CS 2110
Software Design Principles II

Juan Altmayer Pizzorno
port25.com
Overview

- Last week:
  - Design Concepts & Principles
  - Refactoring

- Today: Test-Driven Development
  - TDD + JUnit by Example
The Example

- A collection class SmallSet
  - 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.cornell.cs.cs2110;

public class SmallSet {
    ...
}

SmallSetTest.java
package edu.cornell.cs.cs2110;

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

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

    @Test public void testBar() {
        ...
    }
}
A List of Tests

• 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
SmallSet
class SmallSet {
}

SmallSetTest
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
Red Bar

- We **need** the test to **fail**, so we define `size()`

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

- Running the test yields a red bar indicating failure:

- We’ve **tested** the test, and it works!
Green Bar

• What’s the simplest way to make the test pass?

```java
SmallSet
class SmallSet {
    public int size() {
        return 0;
    }
}
```

• “Fake it till you make it”

• Re-running yields the legendary JUnit Green Bar:

![JUnit Green Bar](image)

• 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 code...
    }
  }
  @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
class SmallSet {
  public int size() {
    // Method implementation...
  }
  public void add(Object o) {
    // Method implementation...
  }
  }
  ```
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 0;
    return _size;
}

public void add(Object o) {
    ++_size;
}
```

- And we get a green bar:
Adding Something Again

• 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);
        assertEquals(1, s.size());
    }
}
```

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

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

```java
SmallSet
private int _size = 0;
public static final int MAX = 10;
private Object _items[] = new Object[MAX];

...
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...

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

SmallSet (after)
private boolean inSet(Object o) {
  for (int i=0; i < MAX; i++) {
    if (_items[i] == o) {
      return true;
    }
  }
  return false;
}

public void add(Object o) {
  if (!inSet(o)) {
    _items[_size] = o;
    ++_size;
  }
}

- 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`

• “Array...” makes no sense on a Set abstraction...
Size Matters

• We first have `add()` check the size,

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

• ... re-run the tests, check for green, define our own exception...

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

• ... re-run the tests, check for green, and...
Testing for Exceptions

- ...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”
Fixing a Bug

• What if after releasing we found a bug?
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!
Not the Whole Story

• 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