Today

- Quiz 10
- Exam overview
- Test Driven Development (TDD)

Homework

- HW 5 due
- HW 6 out

Reminder

- Project updates in 1 week
  - 6 minute presentation with demos
  - see Proj 8
Exam 2 Overview

Basics:

- Closed book, closed notes
- Worth 10% of final grade

Topics:

- Planning
- Carrying out iterations
- Task boards, burn-down
- Estimation (count-compute-judge, why accuracy)
- SCM and svn
- Testing basics
- JUnit
- Test-driven development (today)
Test-Driven Development

“After implementing a small part of the feature, I found that some of the implemented functionality was similar to something in another class, therefore I started changing that class so I could use it instead. During that work I discovered a defect that I then began to fix. However, to fix it I had to change the internal data structure a lot, which led to adding a parameter to a few of the class methods. Therefore I had to change all the places where these methods were called. Then I noticed ...”

[Christensen]

Test-Driven Development (TDD) helps avoid wild (coding) goose chases
TDD emphasizes **taking small steps** and **keeping focused**

- make one tiny change at a time
- focus on 1 step at a time, and write down others for later

**TDD emphasizes testing first!**

1. Write (failing) tests
2. Make tests pass
3. Refactor

**Fix design issues**

The basic TDD design process: ... about **design** as much as testing

- Create a **walking skeleton** (basic design)
- **Grow the design** in small steps
  - tests (go from user stories → API)
  - refactor (improve code/design)
- **Focus** on only what is needed
  - keeping focus is one of the hardest aspects of TDD!
Refactoring

Improve design of **existing** code

- modify and **restructure** source code
- to improve **flexibility** and **maintainability**
- without changing external behavior

That is, “Designing w/ a purpose”

After refactoring **no new features** added

- but make it **easier** to change later
  - remove duplicate code
  - rename methods
  - reorganize methods
  - change internal data structures
- or make it more **robust to change**
  - small generalizations
  - adopt software design patterns
Basic TDD Principles

Test First

Automate Tests

Use “Test Lists”

- write tests you know are needed
- add more as you find new ones

Follow 5-Step TDD “Rhythym”

1. (Quickly) add a test
2. Run all tests, see new one fail
3. Make a little change (to make it pass) ... more on this soon
4. Run tests, see them succeed
5. Refactor to remove duplication

- move quickly (takes practice)
- stay focused (do only what is needed to get tests to pass)
TDD Example [Christensen]

Parking lot pay station software

Basic user stories

- buy a parking ticket ... add coins, 5 cents for 2 minutes, get receipt
- cancel transaction ... add coins, cancel, get coins back
- reject illegal coins

Basic UI and architecture
The PayStation interface:

```java
public interface PayStation {
    public void addPayment(int coinValue)
        throws IllegalCoinException;
    public int readDisplay();
    public Receipt buy();
    public void cancel();
}

public interface Receipt {
    public int value();
}
```

- Super simple to illustrate TDD
  - coins represented by int values
  - coin validation done in pay station

Q: Initial Test List?

- Accept legal coin
- 5 cents should give 2 minutes parking time
- reject illegal coin
- read display
- buy produces valid receipt
- cancel resets pay station
PASS 1

Q: Which test to write first?

**TDD Principle: One step test**

Pick a test that:

- will **teach** you something
- you are **confident** you can implement

For example

- **read display** is trivial (won’t learn anything new about system)
- sometimes a list will have tests you’ll have no clue how to implement (at that point)
- **buy produces valid receipt** seems more complicated, but relies on other features (so wait until we learn more)
- **5 cents** story seems good ...
STEP 1: Add a test

```java
public class PayStationImplTest {
    @Test
    public void shouldDisplay2minFor5Cents() throws IllegalCoinException {
        PayStation s = new PayStationImpl();
        s.addPayment(5);
        assertTrue(s.readDisplay() == 2);
    }
}
```

STEP 2: Run all tests, see new one fail

Q: What is going to happen?

- We haven’t written any code yet!
- Lots of compile errors

So, we’ll implement a minimal PayStationImpl using “stubs”:

- all method bodies empty
- return 0, null, etc.
public class PayStationImpl implements PayStation {
    public void addPayment(int coinValue) throws IllegalCoinValue {
    
    }

    public int readDisplay() {
        return 0;
    }

    public void cancel() {
    }
}

In TDD, always take small steps!

Now we can run and see the test fail

STEP 3: Make a little change to make it pass

    Q: What should we do now?

TDD Principle: Fake it ('til you make it)

    • e.g., return a constant ... some poeple don’t like this
Our change:

```java
public int readDisplay() {
    return 2;
}
```

Note this is exactly what is needed to get the test to pass!

**STEP 4:** Run all tests, and see them succeed

- we actually tested many of our initial cases:
  - accept legal coin
  - 5 cents for 2 minutes
  - read display

- But is there a problem?
  - although passes, our code is **incomplete**!
  - only works for one case (2 mins)

Q: but why waste time, when it’ll have to change?
Remember

• focus was only the 2 min case
• we implement the smallest amount for this case
• our code is driven by our tests

Key Point

• only implement what your tests demand
• if you need more functionality, you need more/better tests!

TDD Principle: Triangulation

• Abstract only when you have 2 or more test cases

So, we need to add more tests here

• e.g., 25 cents should give 10 minutes parking time

Faking it here helped us stay focused and take small steps

STEP 5: Refactor to remove duplication

• no duplication, so we skip this step
PASS 2

STEP 1: try to get rid of “fake it” code

• don’t want it to stay long and/or accumulate

Pick: 25 cents should give 10 minutes

How do we add a test?

• to previous (JUnit) test?
• as a new (JUnit) test? ... Yes!

TDD Principle: Isolated Tests

• Tests should never affect one another (and test only one thing!)
• helps find and fix bugs faster (1 test fails, 1 simple thing)

Our new test:

```java
@Test
public void shouldDisplay10minFor25Cents() throws IllegalCoinException {
    PayStation s = new PayStationImpl();
    s.addPayment(25);
    assertTrue(s.readDisplay() == 10);
}
```
STEP 2: Obviously fails

STEP 3: Multiply by whatever entered so far

```java
public class PayStationImpl implements PayStation {
    private int inserted;
    public void addPayment(int coinValue)
        throws IllegalCoinException
    {
        inserted = coinValue;
    }
    public int readDisplay() {
        return (inserted / 5) * 2;
    }
}
```

Note the code is still **incomplete**!

- but += not yet driven by a test!
- we need to add a test: enter 2 or more legal coins

STEP 4: All tests pass!

STEP 5:

- no code duplication
- but we do have some test code duplication
- creating new Pay Stations in each test ...
JUnit “Fixtures”

- test setup code
- is run before every test is run (not just once)

```java
private PayStation s;
@Before
public void setUp() {
    p = new PayStationImpl();
}
@Test
public void shouldDisplay10minFor25Cents() throws IllegalCoinException {
    p.addPayment(25);
    assertTrue(p.readDisplay() == 10);
}
```

TDD Principle: Evident Data

- represent the intent of data
- include expected and actual results in the test
- make their relationships apparent
- write tests for the reader
For example:

```java
    p.addPayment(25);
    assertTrue( p.readDisplay() == (25/5)*2 );
```

- put the actual calculation here
- adding comments also helps
- don’t just copy and paste though

After you refactor, make sure all your tests pass

Q: What is this called? regression testing