50002 - Software Engineering - Lecture 2

Oliver Killane

15/10/21

Lecture Recording

Lecture recording is available here

Tests

```java
import org.junit.Test;

public class ObjectNameTest {

    // Code here is re-run before each test.
    private final Object somobject = new Object();

    @Test
    public void LongSentenceDescribingTest() {

        // Test Contents

        // Asserts
        assertEquals(expected, actual);
        assertTrue(expression);
        assertNull(expression);
        fail("Just fail the test");

        // AssertThat recommended where possible
        assertThat(a, is(b));
    }

    ...
}
```

Refactoring

1. Joshua Kerievsky defines refactoring as the process of improving the design of a piece of code, without changing it's behaviour.
2. The behaviour as observed through the public api should be the same.
3. When using TDD, we should only refactor when our tests are passing.

Joshua Kerievsky

The CEO of Industrial Logic. They are an agile consultation agency that helps companies improve their development cycles. He has also written several books such as "Refactoring to Patterns" and several e-learning courses.

By refactoring frequently, we help to eliminate a buildup of technical debt, and ensure that the codebase is kept at a high quality even as new features are added quickly.

Technical Debt

Ineligant, bloated, uncommented/obfuscated code can be itnroduced when new features are added quickly. If we do not fix these issues, it will become more and more difficult to understand the codebase, and add further new features. This deficit of code quality is the debt.

We sometimes introduce technical debt in an effort to get a new feature added quickly, we must be aware of the costs of leaving our repayment (refactoring) until later.

- **Mechanical Transformation**
  A large refactoring can be madee of many small transformations, each not altering the public behaviour.

  We can use modern development tools such as IDEs to perform these transformations more quickly and reliably than manual.

- **Hygiene**
  We can continually apply refactoring technique to make small improvements in our code. This prevents a large buildup of technical debt, or needing a large refactoring project to pay the debt off.

  The goal is to avoid major surgery on the codebase.

Catalogue of Refactorings

Extract Method

Take some code out of a method, and create a new one to hold it. This is called *extract method* in intellij.
Java Example:

```
1  class MyClass {
2      ...
3      public void myMethod() {
4          // CODE A
5
6          for (thing : things) {
7              // CODE B
8          }
9      }
10     ...
11 }
```

⇒

```
1  class MyClass {
2      ...
3      public void myMethod() {
4          codeA();
5
6          for (thing : things) {
7              codeB()
8          }
9      }
10
11     private void codeA(){
12         // CODE A
13     }
14
15     private void codeB(){
16         // CODE B
17     }
18     ...
19 }
```

Inlining Variables

Inline variables only used once.
C Example:

```c
int main(int arc, char **argv)
{
    int value;

    value = someCalculation();
    doSomething(value);

    return EXIT_SUCCESS;
}
```

```c
int main(int arc, char **argv)
{
    doSomething(someCalculation())
        ↪ ;

    return EXIT_SUCCESS;
}
```

Separate Responsibility

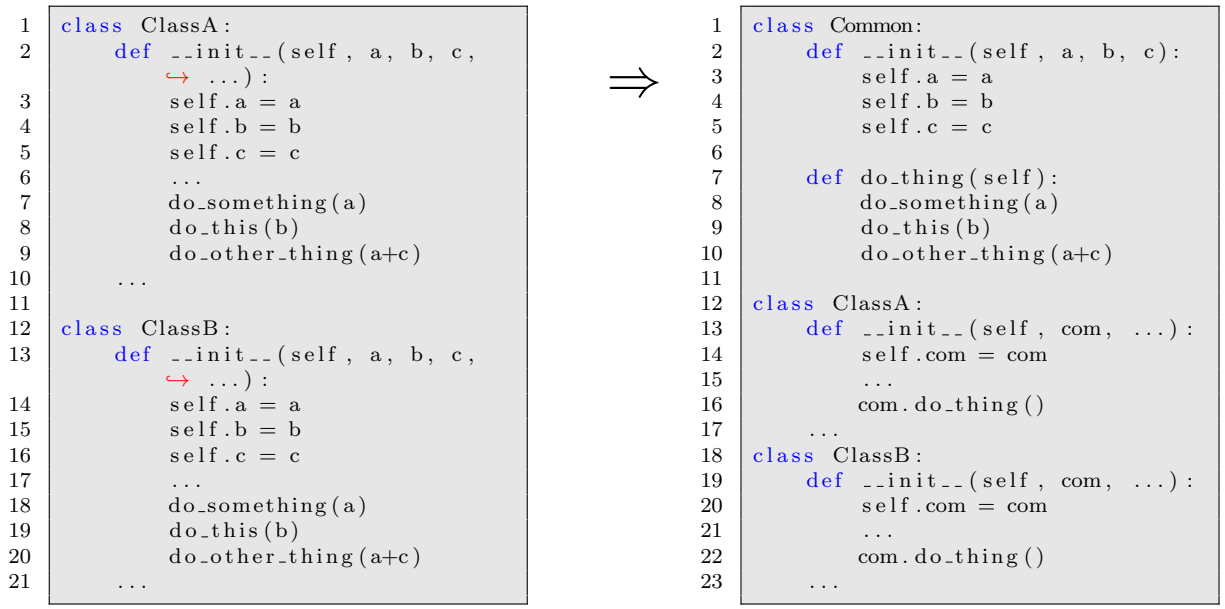
Separate the responsibilities for loops, functions etc. Each only does one 'thing'.
Rust Example:

```rust
fn some_calc(some_items : Vec<i32
    ↪ >) -> (i32, i32) {
    // two separate properties
    let mut a : i32 = 0;
    let mut b : i32 = 0;

    for item in some_items {
        a += a_calc(item);
        b += b_calc(item);
    }

    (a,b)
}
```

```rust
fn some_calc(some_items : Vec<i32
    ↪ >) -> (i32, i32) {
    // two separate properties
    let mut a : i32 = 0;
    let mut b : i32 = 0;

    for item in some_items {
        a += a_calc(item);
    }

    for item in some_items {
        b += b_calc(item);
    }

    (a,b)
}
```

Extract Class

Extract common behaviour out to another class, and compose (can alternatively inherit, though *composition* > *inheritance*).
Python Example:



```
class ClassA:
    def __init__(self, a, b, c,
        ↪ ...):
        self.a = a
        self.b = b
        self.c = c
        ...
        do_something(a)
        do_this(b)
        do_other_thing(a+c)
    ...

class ClassB:
    def __init__(self, a, b, c,
        ↪ ...):
        self.a = a
        self.b = b
        self.c = c
        ...
        do_something(a)
        do_this(b)
        do_other_thing(a+c)
    ...
```

⇒

```
class Common:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def do_thing(self):
        do_something(a)
        do_this(b)
        do_other_thing(a+c)

class ClassA:
    def __init__(self, com, ...):
        self.com = com
        ...
        com.do_thing()
    ...
class ClassB:
    def __init__(self, com, ...):
        self.com = com
        ...
        com.do_thing()
    ...
```

Replace conditionals with polymorphism

Rather than replying on conditionals to determine behaviour, use polymorphism by having objects of a n interface behave differently.
Java Example:

```
1  class MyClass {
2      ...
3      public void workOnObject(
           ↪ MyObject obj) {
4          if (obj.property()) {
5              obj.doThis();
6          } else {
7              obj.doThat();
8          }
9      }
10     ...
11 }
12
13 class MyObject {
14     ...
15 }
```

```
1  class MyClass {
2      ...
3      public void workOnObject(
           ↪ MyObject obj) {
4          obj.do();
5      }
6      ...
7  }
8
9  interface MyObject {
10     void do();
11 }
12
13 class MyObjProp implements
       ↪ MyObject {
14     void do() {
15         //doThis
16     }
17 }
18
19 class MyObjNoProp implements
       ↪ MyObject {
20     void do() {
21         //doThat
22     }
23 }
```