MONTANA

STATE UNIVERSITY

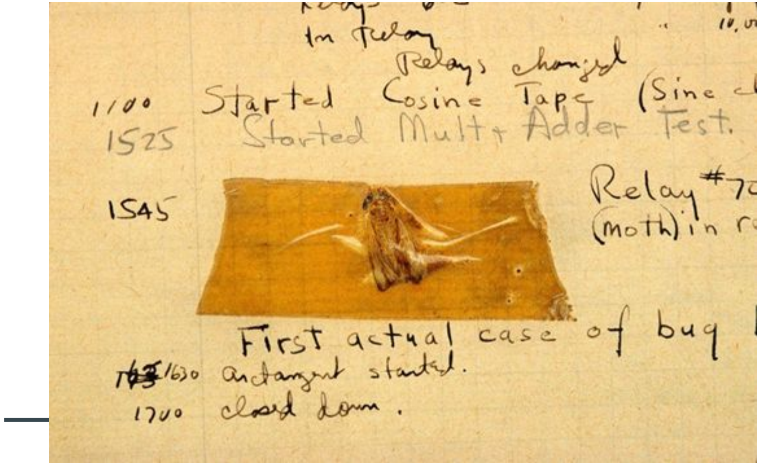
Testing Code

•••

The Nitty Gritty

The Birth Of Computer Bugs

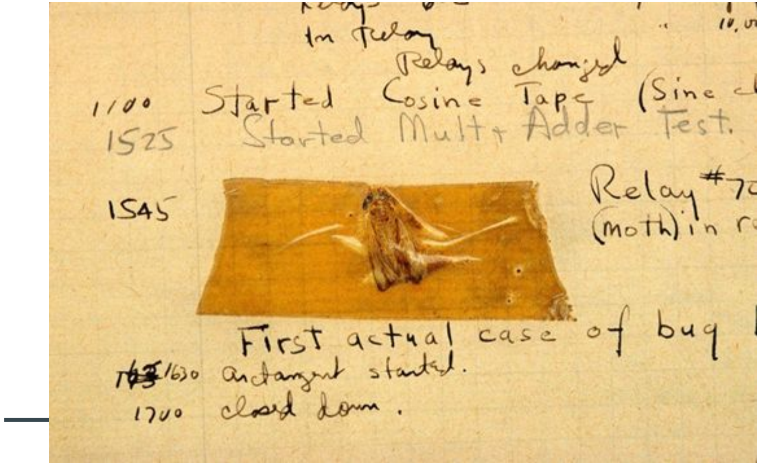
- In 1947 Harvard University was operating a room-sized computer called the Mark II
  - Mechanical relays
  - Glowing vacuum tubes
  - Programed by hand
- A moth flew into the computer and was killed
  - F
- The first computer bug



in relay
Relays changed
1100 Started Cosine Tape (Sine c
1525 Started Mult+ Adder Test.
1545 Relay #7
(moth) in r
First actual case of bug
~~143~~ 1630 antangent started.
1700 closed down.

Developing Software

- Traditional Waterfall
  - Extensive specification
  - Implementation
  - Acceptance testing (manually)
  - Fix bugs
    - Probably break other stuff
  - Goto Acceptance Testing
    - Unless t > deadline



in relay

Relays changed

1100 Started Cosine Tape (Sine c

1525 Started Mult+ Adder Test.

1545 Relay #70

(moth) in r

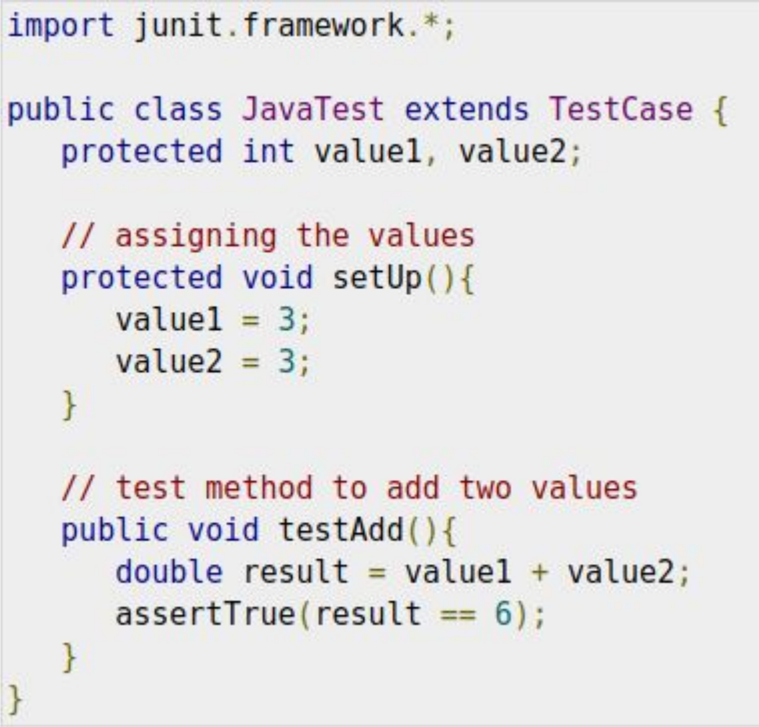
First actual case of bug

~~143~~1630 Antangent started.

1700 closed down.

Developing Software

- Waterfall was very brittle
  - Particularly costly due to the manual testing phase
- In the 90s and 2000s, **automated testing** became more and more prominent
- Early 2000s: the emergence of Test Driven Development (TDD)
  - Sometimes called “Test first” development



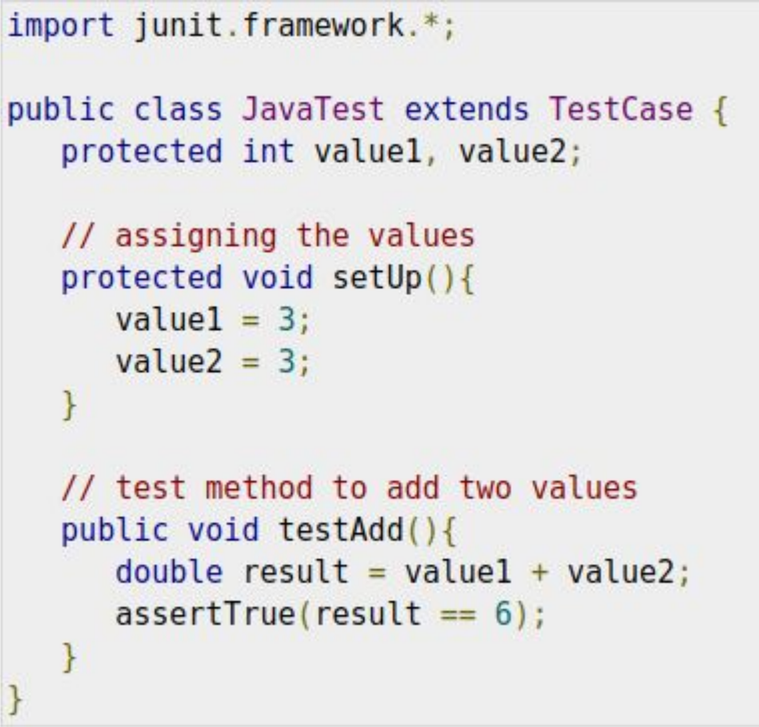
```java
import junit.framework.*;

public class JavaTest extends TestCase {
   protected int value1, value2;

   // assigning the values
   protected void setUp(){
      value1 = 3;
      value2 = 3;
   }

   // test method to add two values
   public void testAdd(){
      double result = value1 + value2;
      assertTrue(result == 6);
   }
}
```

Levels of Testing

- Unit Tests - Small, typically focused on single function or operation
- Integration Tests - Larger, test the interaction between components
- Validation Tests - Often manual, ensure that the unit and integration tests didn’t miss anything with respect to system functionality
  - This touches on a problem
- Acceptance Tests - Almost always manual, does the user accept what we built as solving their use cases?

Unit Tests

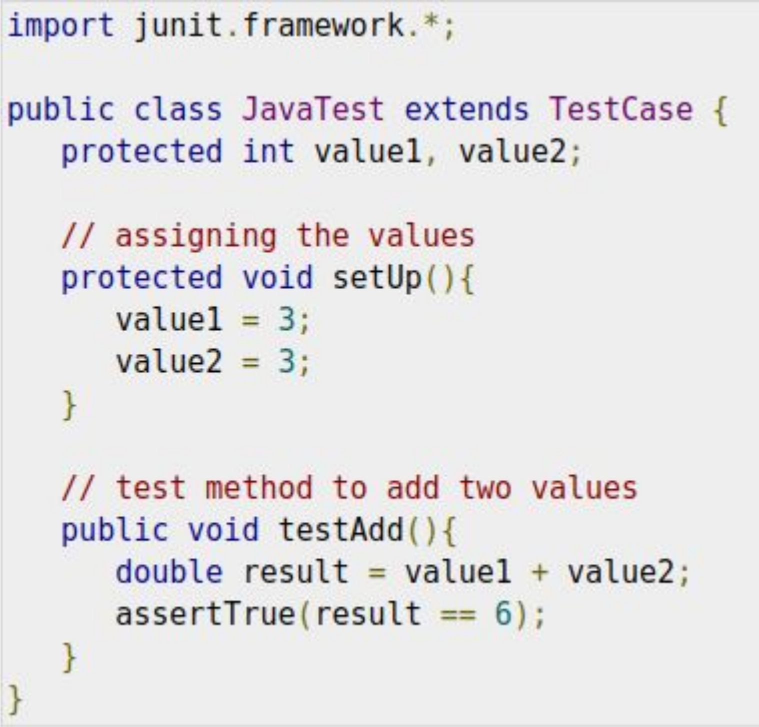
- This is a good example of a unit test taken from java
- Tests the “add” operator
- Is this a useful test?



```
import junit.framework.*;

public class JavaTest extends TestCase {
   protected int value1, value2;

   // assigning the values
   protected void setUp(){
      value1 = 3;
      value2 = 3;
   }

   // test method to add two values
   public void testAdd(){
      double result = value1 + value2;
      assertTrue(result == 6);
   }
}
```

Unit Tests

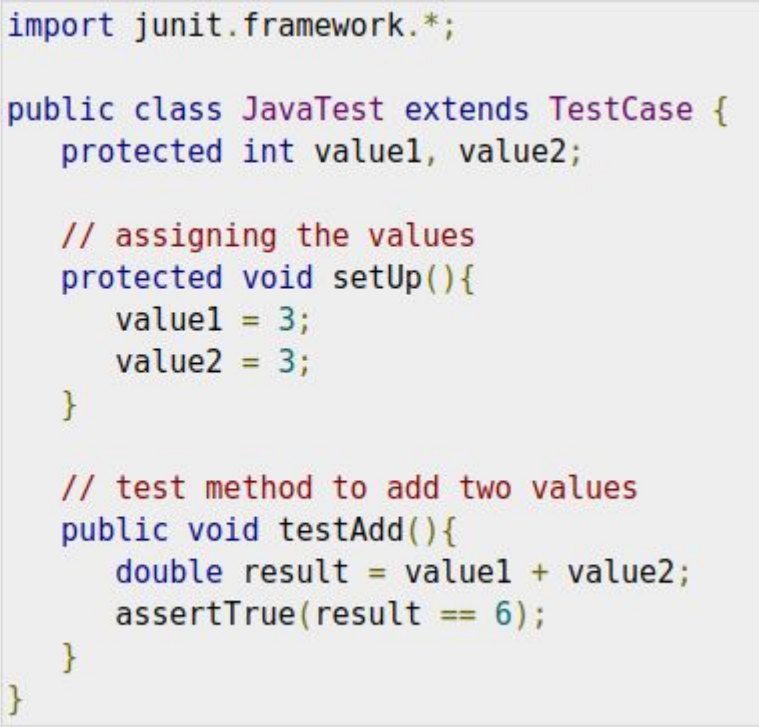
- Shows a common setup/teardown pattern in unit tests
- Shows the common test pattern:
  - Do something
  - Assert something about the state after that



```
import junit.framework.*;

public class JavaTest extends TestCase {
   protected int value1, value2;

   // assigning the values
   protected void setUp(){
      value1 = 3;
      value2 = 3;
   }

   // test method to add two values
   public void testAdd(){
      double result = value1 + value2;
      assertTrue(result == 6);
   }
}
```

Unit Tests - Problems?

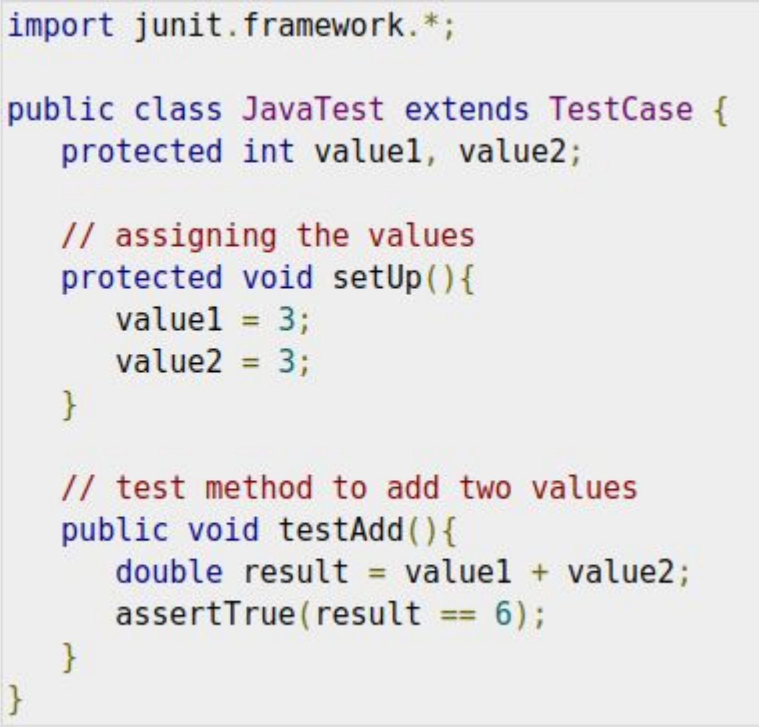
- Can be a bit precious at times
- Do we really need to know if the add operator works?
- Also often tied too closely to an implementation
- If you write 20 unit tests for a method, and that method is removed...



```
import junit.framework.*;

public class JavaTest extends TestCase {
   protected int value1, value2;

   // assigning the values
   protected void setUp(){
      value1 = 3;
      value2 = 3;
   }

   // test method to add two values
   public void testAdd(){
      double result = value1 + value2;
      assertTrue(result == 6);
   }
}
```

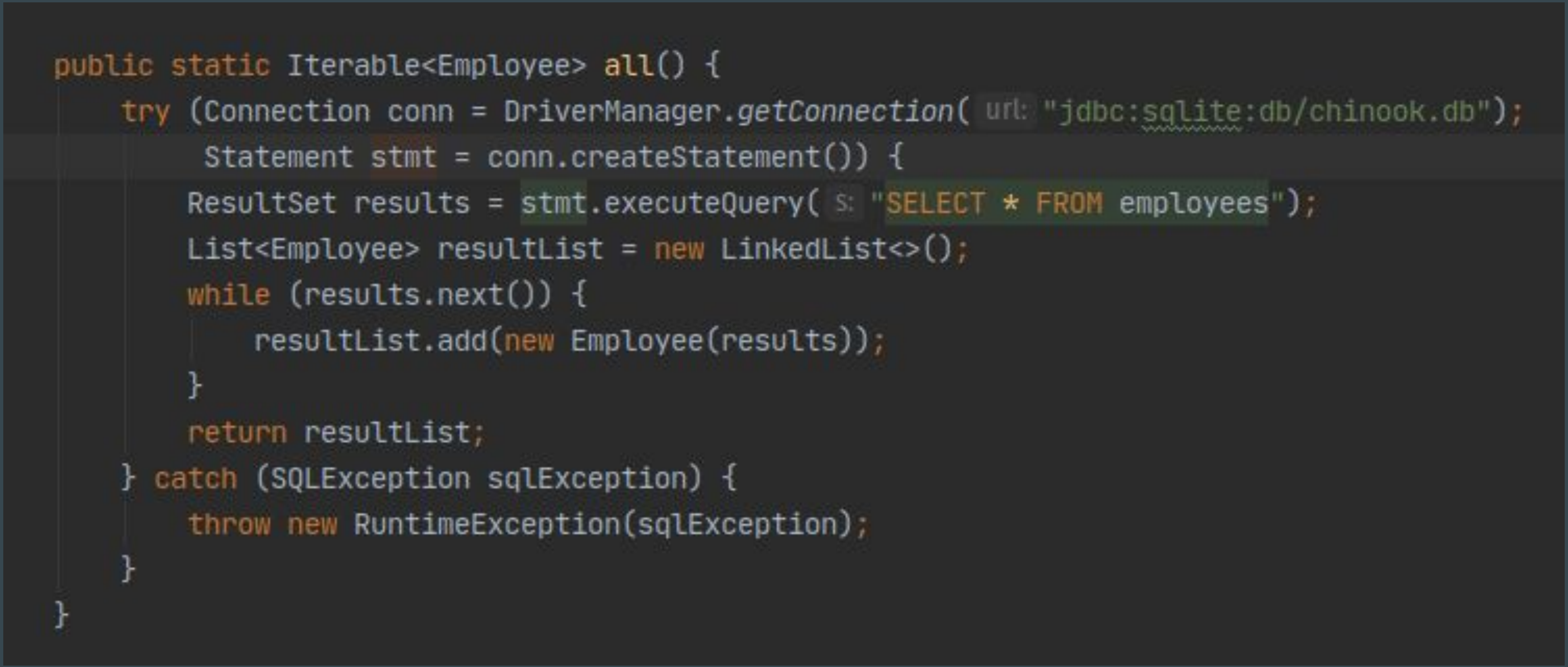
Unit Tests & Mocks

- What if a function requires complex infrastructure support to test?
- E.g. a database
- Solution: “Mock” the infrastructure
- Create a mock database?



```
import junit.framework.*;

public class JavaTest extends TestCase {
   protected int value1, value2;

   // assigning the values
   protected void setUp(){
      value1 = 3;
      value2 = 3;
   }

   // test method to add two values
   public void testAdd(){
      double result = value1 + value2;
      assertTrue(result == 6);
   }
}
```

A Complex Method To Unit Test



```java
public static Iterable<Employee> all() {
    try (Connection conn = DriverManager.getConnection( url: "jdbc:sqlite:db/chinook.db");
         Statement stmt = conn.createStatement()) {
        ResultSet results = stmt.executeQuery( s: "SELECT * FROM employees");
        List<Employee> resultList = new LinkedList<>();
        while (results.next()) {
            resultList.add(new Employee(results));
        }
        return resultList;
    } catch (SQLException sqlException) {
        throw new RuntimeException(sqlException);
    }
}
```

Problems with Mocks

- Often complicated to implement
- **Operational Semantics** often make a big difference in test outcome!
- How do we get mocks into our tests?
    - Pass dependencies in as arguments
    - Dependency Injection

Easy mocking. Better testing.

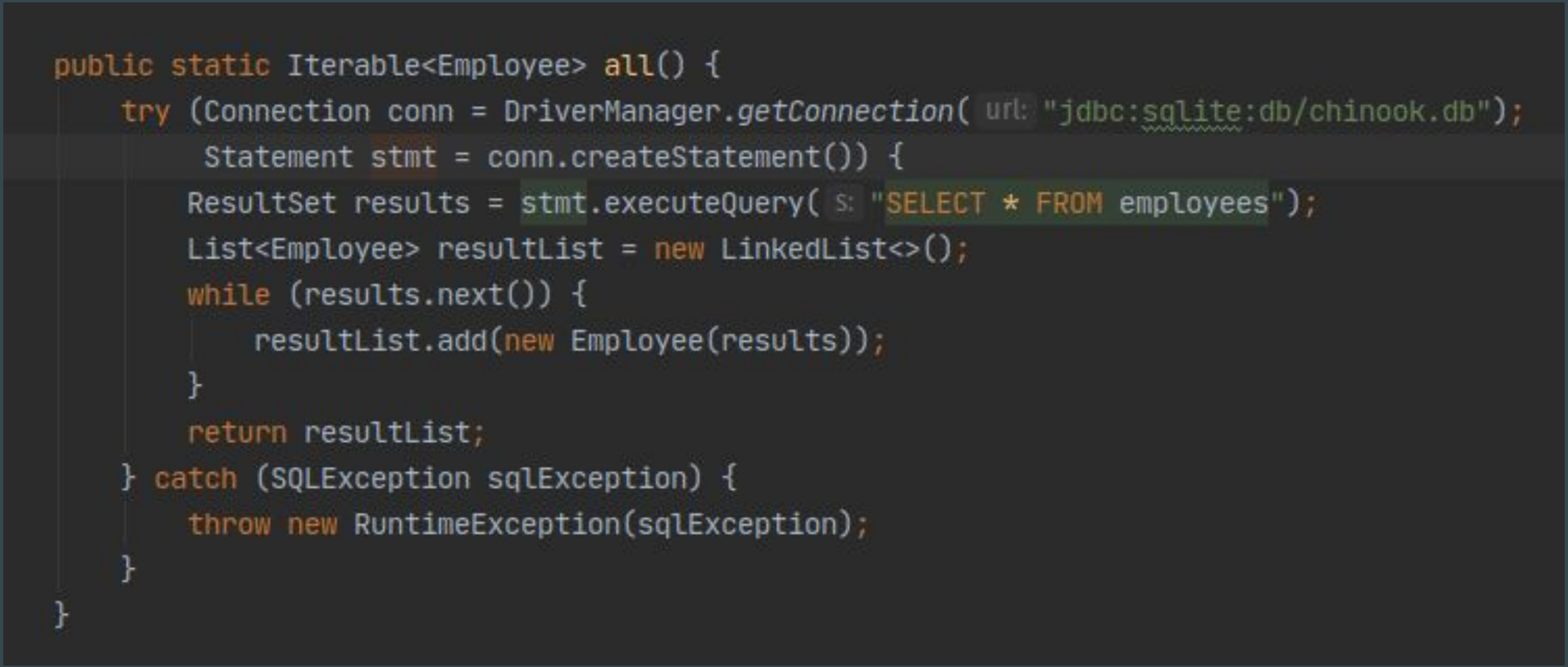
Problems with Mocks

- More practically for us, mocks require a notion of an interface
- C has no native notion of interfaces
- So no mocking! (*)

Integration Tests

- Don’t mock out subsystems
- Use live and full implementations of all dependencies
- You are testing the “integration” between various layers of your software

A Complex Method To Integration Test



```
public static Iterable<Employee> all() {
    try (Connection conn = DriverManager.getConnection( url: "jdbc:sqlite:db/chinook.db");
         Statement stmt = conn.createStatement()) {
        ResultSet results = stmt.executeQuery( s: "SELECT * FROM employees");
        List<Employee> resultList = new LinkedList<>();
        while (results.next()) {
            resultList.add(new Employee(results));
        }
        return resultList;
    } catch (SQLException sqlException) {
        throw new RuntimeException(sqlException);
    }
}
```

Integration Tests - Pros

- Much more realistic
- Allow you to catch tricky operational semantic issues between systems
- Often written at a “higher level” than unit tests
- Therefore they remain relevant longer

Integration Tests - Cons

- Slower
- When something breaks due to a spooky dependency issue they can be quite difficult to debug

Validation & Acceptance Testing

- Typically more of a sociological problem for developers
- Validation Testing is usually done by a Q/A department
    - Q/A/Development dynamics can be difficult
    - “Works For Me”-ism
    - Feedback loop can be very long
- Acceptance Testing is typically done on site with customers
    - Customers can be finicky
    - Feedback loop can be *extremely* wrong
    - Disconnect between what sales promised and what engineering built becomes apparent
        - Guess who gets blamed?

Test Driven Development

- “Write your tests first” (Really, Test *first* development)
- An ideological position for some people
- There is evidence that testing does improve coder productivity
- There is evidence that test-first produces more tests
- Therefore, test-first improves coder productivity
    - *Where’s the flaw in this reasoning?*

My Take

- Unit tests are useful when you get going, but be prepared to throw them out
- Integration tests that test at *the right level of abstraction* are extremely valuable
  - The right level of abstraction: the level at which the semantics of your system remain relatively stable independent of implementation
- Test first isn’t very useful, especially when you are still figuring out what you are building
- All that said: *TEST YOUR CODE*

Continuous Integration

- A big step forward for development has been continuous integration
- The practice of merging all developers' working copies to a shared mainline several times a day
  - And *testing* it
- CI servers now are expected to
  - Compile code
  - Run tests
  - Sometimes deploy code
    - Often to a test branch or staging branch
    - Production is also a possibility (not a fan)

Continuous Integration

- There are lots of CI tools
- They all have their issues
- Many now integrate with Github
- Big Ones
  - Jenkins
  - Bamboo
  - Circle CI

Testing in C

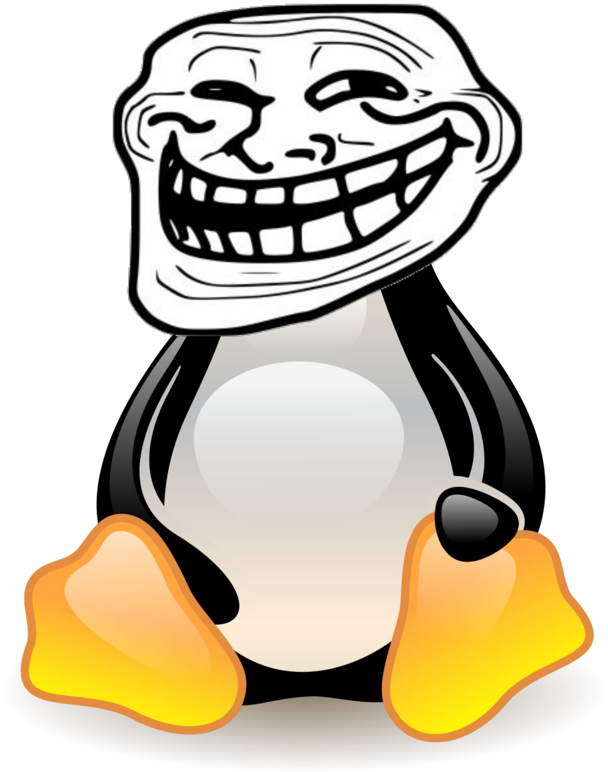
- C does not have a culture of testing

  “The linux kernel has a heavy emphasis on community testing.”

Testing in C

- C does not have a culture of testing

  “The linux kernel has a heavy emphasis on community testing.”



Testing in C

- C++ has a bit more of a culture of testing
- We will be using a C++ testing library, google test, to test our project

Testing in C

- C doesn’t have interfaces?
- True, but C is also *functional*
- We can design functions to be testable
  - Try not to update state
  - Return values that can be asserted against
- What can’t we test in the project?

Testing in C

“The CSCI 366 project has a heavy emphasis on community testing.”

Testing in C

> *“The CSCI 366 project has a heavy emphasis on community testing.”*

MONTANA
STATE UNIVERSITY