

Testing (Part 1/3)

Martin Kellogg

Testing (part 1)

Today's agenda:

- **Reading Quiz**
- What is testing?
- How to write tests
- Different kinds of tests and how to use them
- Continuous integration (or: why most of your tests should be automated)

Reading quiz: testing, part 1

Q1: **TRUE** or **FALSE**: During test-driven development, it's important that you get the code right the first time: you should spend a lot of time making sure that it's right, before re-running your failing tests.

Q2: The author advocates that tests should run in under:

- A. 1 second
- B. 10 seconds
- C. 1 minute
- D. 1 hour

Reading quiz: testing, part 1

Q1: **TRUE** or **FALSE**: During test-driven development, it's important that you get the code right the first time: you should spend a lot of time making sure that it's right, before re-running your failing tests.

Q2: The author advocates that tests should run in under:

- A. 1 second
- B. 10 seconds
- C. 1 minute
- D. 1 hour

Reading quiz: testing, part 1

Q1: **TRUE** or **FALSE**: During test-driven development, it's important that you get the code right the first time: you should spend a lot of time making sure that it's right, before re-running your failing tests.

Q2: The author advocates that tests should run in under:

- A. 1 second
- B. 10 seconds
- C. 1 minute
- D. 1 hour

Testing (part 1)

Today's agenda:

- Reading Quiz
- **What is testing?**
- How to write tests
- Different kinds of tests and how to use them
- Continuous integration (or: why most of your tests should be automated)

Testing (part 1)

Today's agenda:

- Reading Quiz
- **What is testing?**
- How to write tests
- Different kinds of tests and I
- Continuous integration (or: v automated)

Announcements:

- IP1 should be available by the time you see this lecture
 - my advice: start **right away!**
 - last semester, many students failed to complete IP1 on time, because they took this week off
 - don't let this be you!

Testing (part 1)

Today's agenda:

- Reading Quiz
- **What is testing?**
- How to write tests
- Different kinds of tests and how to use them
- Continuous integration (or: why most of your tests should be automated)

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**

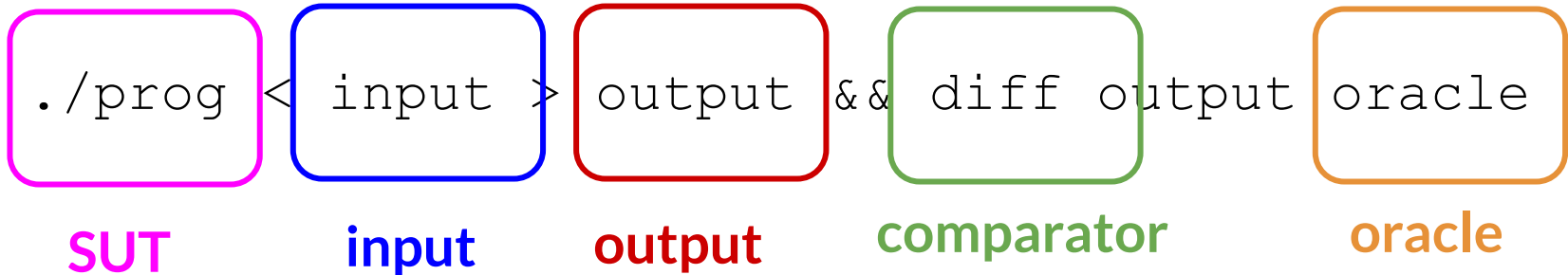
```
./prog < input > output && diff output
```

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**



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?**)

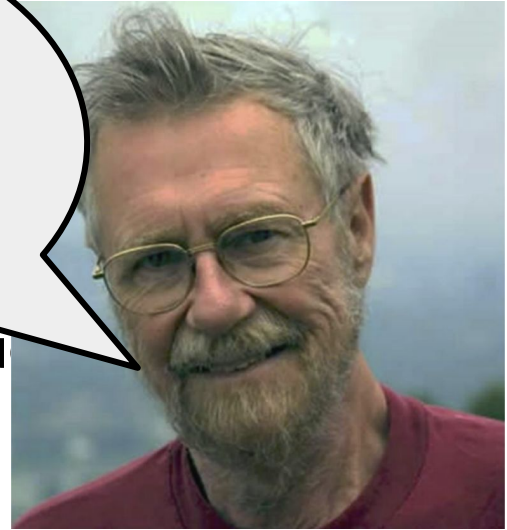
Ideal situation: you can test every input (“**exhaustive testing**”)

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

Building a test case

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

“Tests can show the presence of bugs, but not their absence”



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 space**

We'll talk about these out of order:

- comparators
- oracles
- inputs

Testing (part 1)

Today's agenda:

- Reading Quiz
- What is testing?
- **How to write tests**
- Different kinds of tests and how to use them
- Continuous integration (or: why most of your tests should be automated)

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 **string**)
- Also common:
 - **over-approximation** (“is the output greater than the expected values”, or, more commonly, “is the output greater than the expected value”)
 - **under-approximation** (“does the output contain 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!

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

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

Edge case examples:

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

Choosing inputs

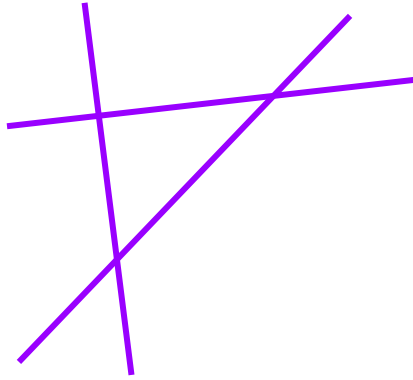
- When writing tests by hand, this is often the hardest part
- Strategies:
 - choose at random (avoid when writing by hand)
 - edge cases
 - **partition testing**

Partition testing

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

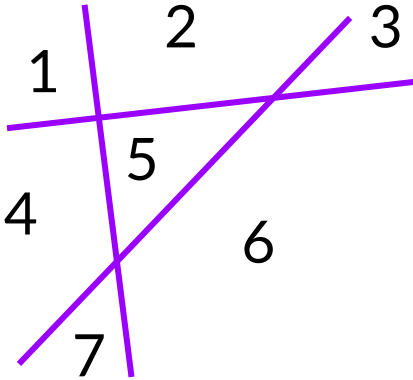
Partition testing

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



Partition testing

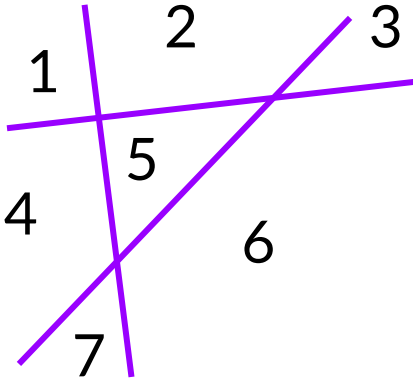
Key idea: split up the input space into redundant “regions”



- write one test **for each region**

Partition testing

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

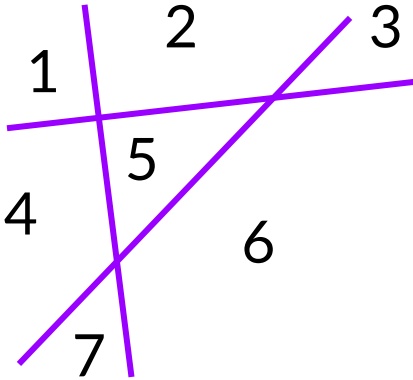


- write one test **for each region**
- possible ways to split up the input:
 - parity (even, odd)
 - positive, negative, zero
 - jpg files vs png files
 - correctly-formatted input vs incorrectly-formatted input

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

Testing (part 1)

Today's agenda:

- Reading Quiz
- What is testing?
- How to write tests
- **Different kinds of tests and how to use them**
- Continuous integration (or: why most of your tests should be automated)

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 element
- typically they should be **small and simple**
- tests features **in isolation**, which means they should not depend on other parts of the system
- 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

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 one minute and, in pairs, generate three examples of things that are hard to test because of their dependencies or other expense factors.

Then, the TA will lead a five-minute-ish discussion of common answers on the whiteboard.

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

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

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

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 write 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

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 (part 1)

Today's agenda:

- Reading Quiz
- What is testing?
- How to write tests
- Different kinds of tests and how to use them
- **Continuous integration** (or: why most of your tests should be automated)

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”

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

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

Continuous integration

Definition: *continuous integration* (CI) “is a software development practice where developers regularly merge their code to a central repository, after which automatic builds are triggered to produce artifacts that are ready to be tested in an automated test environment.”

- use of CI is **practically mandatory**
- **best practices:**
 - use CI for every project, even if it's a small project
 - all changes to a project should be made through a version control system
 - 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**

Takeaways

- A test is an input + a comparator + an oracle
- Use strategies like partition testing when writing test cases by hand
- Different kinds of tests serve different purposes
 - understand the difference between unit, integration tests
 - regression testing prevents bugs (especially when combined with TDD + CI)
- Use TDD + CI to improve software quality
- Next time: test suite quality and mutation testing