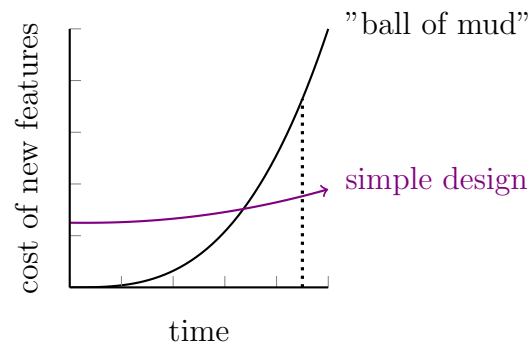


CO220 - Software Engineering Design

8th October 2019

Cost of Change



The "project heat death", denoted by the dotted line, is where the cost of adding new features outweighs the value gained by adding those features. Note that the initial cost of doing a simple design can be more expensive (since it requires more planning).

Elements of Simple Design

This is arranged in a pyramid on the slides (since they "build up on each other") but I will write it as a list, starting from the bottom;

1. behaves correctly

It doesn't matter if the codebase is well structured, or the code is elegant if it doesn't do the right thing (is buggy, or isn't what the customer wanted).

- automated testing
- test-driven development
- mock objects

2. minimises duplication

If something needs to be changed in the future, and it's in multiple places, it will have to be changed in all of those places which will take longer. Additionally, it's also easy to miss, causing bugs.

3. maximises clarity

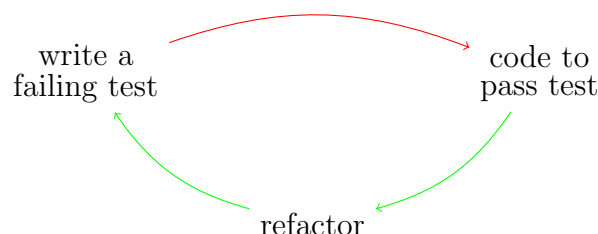
Code should be easy to modify, such that the parts that need to be changed can be easily located. Important especially if working with others.

4. has fewer elements

Less important - we want to focus on the previous levels first, and don't want to lose the benefits by combining elements.

Test-Driven Development / Behaviour Driven Development

Having a test suite provides confidence that the codebase still works, even after major changes.



We start by writing failing tests, which seems counterintuitive, as there is no code to test. However, these tests are written as if the code was working - which gives us a specification on how the code **should** behave. We want to write code as quickly as possible that gets us from the **red** state (failing tests), to a **green** state (passing tests). This code is likely untidy - we can then tidy it up (which shouldn't break the tests).

Additionally, it's not only about testing; we can replace the stages as follows;

- API design (write a failing test)
"I wish there was a method that would take these parameters and do this"
- internals design (code to pass test)
"Just making it work"
- structural design (refactor)
"How can we improve the design?" (pyramid layers)

We focus more on what it should do (how it should behave), and not how it does it. For example, `CustomerLookup` should do the following;

- finds customer by ID
- fails for duplicate customers

The test is named as the expected result, and if it is true, then it behaves correctly.

```
1 public class CustomerLookupTest {
2     @Test
3     findsCustomerById() {
4         ...
5     }
6     @Test
7     failsForDuplicateCustomers() {
8         ...
9     }
10 }
```

Example of TDD

The object `FibonacciSequence` should do the following;

- defines the first two terms to be one
- has each term equal to the sum of the previous two
- is not defined for negative terms

```
1 ... // (FibonacciSequenceTest.java)
2
3 import static org.hamcrest.CoreMatchers.is;
4 import static org.junit.Assert.assertThat;
5
6 import org.junit.Test;
7
8 public class FibonacciSequenceTest {
9
10     @Test
11     public void definesFirstTwoTermsToBeOne() {
```

```

12     assertThat(new FibonacciSequeunce().term(0), is(1));
13     assertThat(new FibonacciSequeunce().term(1), is(1));
14 }
15 }

```

Obviously, none of this will work yet, as the code doesn't exist. However, we can use this to create the code as follows (this is incorrect, but our tests now pass);

```

1 ... // (FibonacciSequence.java)
2
3 public class FibonacciSequence {
4
5     public int term(int i) {
6         return 1;
7     }
8 }

```

We can then add more tests, which should now fail;

```

1 ... // (FibonacciSequenceTest.java)
2
3 public class FibonacciSequenceTest {
4     ...
5
6     @Test
7     public void hasEachTermTheSumOfPreviousTwo() {
8         assert(new FibonacciSequeunce().term(2), is(2));
9         assert(new FibonacciSequeunce().term(3), is(3));
10        assert(new FibonacciSequeunce().term(4), is(5));
11    }
12 }

```

Similarly, we can modify the code again to add a naive implementation which performs it recursively;

```

1 ... // (FibonacciSequence.java)
2
3 public class FibonacciSequence {
4
5     public int term(int i) {
6         if (i < 2) {
7             return 1;
8         }
9         return term(i - 1) + term(i - 2);
10    }
11 }

```

Adding the last bullet point as a test;

```

1 ... // (FibonacciSequenceTest.java)
2
3 public class FibonacciSequenceTest {
4     ...
5
6     @Test
7     public void isNotDefinedForNegativeIndices() {
8         try {
9             new FibonacciSequeunce().term(-1);
10            fail("should have thrown exception")

```

```
11     } catch (IllegalArgumentException iae) {
12         assertThat(iae.getMessage(), containsString("negative index"))
13     }
14 }
15 }
```

Fixing this, we add the following;

```
1 ... // (FibonacciSequence.java)
2
3 public class FibonacciSequence {
4
5     public int term(int i) {
6         if (i < 0) {
7             throw new IllegalArgumentException("negative index not supported");
8         }
9
10        ...
11    }
12 }
```

This is the only time I will actually write out every step, since that's the focus of TDD.