SED

Software Engineering Design

Dr Robert Chatley - rbc@imperial.ac.uk

@rchatley

Welcome to the course on Software Engineering Design.

Traditionally, many software design courses have often concentrated on notations and formal methods for drawing out and specifying how a piece of software should be structured, and how it should behave. This course aims to take a more modern approach, and particularly looks at why it is important to apply good design principles as systems grow larger and more complex…

Waterfall Development

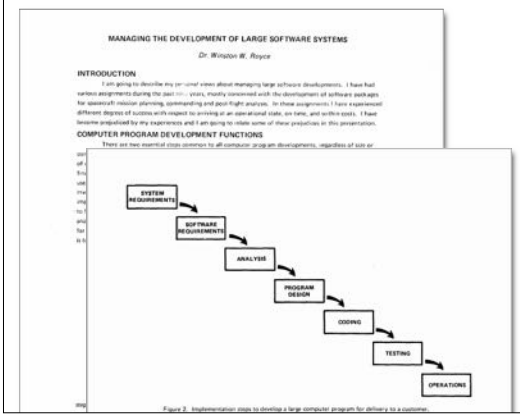


photo: stefg74

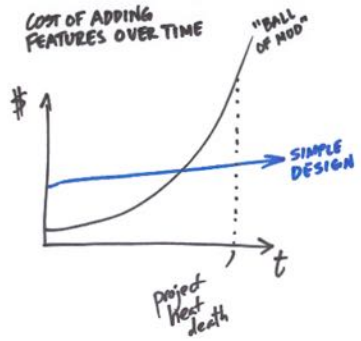
Winston W. Royce wrote a paper in 1970 called "Managing the Development of Large Software Systems". Unfortunately this paper was somewhat misinterpreted by the industry at large, and what emerged was the Waterfall Model of software development. It set out a number of different phases of software design and implementation, which in the Waterfall model flow from one to the other sequentially. Design was done as an early phase, before coding and testing.

One of the problems with this is that we only get one shot at getting things right - there isn't much scope for going back and reworking things after a phase ends. As it happens, this wasn't actually what Royce meant when he wrote the original paper, but it was what people took from it and it stuck for quite a few years.

photo: Rachel Davies @ UnrulyMedia

The waterfall model of upfront design is not very representative of how most software development teams working today do design. Teams tend to work using agile, iterative methods, and to revise and refine the design of their software constantly as they add new features, fix bugs, etc. They still talk about, think about, and draw software designs, but this is not so often a formal stage of the development process. Design is a matter of organising and re-organising code in order to solve problems, and avoid future problems. The focus of this course is on how to do this well - it is much more focussed on code than on notation.

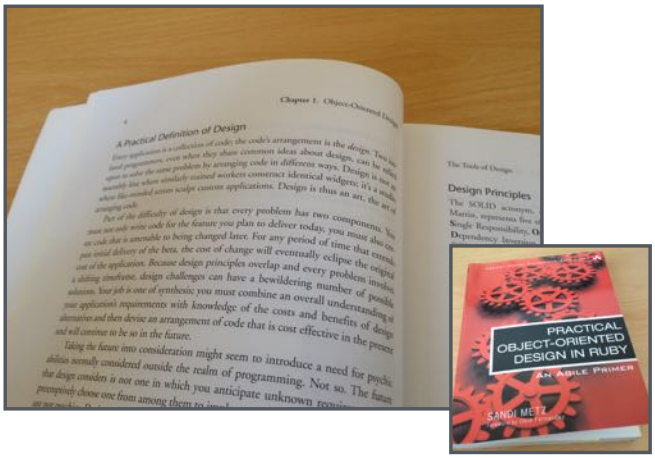
Cost of Change



http://www.jbrains.ca/permalink/the-three-values-of-software

In the article linked on the slide, J.B. Rainsberger refers to design as being one of the three values of software. Without design, the marginal cost of adding a new feature will gradually increase until it becomes too hard to add anything new. Paying attention to improving the design over time will allow us to evolve and maintain the software effectively.

This is one of the most important aspects of software design. As Sandi Metz writes in her book "Practical Object Oriented Design in Ruby": "Changing requirements are the programming equivalent of friction and gravity. They introduce forces that apply sudden and unexpected pressures that work against the best-laid plans. It is the need for change that makes design matter".

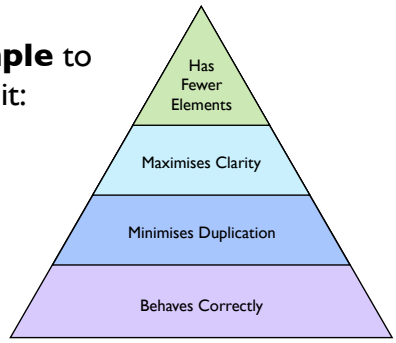


What do we mean by design anyway? In her book on design, Sandi Metz writes "Every application is a collection of code; the code's arrangement is the design."

She goes on, "Part of the difficulty of design is that every problem has two components. You must not only write code for the feature you plan to deliver today, you must also create code that is amenable to being changed later."

Four Elements of Simple Design

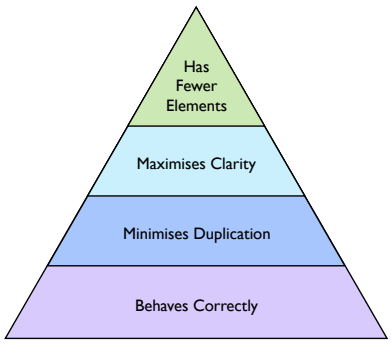
A design is **simple** to the extent that it:



http://www.jbrains.ca/permalink/the-four-elements-of-simple-design

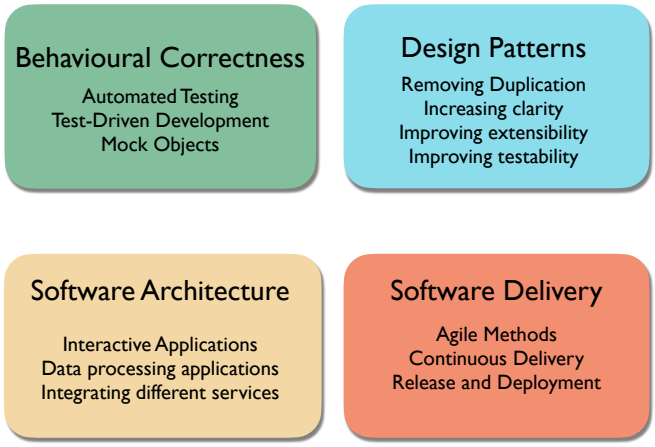
In another article, J.B. Rainsberger sets out his four elements of simple design as follows. A design is simple to the extent that it 1) passes its tests 2) minimises duplication 3) maximises clarity 4) has fewer elements.

Minimising duplication helps to make the design more maintainable and hence more robust. If code is duplicated, then making a change to it may mean making the same change in multiple places, which is more work than we would like. By aiming to maximise clarity, we work to improve comprehension of the design by humans. Rainsberger's fourth principle - to reduce the number of elements - aims to make the overall size of the system smaller, making it simpler and easier to comprehend.



In another article, J.B. Rainsberger sets out his four elements of simple design as follows. A design is simple to the extent that it 1) passes its tests 2) minimises duplication 3) maximises clarity 4) has fewer elements.

Minimising duplication helps to make the design more maintainable and hence more robust. If code is duplicated, then making a change to it may mean making the same change in multiple places, which is more work than we would like. By aiming to maximise clarity, we work to improve comprehension of the design by humans. Rainsberger's fourth principle - to reduce the number of elements - aims to make the overall size of the system smaller, making it simpler and easier to comprehend.



Here is an outline of the topics that we'll cover during this course. Some we will look at in a lot of detail, others we will just touch on to give you some context and something to follow up on later.

There is no one book that this course follows. There are many many good books that can help us to become better software engineers, concentrating on different aspects of technical practice, coding, testing, design, running projects and developing products. The list on this slide is a good starting point - and the blog post lists a lot more suggestions. You don't need to read these books to follow this course, but we will touch on topics from many of them, so you may want to read some of them if you want to look deeper into specific topics - either now or later in your career.

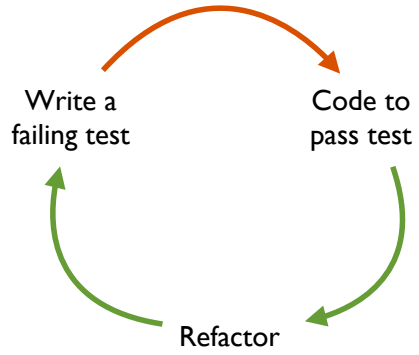
SED

Test Driven Development

Dr Robert Chatley - rbc@imperial.ac.uk

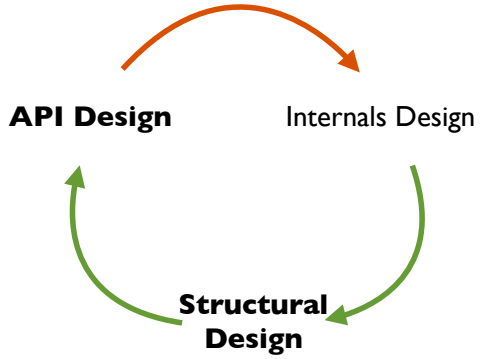
@rchatley

TDD Cycle



When following the Test-Driven Development (TDD) practice, we work in a cycle. We start by writing a test. We write the test first, before writing the implementation. This helps us to specify what we want the code we are about to write to do. How do we expect it will behave when it works properly? Once we have written a test, we watch it fail. This is expected, as we haven't implemented the feature yet. Then we write the simplest possible piece of code we can to make the test pass. After this we *refactor* our design to clean up, remove any duplication, improve clarity etc etc. Then we being the cycle again. When working with unit tests this cycle should be short, and provide us very rapid feedback about our code.

TDD is a Design Activity



TDD is not only about testing. It is a design process. Whenever we write a new test, we are writing an executable specification of how our code should behave. We are designing its public API. Next, when we make a new test pass we know that we have software that behaves correctly. To get to this point we have to do some design of the internal logic of the object. Once we get to a green state, with tests passing, we have the opportunity to improve and simplify our design, perhaps improving structure, removing duplication or increasing the clarity of the code. The tests give us a safety net to make changes without breaking functionality.

Behaviour Driven Development (BDD)

```
CustomerLookup
- finds customer by id
- fails for duplicate customers
- ...

public class CustomerLookupTest {

    @Test
    findsCustomerById() {
        ...
    }
    @Test
    failsForDuplicateCustomers() {
        ...
    }
}
```

http://dannorth.net/introducing-bdd/

We want to use tests to verify the *behaviour* of our system. A well-written set of tests acts as an executable specification for the software, and documents the behaviour. Dan North coined the term Behaviour Driven Development to encourage people to think about the required behaviour (rather than implementation detail) when doing TDD. We can write our specification first in natural language, and then use that to form our test cases and test names.

Developing an Object

```
FibonacciSequence
- 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, 1, 2, 3, 5, 13 …

We start by writing down a few behavioural properties of the object we want to create - an informal specification. You can see how this can act as documentation for how the object behaves. Next we will translate these into automated tests.

Developing an Object

```
FibonacciSequence
- defines the first two terms to be one
- has each term equal to the sum of the previous two
- ...

public class FibonacciSequenceTest {

    @Test public void
    definesTheFirstTwoTermsToBeOne() {
        ...
    }

    @Test public void
    hasEachTermEqualToTheSumOfThePreviousTwo() {
        ...
    }
}
```

This example in is Java using JUnit as a test framework. We create a test class FibonacciSequenceTest to accompany our (to be written) FibonacciSequence class. We define the first test methods one at a time with names matching the textual specification. The @Test annotation tells JUnit to treat this method as a test.

We will probably want to include some assertions in the bodies of our tests. In JUnit we can write these in the form

```
assertTrue(x);
assertFalse(x);
assertThat(x, is(y));
```

Developing an Object

```
FibonacciSequence
- 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
- can be iterated through
```

We step through the specification on line at a time. For each line we add a test, fill it in to make appropriate assertions on the object under test, and then complete the implementation to make it pass. We then assess our current design to see if we can make it simpler or cleaner and tidy it up. Then we repeat the cycle with the next requirement. By following this cycle strictly we make sure that we never add code that isn't being tested.

---

https://pragprog.com/book/bopytest/python-testing-with-pytest

TDD in Python


```
FibonacciSequence
- defines the first two terms to be one
- has each term equal to the sum of the previous two
- ...
```

```python
import pytest

from fibonacci import FibonacciSequence

@pytest.fixture()
def sequence():
    return FibonacciSequence()


def test_can_calculate_a_specific_term(sequence):
    assert sequence.term(0) == 1
    assert sequence.term(4) == 5


def test_allows_iteration(sequence):
    assert sequence.next() == 1
    assert sequence.next() == 1
    assert sequence.next() == 2
```

TDD is practised widely in current software development. It isn't specific to Java. There are unit testing tools for almost every language, and we can apply the principles of TDD to help ensure quality regardless of the language we are working in. Here we show an example in Python using pytest.

Jasmine - TDD in JavaScript

```
FibonacciSequence
- defines the first two terms to be one
- has each term equal to the sum of the previous two
- ...
```

http://javascript.crockford.com
http://pivotal.github.com/jasmine

```javascript
describe("Fibonacci Sequence", function() {

  var fib = new FibonacciSequence();

  it("defines the first two terms to be 1", function() {
    expect(fib.term(0)).toEqual(1);
    expect(fib.term(1)).toEqual(1);
  });

  it("has each term equal to the sum of the previous two", function() {
    expect(fib.term(2)).toEqual(2);
    expect(fib.term(3)).toEqual(3);
    expect(fib.term(5)).toEqual(8);
  });
});
```

Here is another example of developing the FibonacciSequence, this time in JavaScript and using the unit testing tool Jasmine. You can see that the pattern is the same, even though the way that the tests are expressed varies slightly.