CS189A  Testing 1... 2 ... 3
Testing can be great!

- Many people
  - Write code without being sure it will work
  - Press run and pray
  - If it fails, they change something random

- This
  - Never works
  - And ruins your Friday evening social plans

- Test-Driven Development saves the day!
A collection class SmallSet

Semantics for a set data structure

containing up to N objects (hence “small”)

typical operations:

- add: adds item
- contains: item in the set?
- size: # items

we’ll implement add(), size()
Test Driven Development

- We’ll go about in small iterations
  1. add a test
  2. run all tests and watch the new one fail
  3. make a small change
  4. run all tests and see them all succeed
  5. refactor (as needed)

- We’ll use JUnit
JUnit

- What do JUnit tests look like?

SmallSet.java
package edu.ucsb.cs.cs189a;

class SmallSet {
  ...
}

SmallSetTest.java
package edu.ucsb.cs.cs189a;

import org.junit.Test;
import static org.junit.Assert.*;

class SmallSetTest {
  @Test public void testFoo() {
    SmallSet s = new SmallSet();
    ...
    assertTrue(...);
  }

  @Test public void testBar() {
    ...
  }
}
We start by thinking about how to test, not how to implement:

- size=0 on empty set
- size=N after adding N distinct elements
- adding element already in set doesn’t change it
- throw exception if adding too many
- ...

Each test verifies a certain “feature”
A First Test

- We pick a feature and test it:

```java
class SmallSet {}

class SmallSetTest {
  @Test public void testEmptySetSize() {
    SmallSet s = new SmallSet();
    assertEquals(0, s.size());
  }
}
```

- This doesn’t compile: `size()` is undefined

- But that’s all right: we’ve started designing the interface by using it
A test can be defined before the code is written:

```java
class SmallSet {
    public int size() {
        return 42;
    }
}
```

Running the test yields a red bar indicating failure:

If we add the size function and re-run the test, it works!
What’s the simplest way to make a test pass?

“Fake it till you make it”

Re-running yields the legendary JUnit Green Bar:

We could now refactor, but we choose to move on with the next feature instead.
Adding Items

- To implement adding items, we first test for it:

```java
SmallSetTest
class SmallSetTest {
    @Test public void testEmptySetSize() {
        @Test public void testAddOne() {
            SmallSet s = new SmallSet();
            s.add(new Object());
            assertEquals(1, s.size());
        }
    }
}
```

- `add()` is undefined, so to run the test we define it:

```java
SmallSet
public int size() {
    return 0;
}
public void add(Object o) {
    size++;
}
```
Adding Items

- The test now **fails** as **expected**:

- It seems obvious we need to count the number of items:

  ```java
  SmallSet
  private int _size = 0;
  
  public int size() {
    return _size;
  } 
  
  public void add(Object o) {
    ++_size;
  }
  ```

- And we get a green bar:
So what if we added an item already in the set?

```java
SmallSetTest
class SmallSetTest {
    @Test public void testEmptySetSize() ...

    @Test public void testAddOne() ...

    @Test public void testAddAlreadyInSet() {
        SmallSet s = new SmallSet();
        Object o = new Object();
        s.add(o);
        s.add(o);
        s.add(o);
        assertEquals(1, s.size());
    }
}
```

As expected, the test fails...
Remember that Item?...

- We need to remember which items are in the set...

```java
public void add(Object o) {
    for (int i=0; i < MAX; i++) {
        if (_items[i] == o) {
            return;
        }
    }
    _items[_size] = o;
    ++_size;
}
```

- All tests pass, so we can refactor that loop...
Refactoring

- (...) loop) which doesn’t “speak to us” as it could...

- All tests still pass, so we didn’t break it!
Too Many

What if we try to add more than SmallSet can hold?

```java
SmallSetTest
...
@Test public void testAddTooMany() {
    SmallSet s = new SmallSet();
    for (int i=0; i < SmallSet.MAX; i++)
    {
        s.add(new Object());
    }
    s.add(new Object());
}
```

The test fails with an error: ArrayIndexOutOfBoundsException

We know why this occurred, but it should bother us: “ArrayIndex” isn’t a sensible error for a “set”
Size Matters

- We first have `add()` check the size, define our own exception...

```java
public void add(Object o) {
    if (!inSet(o) && _size < MAX) {
        _items[_size] = o;
        ++_size;
    }
}
```

- ... re-run the tests, check for green, and...

```java
public class SmallSetFullException extends Error {}
```

- ... re-run the tests, check for green, and...
... finally test for our exception:

```java
SmallSetTest
@Test public void testAddTooMany() {
    SmallSet s = new SmallSet();
    for (int i=0; i < SmallSet.MAX; i++) {
        s.add(new Object());
    }
    try {
        s.add(new Object());
        fail("SmallSetFullException expected");
    } catch (SmallSetFullException e) {}  
}
```

The test fails as expected, so now we fix it...
Testing for Exceptions

... so now we modify add() to throw:

```java
SmallSet
public void add(Object o) {
    if (!inSet(o) && _size < MAX) {
        if (_size >= MAX) {
            throw new SmallSetFullException();
        }
        _items[_size] = o;
        ++_size;
    }
}
```

All tests now pass, so we’re done:
Review

- Started with a “to do” list of tests / features
  - could have been expanded as we thought of more tests / features

- Added features in small iterations

- “a feature without a test doesn’t exist”
Is testing obligatory?

- Yes and no…
  - When you write code in professional settings with teammates, definitely!
    - In such settings, failing to test your code just means you are inflicting errors you could have caught on teammates!
    - At Google, people get fired for this sort of thing!
  - So… in industry… test or perish!

- But what if code is just “for yourself”?
  - Testing can still help you debug, and if you go to the trouble of doing the test, JUnit helps you “keep it” for re-use later.
  - But obviously no need to go crazy in this case
A bug can reveal a missing test

... but can also reveal that the specification was faulty in the first place, or incomplete
- Code “evolves” and some changing conditions can trigger buggy behavior
  - Perhaps specification needs to evolve
- This isn’t your fault or the client’s fault but finger pointing is common

Great testing dramatically reduces bug rates
- And can make fixing bugs way easier
- But can’t solve everything: Paradise isn’t attainable in the software industry
Reasons for TDD

- By writing the tests first, we
  - test the tests
  - design the interface by using it
  - ensure the code is testable
  - ensure good test coverage

- By looking for the simplest way to make tests pass,
  - the code becomes “as simple as possible, but no simpler”
  - may be simpler than you thought!
There’s a lot more worth knowing about TDD

- What to test / not to test
  - e.g.: external libraries?
- How to refactor tests
- Fixtures
- Mock Objects
- Crash Test Dummies
- ...

Beck, Kent: *Test-Driven Development: By Example*
How people build really big programs

- When applications are small, you can understand every element of the system.

- But as systems get very large and complex, you increasingly need to think in terms of interfaces, documentation that defines how modules work, and your code is more fragmented.

- This forces you into a more experimental style.
  - Test that the code you use works (cover your behind).
JUnit testing isn’t enough

- For example, many systems suffer from “leaks”
  - Such as adding more and more objects to an ArrayList
  - The amount of memory just grows and grows

- Some systems have issues triggered only in big deployments, like cloud computing settings
- Sometimes the application “specification” was flawed, and a correct implementation of the specification will look erroneous to the end user
- But a thorough test plan can reveal all such problems

- Quality Assurance Q/A to the rescue (teams/members testing integration, UI/UX, full system in different settings)
The Q/A cycle

- Real companies have quality assurance teams

- They take the code and refuse to listen to all the long-winded explanations of why it works

- Then they do their own, independent, testing

- And then they send back the broken code with a long list of known bugs!

- Separating development from Q/A really helps
Q/A Can Help But...

- The best code written by professionals will still have some rate of bugs
  - They reflect design oversights, or bugs that Q/A somehow didn’t catch
  - Evolutionary change in requirements
  - Incompatibilities between modules developed by different people, or enhancements made by people who didn’t fully understand the original logic

- So never believe that software will be flawless
- Our goal in CAPSTONE is to do as well as possible
- What if I want to know more: Check out fault tolerance!