Testing Basics

Martin Kellogg

Testing Basics

Today's agenda:

- **Reading Quiz**
- What is testing?
- How to write tests
- Different kinds of tests and how to use them
- Continuous integration
- Test quality
- Test suite quality

Reading Quiz: Testing Basics

Get out a scrap of paper and write your name and UCID. You will have two minutes to answer two questions about the readings. The quiz will be scored out of 4: 2 points for your name/UCID (i.e., attendance) and 1 point for each correct answer.

NOTE: do **not** write a number starting with 31… (i.e., your student ID). You will not receive credit for the quiz if you do. Correctly writing your name looks like this:

Martin Kellogg (mjk76)

Reading Quiz: Testing Basics

Q1: **TRUE** or **FALSE**: test-driven development encourages you to write seemingly-trivial implementations of methods (e.g., a “count()” method that always returns the literal 0) quickly, just to get the tests running

Q2: SQLite maintains which of the following coverage metrics (select all that apply):

A. 100% statement coverage

B. 100% branch coverage

C. 100% condition (MC/DC) coverage

Reading Quiz: Testing Basics

Q1: **TRUE** or **FALSE**: test-driven development encourages you to write seemingly-trivial implementations of methods (e.g., a "count()" method that always returns the literal 0) quickly, just to get the tests running

Q2: SQLite maintains which of the following coverage metrics (select all that apply):

A. 100% statement coverage
B. 100% branch coverage
C. 100% condition (MC/DC) coverage

Reading Quiz: Testing Basics

Q1: **TRUE** or **FALSE**: test-driven development encourages you to write seemingly-trivial implementations of methods (e.g., a "count()" method that always returns the literal 0) quickly, just to get the tests running

Q2: SQLite maintains which of the following coverage metrics (select all that apply):

A. 100% statement coverage
B. 100% branch coverage
C. 100% condition (MC/DC) coverage

Reading Quiz: Testing Basics

Q1: **TRUE** or **FALSE**: test-driven development encourages you to write seemingly-trivial implementations of methods (e.g., a “count()” method that always returns the literal 0) quickly, just to get the tests running

Q2: SQLite maintains which of the following coverage metrics (select all that apply):

A. 100% statement coverage
B. 100% branch coverage
C. 100% condition (MC/DC) coverage

Reading Quiz: Testing Basics

Q1: TRUE or **FALSE**: test-driven development encourages you to write seemingly-trivial implementations of methods (e.g., a “count()” method that always returns the literal 0) quickly, just to get the tests running

Q2: SQLite maintains which of the following coverage metrics (select all that apply):

A. 100% statement coverage
B. 100% branch coverage
C. 100% condition (MC/DC) coverage

Testing Basics

Today's agenda:

- Reading Quiz
- **What is testing?**
- How to write tests
- Different kinds of tests and how to use them
- Continuous integration
- Test quality
- Test suite quality

What is testing?

Definition: a ***test*** executes a given input on a program (the ***system under test*** or ***SUT***) and compares the SUT's output to a given oracle

What is testing?

Definition: a ***test*** executes a given input on a program (the ***system under test*** or ***SUT***) and compares the SUT's output to a given oracle

Aside: testing is the canonical example of a ***dynamic analysis***, which is program analysis that requires running the program

What is testing?

Definition: a *test* executes a given input on a program (the ***system under test*** or ***SUT***) and compares the SUT's output to a given oracle

```
./prog < input > output && diff output oracle
```

What is testing?

Definition: a ***test*** executes a given input on a program (the ***system under test*** or ***SUT***) and compares the SUT's output to a given oracle

```
./prog < input > output && diff output oracle
```

SUT

What is testing?

Definition: a *test* executes a **given input** on a program (the *system under test* or *SUT*) and compares the SUT's output to a given oracle

```
./prog < input > output && diff output oracle
```

input

What is testing?

Definition: a ***test*** executes a **given input** on a program (the ***system under test*** or ***SUT***) and compares the SUT's **output** to a given oracle

```
./prog < input > output && diff output oracle
```

output

What is testing?

Definition: a *test* executes a **given input** on a program (the ***system under test*** or ***SUT***) and **compares** the SUT's **output** to a given oracle

```
./prog < input > output && diff output oracle
```

comparator

What is testing?

Definition: a *test* executes a **given input** on a program (the *system under test* or *SUT*) and **compares** the SUT's **output** to a given **oracle**

```
./prog < input > output && diff output oracle
```

oracle

What is testing?

Definition: a *test* executes a **given input** on a program (the *system under test* or *SUT*) and **compares** the SUT's **output** to a given **oracle**

```
./prog < input > output && diff output oracle
```

SUT **input** **output** **comparator** **oracle**

Building a test case

- You usually know the SUT

Building a test case

- You usually know the SUT
- **You choose** inputs (**how?**)

Building a test case

- You usually know the SUT
- You choose inputs (how?)
- Run the SUT on the chosen inputs to produce the output

Building a test case

- You usually know the SUT
- You choose inputs (how?)
- Run the SUT on the chosen inputs to produce the output
- You choose the comparator (how?)

Building a test case

- You usually know the SUT
- You choose inputs (how?)
- Run the SUT on the chosen inputs to produce the output
- You choose the comparator (how?)
- You choose the oracle (how?)

Building a test case

- You usually know the SUT
- **You choose** inputs (**how?**)
- Run the SUT on the chosen inputs to produce the output
- **You choose** the comparator (**how?**)
- **You choose** the oracle (**how?**)

Ideal situation: you can test every input ("**exhaustive testing**")

Building a test case

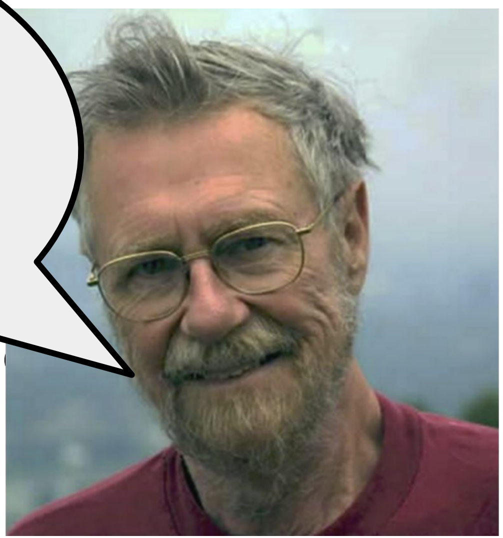
- You usually know the SUT
- You choose inputs (how?)
- Run the SUT on the chosen inputs to produce the output
- You choose the comparator (how?)
- You choose the oracle (how?)

Ideal situation: you can test every input ("**exhaustive testing**")

- in practice, rarely possible: **input space is too large**

Building a test cas

> "Tests can show the presence of bugs, but not their absence"



- You usually know th
- You choose inputs (ho
- Run the SUT on the chosen produ
- You choose the comparator (how?)
- You choose the oracle (how?)

Ideal situation: you can test every input ("exhaustive testing")

- in practice, rarely possible: input space is too large

Building a test case

- You usually know the SUT
- You choose inputs (how?)
- Run the SUT on the chosen inputs to produce the output
- You choose the comparator (how?)
- You choose the oracle (how?)

Ideal situation: you can test every input ("

- in practice, rarely possible: input spac

We'll talk about these out of order:

- comparators
- oracles
- inputs

Testing Basics

Today's agenda:

- Reading Quiz
- What is testing?
- **How to write tests**
- Different kinds of tests and how to use them
- Continuous integration
- Test quality
- Test suite quality

Choosing a comparator

- Most common: **exact match** (often a good choice!)

Choosing a comparator

- Most common: exact match (often a good choice!)
- Also common:
  - over-approximation (“is the output one of these good values”, or, more commonly, “is there any output at all”)

Choosing a comparator

- Most common: exact match (often a good choice!)
- Also common:
  - over-approximation (“is the output one of these good values”, or, more commonly, “is there any output at all”)
  - under-approximation (“does the output contain this expected value”)

Choosing a comparator

- Most common: **exact match** (often a good choice!)
- Also common:
  - **over-approximation** (“is the output one of these good values”, or, more commonly, “is there any output at all”)
  - **under-approximation** (“does the output contain this expected value”)
- But, could be an **arbitrarily-complex boolean** function
  - must be boolean, because test needs to either **pass** or **fail**

Choosing a comparator

- Most common: exact match (often a g
- Also common:
  - **over-approximation** ("is the outp
    values", or, more commonly, "is th
  - **under-approximation** ("does the
    expected value")
- But, could be an **arbitrarily-complex boolean** function
  - must be boolean, because test needs to either pass or **fail**

Choosing a comparator is easy for programs that read and write text. For programs that e.g., have a GUI, this can be a very difficult problem.

Choosing an oracle

- As a human, you get this from the **specification**
  - ask yourself: “what should the program do with this input?”

Choosing an oracle

- As a human, you get this from the **specification**
  - ask yourself: “what should the program do with this input?”
  - advice: always **write down the oracle**
    - common (low quality) oracle: add a `printf` statement to the program, run it, check by hand that the output is what you expect

Choosing an oracle

- As a human, you get this from the **specification**
  - ask yourself: "what should the program do
  - advice: always **write down the oracle**
    - common (low quality) oracle: add a `printf` statement to the program, run it, check by hand that the output is what you expect

Don't do this!
Why not?

Choosing an oracle

- As a human, you get this from the **specification**
  - ask yourself: “what should the program do with this input?”
  - advice: always **write down the oracle**
    - common (low quality) oracle: add a `printf` statement to the program, run it, check by hand that the output is what you expect
- Choosing an oracle automatically is **very hard**
  - key problem in automated test generation
  - we’ll talk about this in more detail later

Choosing inputs

- When writing tests by hand, this is often the hardest part

Choosing inputs

- When writing tests by hand, this is often the hardest part
- Strategies:
  - choose at random (avoid when writing by hand)
  - edge cases
  - partition testing
  - white-box testing
  - black-box testing

Choosing inputs

- When writing tests by hand, this is often the hardest part
- Strategies:
  - choose at random (avoid when writing by hand)
  - **edge cases**
  - partition testing
  - white-box testing
  - black-box testing

Edge case examples:

- 0, 1, -1
- null
- empty list
- empty file
- etc.

Choosing inputs

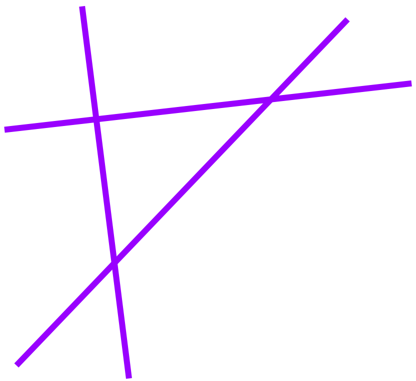
- When writing tests by hand, this is often the hardest part
- Strategies:
    - choose at random (avoid when writing by hand)
    - edge cases
    - **partition testing**
    - white-box testing
    - black-box testing

Partition testing

Key idea: split up the input space into redundant “regions”

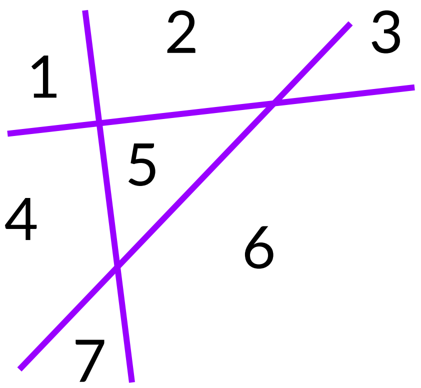
Partition testing

Key idea: split up the input space into redundant "regions"



Partition testing

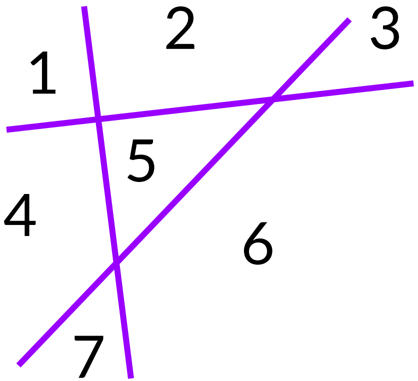
Key idea: split up the input space into redundant "regions"



- write one test **for each region**

Partition testing

Key idea: split up the input space into redundant "regions"

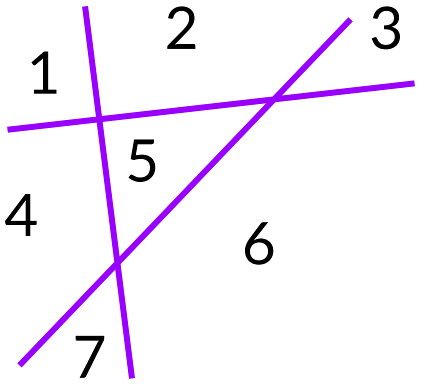


- write one test **for each region**
- possible ways to split up the input:
  - parity (even, odd)
  - positive, negative, zero
  - jpg files vs png files
  - correctly-formatted input vs incorrectly-formatted input

Partition testing

Key idea: split up the input space into

Common technique: split up input space k ways, write 2^k tests



- write one test **for each region**
- possible ways to split up the input:
  - parity (even, odd)
  - positive, negative, zero
  - jpg files vs png files
  - correctly-formatted input vs incorrectly-formatted input

Choosing inputs

- When writing tests by hand, this is often the hardest part
- Strategies:
  - choose at random (avoid when writing by hand)
  - edge cases
  - partition testing
  - **white-box testing**
  - black-box testing

White-box testing

Key idea: choose test inputs based on the program’s source code

White-box testing

Key idea: choose test inputs based on the program's source code

- for example, if there is a condition like `if (x > 10) { … }`, consider choosing 9 and 11 for `x`

White-box testing

Key idea: choose test inputs based on the program's source code

- for example, if there is a condition like `if (x > 10) { … }`, consider choosing 9 and 11 for `x`

Advantages:

White-box testing

Key idea: choose test inputs based on the program's source code

- for example, if there is a condition like `if (x > 10) { … }`, consider choosing 9 and 11 for `x`

Advantages:

- relies primarily on your programming skill
- lets us achieve high coverage (next lecture)

White-box testing

Key idea: choose test inputs based on the program's source code

- for example, if there is a condition like `if (x > 10) { … }`, consider choosing 9 and 11 for `x`

Advantages:

- relies primarily on your programming skill
- lets us achieve high coverage (next lecture)

Disadvantages:

White-box testing

Key idea: choose test inputs based on the program's source code

- for example, if there is a condition like `if (x > 10) { … }`, consider choosing 9 and 11 for `x`

Advantages:

- relies primarily on your programming skill
- lets us achieve high coverage (next lecture)

Disadvantages:

- you have to actually read the code
- easy to accidentally "bias" yourself towards what the code already does

Choosing inputs

- When writing tests by hand, this is often the hardest part
- Strategies:
    - choose at random (avoid when writing by hand)
    - edge cases
    - partition testing
    - white-box testing
    - **black-box testing**

Black-box testing

Key idea: don't look at what the code does at all! Instead, only consider the code's **interface**: the inputs and outputs.

Black-box testing

Key idea: don’t look at what the code does at all! Instead, only consider the code’s **interface**: the inputs and outputs.

- key advantage over white-box testing: you aren’t biased by the implementation

Black-box testing

Key idea: don't look at what the code does at all! Instead, only consider the code's **interface**: the inputs and outputs.

- key advantage over white-box testing: you aren't biased by the implementation
- key disadvantage vs white-box testing: you can't tell how well you're covering the implementation, only the specification
  - hard to choose edge cases, get high coverage, etc.

Black-box testing

Key idea: don't look at what the code does at all! Instead, only consider the code's **interface**: the inputs and outputs.

- key advantage over white-box testing: you aren't biased by the implementation
- key disadvantage vs white-box testing: you can't tell how well you're covering the implementation, only the specification
    - hard to choose edge cases, get high coverage, etc.
- also common in practice: ***grey-box*** testing, which mixes the two styles

Testing Basics

Today’s agenda:

- Reading Quiz
- What is testing?
- How to write tests
- **Different kinds of tests and how to use them**
- Continuous integration
- Test quality
- Test suite quality

Kinds of tests

Many ways to classify tests:

- by size: **how many resources** do the tests need?
- by scope: **what sort of thing** is the SUT?
- by purpose: **why** are we testing?
- by manner: **how** is testing performed?

Kinds of tests

Many ways to classify tests:

- by size: **how many resources** do the tests need?
- by scope: **what sort of thing** is the SUT?
- by purpose: **why** are we testing?
- by manner: **how** is testing performed?

All valid ways to classify tests!

Kinds of tests

We'll discuss the following important kinds of tests:

- **unit** tests
- **integration** tests
  - with a discussion of **mocking**
- **regression** tests

Kinds of tests: unit tests

Definition: a ***unit test*** tests individual “units” of source code: procedures, methods, classes, modules, etc.

Kinds of tests: unit tests

Definition: a ***unit test*** tests individual "units" of source code: procedures, methods, classes, modules, etc.

- unit tests are characterized by **scope**: you can tell a test is a unit test because it tests only a single component

Kinds of tests: unit tests

Definition: a ***unit test*** tests individual “units” of source code: procedures, methods, classes, modules, etc.

- unit tests are characterized by **scope**: you can tell a test is a unit test because it tests only a single component
- typically they should be small and fast

Kinds of tests: unit tests

Definition: a ***unit test*** tests individual "units" of source code: procedures, methods, classes, modules, etc.

- unit tests are characterized by **scope**: you can tell a test is a unit test because it tests only a single component
- typically they should be **small and fast**
- tests features **in isolation**, which makes debugging easier

Kinds of tests: unit tests

Definition: a ***unit test*** tests individual "units" of source code: procedures, methods, classes, modules, etc.

- unit tests are characterized by **scope**: you can tell a test is a unit test because it tests only a single component
- typically they should be **small and fast**
- tests features **in isolation**, which makes debugging easier
- modern frameworks are often based on SUnit (for Smalltalk)
    - e.g., JUnit (Java), unittest (Python), googletest (C++), etc.

Kinds of tests: unit tests

Definition: a ***unit test*** tests individual "units" of source code: procedures, methods, classes, modules, etc.

- unit tests are characterized by **scope**: you can tell a test is a unit test because it tests only a single
- typically they should be **small and**
- tests features **in isolation**, which
- modern frameworks are often based on SUnit (for Smalltalk)
    - e.g., JUnit (Java), unittest (Python), googletest (C++), etc.

Collectively referred to as **xUnit** frameworks

Kinds of tests: unit tests: xUnit

- Test cases “look like other code”
  - They are special methods written to return a boolean or raise assertion failures

Kinds of tests: unit tests: xUnit

- Test cases “look like other code”
  - They are special methods written to return a boolean or raise assertion failures
- A test case **discoverer** finds all such tests
  - Special naming scheme, dynamic reflection, etc.

Kinds of tests: unit tests: xUnit

- Test cases “look like other code”
  - They are special methods written to return a boolean or raise assertion failures
- A test case **discoverer** finds all such tests
  - Special naming scheme, dynamic reflection, etc.
- A test case **runner** chooses which tests to run

Kinds of tests: unit tests: xUnit

- Test cases “look like other code”
  - They are special methods written to return a boolean or raise assertion failures
- A test case **discoverer** finds all such tests
  - Special naming scheme, dynamic reflection, etc.
- A test case **runner** chooses which tests to run
- Each test is run in a “fresh” environment
  - A **test fixture** specifies which code to run before/after the test case to setup/teardown the right environment

Kinds of tests

We'll discuss the following important kinds of tests:

- **unit** tests
- **integration** tests
  - with a discussion of **mocking**
- **regression** tests

Kinds of tests: integration tests

Definition: an ***integration test*** tests that multiple sub-components of a software system work correctly when combined

Kinds of tests: integration tests

Definition: an ***integration test*** tests that multiple sub-components of a software system work correctly when combined

- **Goal**: answer the question "Does our application work from start to finish?"

Kinds of tests: integration tests

Definition: an ***integration test*** tests that multiple sub-components of a software system work correctly when combined

- **Goal**: answer the question "Does our application work from start to finish?"
- Typically **combined with unit testing**: unit test individual components, then test that they integrate together properly

Kinds of tests: integration tests vs unit tests

Question: what determines whether a test is a **unit test** of a module, or an **integration test** of its sub-components?

Kinds of tests: integration tests vs unit tests

Question: what determines whether a test is a **unit test** of a module, or an **integration test** of its sub-components?

Answer: perspective!

Remember, all of computer science is based on **abstractions**. An integration test for layer *n* of a software stack might be a unit test for layer *n+1*

Kinds of tests: integration tests vs unit tests

Question: what determines whether a test is a **unit test** of a module, or an **integration test** of its sub-components?

Answer: perspective!

Remember, all of computer science is based on **abstractions**. An integration test for layer *n* of a software stack might be a unit test for layer *n+1*

This also promotes a modular, decoupled design

Kinds of tests: integration tests vs unit tests

- However, there are some things that unit tests should **never** do: any test that does any of these things is almost certainly an integration test:

Kinds of tests: integration tests vs unit tests

- However, there are some things that unit tests should **never** do: any test that does any of these things is almost certainly an integration test:
  - It talks to a database
  - It communicates across a network
  - It touches the file system
  - You have to do special things to your environment (such as editing configuration files) to run it

Kinds of tests: integration tests vs unit tests

- However, there are some things that unit tests should **never** do: any test that does any of these thi integration test:
  - It talks to a database
  - It communicates across a network
  - It touches the file system
  - You have to do special things to your environment (such as editing configuration files) to run it

What do these things have in common?

They are EXPENSIVE

Kinds of tests: integration tests vs unit tests

- However, there are some things that unit tests should **never** do: any test that does any of these thi… integration test:
  - It talks to a database
  - It communicates across a network
  - It touches the file system
  - You have to do special things to your environment (such as editing configuration files) to run it

What do these things have in common?
They are **EXPENSIVE**

Testing SUTs that are hard to test

What if we want to write unit or integration tests for some SUT, but the SUT has **expensive dependencies**?

Exercise: take two minutes and, in pairs, generate three examples of things that are hard to test because of their dependencies or other expense factors.

Mocking

Definition: ***Mock objects*** are simulated objects that mimic the behavior of real objects in controlled ways.

In testing, **mocking** uses a mock object to test the behavior of some other object.

- analogy: use a crash test dummy instead of real human to test automobiles

Mocking example: Web API Dependency

- Suppose we're writing a single-page web app
- The API we'll use (e.g., Speech to Text) hasn't been implemented yet or costs money to use
- We want to be able to write our frontend (website) code without waiting on the serverside developers to implement the API and without spending money each time
- What should we do?

Mocking example: Web API Dependency

- Solution: make our own “fake” (“mock”) implementation of the API
- For each method the API exposes, write a substitute for it that just returns some hardcoded data (or any other approximation)
    - Why does this work?

Mocking example: Error Handling

- Suppose we're writing some code where certain kinds of errors will occur **sporadically once deployed**, but “never” in development
  - Out of memory, disk full, network down, etc.

Mocking example: Error Handling

- Suppose we're writing some code where certain kinds of errors will occur **sporadically once deployed**, but “never” in development
  - Out of memory, disk full, network down, etc.
- We'd like to apply the same strategy: write a fake version of the function ...
  - But that sounds difficult to do manually, because many functions would be impacted
  - Example: many functions use the disk

Mocking example: Error Handling

- Strategy one: **static** (= “before running the program”) mocking
  - Move all disk access to a wrapper API, use mocking there at that one point (coin flip fake error)
  - Combines modularity/encapsulation with mocking

Mocking example: Error Handling

- Strategy one: **static** (= “before running the program”) mocking
  - Move all disk access to a wrapper API, use mocking there at that one point (coin flip fake error)
  - Combines modularity/encapsulation with mocking
- Strategy two: **dynamic** (= “while running the program”) mocking
  - While the program is executing, have it **rewrite itself** and replace its existing code with fake or mocked versions
  - this approach is common but has serious downsides, so let’s explore it in a little more detail

Dynamic mocking

- Some languages provide **dynamic mocking libraries** that allow you to substitute objects and functions at runtime

Dynamic mocking

- Some languages provide **dynamic mocking libraries** that allow you to substitute objects and functions at runtime
    - For one test, we could use a mocking library to force another line of code inside our target function to throw an exception when reached

Dynamic mocking

- Some languages provide **dynamic mocking libraries** that allow you to substitute objects and functions at runtime
  - For one test, we could use a mocking library to force another line of code inside our target function to throw an exception when reached
- This feature is available in modern dynamic languages with reflection (Python, Java, etc.)
  - the Jest library used by Covey.Town supports this

Dynamic mocking library uses

- Track how many times a function was called and/or with what arguments ("***spying***")
    - How would you do this with dynamic mocking?

Dynamic mocking library uses

- Track how many times a function was called and/or with what arguments ("***spying***")
  - How would you do this with dynamic mocking?
- Add or remove side effects
  - Exceptions are considered a side effect by mocking libraries

Dynamic mocking library uses

- Track how many times a function was called and/or with what arguments ("***spying***")
  - How would you do this with dynamic mocking?
- Add or remove side effects
  - Exceptions are considered a side effect by mocking libraries
- Test locking in multithreaded code
  - e.g., force a thread to stall after acquiring a lock

Dynamic mocking library disadvantages

Dynamic mocking library disadvantages

- Test cases with dynamic mocking can be **very fragile**
    - What if someone moves or removes the call to the operation you mocked?

Dynamic mocking library disadvantages

- Test cases with dynamic mocking can be **very fragile**
  - What if someone moves or removes the call to the operation you mocked?
- Dynamic mocking **requires good integration tests**
  - If we mock dependencies, we need to be extra careful that our data structures play nicely together

Dynamic mocking library disadvantages

- Test cases with dynamic mocking can be **very fragile**
    - What if someone moves or removes the call to the operation you mocked?
- Dynamic mocking **requires good integration tests**
    - If we mock dependencies, we need to be extra careful that our data structures play nicely together
- Dynamic mocking libraries have a **learning curve**
    - Many language-specific caveats, based on the implementation of the library
    - Error messages are often cryptic (modified program)

Kinds of tests

We'll discuss the following important kinds of tests:

- **unit** tests
- **integration** tests
  - with a discussion of **mocking**
- **regression** tests

Kinds of tests: regression tests

Definition: a ***regression test*** tests that the system no longer suffers from a specific bug

Kinds of tests: regression tests

Definition: a ***regression test*** tests that the system no longer suffers from a specific bug

- prevents old bugs from being **reintroduced**
  - by you or someone else

Kinds of tests: regression tests

Definition: a ***regression test*** tests that the system no longer suffers from a specific bug

- prevents old bugs from being **reintroduced**
  - by you or someone else
- theory: **monotonically increasing** software quality

Kinds of tests: regression tests

Definition: a ***regression test*** tests that the system no longer suffers from a specific bug

- prevents old bugs from being **reintroduced**
  - by you or someone else
- theory: **monotonically increasing** software quality
- **best practice**: when you fix a bug, add a test that specifically exposes that bug
  - that test is a regression test

How to use tests

How to use tests

- as **acceptance criteria**
  - for a feature or bug-fix: ***test driven development***

How to use tests

- as **acceptance criteria**
    - for a feature or bug-fix: ***test driven development***
    - or for a customer accepting the work is done:
        - “if these tests pass, we agree the project is finished”

How to use tests

- as **acceptance criteria**
  - for a feature or bug-fix: ***test driven development***
  - or for a customer accepting the work is done:
    - "if these tests pass, we agree the project is finished"
- to **prevent** the recurrence of **past mistakes**
  - ***regression testing***

How to use tests

- as **acceptance criteria**
  - for a feature or bug-fix: ***test driven development***
  - or for a customer accepting the work is done:
    - “if these tests pass, we agree the project is finished”
- to **prevent** the recurrence of **past mistakes**
  - ***regression testing***
- as a **gatekeeper** to prevent breaking changes to the system
  - ***continuous integration***

Test driven development

Definition: ***test driven development*** (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved so that the tests pass.

Test driven development

Definition: ***test driven development*** (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved so that the tests pass.

- **key idea**: using TDD **guarantees** that you have a test for each line of code that you write

Test driven development

Definition: ***test driven development*** (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved so that the tests pass.

- **key idea**: using TDD **guarantees** that you have a test for each line of code that you write
- research shows that TDD **dramatically improves** software quality (as measured by defect density)
    - implication: **always use TDD** if possible

Test driven development: steps

1. “think of a test that will **force** you to add the next few lines of production code”

Test driven development: steps

1. “think of a test that will **force** you to add the next few lines of production code”

requirement: the test must **fail** when first written!

- “run your entire suite of tests and watch the new test fail”

Test driven development: steps

1. "think of a test that will **force** you to add the next few lines of production code"

requirement: the test must **fail** when first written!

- "run your entire suite of tests and watch the new test fail"
- what if your new test ***doesn't*** fail?

Test driven development: steps

1. “think of a test that will **force** you to add the next few lines of production code”

requirement: the test must **fail** when first written!

- “run your entire suite of tests and watch the new test fail”
- what if your new test ***doesn’t*** fail?
  - actually a very common problem!
  - when reporting a bug, this is why you should try to provide a failing test case

Test driven development: steps

1. “think of a test that will **force** you to add the next few lines of production code”
2. write the test and **observe** the test failure

Test driven development: steps

Common mistake: don't actually run the tests, just assume that your test will fail

1. "think of a test that will **force** you t[...] production code"
2. write the test and **observe** the test failure

Test driven development: steps

1. "think of a test that will **force** you to add the next few lines of production code"
2. write the test and **observe** the test failure
3. write **just enough** code to get the test to pass

Test driven development: steps

1. "think of a test that will **force** you to
   production code"
2. write the test and **observe** the test failure
3. write **just enough** code to get the test to pass

Don't worry too much about elegance - goal in step 3 is to get back to **working code**

Test driven development: steps

1. “think of a test that will **force** you to add the next few lines of production code”
2. write the test and **observe** the test failure
3. write **just enough** code to get the test to pass
4. **refactor** your code to improve its quality/elegance, re-running the test after each change to make sure that it still passes

Test driven development: steps

1. "think of a test that will **force** you to add the next few lines of production code"
2. write the test and **observe** the test failure
3. write **just enough** code to get the test to pass
4. **refactor** your code to improve its quality/elegance, re-running the test after each change to make sure that it still passes
5. commit the new code **and the test**; make a PR

Test driven development: steps

1. “think of a test that will **force** you to add the next few lines of production code”
2. write the test and **observe** the test failure
3. write **just enough** code to get the test to pass
4. **refactor** your code to improve its quality/elegance, re-running the test after each change to make sure that it still passes
5. commit the new code **and the test**; make a PR
6. go back to step 1

Why does TDD improve code quality?

Why does TDD improve code quality?

- every behavior has a **regression test** immediately

Why does TDD improve code quality?

- every behavior has a **regression test** immediately
- fast **edit-test-debug cycle**

Why does TDD improve code quality?

- every behavior has a **regression test** immediately
- fast **edit-test-debug cycle**

Definition: the ***edit-test-debug cycle*** is the main loop of software development:

- edit the code
- test to make sure it works
- debug why it doesn't

Research shows that having a fast edit-test-debug cycle is critical for programmer productivity.

Advice: Try to avoid "test" steps of > 30 seconds.

Why does TDD improve code quality?

- every behavior has a **regression test** immediately
- fast **edit-test-debug cycle**

Definition: the ***edit-test-debug cycle*** is the main loop of software development:

- edit the code
- test to make sure it works
- debug why it doesn't

Research shows that having a **fast edit-test-debug cycle** is critical for programmer productivity.

Why does TDD improve code quality?

- every behavior has a **regression test** immediately
- fast **edit-test-debug cycle**

Definition: the ***edit-test-debug cycle*** is the main loop of software development:

- edit the code
- test to make sure it works
- debug why it doesn't

Research shows that having a **fast edit-test-debug cycle** is critical for programmer productivity.

Advice: Try to **avoid** "test" steps of **> 10 seconds**.

Why does TDD improve code quality?

- every behavior has a **regression test** immediately
- fast **edit-test-debug cycle**
- code is **working most of the time** (TDD and Agile are closely related: almost all Agile methodologies advocate for TDD)

Testing Basics

Today’s agenda:

- Reading Quiz
- What is testing?
- How to write tests
- Different kinds of tests and how to use them
- **Continuous integration**
- Test quality
- Test suite quality

Continuous integration

A few slides ago, I mentioned that it's a good idea to avoid edit-test-debug cycles with > 10 second "test" steps

- but what if your tests **take longer** than that to run?

Continuous integration

A few slides ago, I mentioned that it's a good idea to avoid edit-test-debug cycles with > 10 second "test" steps

- but what if your tests **take longer** than that to run?
- answer: move them from the developer's machine to a **continuous integration** server

Continuous integration

Definition: ***continuous integration*** (CI) “is a software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run”

[https://aws.amazon.com/devops/continuous-integration/]

Continuous integration

Definition: ***continuous integration*** (CI) “is a software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run”

- use of CI is **practically mandatory** in industry

[https://aws.amazon.com/devops/continuous-integration/]

Continuous integration

Definition: ***continuous integration*** (CI) “is a software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run”

- use of CI is **practically mandatory** in industry
- **best practices**:
  - use CI for every project, even very small ones
  - all changes to a project should be gated by CI tests passing
  - run all tests (and other quality checks) automatically in CI

[https://aws.amazon.com/devops/continuous-integration/]

Continuous integration

Definition: ***continuous integration*** (CI) “is a software development practice where developers regularl central repository, after which aut

- use of CI is **practically mandat**
- **best practices**:
  - use CI for every project, ev
  - all changes to a project sho
  - run all tests (and other quality checks) automatically in CI

Advice: be very concerned about any project that:

- doesn’t have a CI setup
- doesn’t run all tests in CI
- lets CI builds regularly fail for long periods of time
  - a failing CI build is an **emergency**

[https://aws.amazon.com/devops/continuous-integration/]

Aside: emergencies

- What constitutes an ***emergency*** (from a DevOps perspective)?

Aside: emergencies

- What constitutes an ***emergency*** (from a DevOps perspective)?
  - depends on your service, but typically these qualify:
    - big % of user requests aren’t getting responses
    - big % of user requests have really high latency
    - lots of your servers are unavailable/down (even if users aren’t yet impacted)

Aside: emergencies

- What constitutes an ***emergency*** (from a DevOps perspective)?
  - depends on your service, but typically these qualify:
    - big % of user requests aren't getting responses
    - big % of user requests have really high latency
    - lots of your servers are unavailable/down (even if users aren't yet impacted)
- In a typical software engineering job, emergencies generate ***pages***
  - including e.g., in the middle of the night

Aside: emergencies

- What constitutes an ***emergency*** (from a DevOps perspective)?
    - depends on your service, but typically these qualify:
        - big % of user requests aren't getting responses
        - big % of user requests have really high latency
        - lots of your servers are unavailable/down (even if users aren't yet impacted)
- In a typical software engineering job, emergencies generate ***pages***
    - including e.g., in the middle of the night
    - an ***on-call engineer*** will have to deal with that page

Aside: emergencies

- What constitutes an ***emergency*** (from a DevOps perspective)?
  - depends on your service, but typically these qualify:
    - big % of user requests aren't getting responses
    - big % of user requests have really high latency
    - lots of your servers are unavailable/down (even if users aren't yet impacted)
- In a typical software engineering job, emergencies generate ***pages***
  - including e.g., in the middle of the night
  - an ***on-call engineer*** will have to deal with that page
  - typical reason a DevOps team might write tests: avoid pages

Aside: emergencies

- What constitutes an ***emergency*** (from a DevOps perspective)?
  - depends on your service, but typically these qualify:
    - big % of user requests a
    - big % of user requests h
    - lots of your servers are
      aren't yet impacted)
- In a typical software engineering job, emergencies generate ***pages***
  - including e.g., in the middle of the night
  - an ***on-call engineer*** will have to deal with that page
  - typical reason a DevOps team might write tests: avoid pages

Why is a failing CI build considered an **emergency**?

Aside: emergencies

- What constitutes an ***emergency*** (from a DevOps perspective)?
  - depends on your service, but typically these qualify:
    - big % of user requests a
    - big % of user requests h
    - lots of your servers are
      aren't yet impacted)
- In a typical software engineering job, emergencies generate ***pages***
  - including e.g., in the middle of the night
  - an ***on-call engineer*** will have to deal with that page
  - typical reason a DevOps team might write tests: avoid pages

Why is a failing CI build considered an **emergency**?

- CI should block merging new code, so failing CI = no new code

Testing Basics

Today’s agenda:

- Reading Quiz
- What is testing?
- How to write tests
- Different kinds of tests and how to use them
- Continuous integration
- **Test quality**
- Test suite quality

Test quality: what makes a test good or bad?

Good	Bad

Test quality: what makes a test good or bad?

Good	Bad

In-class exercise: with a partner, spend ~2 minutes making a list of factors that make a test "good" or "bad".

Test quality: what makes a test good or bad?

Good	Bad
• isolated (only tests one thing)	• brittle
• runs quickly	• slow
• strong oracle	• weak oracle
• hermetic	• redundant
• easy to understand	• hard to understand (“mystery”)
• deterministic	• non-deterministic (“flaky”)
• etc.	• etc.

Test quality: what makes a test good or bad?

Good	Bad
• isolated (only tests one thing)	• **brittle**
• runs quickly	• slow
• strong oracle	• weak oracle
• **hermetic**	• redundant
• easy to understand	• hard to understand (“**mystery**”)
• deterministic	• non-deterministic (“**flaky**”)
• etc.	• etc.

“Hermetic” tests

Definition: a ***hermetic*** test is fully self-contained: its behavior doesn’t depend on anything except the test itself and the SUT

“Hermetic” tests

Definition: a ***hermetic*** test is fully self-contained: its behavior doesn’t depend on anything except the test itself and the SUT

- avoid dependencies on the environment (e.g., software installed on the machine, environment variables, contents of other files, operating system behaviors, etc.)
- being hermetic is also important for builds generally (we’ll discuss more in our lecture on build systems later this semester)

Brittle tests

Definition: a ***brittle*** test fails for reasons unrelated to what it ostensibly tests

Brittle tests

Definition: a ***brittle*** test fails for reasons unrelated to what it ostensibly tests

- common causes:
  - not being hermetic
  - testing too much at once
  - comparator or oracle is too specific

Mystery tests

Definition: a ***mystery*** test fails for reasons that are not immediately clear

Mystery tests

Definition: a ***mystery*** test fails for reasons that are not immediately clear

- commonly **co-occurs with brittleness**: test is brittle because it is too complicated, and when it fails it’s not clear why
  - especially common for very large, end-to-end tests

Mystery tests

Definition: a ***mystery*** test fails for reasons that are not immediately clear

- commonly **co-occurs with brittleness**: test is brittle because it is too complicated, and when it fails it's not clear why
  - especially common for very large, end-to-end tests
- **best practice**: tests should give as much information as possible when they fail
  - **implication**: when writing tests, think about why they might fail in the future and document that in the test itself

Flaky tests

Definition: a ***flaky*** test fails non-deterministically: that is, they sometimes pass and sometimes fail

Flaky tests

Definition: a ***flaky*** test fails non-deterministically: that is, they sometimes pass and sometimes fail

- sometimes caused by brittleness (e.g., relying on the network)

Flaky tests

Definition: a *flaky* test fails non-deterministically: that is, they sometimes pass and sometimes fail

- sometimes caused by brittleness (e.g., relying on the network)
- sometimes caused by **non-determinism in the program** itself
  - e.g., relying on randomness, iteration order of hashtables, etc.

Flaky tests

Definition: a ***flaky*** test fails non-deterministically: that is, they sometimes pass and sometimes fail

- sometimes caused by brittleness (e.g., relying on the network)
- sometimes caused by **non-determinism in the program** itself
  - e.g., relying on randomness, iteration order of hashtables, etc.
- are a **major problem in practice**
  - difficult to debug, so waste a lot of developer time
  - detecting them is an active research area

Testing Basics

Today’s agenda:

- Reading Quiz
- What is testing?
- How to write tests
- Different kinds of tests and how to use them
- Continuous integration
- Test quality
- **Test suite quality**

Test suite quality

- We’ve talked about what makes individual test cases good or bad

Test suite quality

- We’ve talked about what makes individual test cases good or bad
- However, programs typically have **more than one** test

Test suite quality

- We’ve talked about what makes individual test cases good or bad
- However, programs typically have **more than one** test

Definition: a ***test suite*** is a collection of tests for the same program

Test suite quality

- We've talked about what makes individual test cases good or bad
- However, programs typically have **more than one** test

Definition: a ***test suite*** is a collection of tests for the same program

Question: what makes one test suite **better or worse** than another?

Test suite quality

- We've talked about what makes individual test cases good or bad
- However, programs typically have **more than one** test

Definition: a ***test suite*** is a collection of tests for the same program

Question: what makes one test suite **better or worse** than another?

- not just the sum of the "goodness" of all the individual tests!

Test suite quality: who cares?

Why would we want to evaluate the quality of a test suite?

Test suite quality: who cares?

Why would we want to evaluate the quality of a test suite?

- testing is **expensive** (e.g., 35% of total IT spending according to Capgemini World Quality Report, 2015)

Test suite quality: who cares?

Why would we want to evaluate the quality of a test suite?

- testing is **expensive** (e.g., 35% of total IT spending according to Capgemini World Quality Report, 2015)
- we want to direct our resources **efficiently**
  - i.e., avoid writing new tests if we already have a good test suite

Test suite quality: who cares?

Why would we want to evaluate the quality of a test suite?

- testing is **expensive** (e.g., 35% of total IT spending according to Capgemini World Quality Report, 2015)
- we want to direct our resources **efficiently**
  - i.e., avoid writing new tests if we already have a good test suite
- we want to know how much confidence our tests give us
  - ideal world: all tests pass = software is 100% correct

Test suite quality: who cares?

Why would we want to evaluate the quality of a test suite?

- testing is **expensive** (e.g., 35% of total IT spending according to Capgemini World Quality Report, 2015)
- we want to direct our resources **efficiently**
  - i.e., avoid writing new tests if we already have a good test suite
- we want to know how much **confidence** our tests give us
  - ideal world: all tests pass = software is 100% correct
- sometimes, we may not even have enough resources to run all tests

Three ways to think about test suite quality

Consider three ways to think about test suite quality:

Three ways to think about test suite quality

Consider three ways to think about test suite quality:

- test suite quality through the lens of **logic**
    - intuition: if we don't test it, we can't find bugs

Three ways to think about test suite quality

Consider three ways to think about test suite quality:

- test suite quality through the lens of **logic**
  - intuition: if we don't test it, we can't find bugs
  - leads to ***coverage*** (subject of next week's lecture)

Three ways to think about test suite quality

Consider three ways to think about test suite quality:

- test suite quality through the lens of logic
  - intuition: if we don’t test it, we can’t find bugs
  - leads to ***coverage*** (subject of next week’s lecture)
- test suite quality through the lens of **statistics**
  - intuition: test what happens to real users

Three ways to think about test suite quality

Consider three ways to think about test suite quality:

- test suite quality through the lens of logic
    - intuition: if we don’t test it, we can’t find bugs
    - leads to ***coverage*** (subject of next week’s lecture)
- test suite quality through the lens of **statistics**
    - intuition: test what happens to real users
- test suite quality through the lens of adversity
    - intuition: inject bugs and see if the test suite catches them

Three ways to think about test suite quality

Consider three ways to think about test suite quality:

- test suite quality through the lens of logic
  - intuition: if we don't test it, we can't find bugs
  - leads to ***coverage*** (subject of next week's lecture)
- test suite quality through the lens of **statistics**
  - intuition: test what happens to real users
- test suite quality through the lens of adversity
  - intuition: inject bugs and see if the test suite catches them
  - leads to ***mutation testing***, which we'll cover later this semester

Lens of Logic: a brief introduction to coverage

Definition: ***Statement coverage*** is the fraction of source statements that are executed by the test suite.

Lens of Logic: a brief introduction to coverage

Definition: ***Statement coverage*** is the fraction of source statements that are executed by the test suite.

- **Key Logical Observation**: If we **never test** line X then testing **cannot rule out** the presence of a bug on line X

Lens of Logic: a brief introduction to coverage

Definition: ***Statement coverage*** is the fraction of source statements that are executed by the test suite.

- **Key Logical Observation**: If we **never test** line X then testing **cannot rule out** the presence of a bug on line X
- Example: if our test executes lines 1 and 2, but there is a bug on line 3, there is **no way** that our test will find the bug!

Lens of Logic: a brief introduction to coverage

Definition: ***Branch coverage*** is a test suite quality metric that counts the total number of conditional branches exercised by that test suite (i.e., if true and if false are counted separately)

Lens of Logic: a brief introduction to coverage

Definition: ***Branch coverage*** is a test suite quality metric that counts the total number of conditional branches exercised by that test suite (i.e., if true and if false are counted separately)

Note that branch coverage can subsume line coverage:

```
foo(a):
  if a > 5:
    print “x”
  print “y”
```

Lens of Logic: a brief introduction to coverage

Definition: ***Branch coverage*** is a test suite quality metric that counts the total number of conditional branches exercised by that test suite (i.e., if true and if false are counted separately)

Note that branch coverage can subsume line coverage:

```
foo(a):
  if a > 5:
    print "x"
  print "y"
```

Test Suite { `foo(7)` } has 100% line coverage but 50% branch coverage.

Lens of Logic: a brief introduction to coverage

Definition: ***Branch coverage*** is a test suite quality metric that counts the total number of conditional branches exercised by that test suite (i.e., if true and if false are counted separately)

Note that branch coverage can subsume line coverage:

```
foo(a):
  if a > 5:
    print "x"
  print "y"
```

Test Suite { `foo(7), foo(4)` } has 100% line coverage and 100% branch coverage.

Lens of Logic: a brief introduction to coverage

- Statement and branch coverage are all you need for this week's homework assignment

Lens of Logic: a brief introduction to coverage

- Statement and branch coverage are all you need for this week's homework assignment
    - but we will cover both in a lot more detail next week

Lens of Logic: a brief introduction to coverage

- Statement and branch coverage are all you need for this week's homework assignment
  - but we will cover both in a lot more detail next week
- Next week we will also cover:
  - how to compute coverage
  - other kinds of coverage (e.g., path coverage)
  - advantages/disadvantages of various kinds of coverage

Using test suite quality metrics

- It is common to have **too many** test cases

Using test suite quality metrics

- It is common to have **too many** test cases
  - Surprisingly, this is **normal in industry**: you almost always have far too few or far too many!

Using test suite quality metrics

- It is common to have **too many** test cases
    - Surprisingly, this is **normal in industry**: you almost always have far too few or far too many!
- This is especially true when using automated test generation tools

Using test suite quality metrics

- It is common to have **too many** test cases
  - Surprisingly, this is **normal in industry**: you almost always have far too few or far too many!
- This is especially true when using automated test generation tools
  - Which many produce many tests but **lower-quality** ones than humans would produce

Using test suite quality metrics

- It is common to have **too many** test cases
  - Surprisingly, this is **normal in industry**: you almost always have far too few or far too many!
- This is especially true when using automated test generation tools
  - Which many produce many tests but **lower-quality** ones than humans would produce
  - A **big cost problem**!

Test suite minimization

Definition: given a set of test cases and coverage information for each one, the ***test suite minimization problem*** is to find the minimal number of test cases that still have the maximum coverage.

Test suite minimization

Definition: given a set of test cases and coverage information for each one, the ***test suite minimization problem*** is to find the minimal number of test cases that still have the maximum coverage.

Example:

- T1 covers lines 1,2,3
- T2 covers lines 2,3,4,5
- T3 covers lines 1,2
- T4 covers lines 1, 6

Test suite minimization

Definition: given a set of test cases and coverage information for each one, the ***test suite minimization problem*** is to find the minimal number of test cases that still have the maximum coverage.

Example:

- T1 covers lines 1,2,3
- T2 covers lines 2,3,4,5
- T3 covers lines 1,2
- T4 covers lines 1, 6

Which of these tests would you pick to minimize the number that need to be run?

Test suite minimization

Definition: given a set of test cases and coverage information for each one, the ***test suite minimization problem*** is to find the minimal number of test cases that still have the maximum coverage.

Example:

- ~~T1 covers lines 1,2,3~~
- T2 covers lines 2,3,4,5
- ~~T3 covers lines 1,2~~
- T4 covers lines 1, 6

Which of these tests would you pick to minimize the number that need to be run?

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

- very similar to test suite minimization (same techniques are useful for both)

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

- very similar to test suite minimization (same techniques are useful for both)
- **question**: how **hard** are these problems?

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

- very similar to test suite minimization (same techniques are useful for both)
- **question**: how **hard** are these problems?
  - theory strikes again!

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

- very similar to test suite minimization (same techniques are useful for both)
- **question**: how **hard** are these problems?
  - theory strikes again!
  - answer: it's "hard" (similar "traditional" problem that you might consider a reduction to: **knapsack**)

Aside: reductions

Aside: reductions

Your CS education is incomplete until you have **reduced** one problem to another

- examples: reducing something to the halting problem to show that it is not computable; reducing something to satisfiability or knapsack to show that it is NP-hard
- should be covered in a theory of computation class (likely near the end of the semester)

Aside: reductions

Your CS education is incomplete until you have **reduced** one problem to another

- examples: reducing something to the halting problem to show that it is not computable; redu
  knapsack to show that it is NP
- should be covered in a theory
  the end of the semester)

Reduction is a **powerful tool** for thinking about problems: it lets you solve difficult problems indirectly by re-using solutions for other, related problems.

Today’s in-class exercise

- That’s all we’ll cover today
- Our in-class exercise today will ask you to achieve high (statement and branch) coverage on a small program
  - I expect you to write tests by hand to do this
- You are welcome to work with a partner
- This assignment should be relatively easy
  - later assignments are harder!