**Debugging and Unit Testing**

Table of Contents

[Debugging 2](#_Toc83393416)

[Debugging Process 2](#_Toc83393417)

[Debugging Rules 3](#_Toc83393418)

[Debugging Strategies 4](#_Toc83393419)

[01. Debug with Purpose 4](#_Toc83393420)

[02. Explain it to Someone Else 4](#_Toc83393421)

[03. Focus on Recent Changes 4](#_Toc83393422)

[04. Dump State 5](#_Toc83393423)

[05. Rest 5](#_Toc83393424)

[06. Use Tools 5](#_Toc83393425)

[07. Think Ahead 5](#_Toc83393426)

[Unit Testing 6](#_Toc83393427)

[Drivers 6](#_Toc83393428)

[Stubs 6](#_Toc83393429)

[Properties of Good Unit Tests 7](#_Toc83393430)

[Writing Good Tests 7](#_Toc83393431)

[Excuses 8](#_Toc83393432)

[Writing a Unit Test 8](#_Toc83393433)

[Best Practices 11](#_Toc83393434)

[Additional Resources 11](#_Toc83393435)

A **bug** is an error, fault or flaw in any computer program or hardware system. Bugs produce **unexpected results** or cause a system to behave unexpectedly. From a developer’s perspective, bugs can be **syntax or logic errors** within the source code of a program.

Bugs can often be fixed using a development tool called a **debugger**. Since bugs are common in large applications, debugging can save lots of money. Bugs **discovered by clients** are far more **costly** to fix.

## Debugging

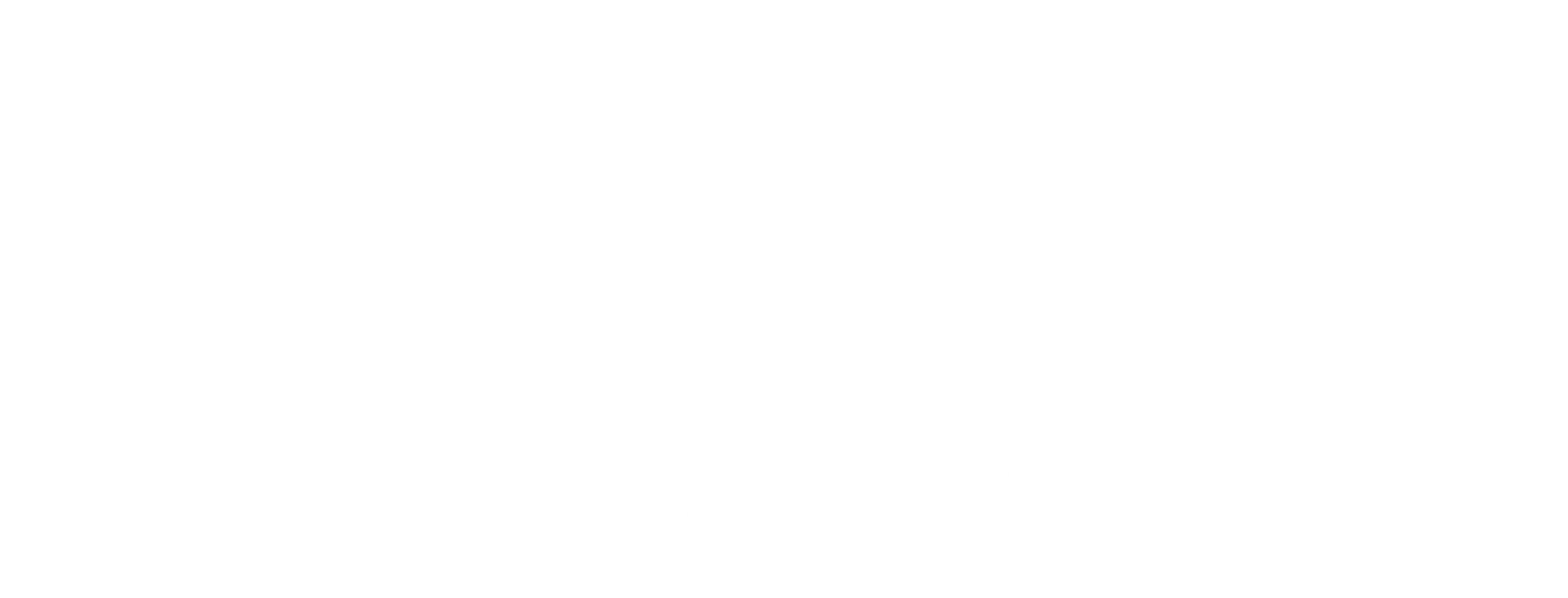
When debugging, we need to think about a few things:

* What **caused** the bug?
* How was it **found**?
* How could it have been **avoided** in the first place?

Debugging is a **sanitization process** consisting of:

* Preventing bugs in the first place
* Detecting bugs when they first arise
* Diagnosing and locating bugs
* Treating bugs

### Debugging Process



1. To **locate** the error, we depend on **test cases**.
2. Once an error is detected, we **design** the repair. This can be done with the help of the **specifications**. We need to check if we implemented the specifications exactly.
3. Finally, we **repair** the error based on our design and **retest** the program.

### Debugging Rules

* If lost or confused, randomly changing things in the code will not help.
* You cannot fix an error without knowing roughly where it is.
* Errors can only be caught and fixed by debugging in a systematic fashion.

## Debugging Strategies

### 01. Debug with Purpose

We should not just change some code and hope it will fix the problem. Instead, we should make the bug **reproducible**. If the problem can be reproduced, we are already half-way towards the solution.

Once the bug has been reproduced, we can use methodical **hypothesis testing** to find the solution.

1. Identify the **simplest input** that produces the bug.
2. Identify **assumptions** that have been made that could be incorrect.
3. Identify how the outcome of a new **test or change** guides you towards the solution.
4. Use pen and paper to stay **organized**.

### 02. Explain it to Someone Else

Often, **explaining the bug** to someone unfamiliar with the program will force us to look at it in a **different way**.

### 03. Focus on Recent Changes

If a **new bug** is found, we should examine **recent changes** to our code. It is very likely that the recent changes introduced the bug. This involves writing and testing code **incrementally**, using **git diff** to see recent changes and **regression testing**, the process of ensuring new changes do not break old code.

### 04. Dump State

In complicated programs, figuring out **where** a bug is can be hard and stepping through a debugger may be **time consuming**. Sometimes, it is easier to just **dump state**, where some log information is produced and examined to find oddities.

### 05. Rest

Sometimes, we have simply been **working too long** on our code. Taking a break and coming back to it may allow us to see issues that we were unable to identify before.

### 06. Use Tools

Sometimes, **error detecting tools** make finding certain bugs easy. For example, there are runtime tools that detect memory errors.

### 07. Think Ahead

Bugs often represent **misunderstanding** of a software interface. When one bug has been identified and resolved, we should:

* Think about whether the bug might **appear again** somewhere else in the code.
* Think about how to **avoid** the bug in the future.

## Unit Testing

The testing done by **programmers** is generally called **unit testing**, as opposed to those done by the testing team. More formally, a unit test is a **piece of code** that invokes a **unit of work** and **checks** one specific **end result** of that unit of work. If the assumptions of the end result turn out to be wrong, the unit test has failed. We are testing **small parts** of the code in isolation.

What exactly is a **unit** though? That depends entirely on us. There are **no hard and fast rules** for this. We can decide that a single function is a unit, or a group of functions is a unit. However, if a unit is **difficult to test**, most likely it needs to be **broken down** into smaller components.

### Drivers

Sometimes, the unit being tested may need input from **other units** or modules that have not been implemented yet. In those cases, we need to **simulate** the input. The module where the inputs are being simulated is called the **driver module**.

The inputs from the driver module may be hardcoded, taken as inputs from the user or read from a file.

### Stubs

The module being tested could also **call** some other module which has not yet been implemented. Such a module can also be **simulated** using a dummy module called a **stub**.

### Properties of Good Unit Tests

* Easy to implement
* Anyone can run
* Runs quickly
* Produces consistent results
* Has full control of the unit under test
* Is fully isolated from other units and tests
* When it fails, it should be easy to detect what the expected output was and determine how to pinpoint the problem

### Writing Good Tests

* The goal is to **expose problems**, so assume the role of the enemy. Managing to make the program fail is your success.
* Test **boundary conditions**, such as values, maximum values, empty arrays, etc.
* Test **different categories of inputs**, such as positive and negative inputs.
* Test **different categories of behaviour**, such as every menu option and every possible error message.
* Test **unexpected inputs**, such as a space in a last name, or a null input, etc.
* Test **representative, normal values**. Obviously, we need to test for random, reasonable values.

### Excuses

Common excuses for not running unit tests, none of which are valid by the way, include:

* It takes too long to run tests.
* It’s not my job.
* I don’t know how the code is supposed to behave.
* But it compiles!
* I’m being paid to write code, not test it.
* I feel guilty about putting the testing team out of work.
* My company won’t let me run unit tests on the live system.
* Tests are expensive.

Possibly the most understandable amongst these excuses is the last one. Tests can be extremely expensive to run. Consider the cost associated with testing air bags in a car! However, despite the cost and the extra effort, tests improve quality, bring quick feedback and provide secure knowledge for newcomers, all of which makes testing worth it.

### Writing a Unit Test

To check if code is behaving as expected, we use an **assertion**, a simple method call that verifies that something is true. We could also use libraries like JUnit.

public void assertTrue (boolean condition)  
{  
 if (!condition) abort();  
}

JAVA

The beautiful part here is, if a test fails, the program is **aborted**.

int a = 2;  
assertTrue(a == 2);

JAVA

Similarly, we can have other methods, such as ones that check if two integers are equal.

public void assertEquals(int a, int b)  
{  
 assertTrue(a == b);  
}

JAVA

Example

Say we have a simple method that is meant to find the largest number in a list.

public class Largest  
{  
 /\*\*  
 \* Return the largest element in a list.  
 \* @param ***list*** A list of integers  
 \* @return The largest number in the given list  
 \*/  
 public static int largest(int[] list)  
 {  
 int index, max = Integer.MAX\_VALUE;  
 for (index = 0; index < list.length - 1; index++)  
 if (list[index] > max) max = list[index];  
 return max;  
 }  
}

JAVA

Just this method can have several test cases:

* The largest number in the list should be the output.
* Changing the order of the list should produce the right result.
* Having duplicate values should still produce the right result.
* If there is just one number in the list, that should be the result.
* With negative numbers, the least negative number should be the result.
* An empty array should produce an error message.
* Passing a null value should produce an error message.

All these test cases can be tested with a series of assertions.

public void testSimple()  
{  
 assertEquals(9, Largest.largest(new int[] {7, 8, 9}));  
}

JAVA

This test will cause an error that the output was wrong. In fact, it will say that the output was some huge value. The mistake we made in our original code was that we initialized a variable to the maximum possible value. Let’s say we fix this by initializing this value to .

Next, we can change the order. Eventually we will get an error for that as well, since the for loop in the code ends at the wrong index.

Similarly, we will also get an error for negative numbers, since the variable value was changed to initialize at , but to accommodate negative values, we need to initialize it to the least possible value.

In this way, by running assertions repeatedly, we can test all possible test cases.

### Best Practices

* Unit test cases should be **independent**. Enhancements and changes in requirements should not affect unit tests.
* Test **one code** at a time.
* Follow clear and consistent **naming conventions** for unit tests.
* In case of **changes** to code for the module, ensure there is a **corresponding unit test case** and that the module passes the test before changing the implementation.
* **Bugs** identified in unit tests must be **fixed** before proceeding to the next phase of the SDLC.
* **Test as you code**. The more code that is written without testing, the more paths appear that need to be checked for errors.

### Additional Resources

[Why Good Developers Write Bad Unit Tests](https://mtlynch.io/good-developers-bad-tests/)