CS 2110

Based on slides originally by
Juan Altmayer Pizzorno
port25.com

Software Design Principles II

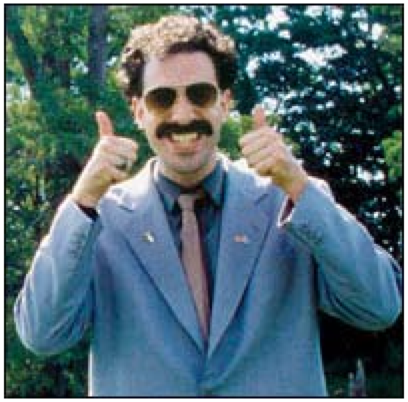
Overview

- Last week:
  - Design Concepts & Principles
  - Refactoring

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

- We use JUnit testing to evaluate your homework assignments…

Tests can be great!



"In my country of Kazakhstan testing is very nice! Make many tests please!

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!

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?

```
[redacted]ornell.cs.cs2110;

SmallSet {
```

```
[redacted]ll.cs.cs2110;

Test;
.junit.Assert.*;

lSetTest {
oid testFoo() {
= new SmallSet();

...);


oid 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:

```
[redacted]allSet {}


          etTest {
          lic void testEmptySetSize() {
          et s = new SmallSet();
          Equals(0, s.size());
```

- T [redacted] t compile: `size()` is undefined
- B [redacted] l right: we’ve started designing the in [redacted] using it

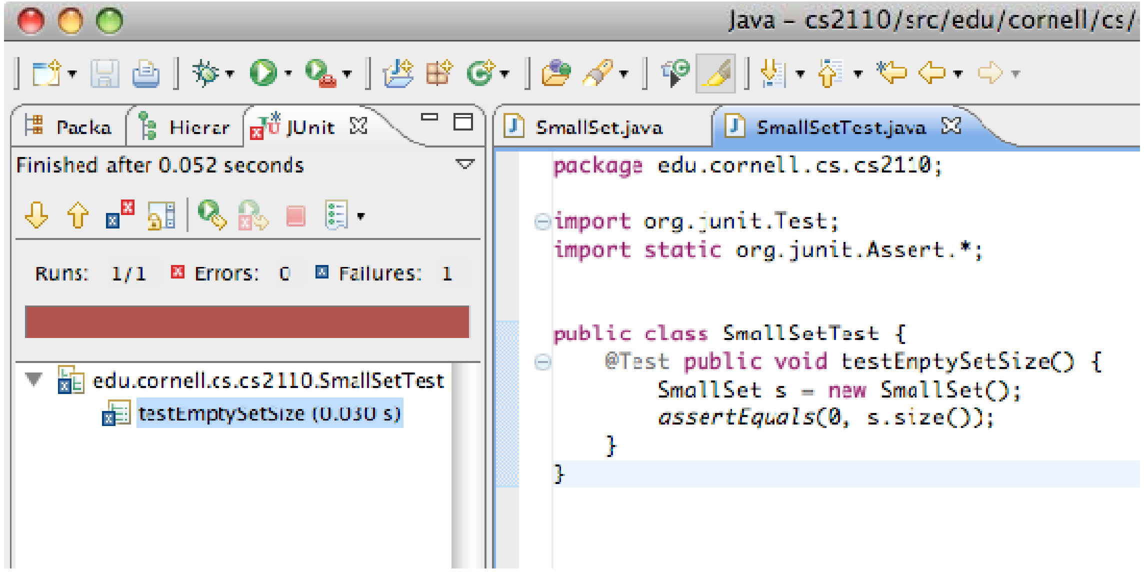
Red Bar

- A test can be defined *before* the code is written

[redacted]

```
allSet {
c int size()

turn 42;
```

ning the test ds a red bar cating failure:



Finished after 0.052 seconds

Runs: 1/1 Errors: 0 Failures: 1

edu.cornell.cs.cs2110.SmallSetTest
testEmptySetSize (0.030 s)

```
package edu.cornell.cs.cs2110;

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


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

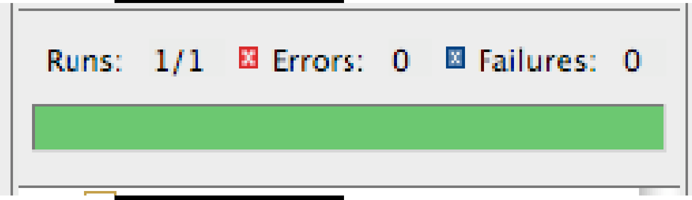
e add the size function and re-run the the , it works!

Green Bar

- What’s the simplest way to make a test pass?

```
[redacted]allSet {
          c int size() {
          turn 0;
```

- “Fake it till you make it”
- Re-running yields the legenda[redacted]nit Green Bar:



```
Runs:  1/1   ☒ Errors:  0   ☒ Failures:  0
```

- We could now refactor, but w[redacted]se to move on with the next feature inste[redacted]

Adding Items

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

```
[redacted]etTest {
lic void testEmptySetSize() ...

lic void testAddOne() {
et s = new SmallSet();
new Object());
Equals(1, s.size());
```

- `add(`[redacted]ined, so to run the test we defi[redacted]

```
[redacted]nt size() ...

oid add(Object o) {}
```

Adding Items

- The test now fails as expected:
- It seems obvious we need to count the number of items:

[redacted]

```
int _size = 0;

nt size() {
n 0;
n _size;


oid add(Object o) {
ze;
```

- And we get a gre

Adding Something Again

- So what if we added an item already in the set?

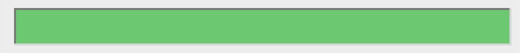
```
[redacted]etTest {
lic void testEmptySetSize() ...

lic void testAddOne() ...

lic void testAddAlreadyInSet() {
et s = new SmallSet();
 o = new Object();
o);
o);
Equals(1, s.size());
```

- As expe test fails...

Remember that Item?...

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

```
[redacted]
int  size = 0;
tatic final int MAX = 10;
Object  items[] = new Object[MAX];

oid add(Object o) {
int i=0; i < MAX; i++) {
 (_items[i] == o) {
 return;



s[_size] = o;
ze;
```



- All test s, so we can refactor that loop...

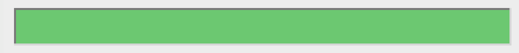
Refactoring

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

```
[redacted] (before)
oid add(Object o) {
int i=0; i < MAX; i++) {
 (_items[i] == o) {
 return;




s[_size] = o;
ze;
```

```
[redacted]inSet(Object o)

 i < MAX; i++)

[i] == o) {
true;






Object o) {
) {
ze] = o;
```

ll tests still pass, so [redacted] reak it!

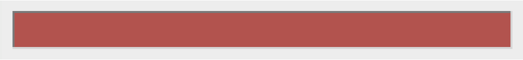


Too Many

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

```
[redacted] void testAddTooMany() {
s = new SmallSet();
i=0; i < SmallSet.MAX; i++)

new Object());

 Object());
```

- The test f [redacted] an error: `ArrayInde` [redacted] `ndsException`



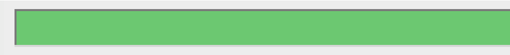
- We know [redacted] occurred, but it should bother us: “Array [redacted] n’t a sensible error for a “set”

Size Matters

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

```
[redacted]oid add(Object o) {
          inSet(o) && _size < MAX) {
          tems[_size] = o;
          _size;
```

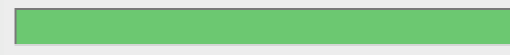
- .. un the tests, check for green,



  d our own exception...

```
[redacted]FullException extends Error {}
```

- .. ts, check for green,



  a

Testing for Exceptions

- ... finally test for our exception:

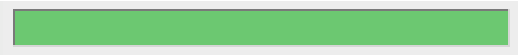
```
[redacted] void testAddTooMany() {
s = new SmallSet();
i=0; i < SmallSet.MAX; i++) {
new Object());


new Object());
SmallSetFullException expected”);

allSetFullException e) {}
```

- T ails as expected,
  s e fix it...

Testing for Exceptions

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

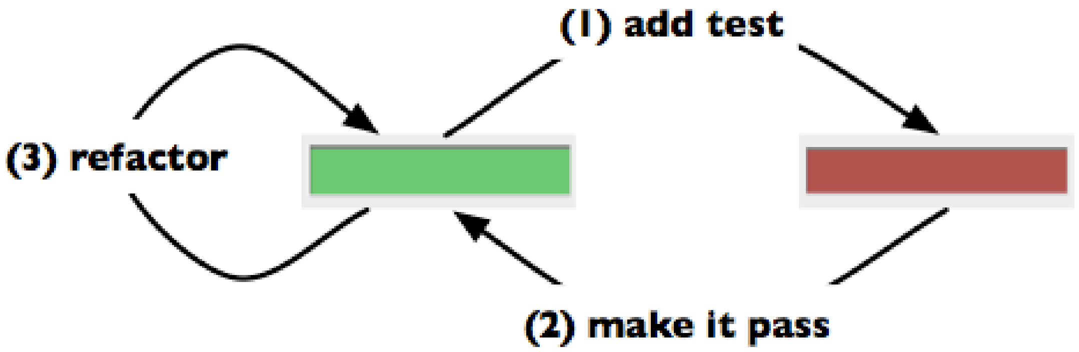
```
[redacted]oid add(Object o) {
  ~~inSet(o) &&~~ size < MAX) {
   (_size >= MAX) {
    throw new SmallSetFullException();

  tems[_size] = o;
  _size;
```



- sts 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

Fixing a Bug

- What if after releasing we found a bug?

Famous last words: “It works!”

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
  - 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!

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*

Even so…

- The best code written by professionals will still have some rate of bugs
  - They reflect design oversights
  - Evolutionary change in requirements
  - Incompatibilities between modules developed by different people

- So never believe that software will be flawless
- Our goal in cs2110 is to do as well as possible
- In later cs courses we’ll study “fault tolerance”!