COMP1531

✅ Correctness - Dynamic Verification

Lecture 2.3

Author(s): Hayden Smith, Dr. Yuchao Jiang

In This Lecture

- **Why?** 🤔
  - Writing tests are critical to ensure an application works
  - Approaching testing the right way will yield better results
  - We need to be able to able to understand the characteristics of programming languages and approaches in terms of how they may prevent bugs
- **What?** 📰
  - Abstraction & Black boxes
  - Design by contract
  - jest
  - safety

👩‍🔬 Software Testing

Quality control when developing software is important.

In extreme cases, a bug or defect can degrade interconnected systems or cause serious issues.

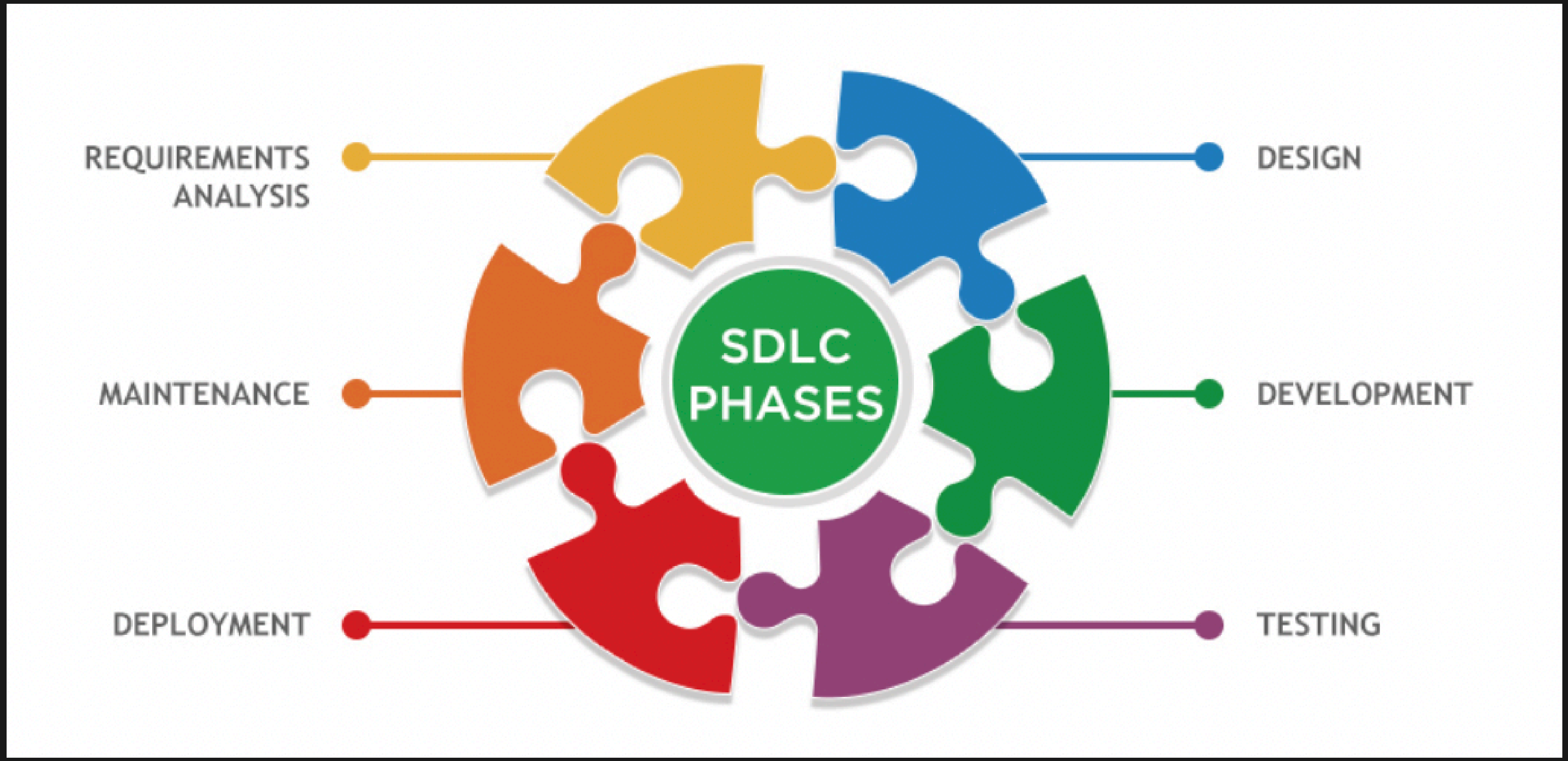
- Consider Nissan having to recall over 1 million cars due to a software defect in the airbag sensor detectors.
- Or a software bug that caused the failure of a USD 1.2 billion military satellite launch.

The numbers speak for themselves. Software failures in the US cost the economy USD 1.1 trillion in assets in 2016. What’s more, they impacted 4.4 billion customers.

A successful test shows that the system operates as intended.

👩‍🔬 Testing In The SDLC

- Software testing is the process of evaluating and verifying that a software product or application does what it is supposed to do.
- The benefits of testing include preventing bugs, reducing development costs and improving performance.
- Because many teams immediately test the code they write, the testing phase often runs parallel to the development phase.



source

👩‍🔬 Software Testing

There are many different types of software tests:

- Acceptance testing
- Integration testing
- Unit testing
- Usability testing
- Functional testing
- Performance testing
- ...

👩‍🔬 Software Testing

- Software testing concerns with exercising and observing product behaviour (dynamic verification).
- The system is executed with test data and its operational behaviour is observed.

👀 Verification

Verification can be broken up into two main types:

- **Static verification**: Testing before executing the code. Sometimes we call these compile-time checks.
- **Dynamic verification**: Testing whilst executing the code. Sometimes we call these run-time checks.

🔓 Software Safety

We can consider verification as a process to help make software safe. This is different from making software secure.

- **Safety**: Protection from accidental misuse
- **Security**: Protection from deliberate miuse

Software becomes unsafe when its design or implementation allow for unexpected or unintended behaviours, particularly during runtime. For example:

- Reading uninitalized memory
- Writing outside array bounds

E.G. Around 94% of spreadsheets contain errors*. For any given spreadsheet formula, there's a 1% chance it contains an error**

💿 Example: Memory Safety

C is not considered a memory safe language. However, Javascript is.

Javascript prevents access to memory that hasn't been initialised or allocated. It does this by **dynamically** checking at runtime.

Whereas in C there is no bounds checking performed for array accesses. Pointers can be dereferenced even if they don't point to allocated memory.

C prioritises performance over memory safety. Javascript vice-versa.

⚡ Static Verification

Static verification is usually considered a more robust and reliable form of testing. However, it's limited as not everything can be verified statically.

In this course we will talk about **Static Typing** and **Linting**. We will talk about static verification later.

🧨 Dynamic Verification

- Verification performed during the execution of software
- This is often what we mean when we say "testing".
- Typically falls into one of two categories:
  - Testing in the small
  - Testing in the large

“Testing Shows The Presence, Not The Absence Of Bugs” — *Edsger W. Dijkstra*

🧨 Dynamic Verification - Why?

- In the real world terrible things happen to programs all the time. Invalid input, unexpected data.
- Dynamic testing is about making your program more robust in the face of **real and inevitable things that go wrong while a program is running.**
- Whilst testing helps show your program is correct, the other long term benefit is helping you ensure that as your code changes over time that it's not expriencing **regressions**. People forget things. Tests tend not to.

“Testing Shows The Presence, Not The Absence Of Bugs” — *Edsger W. Dijkstra*

🧨 Dynamic: Testing In The Small

- Unit testing is the process of testing individual components in isolation.
  - Unit tests are often just testing particular functions in isolation.

🧨 Dynamic: Testing In The Large

Larger tests are tests performed to expose defects in the interfaces and in the interactions between integrated components or systems (ISTQB definition).

These tests tend to be black-box tests, and are written by either developers or independent testers.

🧨 Dynamic: Testing In The Large

Typically these tests fall into these categories:

- Module tests (testing specific module)
- Integration tests (testing the integration of modules)
- System tests (testing the entire system)

👶 Let's Try To Test Naively!

If I left you alone right now, how would you check if this function works correctly?

```
// Even numbers are integers
// exactly divided by 2, mea
// leave no remainder when d
function getEven(nums) {
  const evens = [];
  for (const number of nums)
    if (number % 2 === 0) {
      evens.push(number);
    }
  }
  return evens;
}
```

even_testing1.js

👶 Let's Try To Test Naively!

If I left you alone right now, how would you check if this function works correctly?

```javascript
// Even numbers are integers
// exactly divided by 2, mea
// leave no remainder when d
function getEven(nums) {
  const evens = [];
  for (const number of nums)
    if (number % 2 === 0) {
      evens.push(number);
    }
  }
  return evens;
}
```

even_testing1.js

Would you do something like this?

```javascript
console.log(getEven([1,2,3]));
console.log(getEven([4,5,6]));
console.log(getEven([7]));
```

👶🏾 Let's Try To Test Naively!

However, This Is Not Testing.

- Printing errors or visually inspecting output is a method of debugging not testing. You can't call something a testing method if it doesn't scale well.
- Before we learn about testing, let's quickly talk about what the "black-box" part of black-box testing is.

🐧 Blackbox Testing, Abstraction

- A great type of testing relies on testing **abstractions**.
- Abstraction is the notion of focusing on a higher level understanding of the problem and not worrying about the underlying detail.
- We do this all the time when we drive a car, use our computers, order something online. You're typically focused on expressing an input and wanting an output, with little regard for how you get that output.
- When we look at systems in an abstract way we could also say that we're treating them like **black boxes**.

🐧 Blackbox Testing, Abstraction

- When we're **testing** our code, we always want to view the functions we're testing as abstractions / black boxes.
- Let's try and write some tests for these stub functions.

```
// Returns a new string with vowels removed
function removeVowels(string) {
  return 0;
}

// Calculates the factorial of a number
function factorial(num) {
  return 0;
}
```

blackbox.js

Blackbox Testing, Abstraction

What do we notice when writing these tests?

- We wrote the tests, even if they aren't being passed
- We don't need to know how the function is implemented to test the function
- Now we can go and implement it, and we have tests already done!

```javascript
// Returns a new string with vowels removed
function removeVowels(string) {
  return 0;
}

// Calculates the factorial of a number
function factorial(num) {
  return 0;
}

console.log(removeVowels('abcde') === 'bcd');
console.log(removeVowels('frog') === 'frg');
console.log(factorial(3) === 6);
console.log(factorial(5) === 120);
```

blackbox2.js

🤡 Jest: Testing In JS

To test at scale we need a real testing framework. `jest` is a popular framework for nodejs/javascript. It is installed with NPM.

Let's install jest: `npm install --save-dev jest.`

We add `--save-dev` because its a dependency being used for development and testing, but wouldn't be used in production.

🤡 Jest: Testing In JS

Let's setup a simple example.

Let's write some `jest` tests for our function.

```javascript
// Returns a new string with vowels removed
function removeVowels(string) {
  string = string.replaceAll('a', '');
  string = string.replaceAll('e', '');
  string = string.replaceAll('i', '');
  string = string.replaceAll('o', '');
  string = string.replaceAll('u', '');
  return string;
}

export { removeVowels };
```

jest_lib.js

🤡 Jest: Testing In JS

```
import { removeVowels } from './jest_lib';

test('deals with no vowels', () => {
  const example1 = removeVowels('bcd');
  expect(example1).toEqual('bcd');
});
```

Now we can run

```
./node_modules/.bin/jest src/jest_lib.test.js
```

Using Matchers in Jest

🤡 Jest: Testing In JS

```js
import { removeVowels } from './jest_lib';

describe('removeVowels', () => {
  test('deals with no vowels', () => {
    const example1 = removeVowels('bcd');
    const example2 = removeVowels('lkj');
    expect(example1).toEqual('bcd');
    expect(example2).toEqual('lkj');
  });
  test('deals with only vowels', () => {
    expect(removeVowels('aei')).toEqual('');
    expect(removeVowels('oiu')).toEqual('');
  });
  test('deals with starting vowels', () => {
    expect(removeVowels('ant')).toEqual('nt');
    expect(removeVowels('old')).toEqual('ld');
  });
  test('deals with ending vowels', () => {
    expect(removeVowels('bee')).toEqual('b');
    expect(removeVowels('hi')).toEqual('h');
  });
  test('deals with complex words', () => {
    expect(removeVowels('cannot')).toEqual('cnnt');
    expect(removeVowels('delicious')).toEqual('dlcs');
  });
});
```

jest_lib.test.js

Now we can run

```
./node_modules/.bin/jest jest_lib.test.js
```

🤡 Jest: Testing In JS

Then we'll add `"test": "jest"` to our `package.json` scripts.

Now when we run `npm run test` it will execute `./node_modules/.bin/jest`.

🤡 Jest: Testing In JS

- The general structure is that you have a series of outermost `describe`s that are for a broad area (e.g. auth).
- Then some `describe`s inside that might cover for instance a particular function.
- Then inside that, we have `test`s for each property or use case we're looking to test.
- Then inside that we use one of jest's expect functions (please see the docs!

🤡 Jest: Testing In JS

```
describe('auth capabilities', () => {
  describe('auth_register', () => {
    test('fails on invalid email', () => {
      // Execute some logic
      expect(true).toEqual(true);
    });
  });
});
```

Design By Contract

- When we're testing or implementing a function, we will typically be working with information that tells us the constraints placed on at least the inputs.
- The documentation can come in a variety of forms.
- This information tells us what we do and don't need to worry about when writing tests.

```javascript
// Returns a new string with vowels removed
// Input is a non-empty string type
// Return type is another string
function removeVowels(string) {
  return 0;
}

// Calculates the factorial of a number
// Input is a number between 1 and 10
// Output is a positive number
function factorial(num) {
  return 0;
}
```

design_contract.js

😱 Behaviour

When we look at how programs respond to being used, we often use the word `behaviour` to describe what they do under certain conditions.

When we test, we are essentially just testing the behaviour of a program.

However, it's not always possible to understand the behaviour of every possible scenario from a spec perspective, often either due to:

- It is not feasible to define every possible edge case
- The implementation details are not critical to the spec

🤔 Undefined Behaviour

Undefined behaviour is where a behaviour is not defined at all, often because the use case is beyond a reasonable scope of expectation, and that behaviour generally shouldn't occur within a reasonable usage of the system.

An example might be trying to find the square root of negative 1.

😎 Implementation-Defined Behaviour

Implementation defined behaviour is where a behaviour needs to be defined and considered, but it is up to the individual implementing it to make decisions about it.

An example might be whether text inputs are ASCII or UTF-8 encoding.

👂 Feedback

Or go to the form here.