COMP1531

✅ Correctness - Dynamic Verification

Lecture 2.3

Author(s): Hayden Smith

(Download as PDF)

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

👀 Verification

Verification in a system life cycle context is a set of activities that **compares a product of the system life cycle against the required characteristics for that product**. This may include, but is not limited to, specified requirements, design description and the system itself.

I.E. Verification is checking if the **system has been built correctly**.

👀 Verification

"Poor software quality costs more than $500 billion per year worldwide" – Casper Jones

Systems Sciences Institute at IBM found that it costs four-to five-times as much to fix a software bug after release, rather than during the design process

👀 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, broken I/O.

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

We often refer to small tests as unit tests, which the ISTQB defines as the testing of individual software components.

These can be white-box or black-box tests (we'll come back to this later), and are written often by engineers who will implement work.

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.

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?

```
1 function getEven(nums) {
2   const evens = [];
3   for (const number of nums)
4     if (number % 2 === 0) {
5       evens.push(number);
6     }
7   }
8   return evens;
9 }
```

2.3_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?

```
1 function getEven(nums) {
2   const evens = [];
3   for (const number of nums)
4     if (number % 2 === 0) {
5       evens.push(number);
6     }
7   }
8   return evens;
9 }
```

2.3_even_testing1.js

Would you do something like this?

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

Let's Try To Test Naively!

```
1 function getEven(nums) {
2   const evens = [];
3   for (const number of nums) {
4     if (number % 2 === 0) {
5       evens.push(number);
6     }
7   }
8   return evens;
9 }
```

2.3_even_testing1.js

Or something like this?

```
1 if (getEven([1,2,3]) != [2]) {
2   console.log("Doesn't work 1");
3 }
4 if (getEven([4,5,6]) != [4,6]) {
5   console.log("Doesn't work 2");
6 }
7 if (getEven([7]) != []) {
8   console.log("Doesn't work 3")};
9 }
```

👶🏾 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.

A scaled approach is what makes something become testing rather than debugging.

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.

```
1 // Returns a new string with vowels removed
2 function removeVowels(string) {
3   return 0;
4 }
5
6 // Calculates the factorial of a number
7 function factorial(num) {
8   return 0;
9 }
```

2.3_blackbox.js

Blackbox Testing, Abstraction

What do we notice when writing these tests?

- The tests are complete, 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!

```
// 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);
```

2.3_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.

```
describe('users', () => {
  const name = 'Hayden';
  test('check name', () => {
    expect(name).toEqual('Hayden');
  });
});
```

🤡 Jest: Testing In JS

Let's setup a simple example.

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

```
1 // Returns a new string with vowels removed
2 function removeVowels(string) {
3   return 0;
4 }
5
6 export { removeVowels };
```

2.3_jest_lib.js

🤡 Jest: Testing In JS

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

```
1 import { removeVowels } from './2.3_jest_lib';
2
3 test('deals with no vowels', () => {
4   const example1 = removeVowels('bcd');
5   expect(example1).toEqual('bcd');
6 });
```

Now we can run

`./node_modules/.bin/jest 2.3_blackbox.test.js`

🤡 Jest: Testing In JS

```
1  import { removeVowels } from './2.3_jest_lib';
2
3  describe('removeVowels', () => {
4    test('deals with no vowels', () => {
5      const example1 = removeVowels('bcd');
6      const example2 = removeVowels('lkj');
7      expect(example1).toEqual('bcd');
8      expect(example2).toEqual('lkj');
9    });
10   test('deals with only vowels', () => {
11     expect(removeVowels('aei')).toEqual('');
12     expect(removeVowels('oiu')).toEqual('');
13   });
14   test('deals with starting vowels', () => {
15     expect(removeVowels('ant')).toEqual('nt');
16     expect(removeVowels('old')).toEqual('old');
17   });
18   test('deals with ending vowels', () => {
19     expect(removeVowels('bee')).toEqual('b');
20     expect(removeVowels('hi')).toEqual('h');
21   });
22   test('deals with complex words', () => {
23     expect(removeVowels('cannot')).toEqual('cnnt');
24     expect(removeVowels('delicious')).toEqual('dlcs}');
25   });
26 });
```

2.3_jest_lib.test.js

Now we can run

`./node_modules/.bin/jest 2.3_blackbox.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 `describes` that are for a broad area (e.g. auth).

Then some `describes` inside that might cover for instance a particular function.

Then inside that, we have `tests` 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);
    });
  });
});
```

🤡 Jest: Testing In JS

NOTE: the `jest` syntax relies on some ideas around function syntax and function callbacks that won't make a lot of sense for another week or two. But that's OK! Just copy the code examples for now and change the lines that matter.

ALSO NOTE: To use `jest` in COMP1531 we've had to set a few environment attributes - this means that you will only be able to get `jest` working in week 3 labs or iteration 1 onward.

📜 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.

```
1  // Returns a new string with vowels removed
2  // Input is a non-empty string type
3  // Return type is another string
4  function removeVowels(string) {
5    return 0;
6  }
7
8  // Calculates the factorial of a number
9  // Input is a number between 1 and 10
10 // Output is a positive number
11 function factorial(num) {
12   return 0;
13 }
```

2.3_design_contract.js

🦻 Feedback

Or go to the form here.