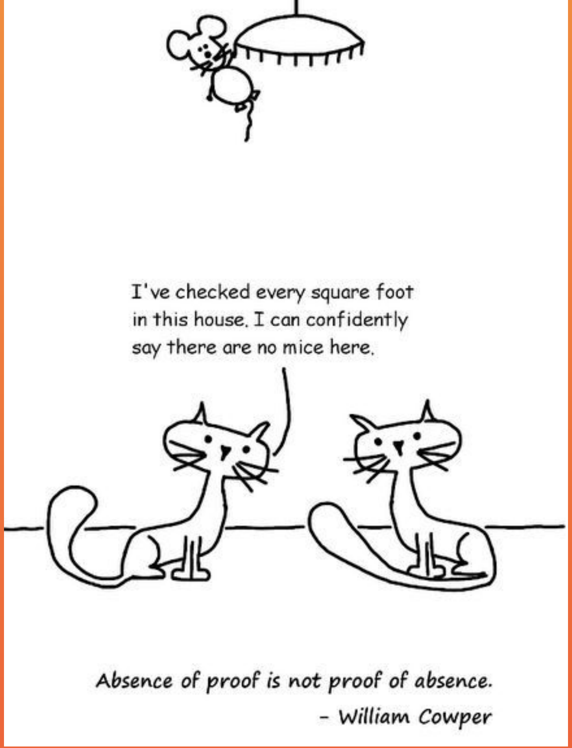

I've checked every square foot in this house. I can confidently say there are no mice here.
Absence of proof is not proof of absence.
- William Cowper

Module 2: Testing

CPSC 310

Reid Holmes

C0

- Only 67/556 repos have been checked out.
- Problems with prettier fixed.
- 3 AutoTest runs in 24h, instead of 2.
- Deadline 1800 this Friday.

Examinable skills

Testing

- Explain different kinds of tests (system, etc).
- Explain the difference between black box and glass box testing, both in terms of when they happen, and how the tests are devised.
- Explain how tests for a particular version of a system become regression tests as the system evolves.
- Properly form a test with the right parts (setup, call, check).
- Explain why line coverage is often considered insufficient as a measure of suite completeness.
- Create test suites that achieve line and branch coverage, and that achieve boundary and equivalence class testing.
- Assess the above kinds of coverage in a given test suite.
- Explain why mock objects are helpful.
- Provide the design/implementation for a mock object given the component to test and a "real" component's interface.
- Explain mutation testing.
- Suggest mutants given some code, or find mutants that some tests might not find.
- Explain how to achieve the various kinds of testability and be able to spot when they are not achieved.
- Explain why TDD affords testability.
- Explain and identify architectural choices to afford improved testability.
- Explain four phase test and given-when-then
- Explain fuzz testing.
- Contrast random fuzzing, generator-based fuzzing, and mutational fuzzing.
- Describe different kinds of assertions and justify their benefits and tradeoffs.

“Testing shows the presence, not the absence of bugs.”

— Dijkstra

Testing

Testing Intro

Arnaud LEMAIRE
@Lilobase

When part of your code doesn't have tests

Why Software Testing?

http://youtu.be/gp_D8r-2hwk

The Ariane 5 ….

An Uncaught Exception!

Sadly, the primary cause was found to be a **piece of software which had been retained from the previous launchers systems and which was not required during the flight of Ariane 5**. The software was used in the Inertial Reference System (SRI) to calculate the attitude of the launcher. In Ariane 4, this software was allowed to continue functioning during the first 50 seconds of flight as it could otherwise delay launching if the countdown was halted for any other reason, this was not necessary for Ariane 5. As well, the software contained implicit assumptions about the parameters, in particular the horizontal velocity that were safe for Ariane 4 but not Ariane 5.

The failure occurred because the horizontal velocity exceeded the maximum value for a 16 bit unsigned integer when it was converted from its signed 64 bit representation. **This failure generated an exception in the code which was not caught and thus propagated up through the processor and ultimately caused the SRI to fail.** The failure triggered the automatic fail-over to the backup SRI which had already failed for the same reason. This combined failure was then communicated to the main computer responsible for controlling the jets of the rocket, however, this information was misinterpreted as valid commands. As a result of the invalid commands, the engine nozzles were swung to an extreme position and the launcher was destroyed shortly afterwards.

The failure was thus entirely due to a single line of code.

This should have been caught *in testing!!!*

DAD! TECHNICAL DEBT!
Just ignore it and it will go away
DAD... I think it is getting bigger.
Everything is cool.
www.lunarbaboon.com

[http://neilstudd.ghost.io/2017/02/11/screen-testing-preview-deepwater-horizon/]

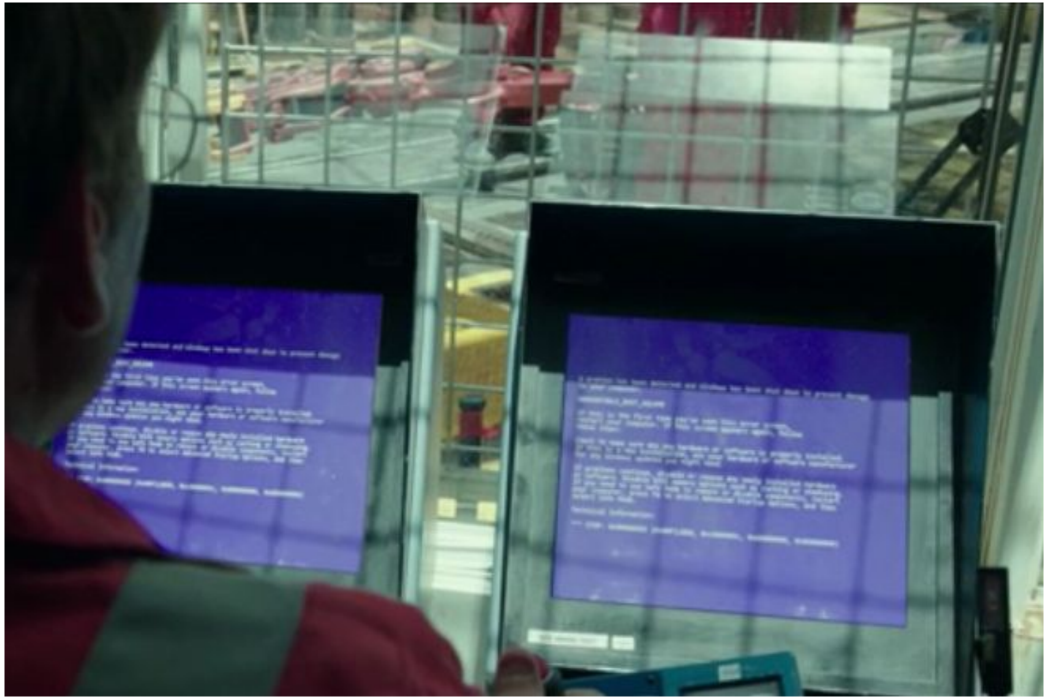
Deepwater Horizon

“Tight deadlines encourage bad decisions”

[http://neilstudd.ghost.io/2017/02/11/screen-testing-preview-deepwater-horizon/]

Deepwater Horizon

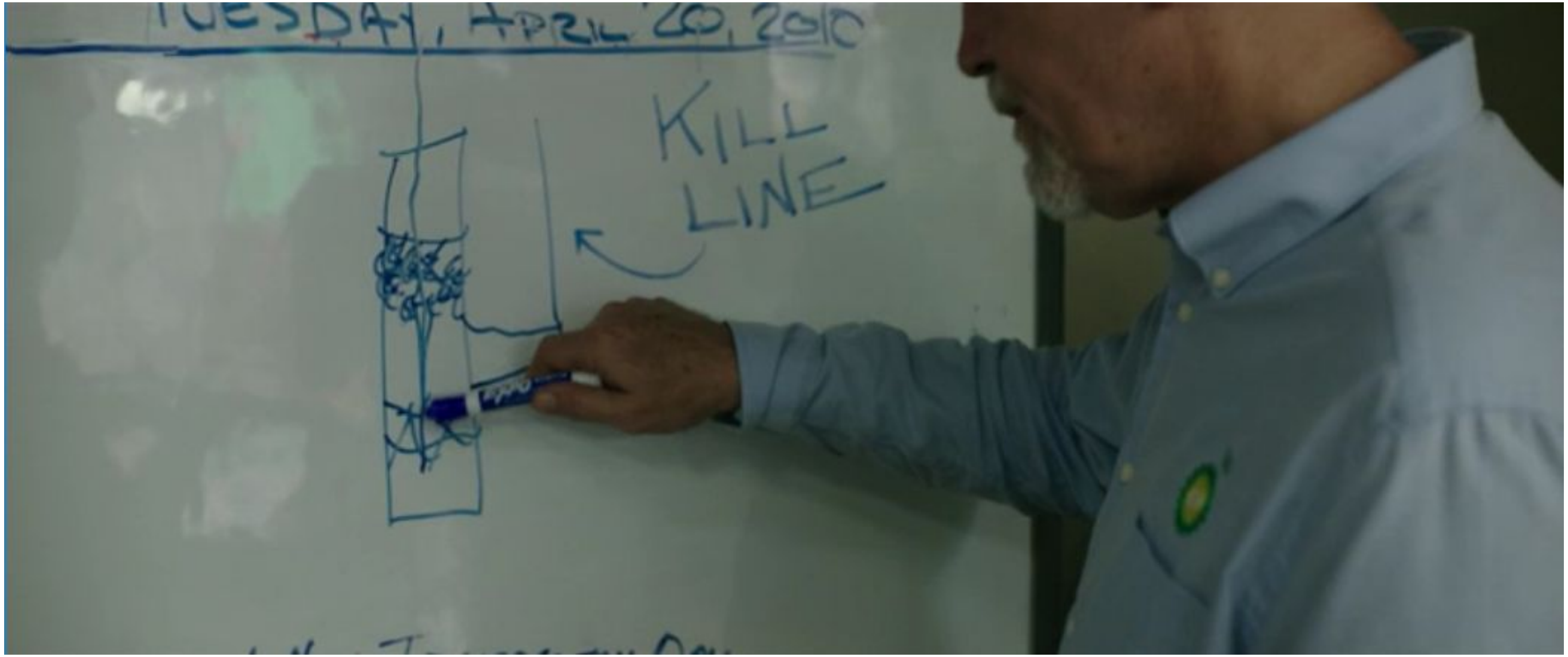
“No slack to fix technical debt”



[http://neilstudd.ghost.io/2017/02/11/screen-testing-preview-deepwater-horizon/]

Deepwater Horizon

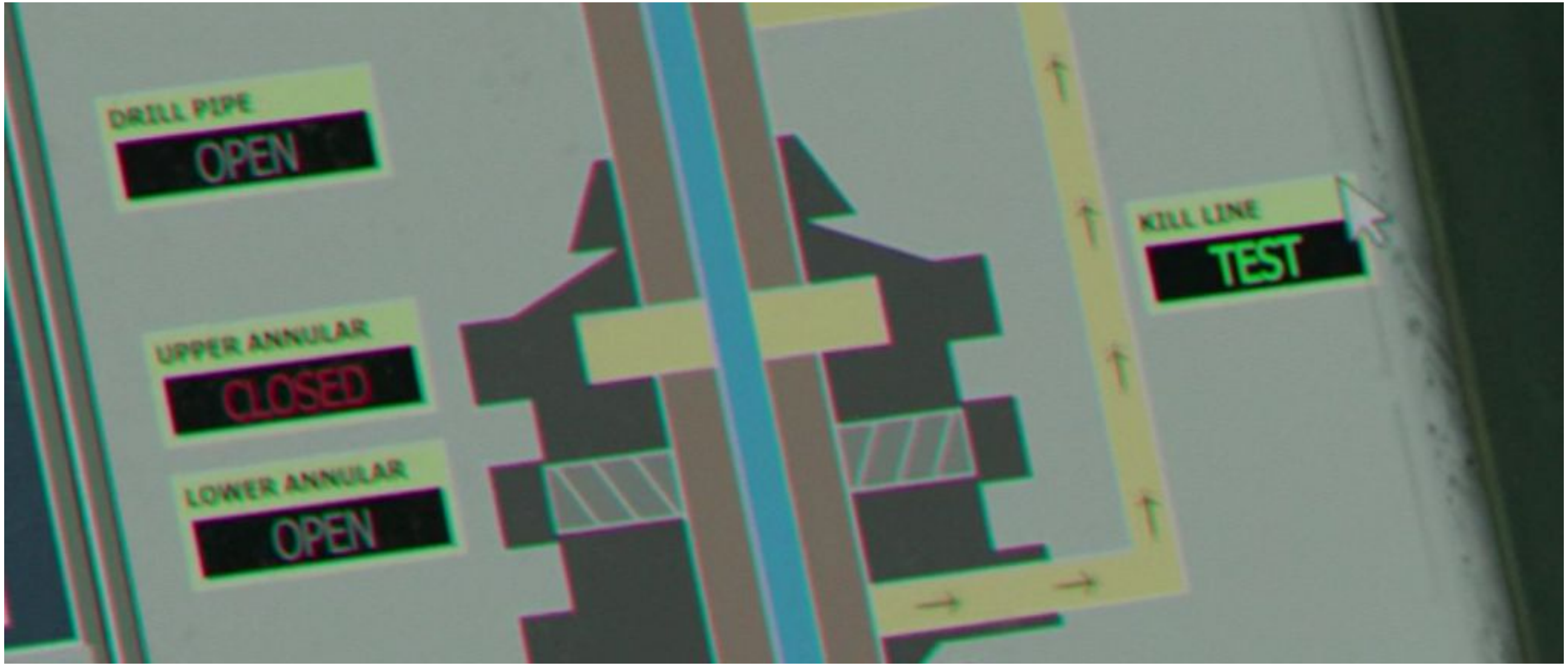
“Poor communication across reporting lines”



[http://neilstudd.ghost.io/2017/02/11/screen-testing-preview-deepwater-horizon/]

Deepwater Horizon

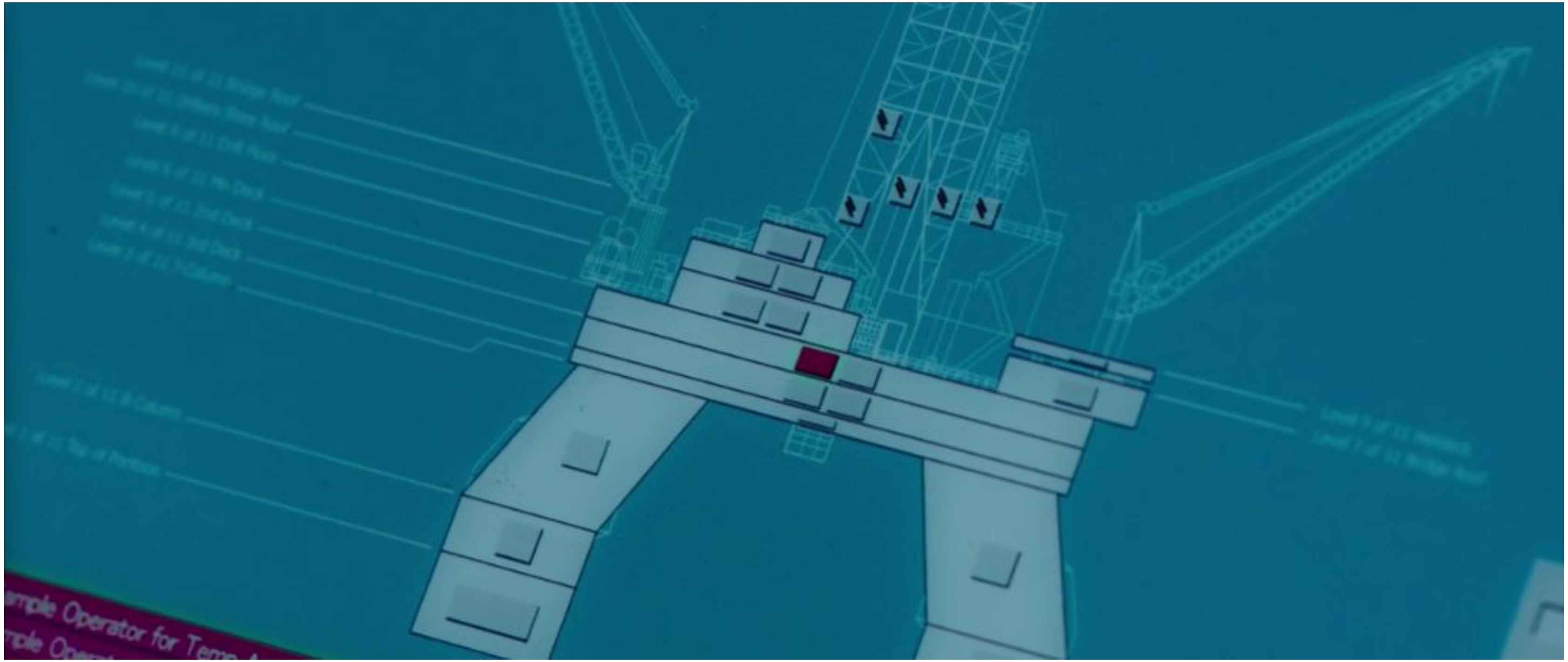
"Interpreting the results of tests"



[http://neilstudd.ghost.io/2017/02/11/screen-testing-preview-deepwater-horizon/]

Deepwater Horizon

“The perfect storm”



Deepwater Horizon

Given a finite amount of time and resources, testing enables validating that a system has an acceptable risk of costly or dangerous defects.

—Bob Binder

Why not test?

- Good reasons:
  - I don’t know how!
  - Legacy code
- Bad reasons:
  - Bad design
  - Doesn’t catch bugs (now)
  - Slow
  - Boring
  - Hard to change
  - That’s QA’s job

Common testing assumptions

- "The cost of fixing faults rises exponentially with how late [e.g., requirements, design, implementation, deployment] they are detected."
  - This is commonly stated but is based on evidence from 20+ years ago.
  - This assumption does not seem to hold for modern processes, tools, and languages.
- Still necessary to validate that the system works.
- Important that system continues to work as it evolves.

Red, green, refactor

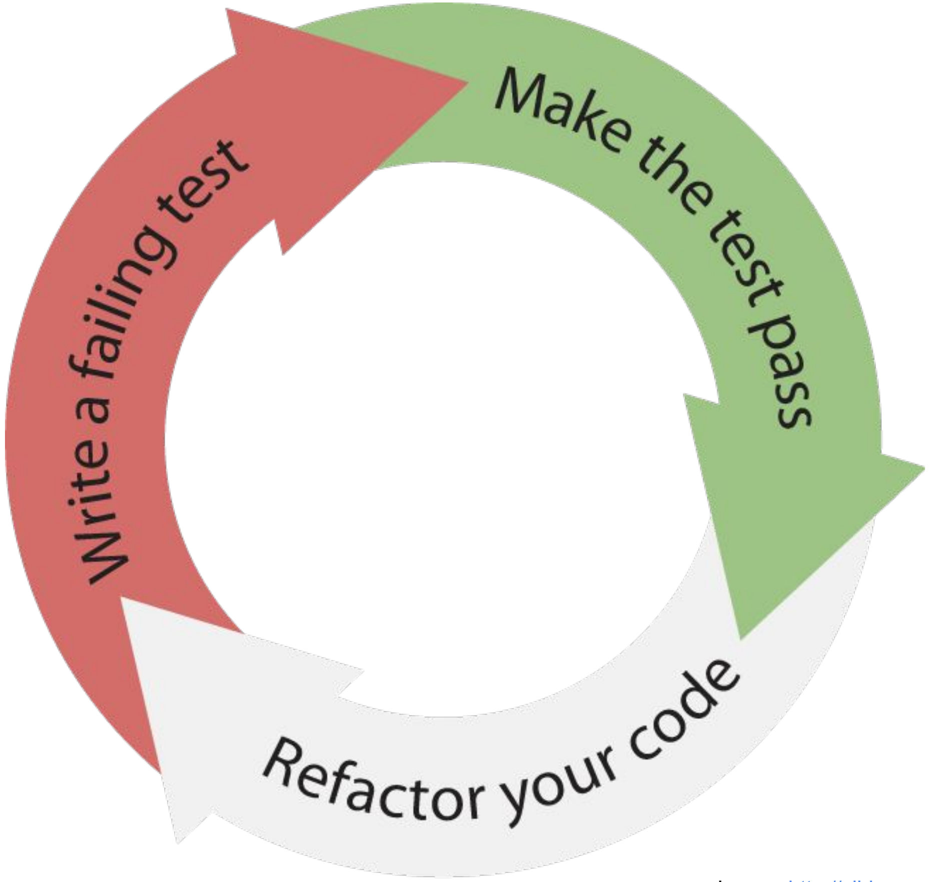


Image: http://slides.com/tonygaskell/mocha-chai#/1/1

Red, green, refactor

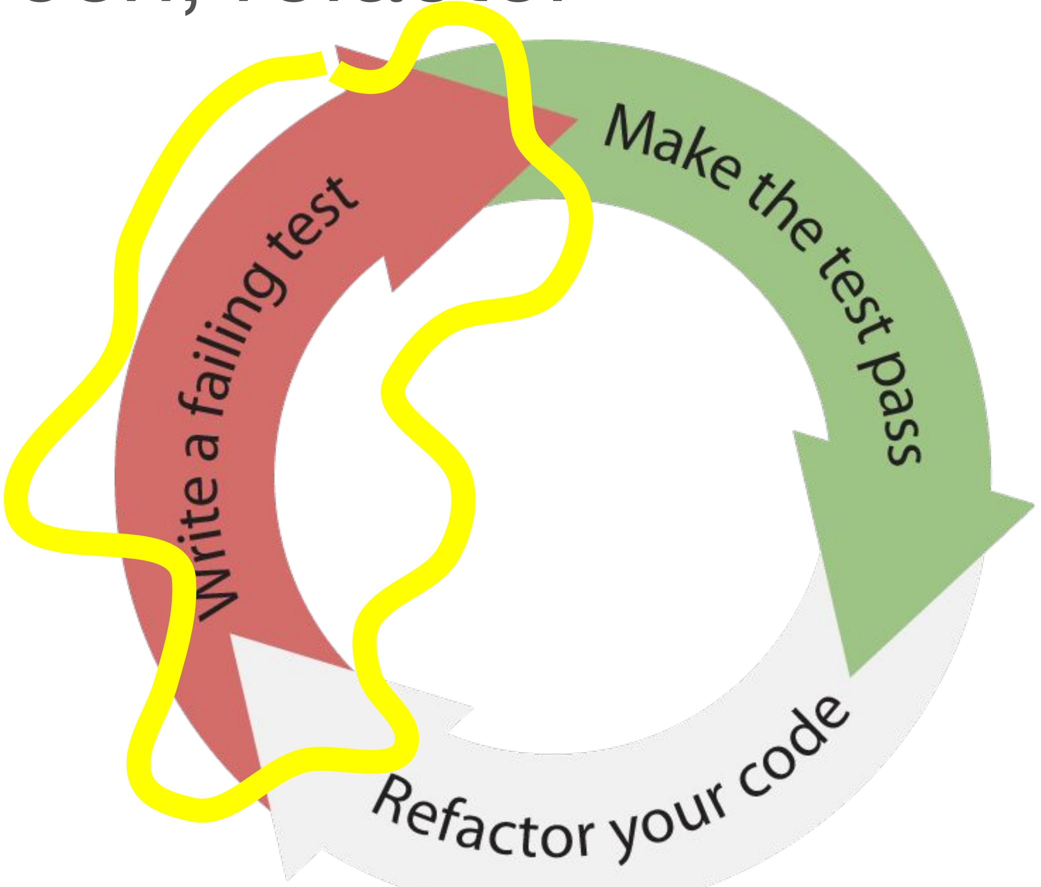
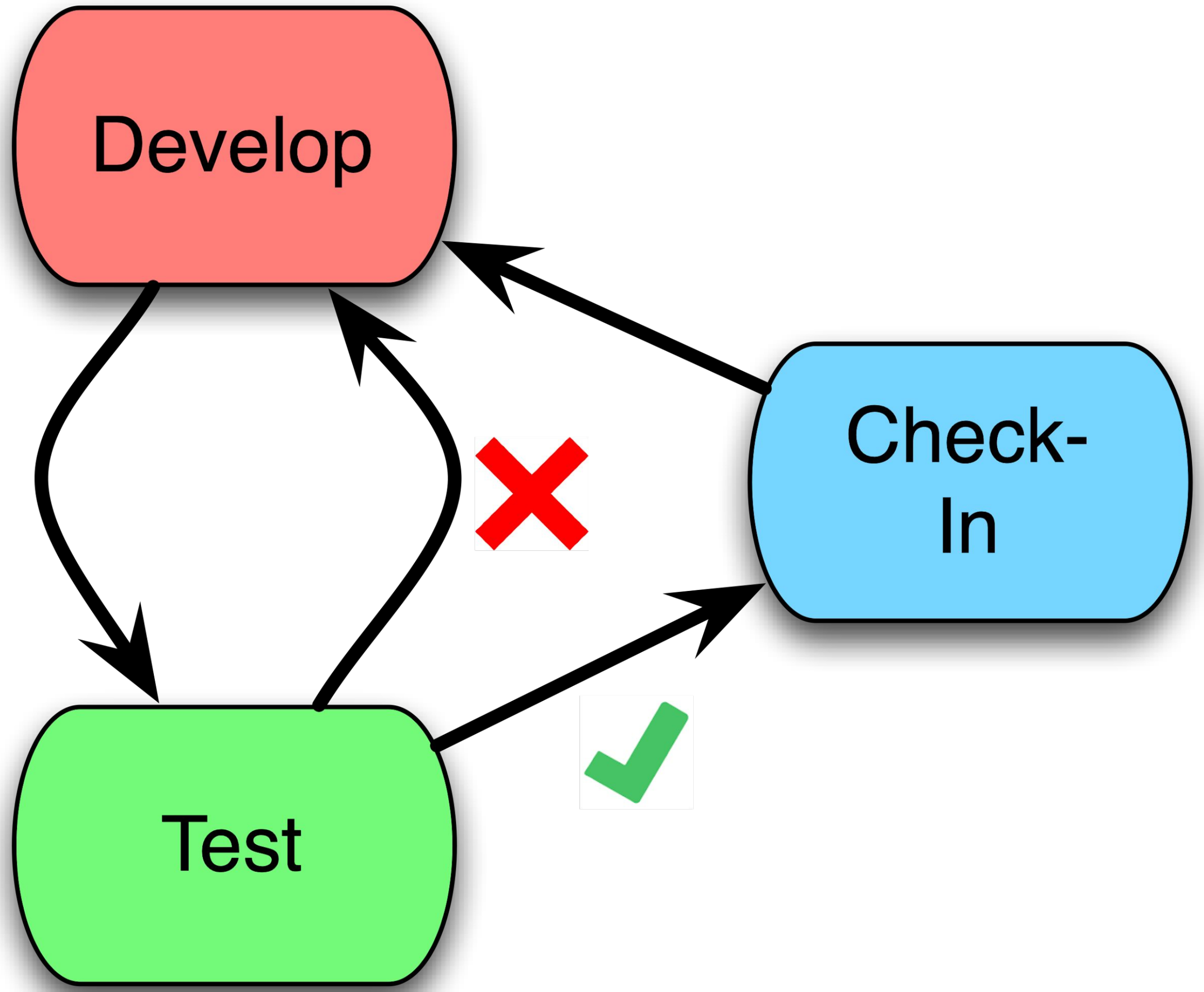


Image: http://slides.com/tonygaskell/mocha-chai#/1/1


Develop
Test
Check-In

Test Terminology

- SUT/CUT: System/Code Under Test.
- Glass-box: Tests that consider internals of CUT.
- Black-box: Tests that are oblivious of internals of CUT.
- Effectiveness: The probability of detecting a bug per unit of effort.
- Higher testability: More effective tests, same cost.
- Lower testability: Fewer weaker tests, same cost.
- Repeatability: The likelihood that running the same test twice will yield the same result.
- Flaky test: Tests that pass/fail non-deterministically.

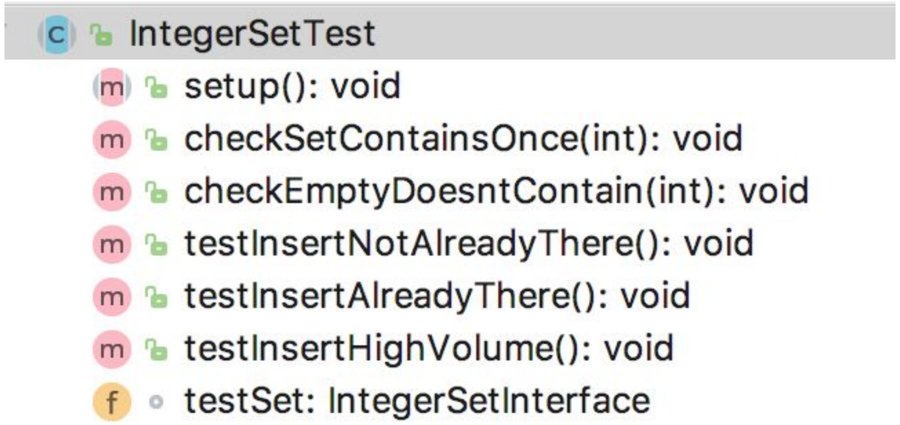
CPSC 210 Reminder: Each Unit Test Implementation

- Each test has:
    1) A **meaningful name** that describes the reason for the test. These names are important as tests are a crucial form of documentation.
    2) **Setup**: Required code to get into desired state. Sometimes this is refactored into the @Before code.
    3) **Execution**: Invokes the CUT.
    4) **Validation**: Validates that the CUT behaved as expected.
    5) **Teardown**: Sometimes it is necessary to reset parts of the system after a test; often in @After code.

Unit Test Suite

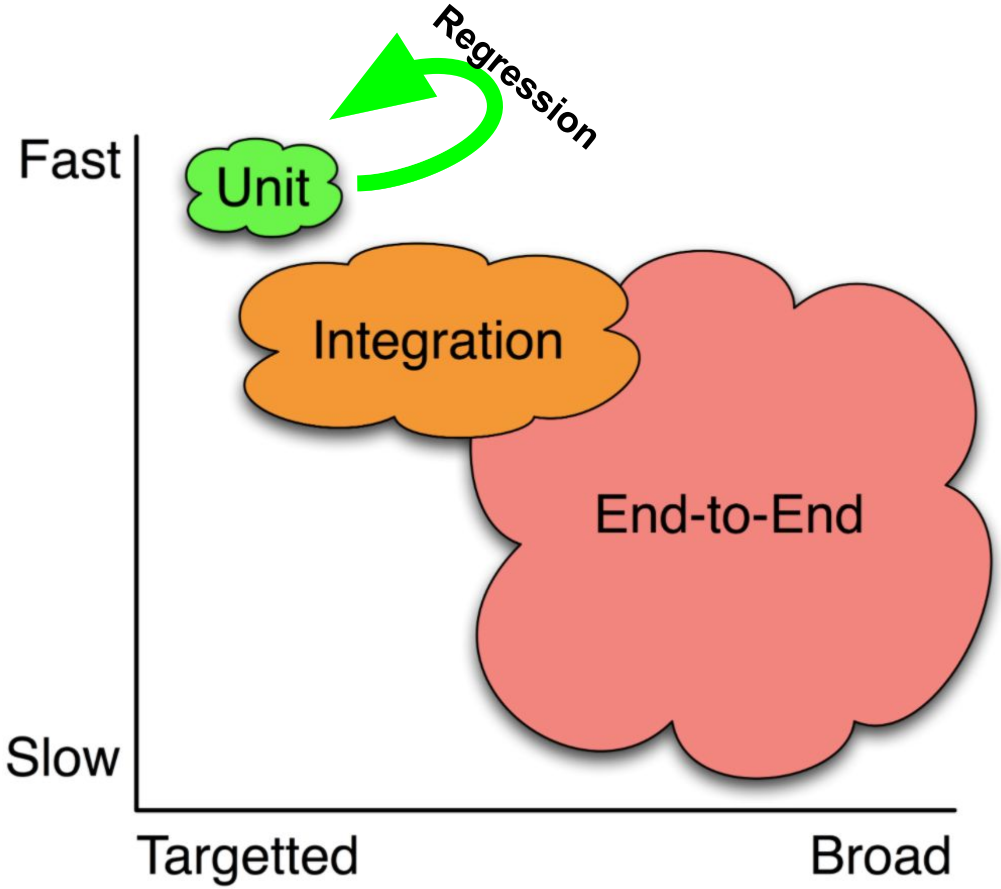
Test Suite:

Contains the collection of tests for one project, sometimes broken down by component (like tests for a particular class).

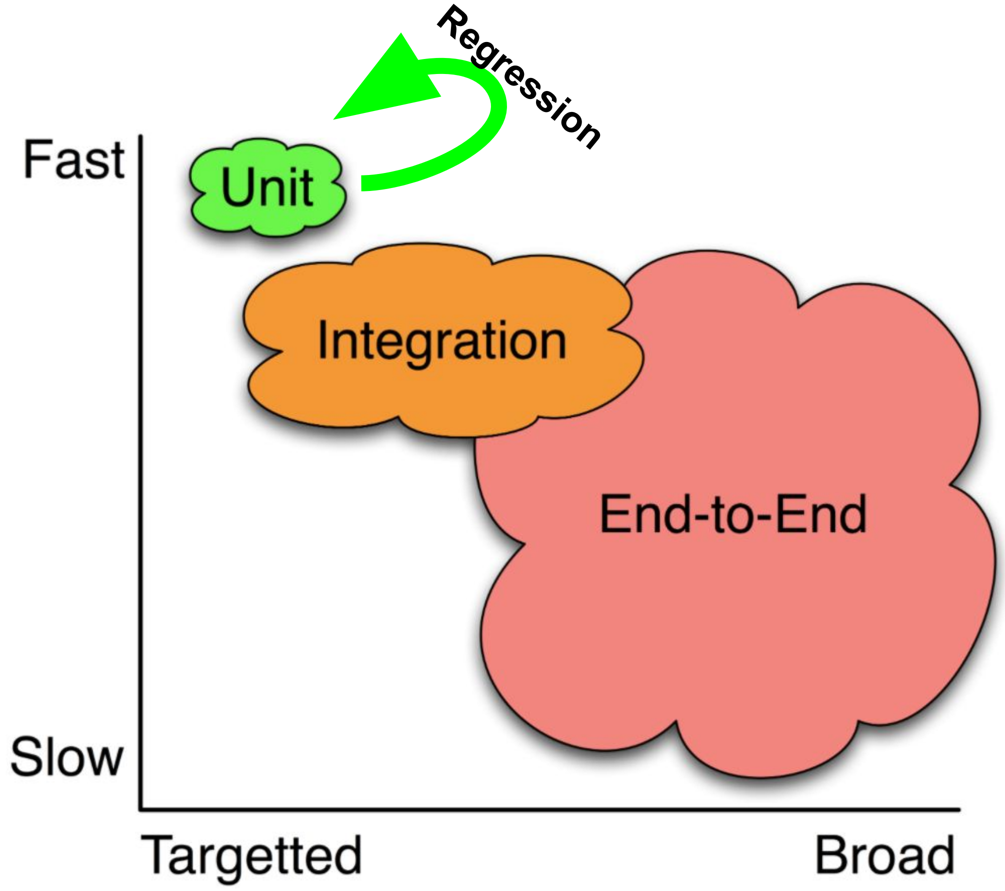


```
IntegerSetTest
  setup(): void
  checkSetContainsOnce(int): void
  checkEmptyDoesntContain(int): void
  testInsertNotAlreadyThere(): void
  testInsertAlreadyThere(): void
  testInsertHighVolume(): void
  testSet: IntegerSetInterface
```

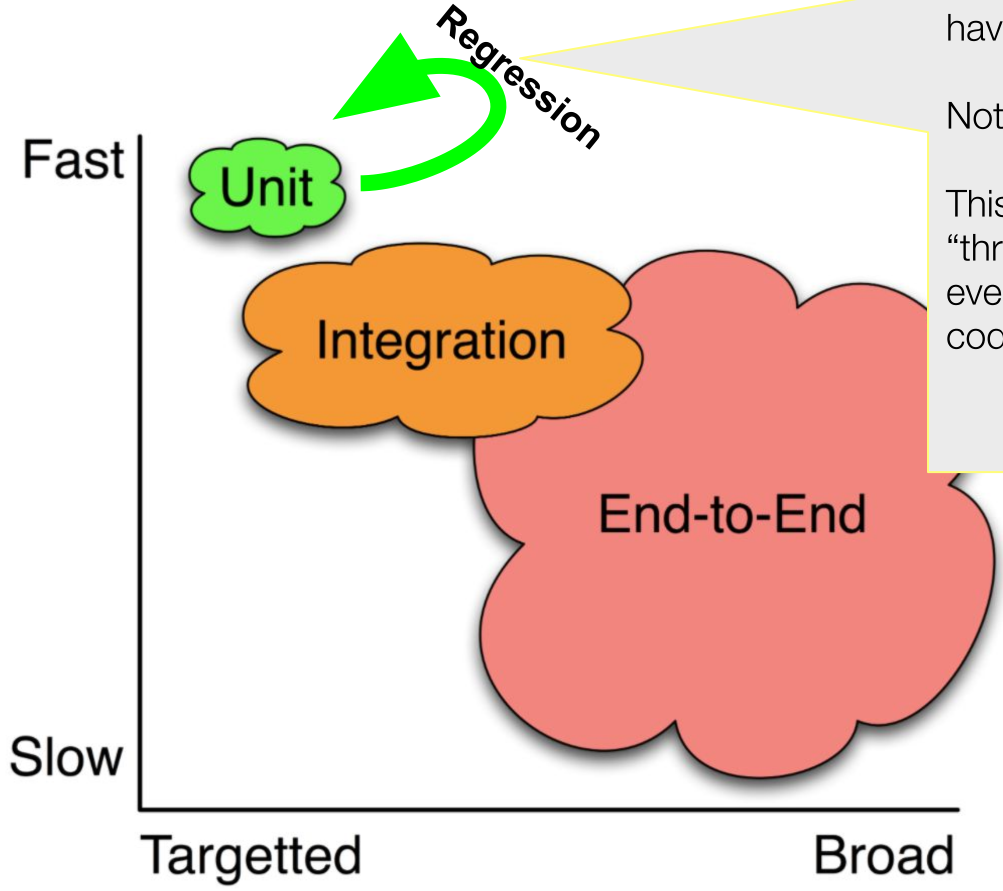
Kinds of Tests



Unit/Smoke Tests



Regression Tests

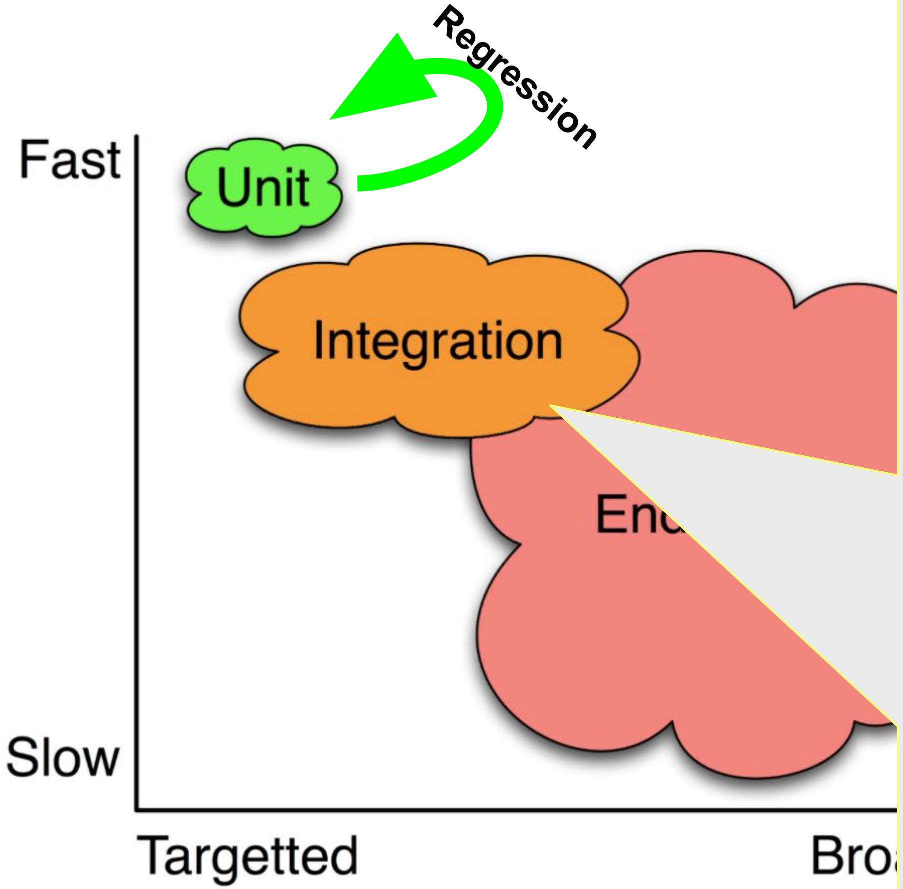


Testing the system to check that changes have not ‘broken’ previously working code.

Not new tests, but repetition of existing tests.

This is why it is so important not to write “throwaway tests”, or test *outside* the spec, even it if works in this specific instance of the codebase.

Integration Tests



Making sure all the parts of the system work together.

If "continuous integration testing" then everyone commits to mainline (a designated branch) every day.

Every commit/merge results in an automated sequence of events: run unit tests, run integration tests, run performance tests.

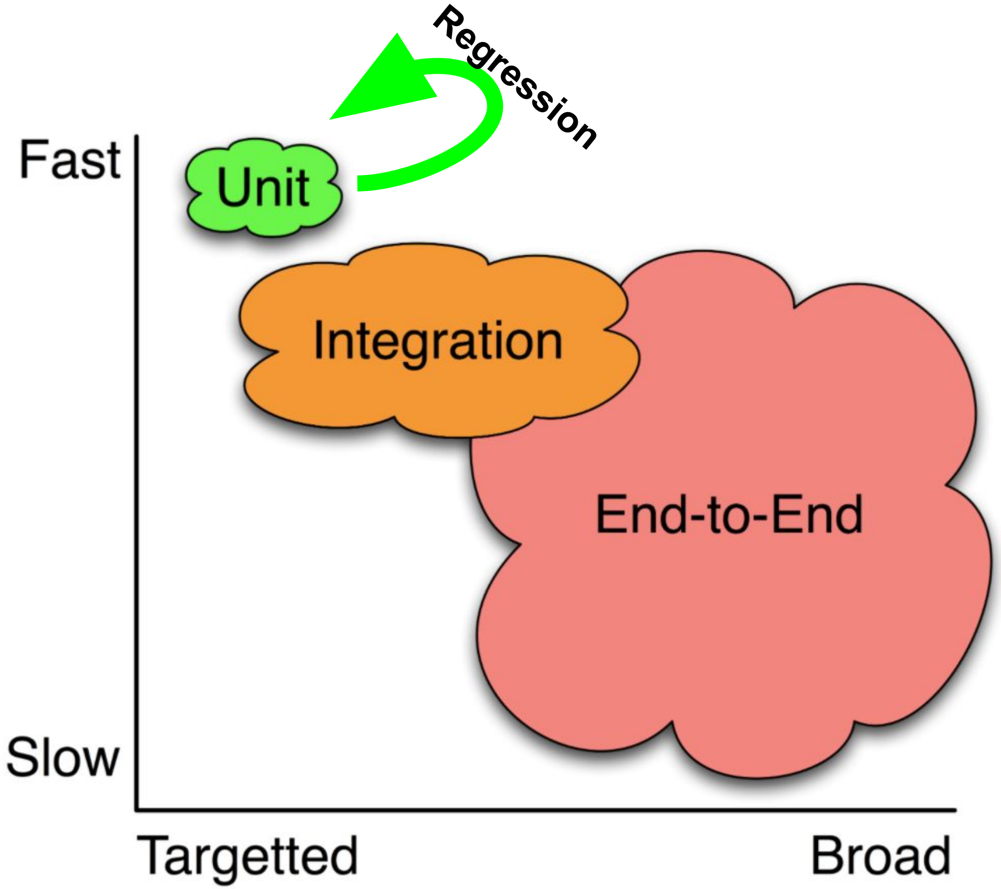
There is direct tool-support for continuous integration.

All software developers must do integration testing, and performing continuous integration testing is a great idea!!

Obviously relies HEAVILY on automated tests.

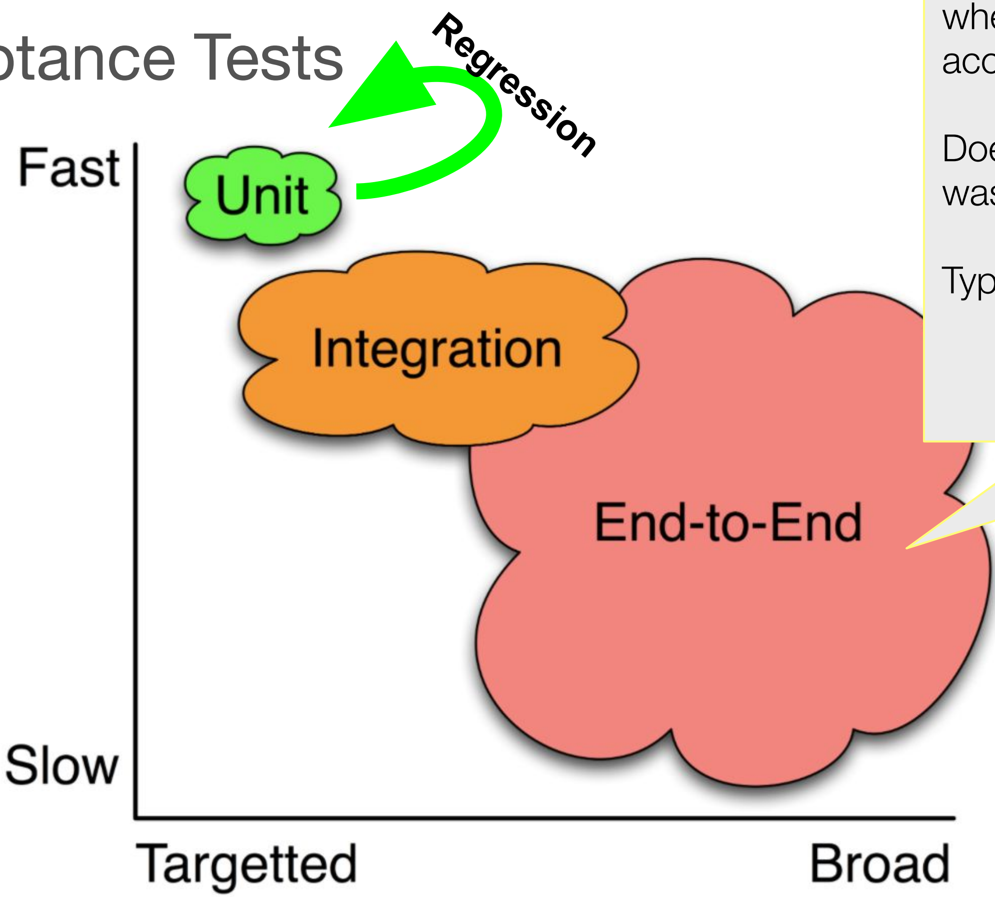
Mock objects get replaced by real ones!

System/End-to-End Tests



System/End-to-End Tests

Acceptance Tests



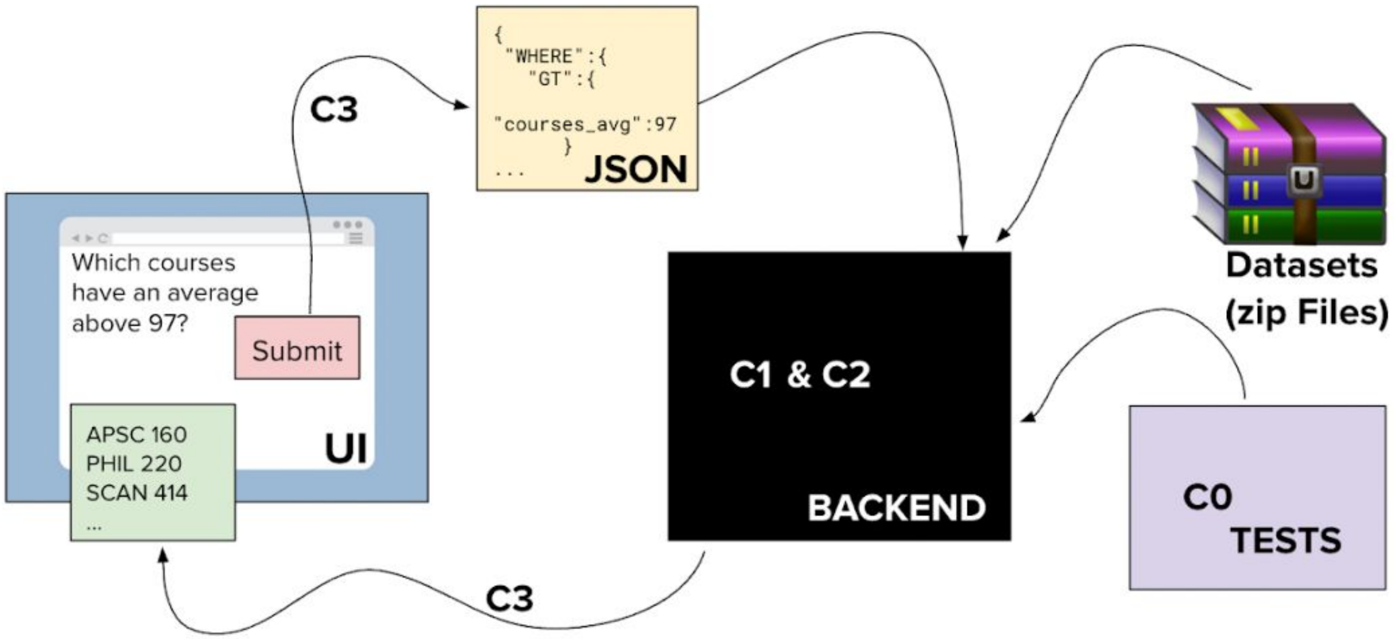
Formal testing with respect to user needs and requirements conducted to determine whether or not the software satisfies the acceptance criteria.

Does the end product solve the problem it was intended to solve?

Typically incremental:

- Alpha test : At production site.
- Beta test : At user's site.

How do Test Types apply to the Project?



How the project is split into the four checkpoints

We’ll go over...

UNIT TESTING

Glass Box (Structural) Testing → Checking the completeness of your suite in various ways, including statement coverage and mutation testing (and then writing tests to fill in the gaps).

Black Box (Functional) Testing → Writing tests based on the specification (using reasoning about equivalence classes, and boundaries).

Test driven design.

Enhancing Testability using Mock Objects.

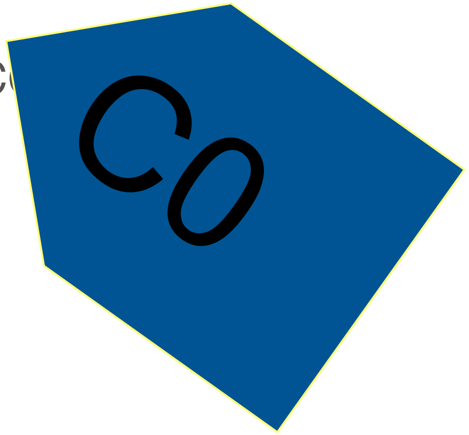
Kinds of Tests → Unit, Acceptance, Integration & Smoke, System.

Designing our test suite

We want to test as much as we can about our system.

We want to consider test inputs to test appropriate scenarios.

We want to get “good coverage” of the code onc



Testing Strategy

- For testing to be effective, it must be intentionally performed.
- This is not an ad hoc process: specifications are carefully examined to determine correct behaviour.
- Creating tests often uncovers incomplete specifications.
- Tests are code too! Since they are expensive to write and maintain, they must be thought of as first-class artifacts.
- White Box testing considers the implementation of a specification.
- Black Box testing does not consider (or even have access to) an implementation.

Black Box Testing

Black Box testing (no access to system internals)

1: Read specification.

2: Write tests.

3: Go to 1.

Black Box testing (no access to system internals)

1: Read specification.

2: Write tests.

3: Go to 1.

On test failure, three options are possible:

- Fix system.
- Fix specification.
- Fix test.

Black Box testing (no access to system internals)

1: Read specification.

2: Write tests.

3: Go to 1.

- ➔ What are the advantages of black box over glass box testing?

Black Box Testing (no access to system internals)

1: Read specification.

2: Write tests.

3: Go to 1.

- ➔ What are the advantages of black box over glass box testing?
  - ◆ Finds high severity bugs: act like a user (follow APIs).
  - ◆ Easy to get started: more clear what to test (client-facing methods).
  - ◆ Less overfitting to implementation: emphasis on specification.
  - ◆ Parallelize development and testing: no need to access implementation.

Black Box Testing (no access to system internals)

1: Read specification.

2: Write tests.

3: Go to 1.

- ➔ What are the advantages of black box over glass box testing?
- ➔ What are the disadvantages of black box testing?

Black Box Testing (no access to system internals)

1: Read specification.

2: Write tests.

3: Go to 1.

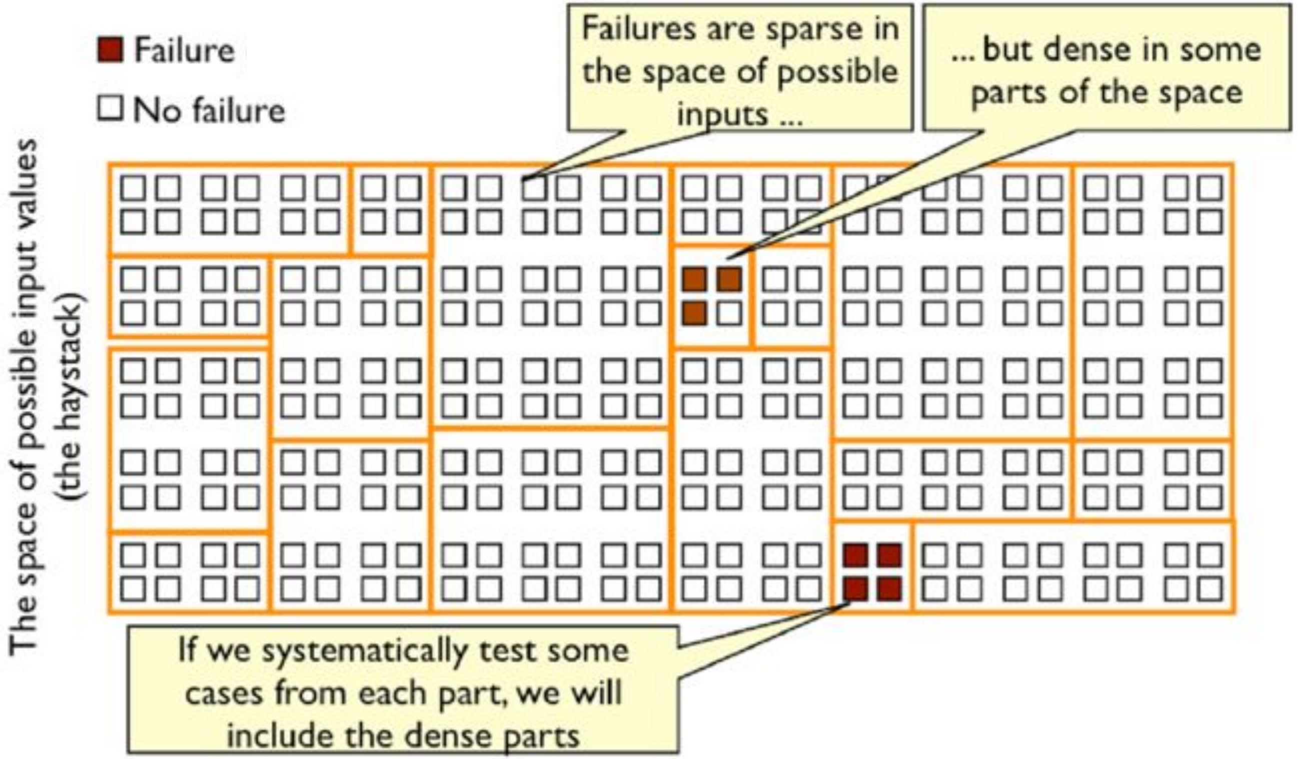
- ➔ What are the advantages of black box over glass box testing?
- ➔ What are the disadvantages of black box testing?
  - ◆ Need well-defined specifications :-)
  - ◆ Poor explainability: test fails… but why?
  - ◆ Intentional handicap: can’t use the implementation!

Black Box Testing (no access to system internals)

When reading a method specification to write tests, look for:

- Input/output equivalence classes:
  - A systematic way of covering the (possibly unbounded) space of inputs/outputs.
  - Help you to avoid redundant tests.
- Boundary cases:
  - Help to find most common errors hiding in edge cases.

Equivalence Classes



■ Failure
□ No failure
The space of possible input values (the haystack)
Failures are sparse in the space of possible inputs ...
... but dense in some parts of the space
If we systematically test some cases from each part, we will include the dense parts

Input/Output Partitioning

- Partitioning of all the possible inputs to CUT into equivalence classes which characterize sets of **inputs** with *something in common.*
- We do *not* consider intentions/actions/outputs of the CUT, except to assert on the correctness of an output for an input.
    - Strength of assertion will depend on how well the spec describes relationship between inputs and outputs.
    - *REQUIRES* clause constrain our input partitioning (they codify assumptions CUT makes about inputs).
    - Input partitioning must also respect PL inputs' types, if known.
    - In general, additional knowledge of CUT may constrain inputs.

Input/Output Partitioning

- Partitioning all possible inputs to CUT into equivalence classes which characterize sets of inputs with *something in common*.

```
// Computes the string representation of the boolean.
//
// If the argument is true, a string equal to "true" is returned;
// otherwise, a string equal to "false" is returned.
public Boolean::toString(): string

e.g., new Boolean(arg).toString();
```

What are the **input** for **toString()**?

Input/Output Partitioning

- Partitioning all possible inputs to CUT into equivalence classes which characterize sets of inputs with *something in common*.

```
// Computes the string representation of the boolean.
//
// If the argument is true, a string equal to “true” is returned;
// otherwise, a string equal to “false” is returned.
public Boolean::toString(): string
```

The Boolean can only take on two possible values. The two input partitions are finite and contain one element each:

- new Boolean(b) where b is true
- new Boolean(b) where b is false

Input/Output Partitioning

- Partitioning all possible inputs to CUT into equivalence classes which characterize sets of inputs with *something in common*.

```
// Computes the string representation of the boolean.
//
// If the argument is true, a string equal to "true" is returned;
// otherwise, a string equal to "false" is returned.
public Boolean::toString(): string
```

The Boolean can only take on two possible values. The two input partitions are finite and contain one element each:

- b is true → expect(new Boolean(true).toString()).to.equal(“true”);
- b is false → expect(new Boolean(false).toString()).to.equal(“false”)

Choose representative from each partition for the concrete test

Input/Output Partitioning

- Partitioning all possible inputs to CUT into equivalence classes which characterize sets of inputs with *something in common*.

```
// Computes the absolute value of a number. If the argument is not
// negative, the argument is returned. If the argument is negative, the
// negation of the argument is returned. If the parameter n is non-coercible
// into a number, NaN is returned.
public static Math::abs(n: number): number
```

Input/Output Partitioning

- Partitioning all possible inputs to CUT into equivalence classes which characterize sets of inputs with *something in common*.

```
// Computes the absolute value of a number. If the argument is not
// negative, the argument is returned. If the argument is negative, the
// negation of the argument is returned. If the parameter a is non-coercible
// into a number, NaN is returned.
public static Math::abs(n: number): number
```

The spec calls out ‘not negative’ and ‘negative’ argument values as well as NaN. We therefore derive the following 3 partitions:

- n is coercible and is a negative value
- n is coercible and is a positive value
- n is non-coercible

Input/Output Partitioning

- Partitioning all possible inputs to CUT into equivalence classes which characterize sets of inputs with *something in common*.

```
// Computes the absolute value of a number. If the argument is not
// negative, the argument is returned. If the argument is negative, the
// negation of the argument is returned. If the parameter a is non-coercible
// into a number, NaN is returned.
public static Math::abs(n: number): number
```

Choose representative from partition for the concrete test

The spec calls out ‘not negative’ and ‘negative’ argument values as well as NaN. We therefore derive the following 3 partitions:

- n is coercible and is a negative value → expect(Math.abs(-1)).to.equal(1);
- n is coercible and is a positive value → expect(Math.abs(1)).to.equal(1);
- n is non-coercible → expect(Number.isNaN(Math.abs({}))).to.be.true;

Input/**Output** Partitioning

- Partitioning all possible outputs to CUT into equivalence classes which characterize sets of outputs with *something in common.*
- Here, we do not consider inputs to CUT, except to assert on the correctness of the output for the given input.
    - Note: output partitioning requires an extra step: to test an output partition we need to formulate inputs that cause an output in that partition. (This may be difficult)

Input/**Output** Partitioning

- Partitioning of all possible outputs to CUT into equivalence classes which characterize sets of outputs with *something in common.*

```
// Computes the absolute value of a number. If the argument is not
// negative, the argument is returned. If the argument is negative, the
// negation of the argument is returned. If the parameter n is non-coercible
// into a number, NaN is returned.
public static Math::abs(n: number): number
```

What are the **output** partitions for **abs**?

Input/**Output** Partitioning

- Partitioning of all possible outputs to CUT into equivalence classes which characterize sets of outputs with *something in common.*

```
// Computes the absolute value of a number. If the argument is not
// negative, the argument is returned. If the argument is negative, the
// negation of the argument is returned. If the parameter a is non-coercible
// into a number, NaN is returned.
public static Math::abs(n: number): number
```

From this spec we expect two kinds of outputs: positive values and NaN.
This gives us two output partitions:

- Output is a positive number
- Output is Number.NaN

Input/**Output** Partitioning

- Partitioning of all possible outputs to CUT into equivalence classes which characterize sets of outputs with *something in common.*

```
// Computes the greater of two int values. That is, the result is the
// argument closer to the value of Number.MAX_VALUE. If the arguments
// have the same value, the result is that same value.
public static Math::max(n: number, m: number): number
```

Input/**Output** Partitioning

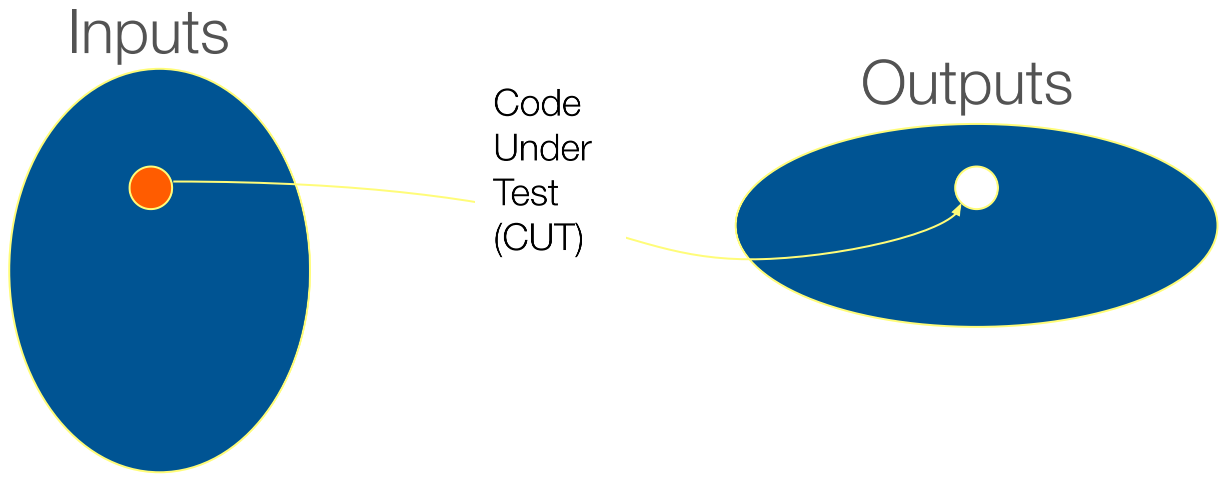
- Partitioning of all possible outputs to CUT into equivalence classes which characterize sets of outputs with *something in common.*

```
// Computes the greater of two int values. That is, the result is the
// argument closer to the value of Number.MAX_VALUE. If the arguments
// have the same value, the result is that same value.
public static Math::max(n: number, m: number): number
```

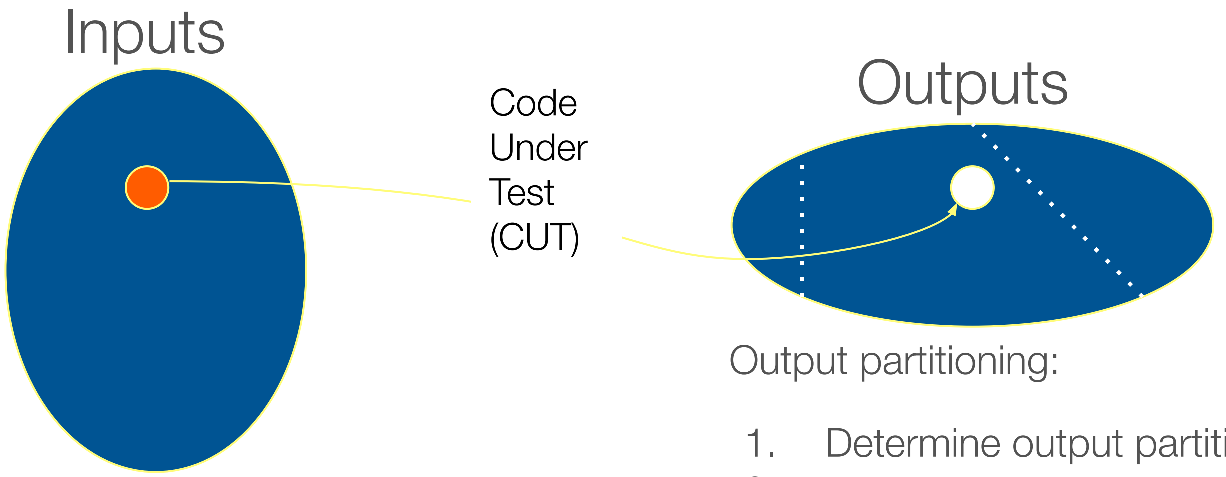
The most we can say about the output of this function is that it must be a number. We don't have anything better than the number type to help us with output partitioning. So, we could partition the outputs into:

- negative values
- 0
- positive values

Relating Input/Output Partitioning



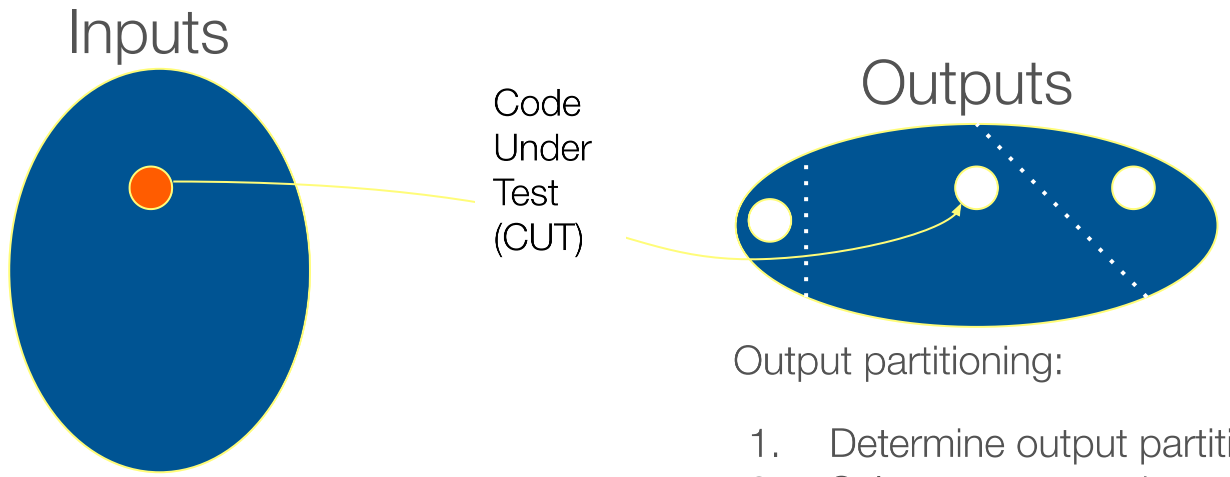
Relating Input/Output Partitioning



Output partitioning:

1. Determine output partitions
2.
3.

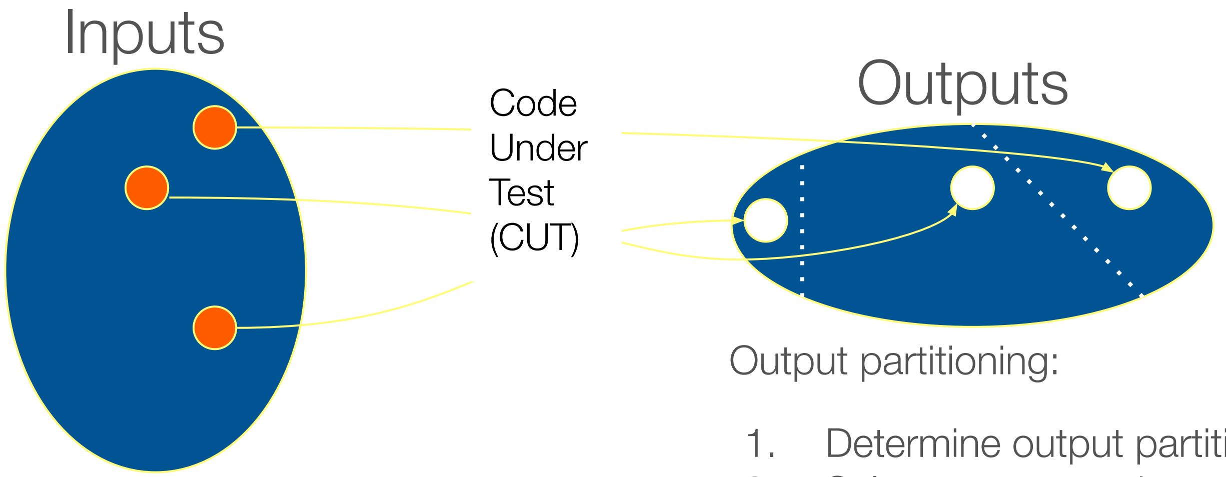
Relating Input/Output Partitioning



Output partitioning:

1. Determine output partitions
2. Select representative outputs per partition
3.

Relating Input/Output Partitioning



Output partitioning:

1. Determine output partitions
2. Select representative outputs per partition
3. Determine inputs that map to outputs

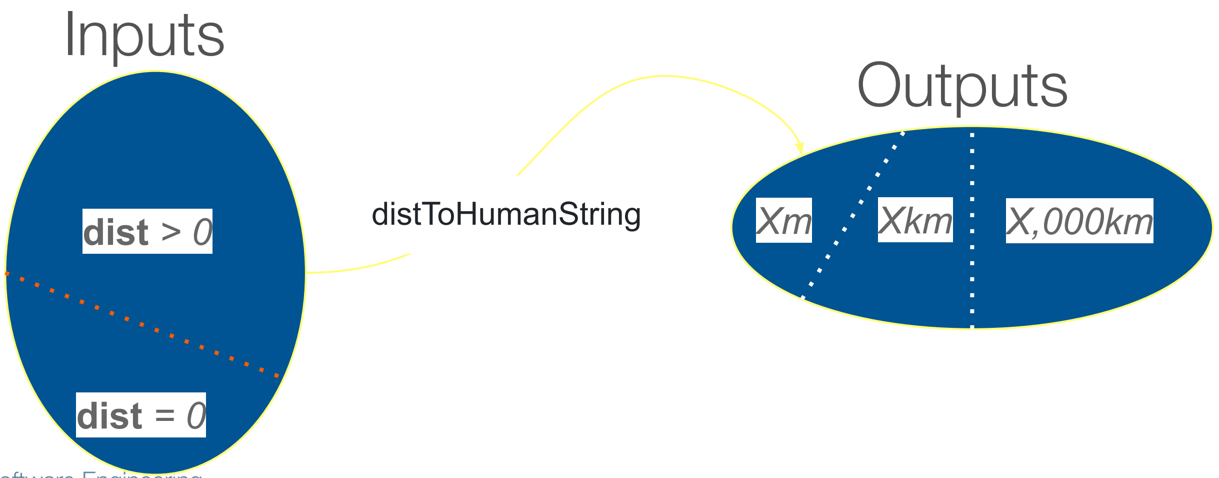
Relating Input/Output Partitioning

Concrete example:

```
// Computes a human readable version of distance.
// Input: distance dist in meters
// Requires: dist >= 0
// Returns: A string representation of dist.
// Example return values: 0m, 100m, 1km, 1.2km, 1,000km
public distToHumanString(dist: number): string
```

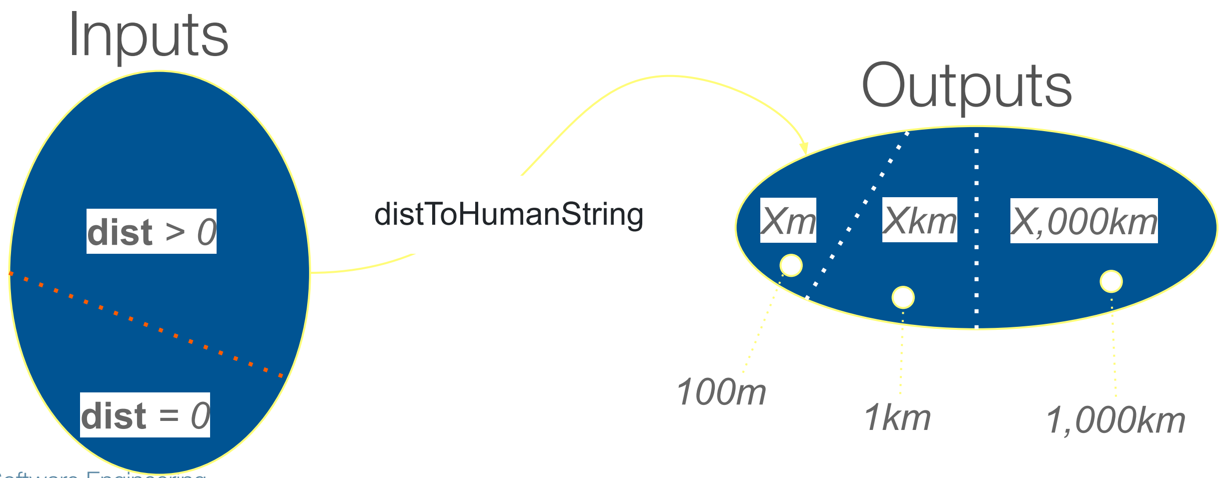
Relating Input/Output Partitioning

```
// Computes a human readable version of distance.
// Input: distance dist in meters
// Requires: dist >= 0
// Returns: A string representation of dist.
// Example return values: 0m, 100m, 1km, 1.2km, 1,000km
public distToHumanString(dist: number): string
```



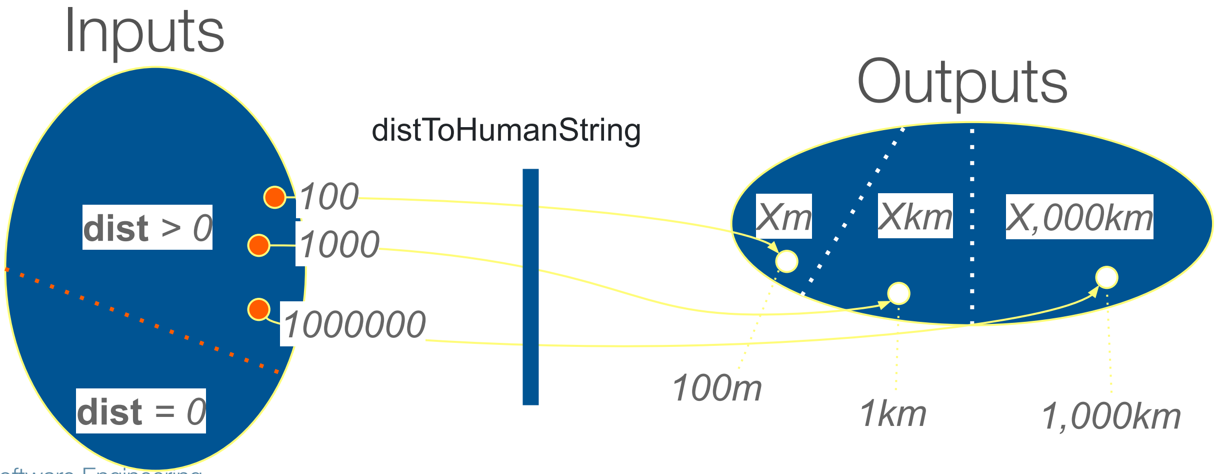
Relating Input/Output Partitioning

```
// Computes a human readable version of distance.
// Input: distance dist in meters
// Requires: dist >= 0
// Returns: A string representation of dist.
// Example return values: 0m, 100m, 1km, 1.2km, 1,000km
public distToHumanString(dist: number): string
```



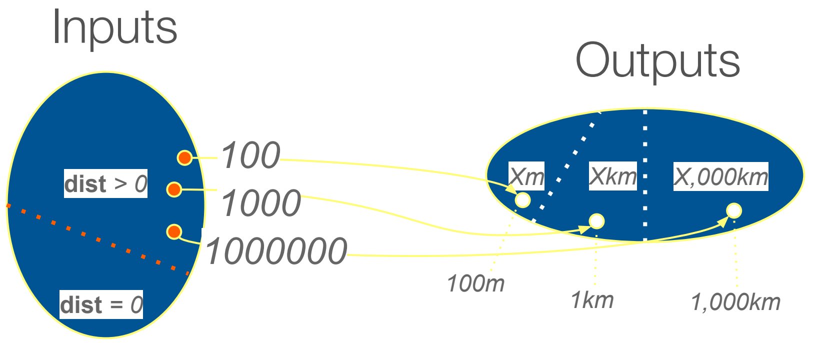
Relating Input/Output Partitioning

```
// Computes a human readable version of distance.
// Input: distance dist in meters
// Requires: dist >= 0
// Returns: A string representation of dist.
// Example return values: 0m, 100m, 1km, 1.2km, 1,000km
public distToHumanString(dist: number): string
```



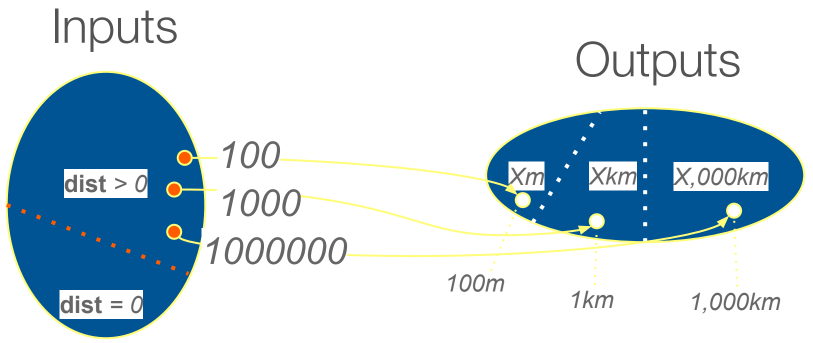
Relating Input/Output Partitioning

```
// Computes a human readable version of distance.
// Input: distance dist in meters
// Requires: dist >= 0
// Returns: A string representation of dist.
// Example return values: 0m, 100m, 1km, 1.2km, 1,000km
public distToHumanString(dist: number): string
```



Relating Input/Output Partitioning

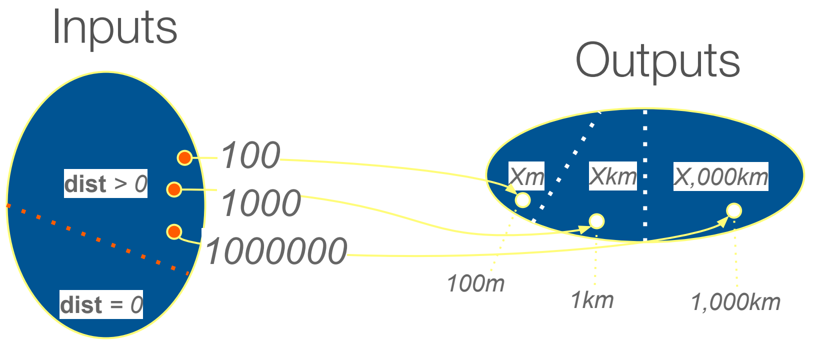
```
// Computes a human readable version of distance.
// Example return values: 0m, 100m, 1km, 1.2km, 1,000km
public distToHumanString(dist: number): string {
    assert(dist>0);
    const output = dist;
    let suffix = null;
    if (dist < 1000) {
        suffix = "m";
    } else {
        suffix += "km";
    }
    // Compute output String
    return (output + suffix);
}
```



Relating Input/Output Partitioning

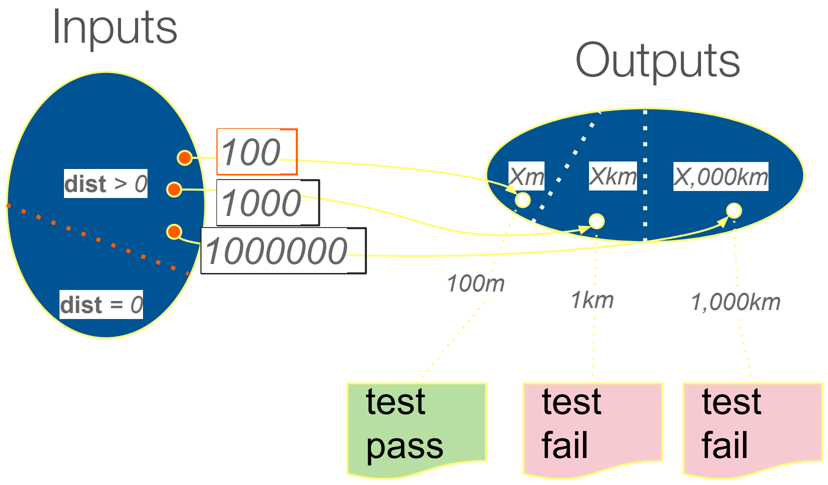
```
// Computes a human readable version of distance.
// Example return values: 0m, 100m, 1km, 1.2km, 1,000km
public distToHumanString(dist: number): string {
    assert(dist>0);
    const output = dist;
    let suffix = null;
    if (dist < 1000) {
        suffix = "m";
    } else {
        suffix += "km";
    }
    // Compute output String
    return (output + suffix);
}
```

Spot the bug(s)



Relating Input/Output Partitioning

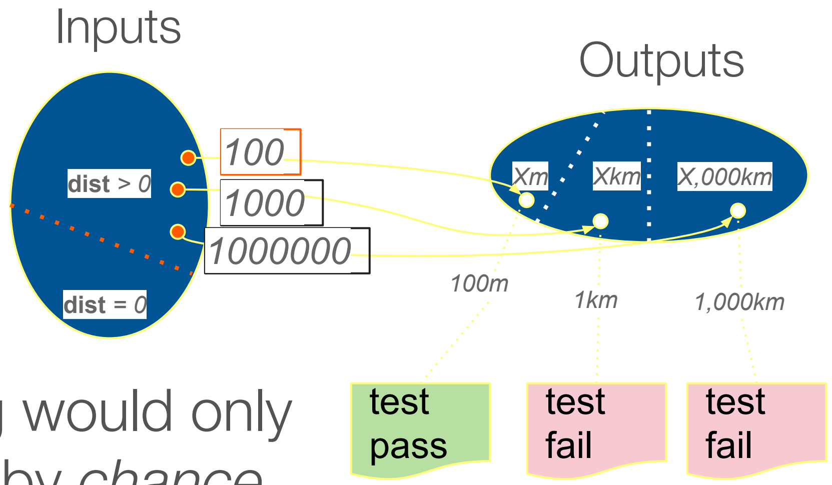
```
// Computes a human readable version of distance.
// Example return values: 0m, 100m, 1km, 1.2km, 1,000km
public distToHumanString(dist: number): string {
    assert(dist>0);
    const output = dist;
    let suffix = null;
    if (dist < 1000) {
        suffix = "m";
    } else {
        suffix += "km";
    }
    // Compute output String
    return (output + suffix);
}
```



Relating Input/Output Partitioning

```
// Computes a human readable version of distance.
// Example return values: 0m, 100m, 1km, 1.2km, 1,000km
public distToHumanString(dist: number): string {
    assert(dist>0);
    const output = dist;
    let suffix = null;
    if (dist < 1000) {
        suffix = "m";
    } else {
        suffix += "km";
    }
    // Compute output String
    return (output + suffix);
}
```

In this case output partitioning is better at finding the bug.



Input partitioning would only find these bugs by *chance*.

Black Box Testing

When reading a method specification to write tests, look for:

- Input/output equivalence classes:
  - A strategic way of covering the (often unbounded) space of inputs/outputs.
  - Help you to avoid redundant tests.
- Boundary cases:
  - Help to find most common errors hiding in edge cases.

Boundary testing

Typos like <= instead of < will generate a LOT of bugs! So test on either side of each boundary!

```
// Computes mortgage interest rate based on credit score.
// Input: credit score between 30 and 85, or throws
// Returns: the interest rate bucket:
//   - high for poor credit scores up to 69
//   - medium for fair credit scores between 70 and 73
//   - low for good credit scores 74 and over
public getMortgageRate(score: number): “low” | “medium” | “high”
```

Boundary testing

Typos like <= instead of < will generate a LOT of bugs! So test on either side of each boundary!

```
// Computes mortgage interest rate based on credit score.
// Input: credit score between 30 and 85, or throws
// Returns: the interest rate bucket:
//    - high for poor credit scores up to 69
//    - medium for fair credit scores between 70 and 73
//    - low for good credit scores 74 and over
public getMortgageRate(score: number): “low” | “medium” | “high”
```

[29, 30, 31]

Boundary testing

Typos like <= instead of < will generate a LOT of bugs! So test on either side of each boundary!

```
// Computes mortgage interest rate based on credit score.
// Input: credit score between 30 and 85, or throws
// Returns: the interest rate bucket:
//    - high for poor credit scores up to 69
//    - medium for fair credit scores between 70 and 73
//    - low for good credit scores 74 and over
public getMortgageRate(score: number): “low” | “medium” | “high”
```

[29, 30, 31]

Boundary testing

Typos like <= instead of < will generate a LOT of bugs! So test on either side of each boundary!

```
// Computes mortgage interest rate based on credit score.
// Input: credit score between 30 and 85, or throws
// Returns: the interest rate bucket:
//    - high for poor credit scores up to 69
//    - medium for fair credit scores between 70 and 73
//    - low for good credit scores 74 and over
public getMortgageRate(score: number): “low” | “medium” | “high”
```

[29, 30, 31]

[84, 85, 86]

Boundary testing

Typos like <= instead of < will generate a LOT of bugs! So test on either side of each boundary!

```
// Computes mortgage interest rate based on credit score.
// Input: credit score between 30 and 85, or throws
// Returns: the interest rate bucket:
//    - high for poor credit scores up to 69
//    - medium for fair credit scores between 70 and 73
//    - low for good credit scores 74 and over
public getMortgageRate(score: number): “low” | “medium” | “high”
```

[29, 30, 31]

[68, 69, 70]

[84, 85, 86]

Boundary testing

Typos like <= instead of < will generate a LOT of bugs! So test on either side of each boundary!

```
// Computes mortgage interest rate based on credit score.
// Input: credit score between 30 and 85, or throws
// Returns: the interest rate bucket:
//    - high for poor credit scores up to 69
//    - medium for fair credit scores between 70 and 73
//    - low for good credit scores 74 and over
public getMortgageRate(score: number): "low" | "medium" | "high"
```

[29, 30, 31] [69, 70, 71] [84, 85, 86]

[68, 69, 70] [73, 74, 75] [72, 73, 74]

Boundary testing

Typos like <= instead of < will generate a LOT of bugs! So test on either side of each boundary!

```
// Computes mortgage interest rate based on credit score.
// Input: credit score between 30 and 85, or throws
// Returns: the interest rate bucket:
//    - high for poor credit scores up to 69
//    - medium for fair credit scores between 70 and 73
//    - low for good credit scores 74 and over
public getMortgageRate(score: number): "low" | "medium" | "high"
```

[29, 30, 31]   [68, 69, 70]   [69, 70, 71]   [73, 74, 75]   [72, 73, 74]   [84, 85, 86]

Cases: [**29, 30, 31, 68, 69, 70, 71, 72, 73, 74, 75, 84, 85, 86**]

Boundary testing

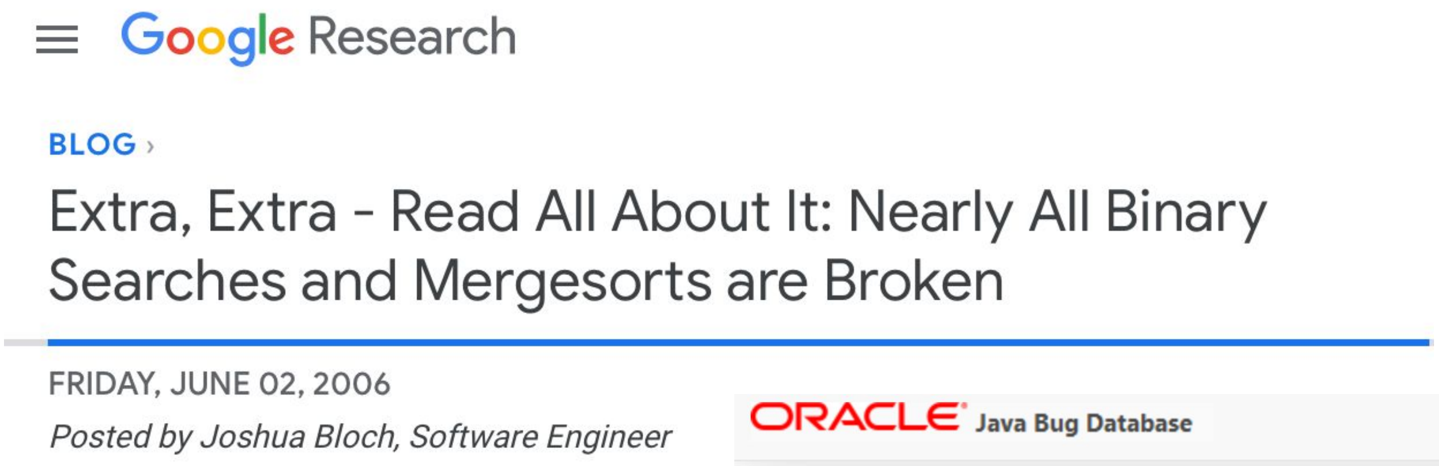
Typos like <= instead of < will generate a LOT of bugs! So test on either side of each boundary!

```
// Computes mortgage interest rate based on credit score.
// Input: credit score between 30 and 85, or throws
// Returns: the interest rate bucket:
//    - high for poor credit scores up to 69
//    - medium for fair credit scores between 70 and 73
//    - low for good credit scores 74 and over
public getMortgageRate(score: number): “low” | “medium” | “high”
```

Cases: $[29, \underbrace{30, 31, 68, 69,} \underbrace{70, 71, 72, 73,} \underbrace{74, 75, 84, 85,} 86]$

Expect: **error** **high** **medium** **low** **error**

Boundary testing



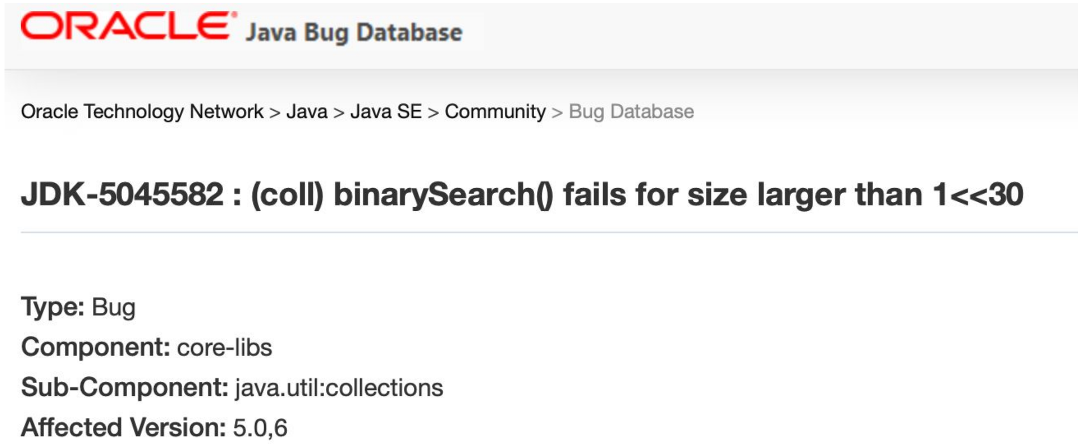
Google Research

BLOG ›

Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken

FRIDAY, JUNE 02, 2006

Posted by Joshua Bloch, Software Engineer



ORACLE Java Bug Database

Oracle Technology Network > Java > Java SE > Community > Bug Database

JDK-5045582 : (coll) binarySearch() fails for size larger than 1<<30

Type: Bug
Component: core-libs
Sub-Component: java.util:collections
Affected Version: 5.0,6

[https://blog.research.google/2006/06/extra-extra-read-all-about-it-nearly.html]

Boundary testing

```java
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
             low = mid + 1
         else if (midVal > key)
             high = mid - 1;
         else
             return mid; // key found
     }
     return -(low + 1);  // key not found.
 }
```

[https://blog.research.google/2006/06/extra-extra-read-all-about-it-nearly.html]

Boundary testing

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mi
        int mi

        if (mi
            l
         else
            h
         else
            return mid; // key found
     }
     return -(low + 1);  // key not found.
 }
```

"The general lesson that I take away from this bug is humility: It is hard to write even the smallest piece of code correctly [..] We must program carefully, defensively, and remain ever vigilant."

- *Joshua Bloch*

[https://blog.research.google/2006/06/extra-extra-read-all-about-it-nearly.html]

Assertions

From execution to validated behavior

- Just because you can execute a SUT, does not mean you can evaluate the correctness of its behavior.
  - The only exception is crashing bugs (never desired).
- Assertions bridge this gap:
  - We assert on the behavior of the execution.
  - Want to consider both **valid** and **invalid** behaviors:
    - `assert(Valid)`: good thing happened.
    - `assert(!Invalid)`: bad thing did not happen.

equality

Assertions

above, below, near

equality

type

set membership

throws

contains

key existence (+ any, + all)

Strength of assertion

```
const actual = exists('cpsc310project');
const expected = true;
```

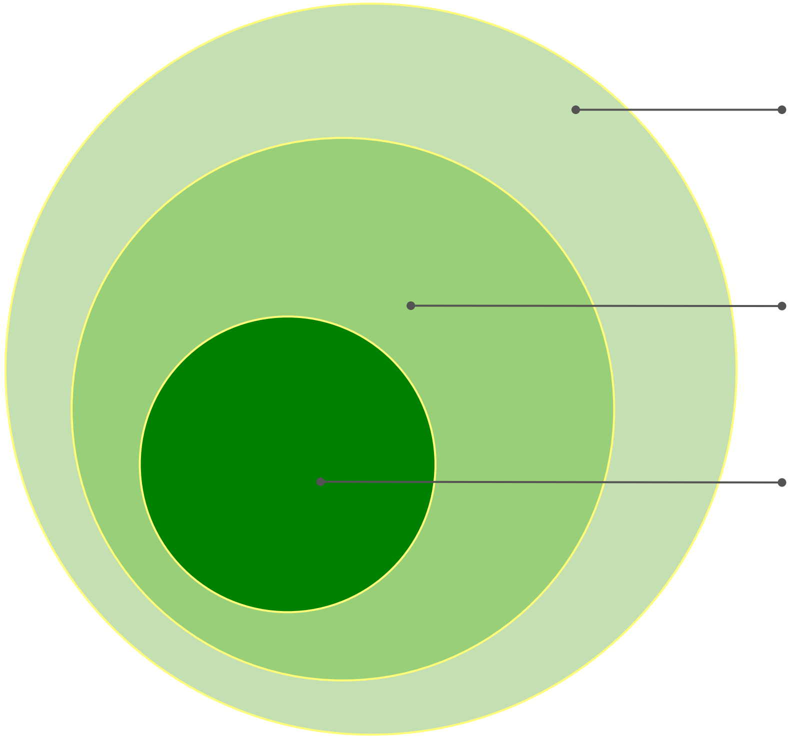
```
expect(actual).to.be.defined;
```

```
expect(actual).to.be.a.boolean;
```

```
expect(actual).to.equal(expected);
```

Strength of assertion

```
const actual = exists('cpsc310project');
const expected = true;
```



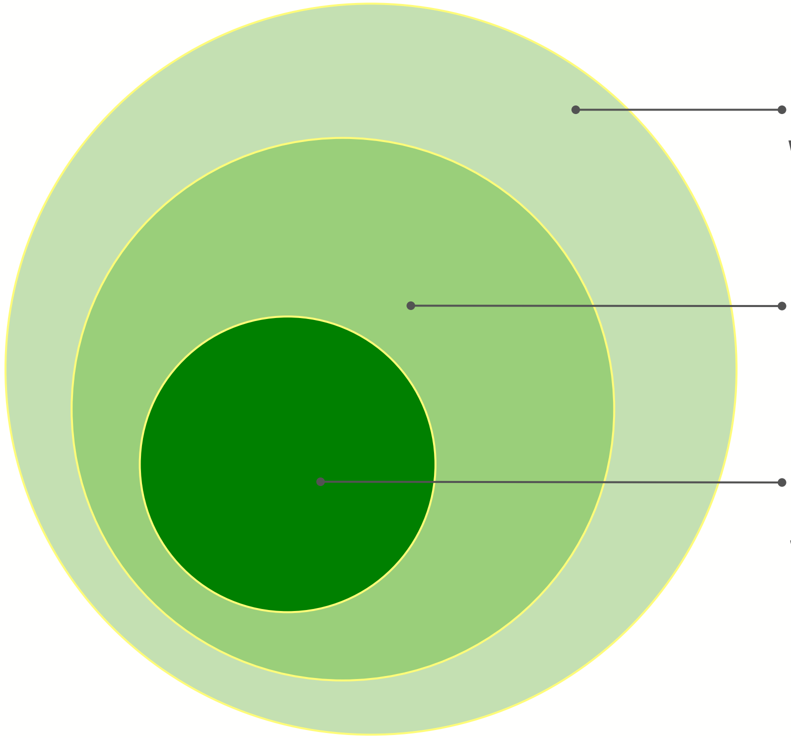
```
expect(actual).to.be.defined;

expect(actual).to.be.a.boolean;

expect(actual).to.equal(expected);
```

Strength of assertion

```
const actual = exists('cpsc310project');
const expected = true;
```



```
expect(actual).to.be.defined;
```

Weaker = more broad

```
expect(actual).to.be.a.boolean;
```

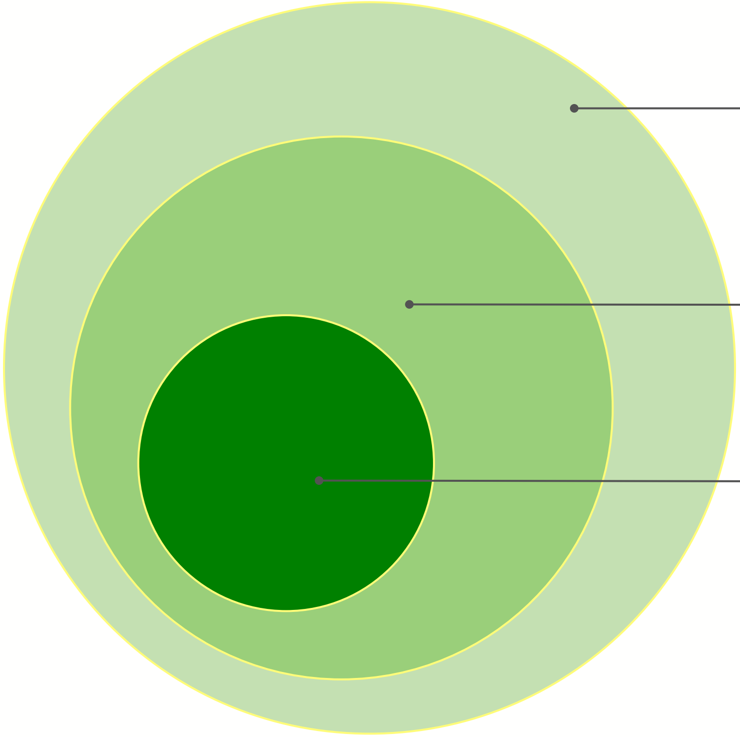
```
expect(actual).to.equal(expected);
```

Stronger = more narrow

Strength of assertion

```
const actual = exists('cpsc310project');
const expected = true;
```

Automated blackbox tests (e.g., fuzzing), typically have weaker assertions



```
expect(actual).to.be.defined;
```

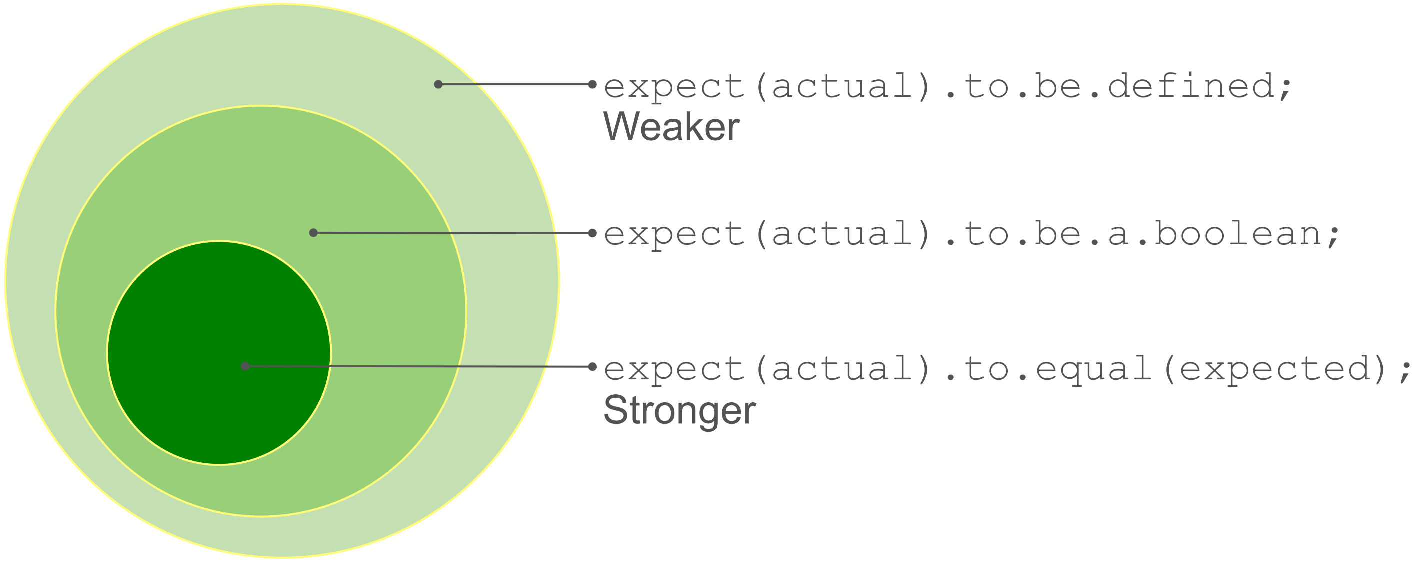
Weaker = more broad

```
expect(actual).to.be.a.boolean;
```

```
expect(actual).to.equal(expected);
```

Stronger = more narrow

Strength of assertion



`expect(actual).to.be.defined;`
Weaker
`expect(actual).to.be.a.boolean;`
`expect(actual).to.equal(expected);`
Stronger

- Assertions that are stronger, usually closely follow the spec.
- Weaker assertions are still useful to help with debugging.
  - E.g., *actual* not equal to *expected…* but why? Not a *boolean*!

Input value checks

```
// checks if a repo exist in GitHub
exists(repoName: string): boolean

expect(exists('')).to.equal(false)
expect(exists(undefined)).to.throw(Error)
expect(exists(null)).to.throw(Error)
expect(exists('null')).to.equal(false)
```

What about this one? (5 mins)

```
createTeam(org: string,
                          name: string,
                          members: string[]): Team
```

What about this one? (5 mins)

```
createTeam(org: string,
                        name: string,
                        members: string[]): Team
```

```
expect(createTeam(bad-inputs)).to.be.null
expect(createTeams(good-inputs)).to.be.an('object')
expect(createTeams(good-inputs).id).to.be.defined
expect(createTeams(good-inputs).id).to.be.positive
```

What about this one? (5 mins)

```
createTeam(org: string,
                        name: string,
                        members: string[]): Team
```

```
// too many options to be exhaustive
createTeam('org','', ['rtholmes'])
createTeam('org', ['rtholmes'])
createTeam('org','', [1,2])
createTeam('org','', {})
createTeam({},'', ['rtholmes'])
createTeam(null,'', ['rtholmes'])
createTeam('org','', ['rtholmes','rthse2','fooz'])
```