

Software Testing

SOFTWARE DESIGN - FALL 2024



Software is not like engineering something easy like a bridge, where you start with a blueprint, build it to spec, then forget about it. Software is dynamic, with a lot of moving parts and requirements that evolve over time. Developers build apps on top of a mountain of abstractions, and nobody fully understands how every layer works

we just need to make sure that our code matches
the requirements of the product



Software Testing

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

Test-driven development is scientifically proven to reduce defects and improve the maintainability of a codebase.



Software Testing

- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.



V & V

- **Verification:** refers to the set of tasks that ensure that software correctly implements a specific function.
- **Validation:** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
- **Verification:** "Are we building the product right?"
- **Validation:** "Are we building the right product?"

Who Tests the Software?



developer

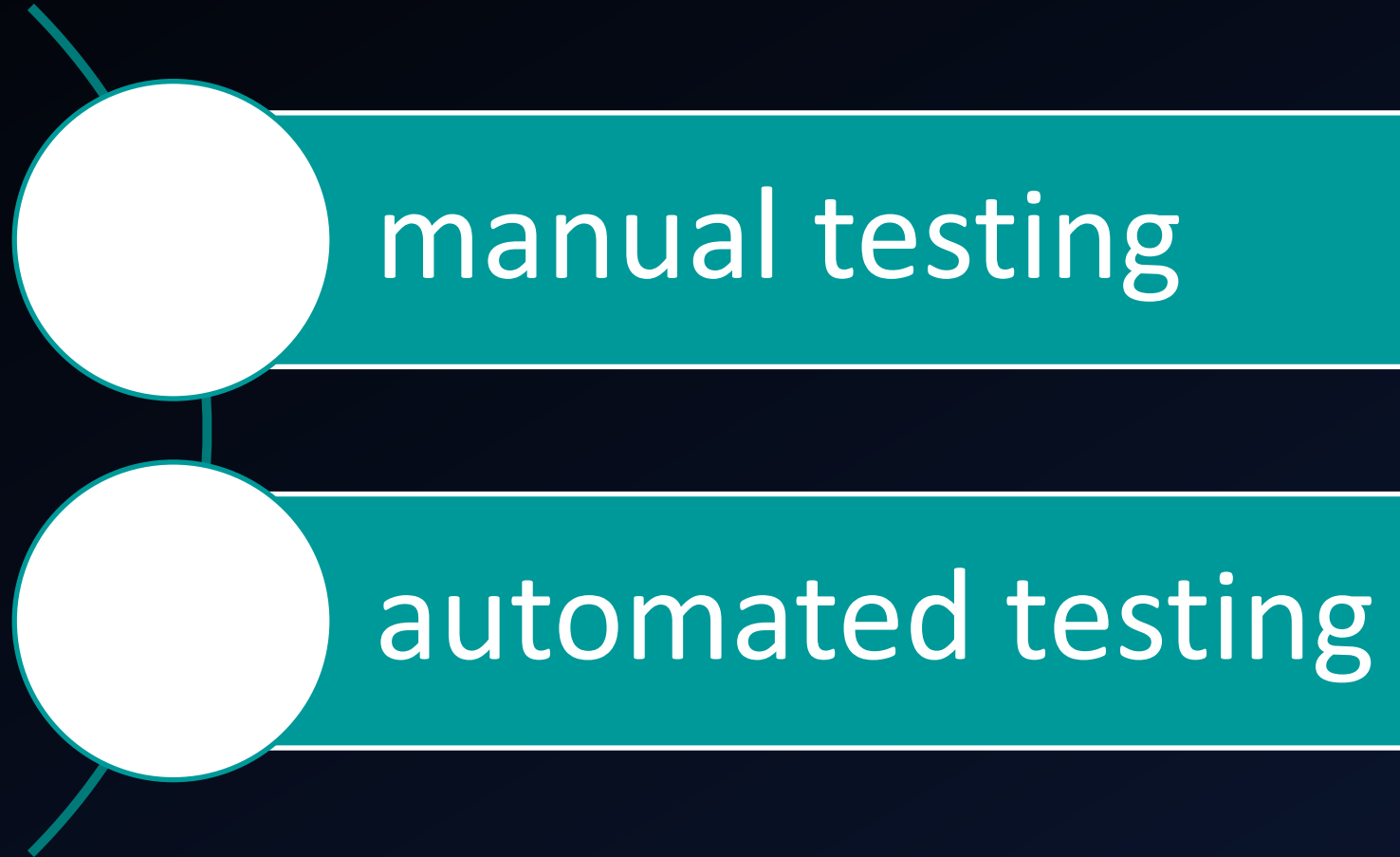
Understands the system
but, will test "gently"
and, is driven by "delivery"



independent tester

Must learn about the system,
but, will attempt to break it
and, is driven by quality

Testing Types



Manual Testing

human being clicks on every button and fills out every form, then assigns a bunch of Jira tickets so they can be backlogged by the developers



Automated Testing

automated testing is basically just a way to write code that describes your requirements and validates your main application code

```
app.spec.js app.spec.js\...  
  
describe('My killer feature 💣', () => {  
  it('finds the sum of a and b', () => {  
    const x = add(2, 2);  
  
    expect(x).toEqual(4);  
    expect(x).not.toBe(null);  
    expect(x).toBeInstanceOf(Number);  
  });  
});
```

individual tests(describe the behavior of the code in human-readable terms)

test suite(describes the feature or thing that's being tested) :
A collection of tests

Test runners



✓ All checks have passed
4 successful checks

✓ build Successfully in 59s — build

✓ test Successfully in 59s — build

✓ publish Successfully in 59s — build

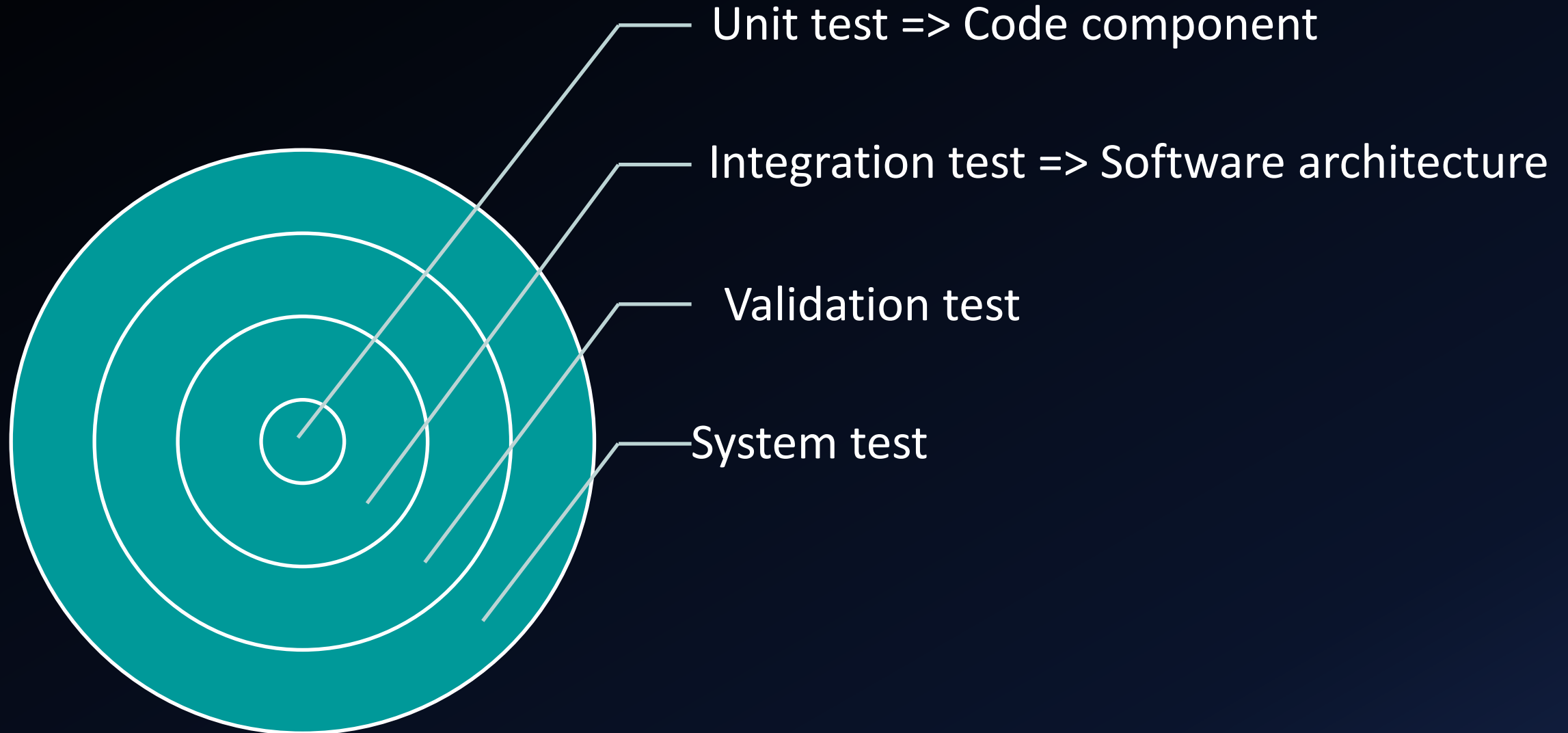
✓ This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request You can also open this in GitHub Desktop or view

```
PASS app/actions/__tests__/activi
PASS app/actions/__tests__/messag
PASS app/actions/__tests__/authen
PASS app/actions/__tests__/paymen
PASS app/actions/__tests__/fellow
PASS app/actions/__tests__/workfl
PASS app/actions/__tests__/enviroi
PASS app/actions/__tests__/loadin
PASS app/actions/__tests__/activi
PASS app/reducers/errors/__tests_
PASS app/actions/__tests__/errors
PASS app/reducers/mixpanel/__test
PASS app/actions/__tests__/mixpan
PASS app/reducers/loading/__tests
PASS app/reducers/messaging/__tes
PASS app/reducers/workflow/__test
PASS app/components/custom/carousi
PASS app/components/custom/v2/act
PASS app/components/custom/v2/nav
PASS app/components/custom/v2/act
PASS app/components/custom/carousi
PASS app/components/custom/v2/nav
PASS app/components/custom/header:
PASS app/components/custom/v2/nav
PASS app/components/custom/header:
PASS app/components/custom/header:
PASS app/components/custom/header:
PASS app/components/authentication
PASS app/components/custom/header:
PASS app/components/custom/add_is
PASS app/components/custom/v2/nav
PASS app/components/custom/header:
PASS app/components/authentication
```

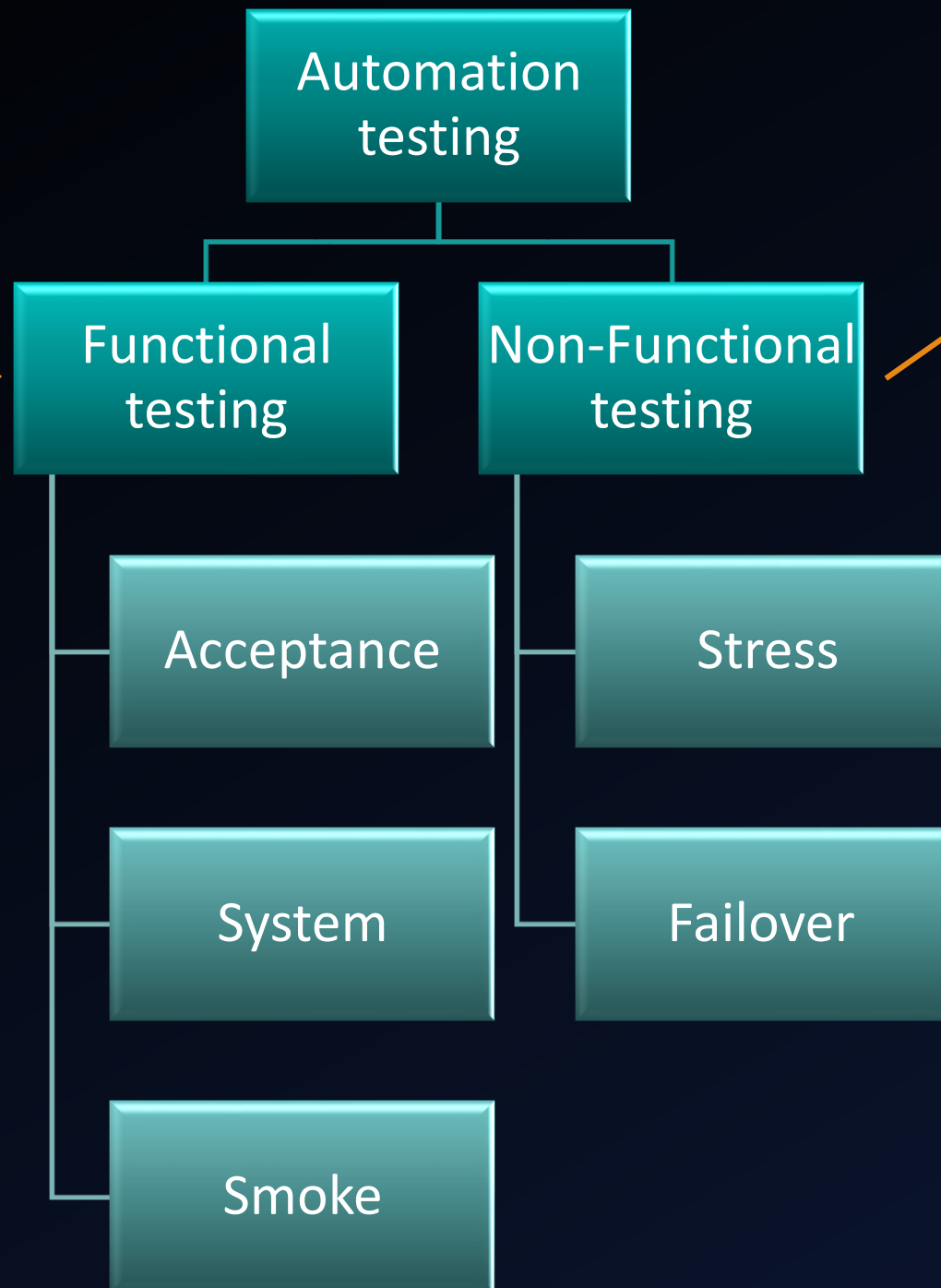
```
Test Suites: 33 passed, 33 total
Tests: 229 passed, 229 total
Snapshots: 229 passed, 229 total
Time: 14.782s
Ran all test suites.
```

Different automated testing strategies



**Test actual
code**

**Performance
Usability
Security**





A pyramid diagram illustrating the hierarchy of testing types. The pyramid is divided into four horizontal layers. From top to bottom, the layers are: MANUAL (dark teal), END-TO-END (medium teal), INTEGRATION TESTS (light gray), and UNIT TESTS (bright cyan). To the right of the pyramid, a vertical arrow points upwards, with three text labels: 'Slower' at the top, 'More complex' in the middle, and 'More time to maintain' at the bottom. The background is dark blue with some abstract teal lines on the left and bottom right.

MANUAL

END-TO-END

INTEGRATION TESTS

UNIT TESTS

Slower

More complex

More time
to maintain

Unit Test

The goal is to validate the behavior of individual functions, methods, or just units of code.

We write unit test to make sure that our program is working correctly at the lowest level.


when we write unit tests to test every single line of code this is what we refer to as code coverage typically one hundred percent code coverage refers to line coverage

Unit Test

let's say you have an if statement that has three different conditions in it therefore under 100% coverage you need to write at least eight different unit tests to cover all the different scenarios

C# Program.cs X

```
1  using System;
2
3  public class Program
4  {
5      public static void Main()
6      {
7          SuperImportantFunction(true, true, true);
8      }
9
10     private static void SuperImportantFunction(bool condition1, bool condition2, bool condition3)
11     {
12         if (condition1 && condition2 && condition3)
13         {
14             Console.WriteLine("Hello");
15         }
16     }
17 }
18
```



	CONDITION 1	CONDITION 2	CONDITION 3
1	FALSE	FALSE	FALSE
2	TRUE	FALSE	FALSE
3	FALSE	TRUE	FALSE
4	FALSE	FALSE	TRUE
5	TRUE	TRUE	FALSE
6	TRUE	FALSE	TRUE
7	FALSE	TRUE	TRUE
8	TRUE	TRUE	TRUE

```
class Stack {
  constructor() {
    this.top = -1;
    this.items = {};
  }

  get peek() {
    return this.items[this.top];
  }

  push(value) {
    this.top += 1;
    this.items[this.top] = value;
  }

  pop() {
    this.top -= 1;
  }
}
```

```
describe('My Stack', () => {
  let stack;

  beforeEach(() => {
    stack = new Stack();
  });

  it('is created empty', () => {
    expect(stack.top).toBe(-1);
    expect(stack.items).toEqual({});
  });

  it('can push to the top', () => {
    stack.push('🍪');
    expect(stack.top).toBe(0);
    expect(stack.peek).toBe('🍪');

    stack.push('🍩');
    expect(stack.top).toBe(1);
    expect(stack.peek).toBe('🍩');
  });

  it('can pop off', () => {
    stack.push('🍪');
    stack.pop()
    expect(stack.top).toBe(-1)
    expect(stack.peek).toEqual(undefined);

    stack.push('🍪');
    stack.push('🍩');
    stack.pop()
    expect(stack.top).toBe(0)
    expect(stack.peek).toBe('🍪');
  });
});
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

PASS test/**stack.test.js**

My Stack

✓ is created empty (8 ms)

✓ can push to the top (1 ms)

✓ can pop off (1 ms)

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	0	0	0	0	

Test Suites: 1 passed, 1 total**Tests:** 3 passed, 3 total**Snapshots:** 0 total**Time:** 2.376 s

Ran all test suites.

Integration Test

individual modules are combined and tested as a group.
data transfer between the modules is tested thoroughly.

For example, you might have a React component and a hook to fetch something from a database. You can unit test each of these individually, but then have an integration test to see how well they work together, like is the component actually able to use the hook to get the data that it needs for the UI?

```
function Feature() {  
  const [data] = useDatabase('url')  
  return <main>{data}</main>  
}
```

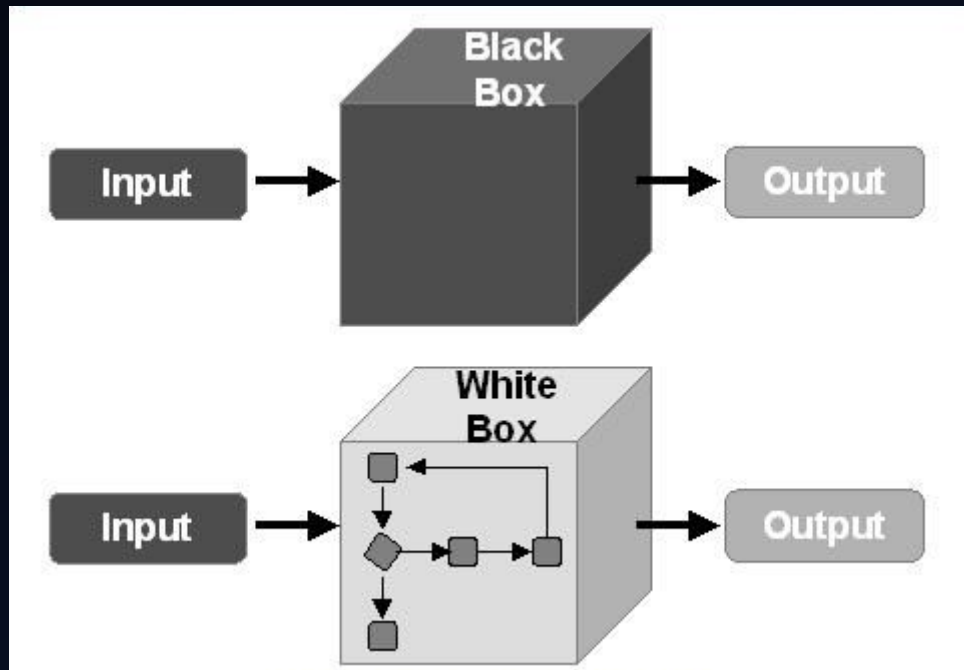
Integration Test

a lot of developers get confused by integration tests thinking they need to test all of their application but that isn't the case you just need to test the Integrations between your components



White Box vs Black Box

depending on who writes your integration tests
whether it be a developer or a tester will
determine whether they're considered white box
testing or Black Box testing.



If integration tests are written by a tester then they might be calling the API and then seeing whether there's something in the database this would therefore be more of a black box because they don't need to know the internals of the application



```
// math.test.js

const { add, subtract } = require('../math');

describe('Math operations', () => {
  it('should add two numbers', () => {
    const result = add(2, 3);
    expect(result).toBe(5);
  });

  it('should subtract two numbers', () => {
    const result = subtract(5, 3);
    expect(result).toBe(2);
  });

  it('should correctly integrate add and subtract', () => {
    const result = subtract(add(7, 3), 2);
    expect(result).toBe(8);
  });
});
```

PASS test/math.test.js

Math operations

✓ should add two numbers (9 ms)

✓ should subtract two numbers

✓ should correctly integrate add and subtract (1 ms)

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
math.js	100	100	100	100	

Test Suites: 2 passed, 2 total

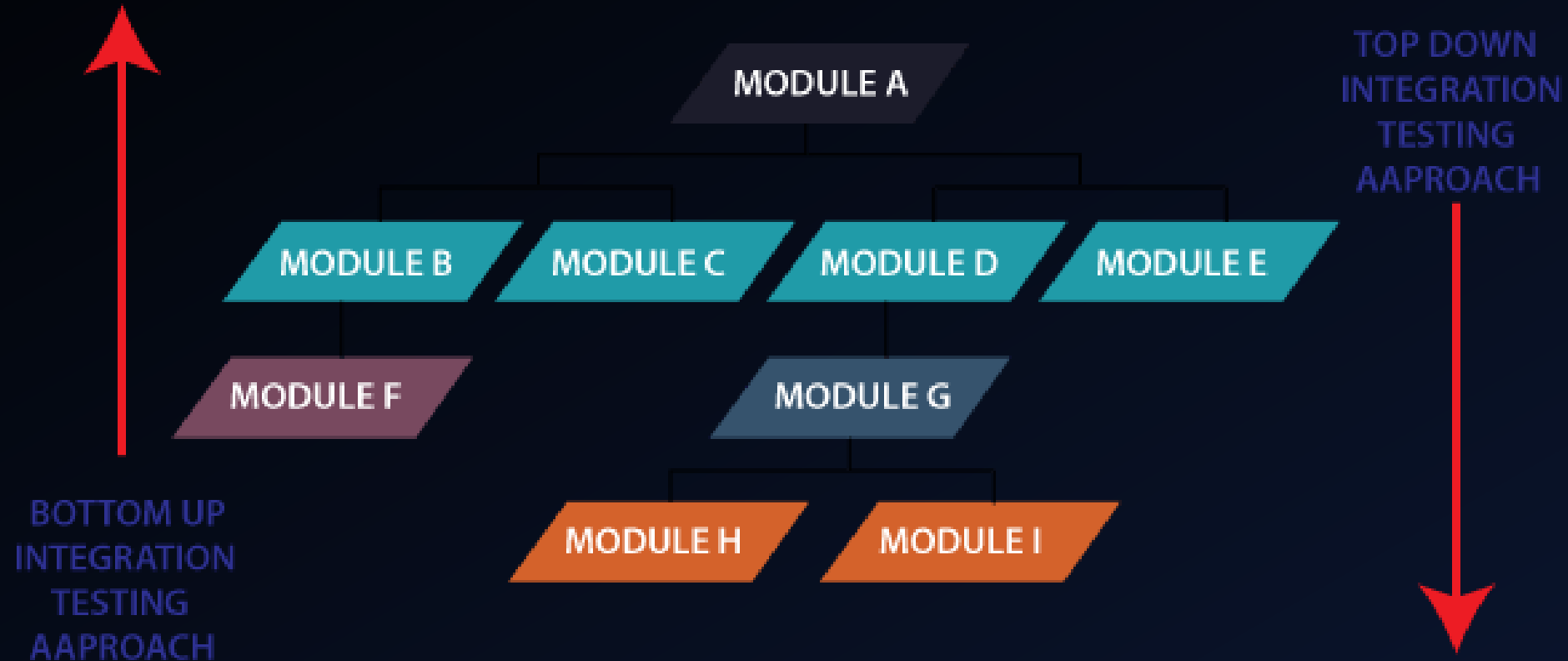
Tests: 6 passed, 6 total

Snapshots: 0 total

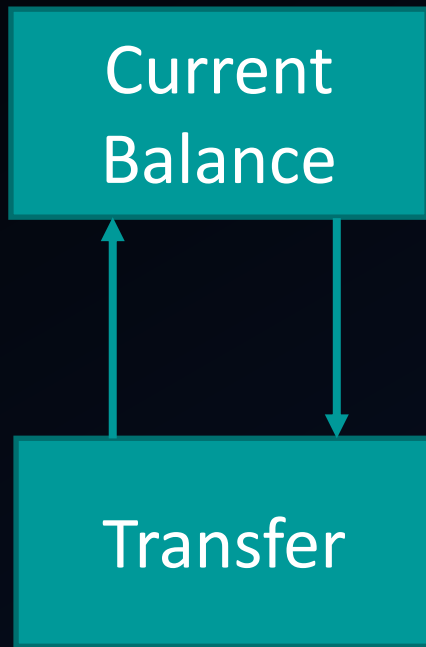
Time: 0.92 s, estimated 1 s

Ran all test suites.

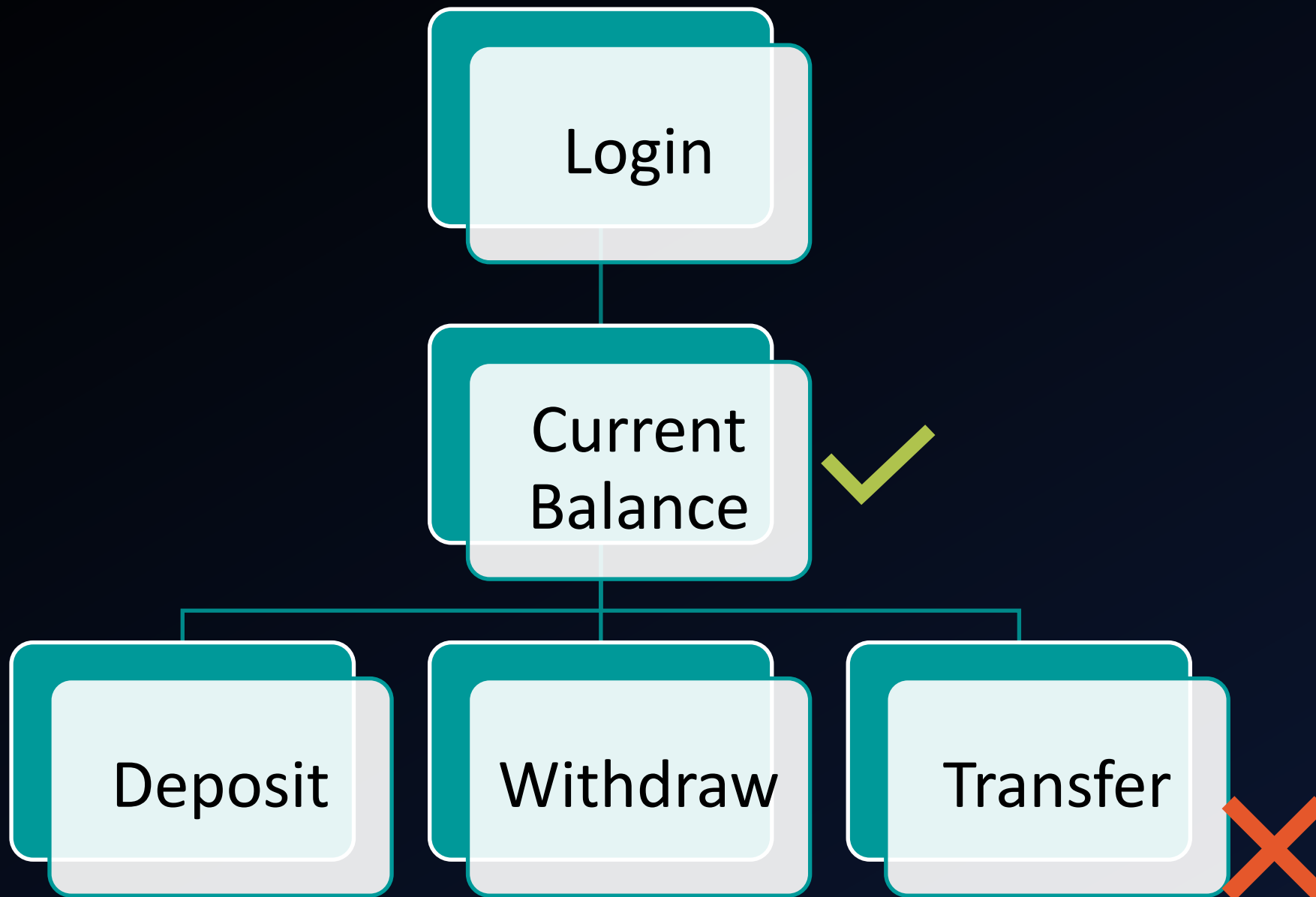
Integration Test Strategies



Integration Testing scenario: Banking Application



A customer is using the current balance module his balance is one thousand he navigates to the transfer module and transfers 500 to a third party account the customer navigates back to the current balance module and now his latest balance should be 500



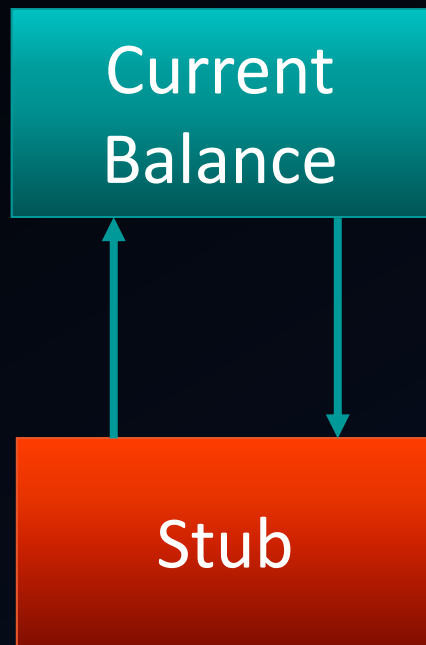
Big Bang Integration Testing

- wait for all modules to be developed before you begin testing:
 - ❑ increases project execution time
 - ❑ it becomes difficult to trace the root cause of defects

Incremental Integration Testing

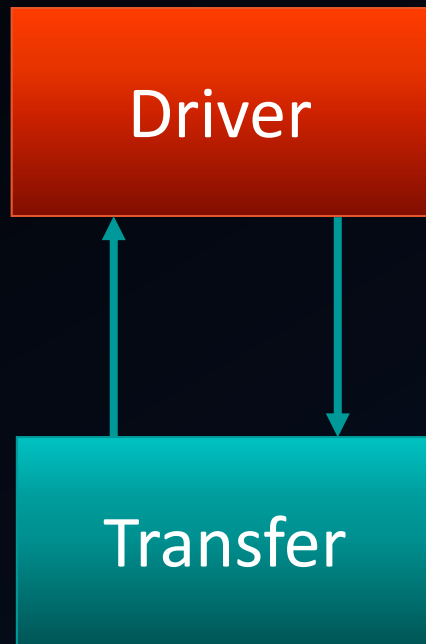
- modules are checked for integration as and when they are available

Top Down Integration



We will create a stub which will accept and give back data to the current balance module
note that this is not a complete implementation of the transfer module which will have lots of checks like whether the third party account number is entered correct the amount to transfer should not be more than the amount available in the account and so on but it will just simulate the data transfer that takes place between the two modules to facilitate testing on the contrary

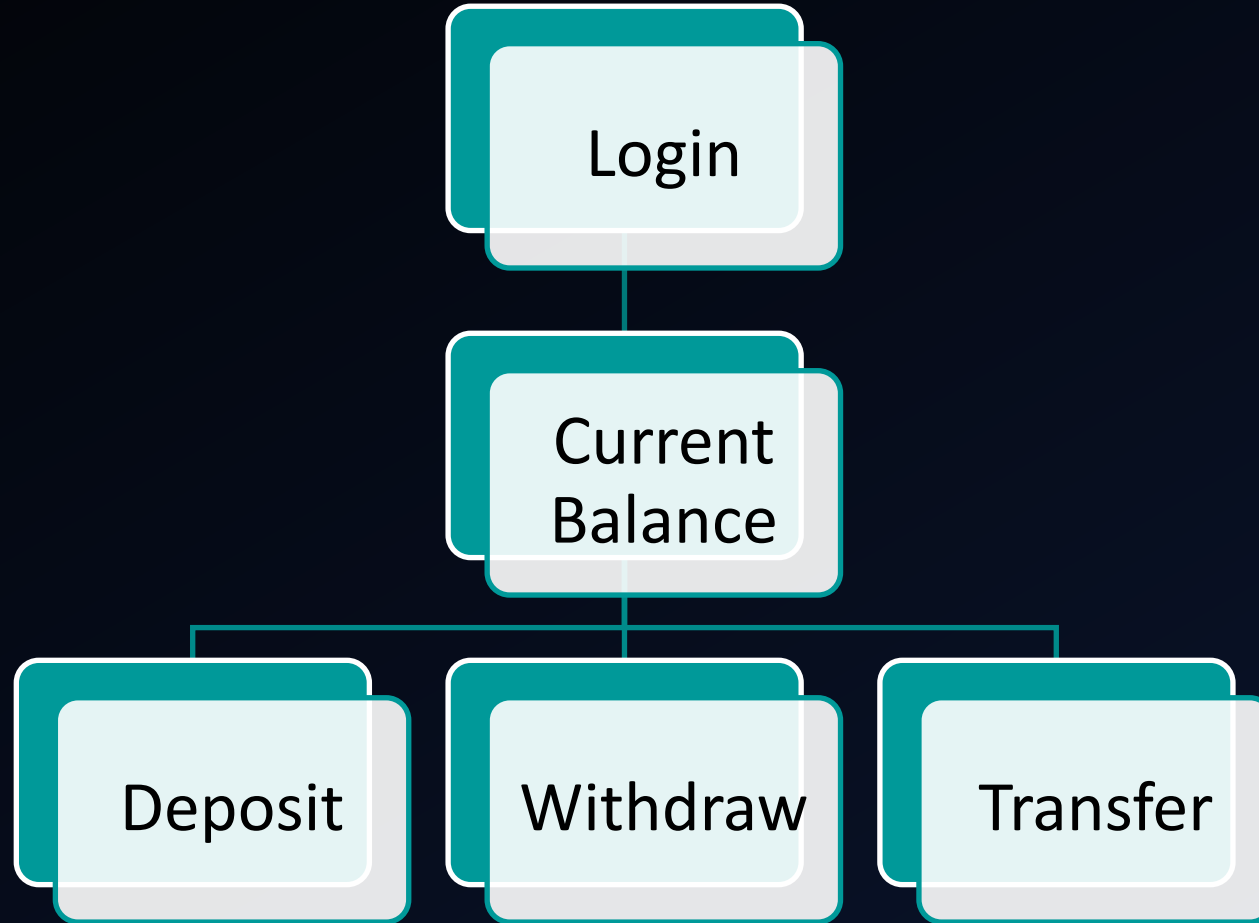
Down Top Integration



if the transfer module is ready but the current balance module is not developed you will create a driver to simulate transfer between the modules to increase the effectiveness of the integration testing

Top to
Bottom

creation
of stubs

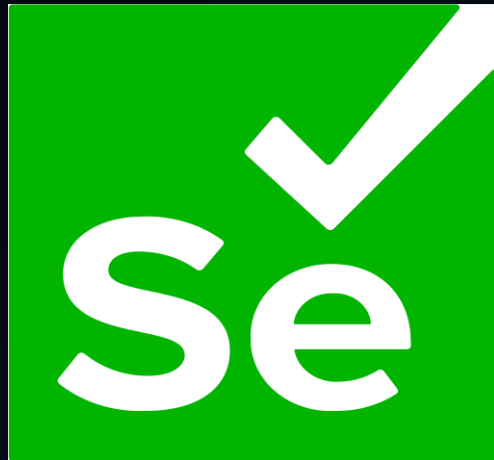


Bottom to
Top

creation of
drivers

End-To-End Test

It runs your app in a simulated environment and attempts to emulate actual user behavior

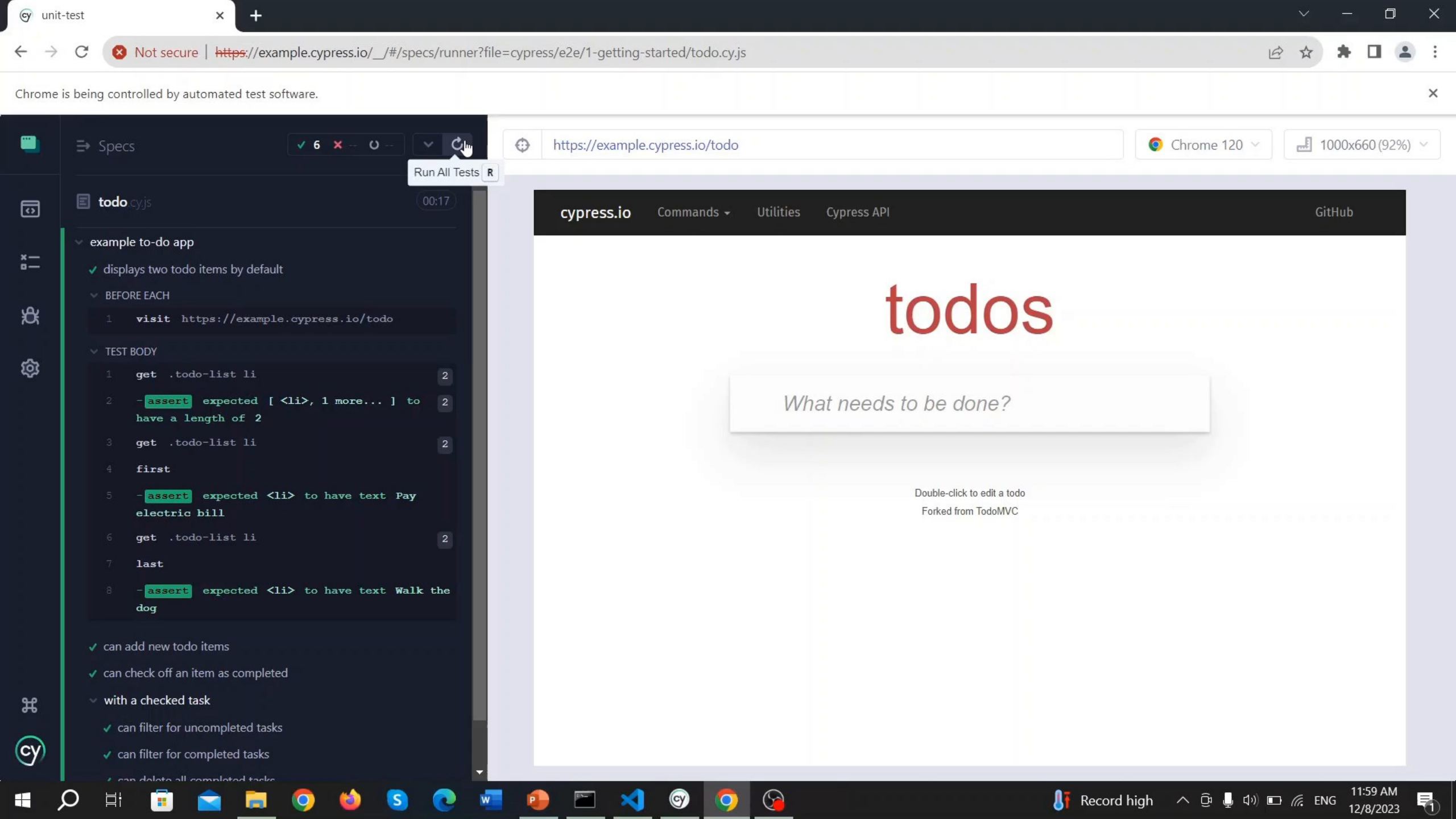


Selenium



Cypress

Cypress has a browser-based test runner that allows you to program tests where users click on buttons, fill out forms, ...

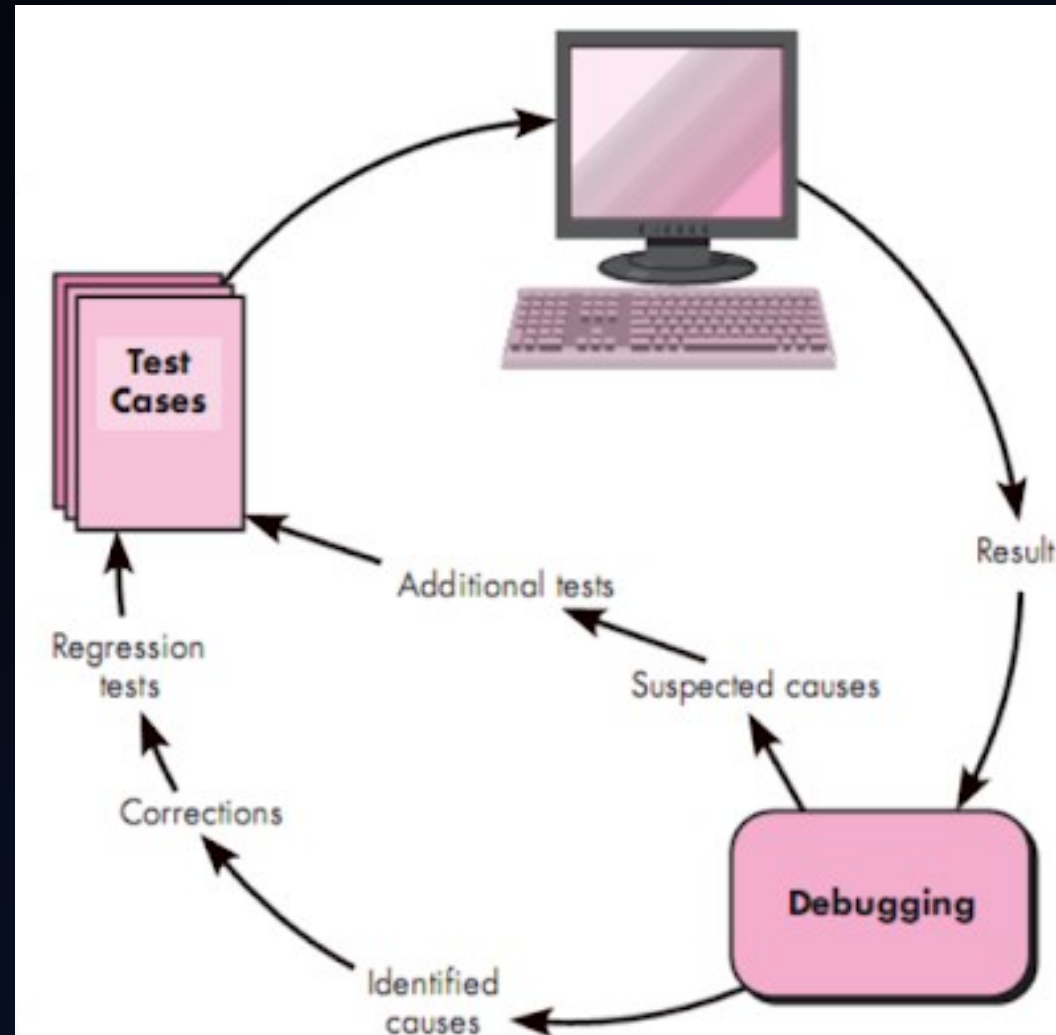


Manual Test

These are the tests that are either too complicated to try and automate or they're not worth the time in trying to do it.

Note: it's always better to find it lower down the pyramid than it is near the top let's say you find a bug while you're doing your manual testing you now have to search for your logs and try and work out where exactly your application failed compare that to finding a bug in your unit tests and you'll be given a stack Trace that shows you the exact line where the problem occurred

The Debugging Process



Regression Testing

- Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors

What is a “Good” Test?

- A good test has a high probability of finding an error
- A good test is not redundant.
- A good test should be “best of breed”
- A good test should be neither too simple nor too complex