Today

- Test Driven Development (wrap up)
- Estimation (cont.)

Homework

- HW 6 now due Tues.
- HW 7 and 8 assigned Tues.
Basic TDD Principles (review)

Test First

Automate Tests

Use “Test Lists”

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

Follow 5-Step TDD “Rhythm”

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

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 people 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

    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
Estimation (cont.)

Estimation questions ...

- Why worry about accurate estimates?
- Where do estimation errors come from?
- How should estimates be presented?
- What is being estimated?

Where do estimation errors come from?

1. information about the project being estimated
2. information about the capabilities of the organization
3. too much “chaos” in the project (estimating a moving target)
4. the estimation process itself

We’ll focus on the (inherent) uncertainty lingering in the estimation process
Estimation Uncertainty

Single-point estimates suggest 100% probability (no uncertainty)

- this is not realistic!
A common **assumption** is that software project outcomes follow a **bell curve** (normal distribution) ...

- The area under the curve adds to 100%
- 1 standard deviation adds to \( \approx 68\% \)
- 2 standard deviations add to \( \approx 95\% \)
- 3 standard deviations add to \( \approx 99\% \)
Q: What’s wrong with assuming a normal distribution here?

- there is a limit to how well a project can be conducted
- there is no limit to how poorly a project can go!

- this complicates things ...
- we’ll assume a normal distribution
- but you should keep in mind the non-normal distribution

**The Plan**: Revisit best and worst case (from the exercise)

- it turns out that we can use these to help clarify estimation uncertainty
- and this helps reduce estimation errors (from single-point estimates)
- can also help to refine our estimation “accuracy” over time