Software Engineering

- The art by which we start with a problem statement and gradually evolve a solution.

- There are whole books on this topic and most companies try to use a fairly uniform approach that all employees are expected to follow.

- The IDE can help by standardizing the steps.

Top-Down Design
- Building a Search Engine:
  - Search Engine
  - Crawler
  - Indexer
  - User Interface
  - Queue
  - HTTP Client
  - Database
  - Inverted Index
  - Ranking Function
  - Spelling Correction
  - HTTP Server

- Refine the design at each step
- Decomposition / “Divide and Conquer”

Bottom-Up Design
- Just the opposite: start with parts:
  - Search Engine
  - Crawler
  - Indexer
  - User Interface
  - Queue
  - HTTP Client
  - Database
  - Inverted Index
  - Ranking Function
  - Spelling Correction
  - HTTP Server

- Composition
- Build-It-Yourself (e.g. IKEA furniture)

Top-Down vs. Bottom-Up
- Is one of these ways better? Not really!
  - It’s sometimes good to alternate
  - By coming to a problem from multiple angles you might notice something you had previously overlooked
  - Not the only ways to go about it

- With Top-Down it’s harder to test early because parts needed may not have been designed yet
- With Bottom-Up, you may end up needing things different from how you built them

Software Process
- For simple programs, a simple process...
  - “Waterfall”

- But to use this process, you need to be sure that the requirements are fixed and well understood!
  - Many software problems are not like that
  - Often customer refines the requirements when you try to deliver the initial solution!
Incremental & Iterative

- Deliver versions of the system in several small cycles:
- Recognizes that for some settings, software development is like gardening.
- You plant seeds... see what does well... then replace the plants that did poorly.

The Importance of Testing

- Famous last words
  - “It’s all done, I just have not tested it yet”.
- Many people
  - Write code without being sure it will work
  - Press run and pray
  - If it fails, they change something random
  - Never work, and ruins weekend social plans.
- Test-Driven Development!

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

Testing and Test-Driven Development

- A collection class SmallSet
  - containing up to N objects (hence “small”)
  - typical operations:
    | Method | Description |
    |--------|-------------|
    | add    | adds item   |
    | contains | is item in the set? |
    | size   | # items     |
  - we’ll implement add(), size()

The Example

- A collection class SmallSet
  - containing up to N objects (hence “small”)
  - typical operations:
    | Method | Description |
    |--------|-------------|
    | add    | adds item   |
    | contains | is item in the set? |
    | size   | # items     |
  - we’ll implement add(), size()

JUnit

- What do JUnit tests look like?
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 size
  - throw exception if adding too many
- ...  
- Each test verifies a certain “feature”

A First Test

- We pick a feature and test it:

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

```
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

- A test can be defined before the code is written
- Running the test yields a red bar indicating failure:
- If we add the size function and re-run the test, it works!

Green Bar

- What’s the simplest way to make a test pass?
- “Fake it till you make it”
- Re-running yields the legendary JUnit Green Bar:
- Move on with the next feature

Adding Items

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

```
SmallSet
public int size() {
  return _size;
}
```

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

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

```
public int size() {
  return _size;
}
```

- And we get a green bar:
Adding Something Again

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

```
public class SmallSetTest {
    @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...

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

• All tests pass, so we can refactor that loop...

Refactoring

• FOR-loop doesn’t "speak to us" as it could...

```
public void add(Object o) {
    if (!inSet(o) && _size < MAX) {
        _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?

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

• The test fails with an error: ArrayIndexOutOfBoundsException

Testing for Exceptions

• ...finally test for our exception:

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

public void testAddTooMany() {
    SmallSet s = new SmallSet();
    for (int i=0; i < SmallSet.MAX; i++) {
        s.add(new Object());
    }
    s.add(new Object());
    try {
        s.add(new Object());
    } catch (SmallSetFullException e) {}{
```

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

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

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

• All tests now pass, so we’re done:

After all Tests are Passed

• Is the code is correct?
  – Yes, if we wrote the right tests.
• Is the code efficient?
  – Probably used simplest solution first.
  – Replace simple data structures with better data structures.
• Is the code readable, elegant, and easy to maintain?
  – It is very common to find some chunk of working code, make a replica, and then edit the replica.
  – But this makes your software fragile.
    • Later changes have to be done on all instances, or
    • some become inconsistent
    – Duplication can arise in many ways:
      • constants (repeated “magic numbers”)
      • code vs. comment
      • within an object’s state
• ...”DRY” Principle

• Don’t Repeat Yourself
• A nice goal is to have each piece of knowledge live in one place
• But don’t go crazy over it
  – DRYing up at any cost can increase dependencies between code
  – “3 strikes and you refactor” (i.e., clean up)

“DRY” Principle

Simple Refactoring

• Renaming variables, methods, classes for readability.
• Explicitly defining constants:

```java
public double weight(double mass) {
    return mass * 9.80665;
}
```

```java
static final double GRAVITY = 9.80665;
public double weight(double mass) {
    return mass * GRAVITY;
}
```

• If your application later gets used as part of a NASA mission to Mars, it won’t make mistakes
• Every place that the gravitational constant shows up in your program a reader will realize that this is what she is looking at
• The compiler may actually produce better code

Extract Method

• A comment explaining what is being done usually indicates the need to extract a method

```java
public double totalArea() {
    // now add the circle area += PI * pow(radius,2);
    ...
}
```

• One of the most common refactorings

Extract Method

• Simplifying conditionals with Extract Method

```java
before
    if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
        charge = quantity * _winterRate + _winterServiceCharge;
    } else {
        charge = quantity * _summerRate;
    }
```

```java
after
    if (isSummer(date)) {
        charge = summerCharge(quantity);
    } else {
        charge = winterCharge(quantity);
    }
```
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

  (1) add test

  (2) refactor

  (3) make it pass

• “a feature without a test doesn’t exist”

Is testing obligatory?

• 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!
  – 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.
  – “I have never written a program that was correct before I tested and debugged it.” Prof. Joachims

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
How people big 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

Testing is a part of that style!

- Once you no longer know how big parts of the system even work (or if they work), you instead begin to think in terms of
  - Code you’ve written yourself. You tested it and know that it works!
  - Modules you make use of. You wrote experiments to confirm that they work the way you need them to work
  - Tests of the entire complete system, to detect issues visible only when the whole thing is running or only under heavy load

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

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

Why is Q/A a cycle?

- Each new revision may fix bugs but could also break things that were previously working
- Moreover, during the lifetime of a complex application, new features will often be added and those can also require Q/A
- Thus companies think of software as having a very long “life cycle”. Developing the first version is only the beginning of a long road!

Even with fantastic Q/A...

- 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 cs2110 is to do as well as possible
- In later CS courses we’ll study “fault tolerance”!