureExpr.a implies a: OK, passed 100 tests.
[info] + FeatureExpr.creating (a and b) twice creates equal object: OK, passed 100 tests.
[info] + FeatureExpr.creating (a or b) twice creates equal object: OK, passed 100 tests.
[info] + FeatureExpr.creating (not a) twice creates equal object: OK, passed 100 tests.
[info] + FeatureExpr.applying not twice yields an equivalent formula: OK, passed 100 tests.
[info] + FeatureExpr.Commutativity wrt. equivalence: (a and b) produces the same object as (b and a): OK, passed 100
tests.
[info] + FeatureExpr.Commutativity wrt. equivalence: (a or b) produces the same object as (b or a): OK, passed 100 te
sts.
[info] + FeatureExpr.taut(a=>b) == contr(a and !b): OK, passed 100 tests.
[info] + FeatureExpr.featuremodel.tautology: OK, passed 100 tests.
```

Nightly Builds and Smoke Tests

- Build a release of a large project every night
  - Catches integration problems where a change "breaks the build"
  - Breaking the build is a BIG deal—may result in midnight calls to the responsible engineer
- Run simplified "smoke test" on build
  - Tests basic functionality and stability
  - Often: run by programmers before check-in
  - Provides rough guidance prior to full integration testing



Build and Test Automation

- Compile and execute from the command line
- Dependencies to all required libraries included (or downloaded on demand)
- Build tools
  - make
  - ant
  - gradle
  - maven
  - sbt
  - …

```
repositories {
  mavenCentral()
}

apply plugin: 'java'

dependencies {
  testCompile 'junit:junit:4.10'
}

sourceSets {
  main {
    java { srcDir 'src' }
    resources { srcDir 'misc/res' }
  }
```

Project conventions

- Defaults used by several build tools to find source and test files

- lib/
- src/
  - main/
    - java/
      - … java code …
    - resources/
      - … images …
  - test/
    - java/
      - … test code …
- build.gradle

Test Coverage

How much testing?

- Cannot test all inputs
  - too many, usually infinite

- What makes a good test suite?
- When to stop testing?
- How much to invest in testing?

Blackbox: Random Testing / Fuzz Testing

- Try random inputs, many of them
- Observe whether system crashes (exceptions, assertions)
- Try more random inputs, many more

- Successful in certain domains (parsers, network issues, …)
- Many tests execute similar paths
- Often finds only superficial errors
- Can be improved by guiding random selection with additional information (domain knowledge or extracted from source)

Blackbox: Covering Specifications

- Looking at specifications, not code:

- Test representative case
- Test boundary condition
- Test exception conditions
- (Test invalid case)

Structural Analysis for Test Coverage

- Organized according to program decision structure
- Touching: statement, branch

Whitebox testing

```
public static int binsrch (int[] a, int key) {

    int low  = 0;
    int high = a.length - 1;

    while (true) {

        if ( low > high ) return -(low+1);

        int mid = (low+high) / 2;

        if      ( a[mid] < key )  low  = mid + 1;
        else if ( a[mid] > key )  high = mid - 1;
        else    return mid;
    }
}
```

- Will this statement get executed in a test?
- Does it return the correct result?

- Could this array index be out of bounds?

- Does this return statement ever get reached?

Method Coverage

- Trying to execute **each method** as part of at least one test

```
39    public boolean equals(Object anObject) {
40        if (isZero())
41            if (anObject instanceof IMoney)
42                return ((IMoney)anObject).isZero();
43        if (anObject instanceof Money) {
44            Money aMoney= (Money)anObject;
45            return aMoney.currency().equals(currency())
46                                && amount() == aMoney.amount();
47        }
48        return false;
49    }
```

- Does this guarantee correctness?

Statement Coverage

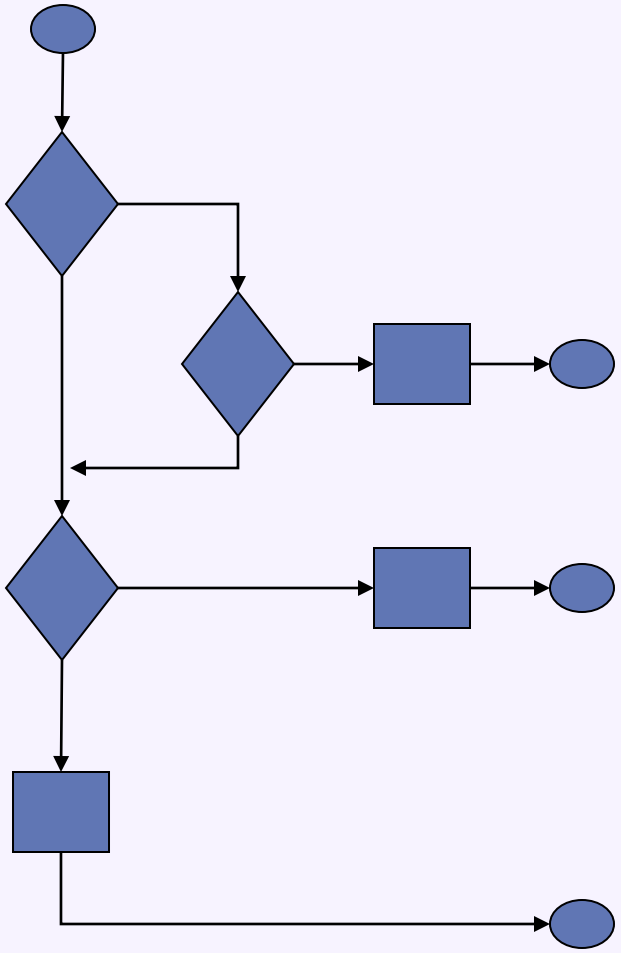
- Trying to test all parts of the implementation
- Execute **every statement** in at least one test

```
39    public boolean equals(Object anObject) {
40        if (isZero())
41            if (anObject instanceof IMoney)
42                return ((IMoney)anObject).isZero();
43        if (anObject instanceof Money) {
44            Money aMoney= (Money)anObject;
45            return aMoney.currency().equals(currency())
46                                && amount() == aMoney.amount();
47        }
48        return false;
49    }
```

- Does this guarantee correctness?

Structure of Code Fragment to Test

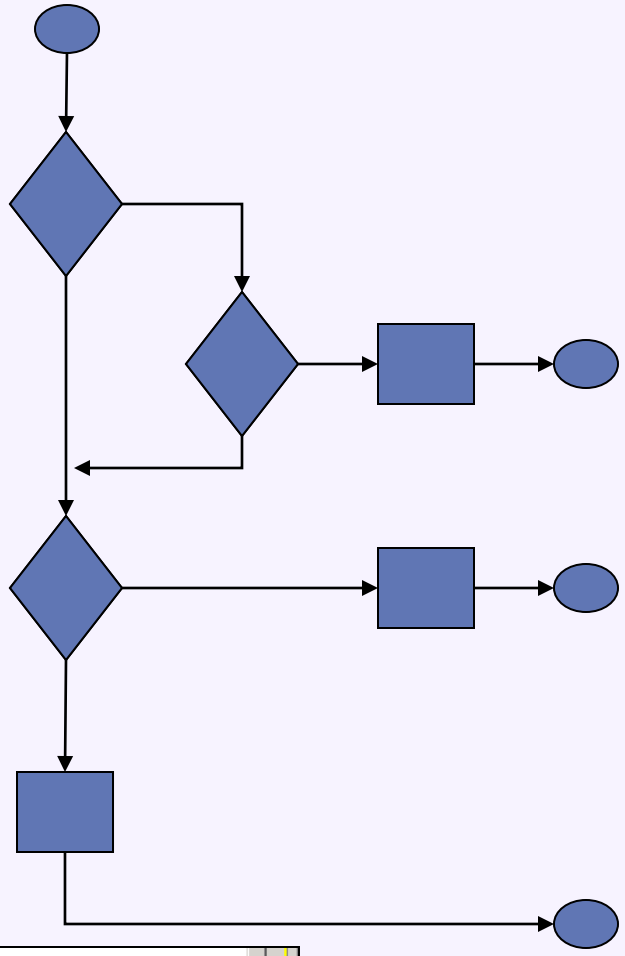
```
39    public boolean equals(Object anObject) {
40        if (isZero())
41            if (anObject instanceof IMoney)
42                return ((IMoney)anObject).isZero();
43        if (anObject instanceof Money) {
44            Money aMoney= (Money)anObject;
45            return aMoney.currency().equals(currency())
46                            && amount() == aMoney.amount();
47        }
48        return false;
49    }
```



Flow chart diagram for junit.samples.money.Money.equals

Statement Coverage

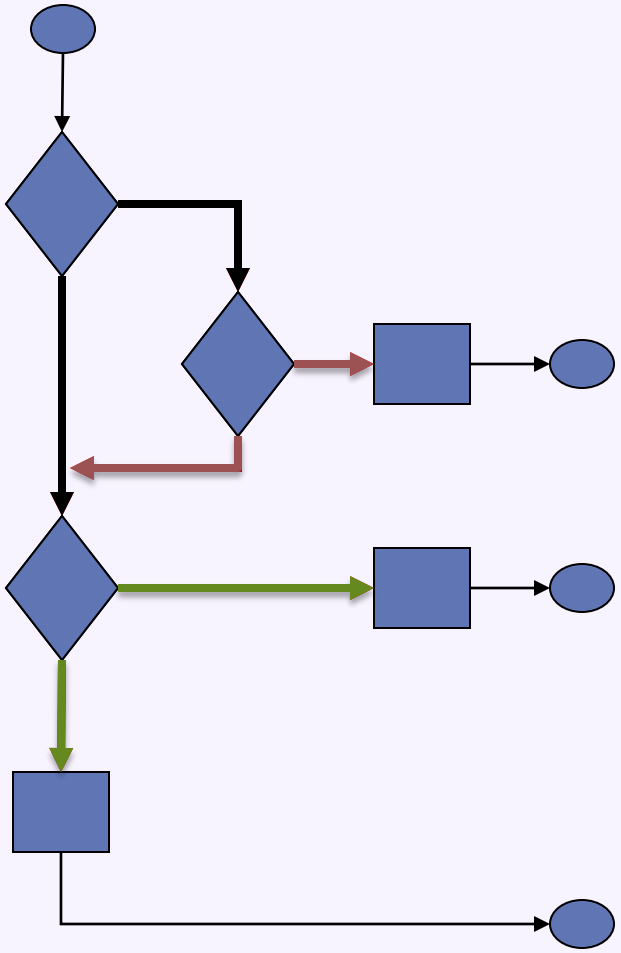
- **Statement coverage**
  - What portion of program statements (nodes) are touched by test cases
- Advantages
  - Test suite size linear in size of code
  - Coverage easily assessed
- Issues
  - Dead code is not reached
  - May require some sophistication to select input sets
  - Fault-tolerant error-handling code may be difficult to “touch”
  - Metric: Could create incentive to *remove* error handlers!



```
39    public boolean equals(Object anObject) {
40        if (isZero())
41            if (anObject instanceof IMoney)
42                return ((IMoney)anObject).isZero();
43        if (anObject instanceof Money) {
44            Money aMoney= (Money)anObject;
45            return aMoney.currency().equals(currency())
46                                && amount() == aMoney.amount();
47        }
48        return false;
49    }
```

Branch Coverage

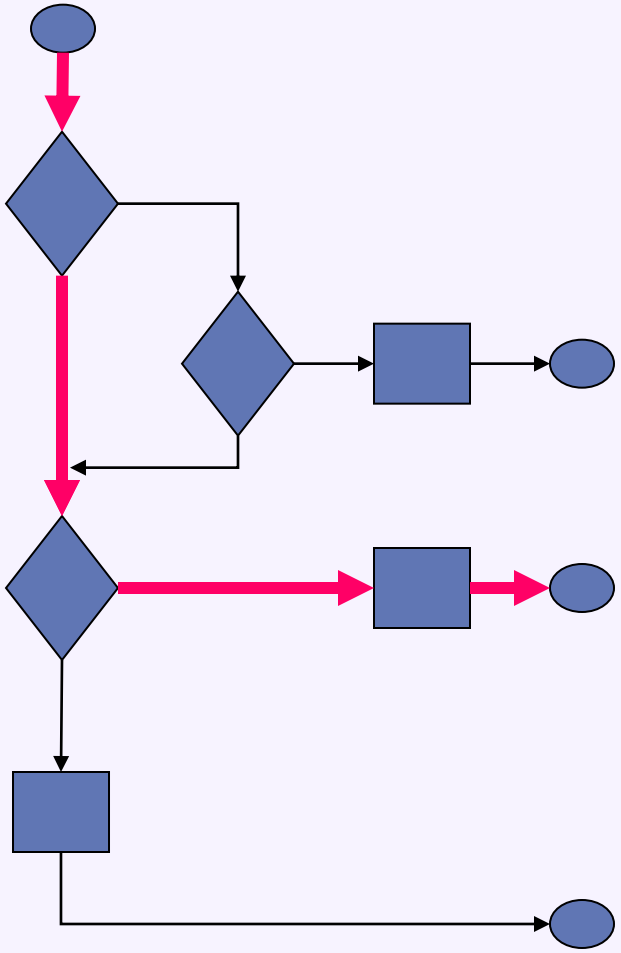
- **Branch coverage**
  - What portion of **condition branches** are covered by test cases?
  - *Or:* What portion of relational expressions and values are covered by test cases?
    - Condition testing (Tai)
  - **Multicondition coverage** – all boolean combinations of tests are covered
- Advantages
  - Test suite size and content derived from structure of boolean expressions
  - Coverage easily assessed
- Issues
  - Dead code is not reached
  - Fault-tolerant error-handling code may be difficult to “touch”



```
39    public boolean equals(Object anObject) {
40        if (isZero())
41            if (anObject instanceof IMoney)
42                return ((IMoney)anObject).isZero();
43        if (anObject instanceof Money) {
44            Money aMoney= (Money)anObject;
45            return aMoney.currency().equals(currency())
46                                && amount() == aMoney.amount();
47        }
48        return false;
49    }
```

Path Coverage

- Path coverage
  - What portion of all possible **paths** through the program are covered by tests?
  - Loop testing: Consider representative and edge cases:
    - Zero, one, two iterations
    - If there is a bound n: n-1, n, n+1 iterations
    - Nested loops/conditionals from inside out
- Advantages
  - Better coverage of logical flows
- Disadvantages
  - Infinite number of paths
  - Not all paths are possible, or necessary
    - What are the *significant* paths?
  - Combinatorial explosion in cases unless careful choices are made
    - E.g., sequence of *n* if tests can yield up to 2^n possible paths
  - Assumption that program structure is basically sound



```
39    public boolean equals(Object anObject) {
40        if (isZero())
41            if (anObject instanceof IMoney)
42                return ((IMoney)anObject).isZero();
43        if (anObject instanceof Money) {
44            Money aMoney= (Money)anObject;
45            return aMoney.currency().equals(currency())
46                            && amount() == aMoney.amount();
47        }
48        return false;
```

Write testable code

```
//700LOC
public boolean foo() {
    try {
        synchronized () {
            if () {
            } else {
            }
            for () {
                if () {
                    if () {
                        if () {
                            if ()?
                            {
                                if () {
                                    for () {
                                    }
                                }
                            }
                        } else {
                            if () {
                                for () {
                                    if () {
                                    } else {
                                    }
                                    if () {
                                    } else {
                                        if () {
                                        }
                                    }
                                    if () {
                                        if () {
                                            if () {
                                                for () {
                                                }
                                            }
                                        }
                                    } else {
                                    }
                                }
```

Unit testing as design mechanism

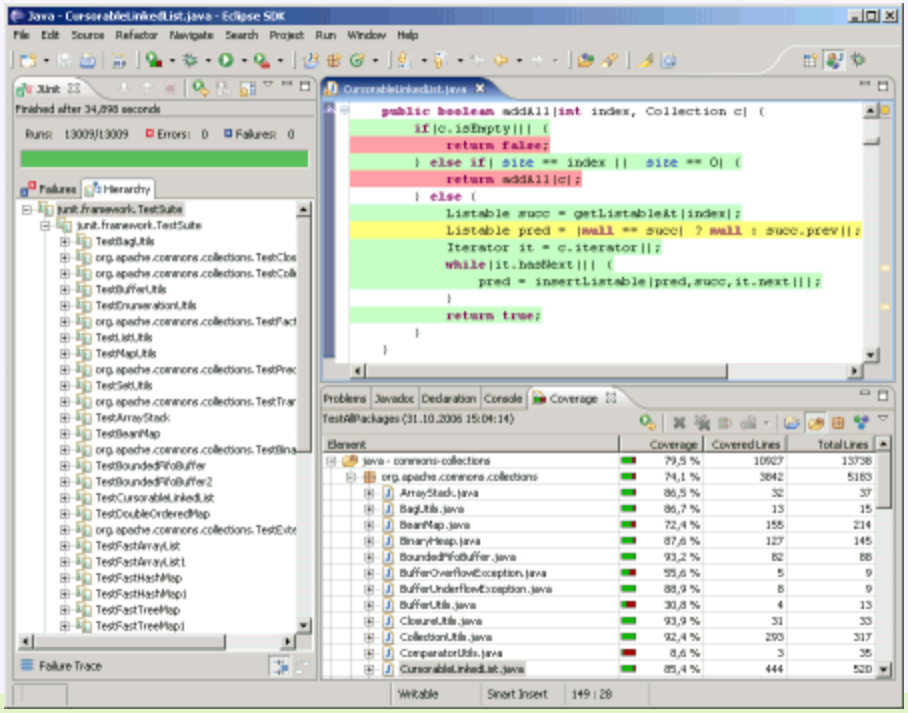
Source:
http://thedailywtf.com/Articles/Coding-Like-the-Tour-de-France.aspx

```
int binarySearch(int[] a, int key) {
        int imin = 0;
        int imax = a.length-1;
        while (imax >= imin) {
                int imid = midpoint(imin, imax);
                if (a[imid] < key)
                        imin = imid + 1;
                else if (a[imid] > key )
                        imax = imid - 1;
                else
                        return imid;
        }
        return -1;
}
```

Find test cases to maximize line, branch, and path coverage.

Test Coverage Tooling

- Coverage assessment tools
  - Track execution of code by test cases
- Count visits to statements
  - Develop reports with respect to specific coverage criteria
  - Instruction coverage, line coverage, branch coverage
- Example: EclEmma tool for JUnit tests



"Coverage" is useful but also dangerous

- Examples of what coverage analysis could miss
  - Unusual paths
  - Missing code
  - Incorrect boundary values
  - Timing problems
  - Configuration issues
  - Data/memory corruption bugs
  - Usability problems
  - Customer requirements issues

- Coverage is not a good **adequacy** criterion
  - Instead, use to find places where testing is *inadequate*

Test coverage – Ideal and Real

- An Ideal Test Suite
  - Uncovers all errors in code
  - Uncovers all errors that requirements capture
    - All scenarios covered
    - Non-functional attributes: performance, code safety, security, etc.
  - Minimum size and complexity
  - Uncovers errors early in the process
- A Real Test Suite
  - Uncovers some portion of errors in code
  - Has errors of its own
  - Assists in exploratory testing for validation
  - Does not help very much with respect to non-functional attributes
  - Includes many tests inserted after errors are repaired to ensure they won't reappear

Testing against the environment (stubs)

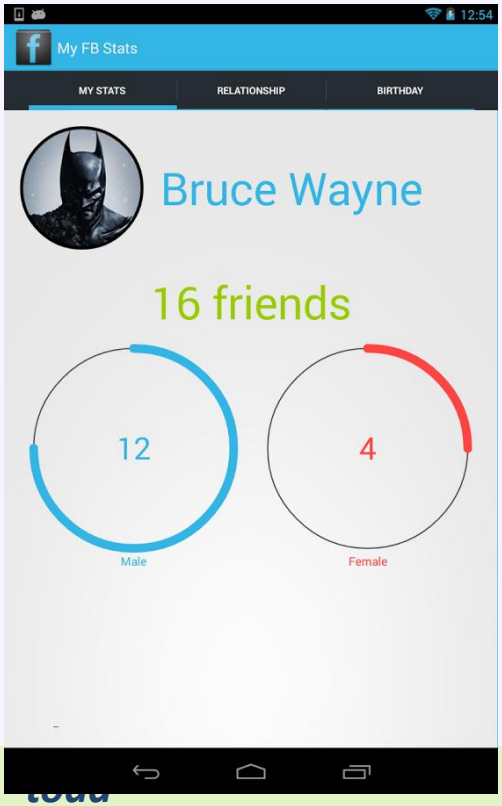
Problems in automating testing

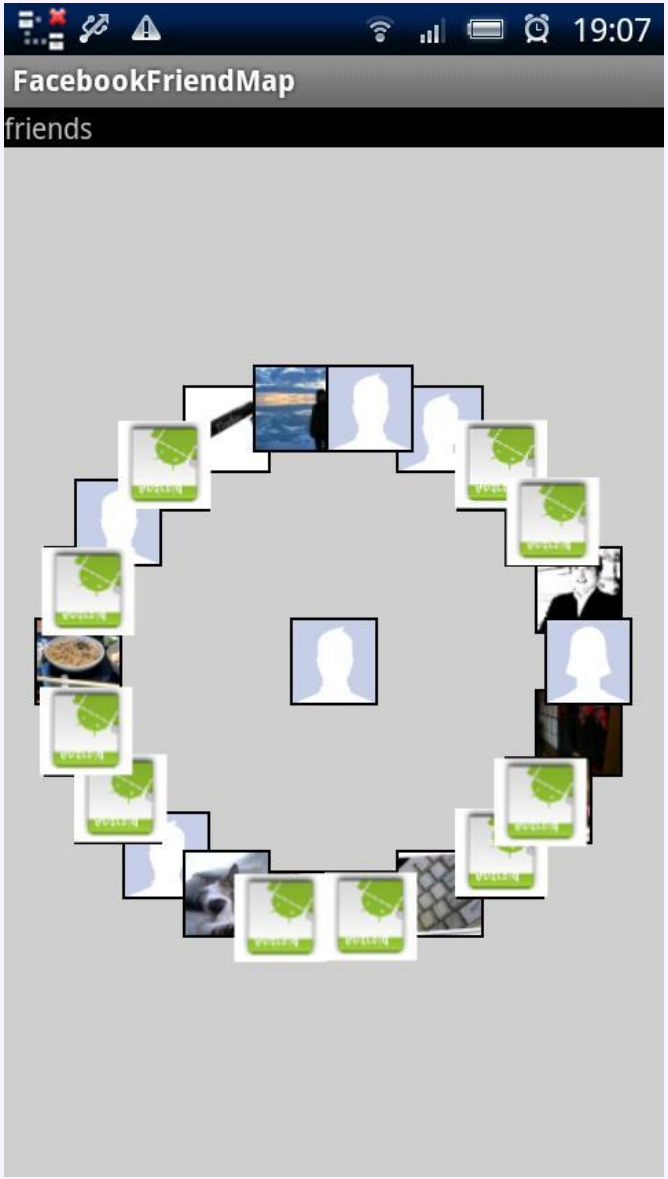
- User interfaces and user interactions
  - Users click buttons, interpret output
  - Waiting/timing issues
- Test data vs. real data
- Testing against big infrastructure (databases, web services, …)
- Testing with side effects (e.g., printing and mailing documents)
- Nondeterministic behavior
- Concurrency (more later and in 15-313)

-> the test environment

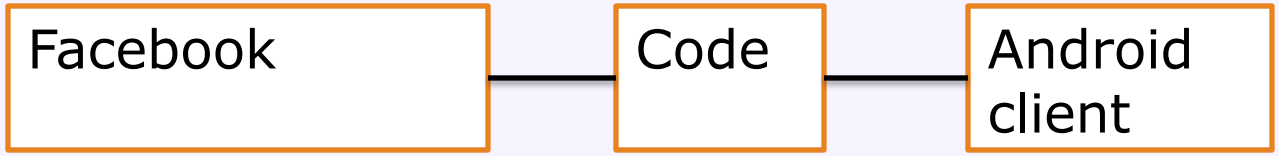
Example

- 3rd party Facebook apps for Android
- User interface for Android
- Internal computations ala HW1
- Backend with Facebook data



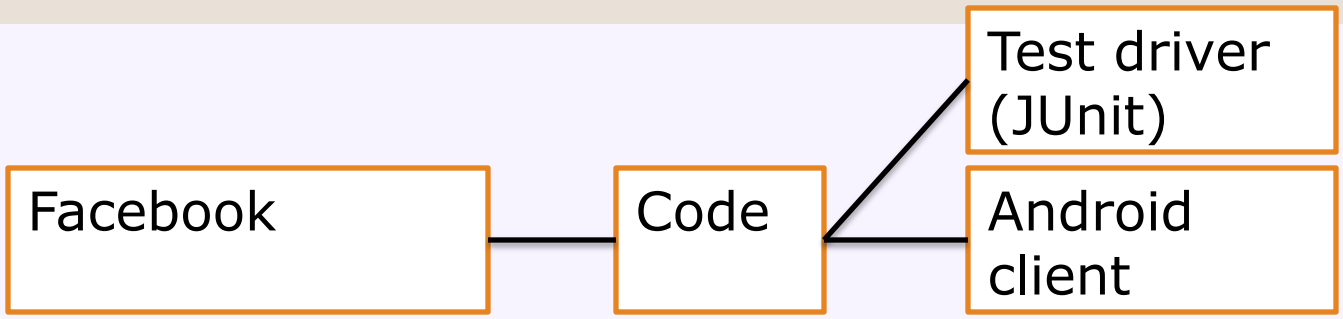


Testing in real environments



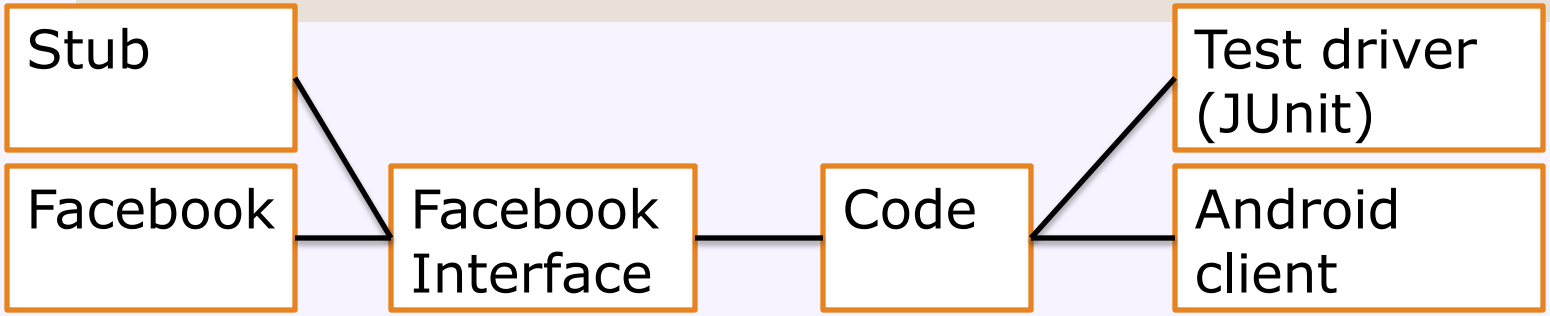

```
void buttonClicked() {
        render(getFriends());
}
Pair[] getFriends() {
        Connection c = http.getConnection();
        FacebookAPI api = new FacebookAPI(c);
        try {
                List<Node> persons = api.getFriends("john");
                for (Node personA: persons) {
                        for (Node personB: persons) {
                                …
                }}
        } catch (…) { … }
        return result;
}
```

Test drivers




```
@Test void testGetFriends() {
        assert getFriends() == …;
}
Pair[] getFriends() {
        Connection c = http.getConnection();
        FacebookAPI api = new FacebookAPI(c);
        try {
                List<Node> persons = api.getFriends("john");
                for (Node personA: persons) {
                        for (Node personB: persons) {
                                …
                }}
        } catch (…) { … }
        return result;
}
```

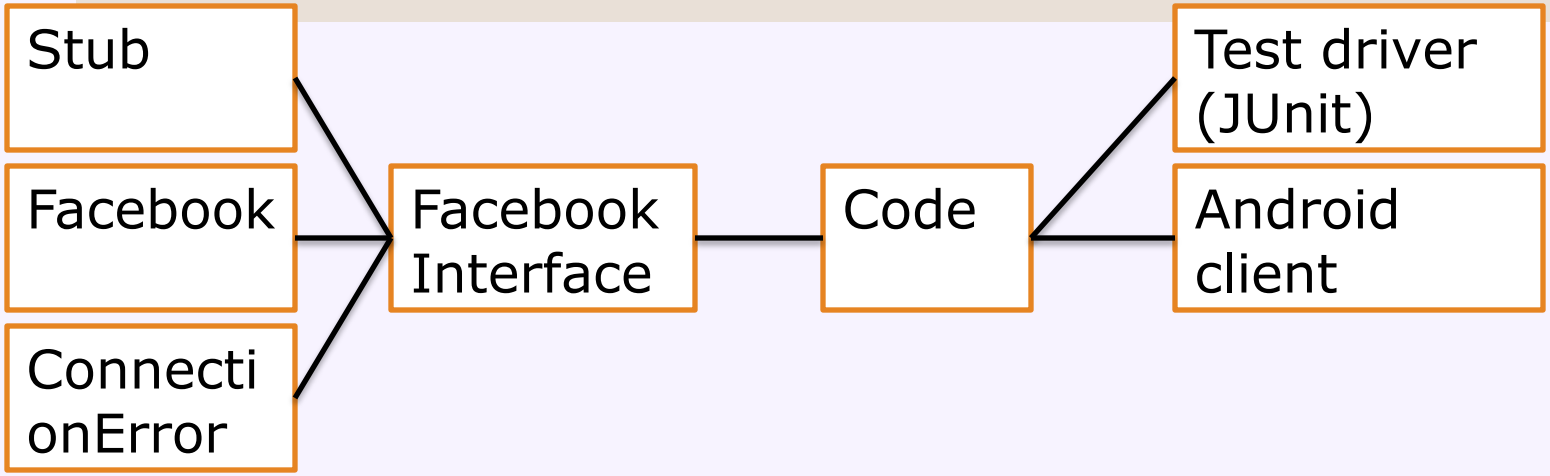
Stubs




```
FacebookInterface fb;
@Before void init() {fb = new FacebookStub(); }

Pair[] getFriends() {
        try {

        } catch
        return r
}
```

```
class FacebookStub implements FacebookInterface {
        void connect() {}
        List<Node> getPersons(String name) {
                if ("john".equals(n)) {
                        List<Node> result=new List();
                        result.add(…);
                        return result;
                }
        }
}
```

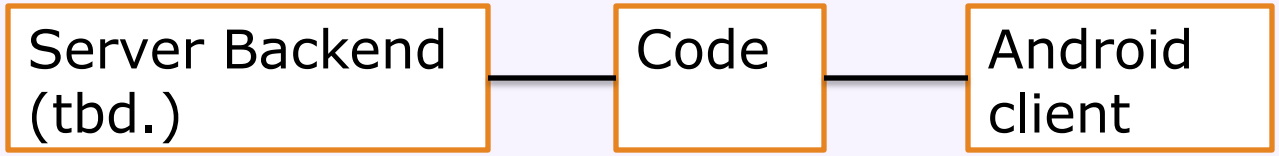
Robustness test




```
class ConnectionError implements FacebookInterface {
        List<Node> getPersons(String name) {
                throw new HttpConnectionException();
        }
}

@Test void testConnectionError() {
        assert getFriends(new ConnectionError) == null;
}
```

Test for expected error conditions by introducing artificial errors through stubs

Testing in real environments



- Separating code (with stubs) allows to test against functionality
  - provided by other teams
  - specified, but not yet implemented

Testing Strategies in Environments

- Separate business logic and data representation from GUI for testing (more later)
- Test algorithms locally without large environment using stubs
- Advantages of stubs
  - Create deterministic response
  - Can reliably simulate spurious states (e.g. network error)
  - Can speed up test execution (e.g. avoid slow database)
  - Can simulate functionality not yet implemented
- Automate, automate, automate

Design Implications

- Write testable code!
- When planning to test with a stub design for it! Abstract the actual subsystem behind and interface.

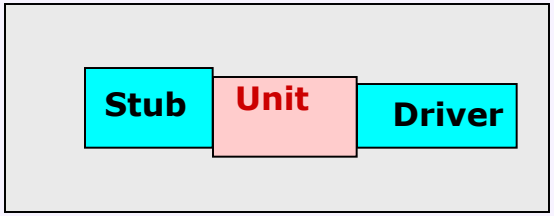
```
int getFreeTime() {
        DB2Database db = new DB2Database("calendar.db");
        return db.execute("select …");
}
```

```
int getFreeTime() {
        IDatabase db =
              databaseFactory.createDb("calendar.db");
        return db.execute("select …");
}
```

```
int getFreeTime(IDatabase db) {
        return db.execute("select …");
}
```

Scaffolding

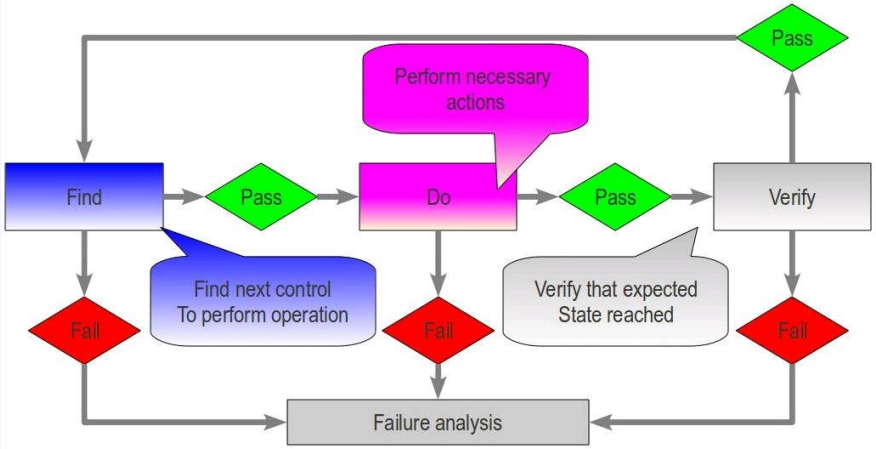
- Catch bugs early: Before client code or services are available
- Limit the scope of debugging: Localize errors



- Improve coverage
  - System-level tests may only cover 70% of code [Massol]
  - Simulate unusual error conditions – test internal robustness
- Validate internal interface/API designs
  - Simulate clients in advance of their development
  - Simulate services in advance of their development
- Capture developer intent (in the absence of specification documentation)
  - A test suite formally captures elements of design intent
  - Developer documentation
- Improve low-level design
  - Early attention to ability to test – "testability"

Automating GUI/Web Testing

- Capture and Replay Strategy
  - Capture mouse actions
  - Capture system events
- Test Scripts
  - (click on button labeled "Start" expect value X in field Y)

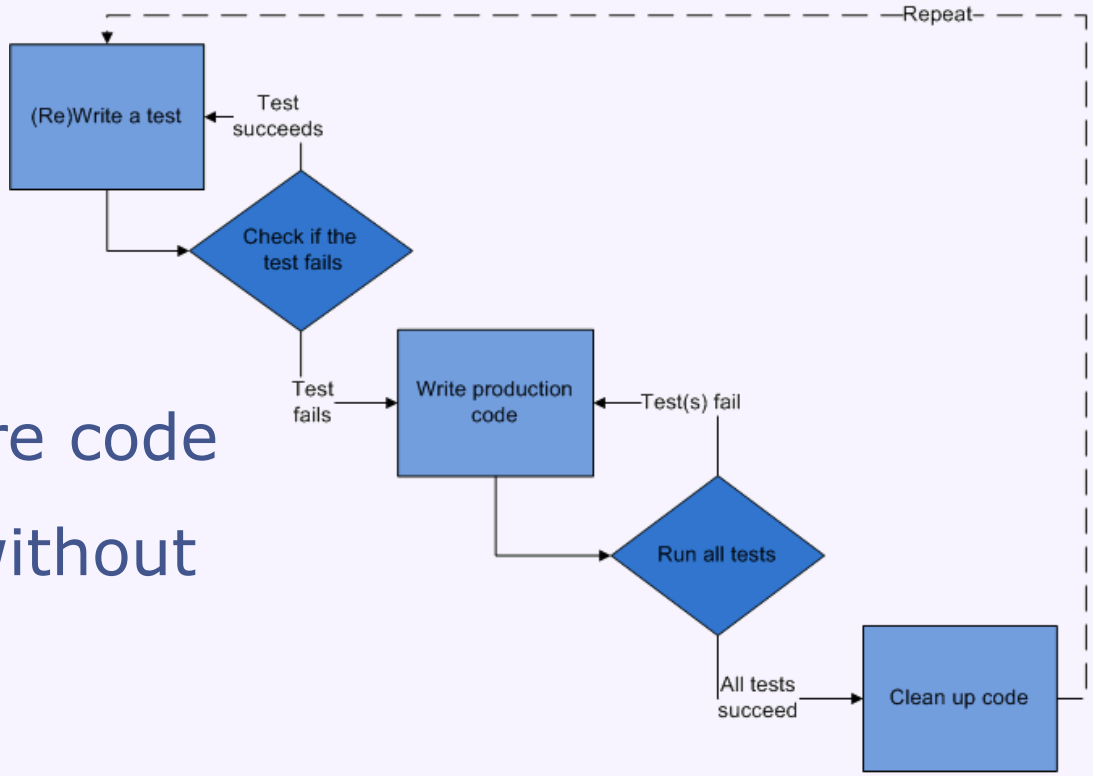


- Lots of tools and frameworks
  - e.g. JUnit + Jemmy for Java/Swing
- (Avoid load on GUI testing by separating model from GUI)

Test-driven development

Test Driven Development

- 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 writing unneeded code
  - Higher product quality (e.g. better code, less defects)
  - Higher test suite quality
  - Higher overall productivity



Summary

- Unit testing is one of many testing approaches
- Unit testing to
  - discover bugs (not prove correctness)
  - document code
  - design testable code
- JUnit details (@Test, …)
- Test coverage: The good, the bad, and the ugly
- Testing against environments - Stubs
- You should be able to write and automate unit tests for all your code now