Week 9 - Testing and Debugging

Overview

In this topic we look at:

- Testing
  - Using Test Harnesses
  - Generating test cases
  - Defensive programming versus validation.
- Debugging

Testing

- Test Harnesses
- Generating Test Cases

Test Harnesses

Test Harnesses

- Is a small program written specifically to test modules.
- Brainstorm some test cases
    - Before you write the code consider what inputs the program might get and what the expected outputs should be for those inputs.

Test Harnesses – Example I

The module prototype is:

```
display_gem(gem)
```

The module takes a gem and displays its fields. The fields are:

```
class Gem_record
  attr_accessor :id, :description, :weight, :price
end
```

The data types are expected to be as follows:

Field	Data Type	Description	Example
id	Integer	An arbitrary integer	9
description	String	Text describing the type of gem	"Emerald"
weight	Float	Weight in grams	3.4
price	Float	Price in AUD	560.45

Test Harnesses – Example II

Considering a variety of possible inputs and expected outputs for the module above:

	Field/Attribute	Input	Expected Output
Test 1	id	1	1
	description	Emerald	“Emerald”
	weight	4	4.00
	price	580.99	$580.99

	Field/Attribute	Input	Expected Output
Test 2	id	2.9	2
	description		“Unknown”
	weight	4.568	4.57
	price	580.99	$580.99

Test Harness - Code

Run RUBY

```ruby

class Gem_record
  attr_accessor :id, :description, :weight, :price
end

def display_gem gem
  puts('Gem information is: ')
  puts gem.id
  puts gem.description
  puts gem.weight
  puts gem.price
end


```

Does the output match what was expected?

Compare Actual with Expected:

	Field/Attribute	Input	Expected Output	Actual Output	
Test 1	id	1	1	1	✓
	description	Emerald	“Emerald”	Emerald	✓
	weight	4	4.00	4	✗
	price	580.99	$580.99	580.99	✗

Correct the code (defensively):

Original:

Corrected:

Run RUBY

```ruby
class Gem_record
  attr_accessor :id, :description, :weight, :price
end

def display_gem gem
  puts('Gem information is: ')
  puts gem.id.to_i
  if (gem.description == nil)
    puts "Unknown"
  else
    puts gem.description
  end
  printf("%.2f\n", gem.weight)
  printf("$%5.2f\n", gem.price)
```

Run the tests again:

Check the output is correct:

Test Harnesses – Example III

- What other tests could we do?
- What other inputs might we get?
- What outputs would we expect?

Defensive programming versus validation

We have been trying to deal with a wide range of different possible inputs.

We see this can make our code increasing complex.

What is another way of addressing this problem?

Validation

Maybe we need to ensure that only a set of “valid” inputs are accepted.

This reduces the variety of different possible variables we need to deal with in later modules.

Eg: dealing with zeros, nil (null) values, etc.

Example of Validation I

The original `read_integer()` in `input_functions.rb`

looks as follows:

It produces the following results:

Example of Validation II

We need to change it to:

This uses regular expression matching see: https://www.safaribooksonline.com/library/view/ruby-programming-for/9781598633979/ch07.html

Run RUBY

```
1
2 # Display the prompt and return the read string
3 def read_string prompt
4   puts prompt
5   value = gets.chomp
6 end
7
8 # Display the prompt and return the read float
9 def read_float prompt
10   value = read_string(prompt)
11   value.to_f
12 end
13
14 # Display the prompt and return the read integer
```

Example of Validation III

This changes the behaviour as follows:

For us to remove our “defensive” code in `display_gem()` we need to be confident that where-ever `Gem` data is created or input into the system it is valid.

The Golden Rule for testing

Test for 0, 1 and 3 elements.

Eg: when testing array processing.

(see test_gem_array.rbin the Code)

Run RUBY

```ruby

class Gem_record
  attr_accessor :id, :description, :weight, :price
end


def display_gems gems
  index = 0
  while (index < gems.length)
    display_gem(gems[index])
    index += 1
  end
  return gems
end
```

The 'Golden Rule' for testing

Test for 0, 1 and 3 elements.

Eg: when testing array processing.

See the code then run it in the terminal.

A Buggy Program

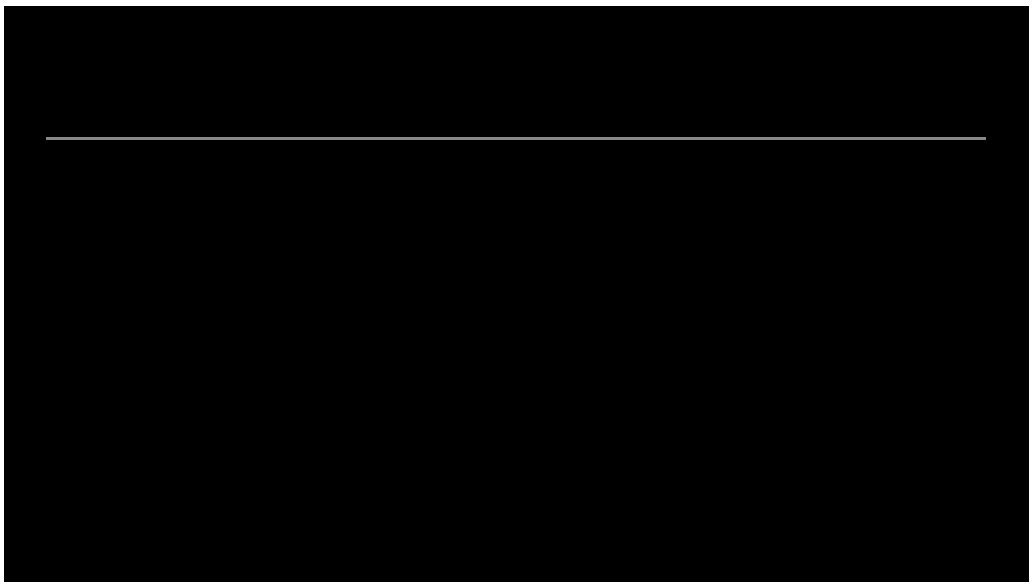
This program only prints out one gem, no matter how many there are.

Bottom-Up Testing/Debugging

I have suggested testing each module before including it in your program

If you did not do this and your program is crashing, you can do this retrospectively (i.e go back and test each module).

This is bottom-up testing or debugging (i.e start with the lowest level modules and test each before working your way up the structure chart).



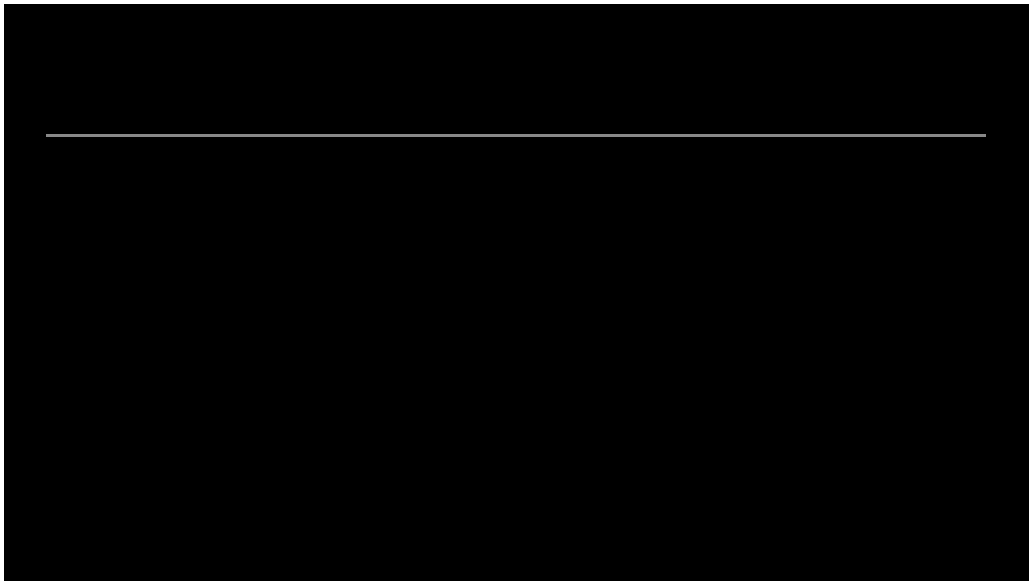
Top-Down Testing/Debugging

Another approach is to write all the modules and then test them together, correcting errors as you go.

This is commonly what programmers do in practice – but it is not as thorough as bottom-up testing – thus the produced code is not as reliable.

When doing top-down debugging you can use tools.

Print out the **state** of variables at different points



Two possible approaches

1.Track the changes to the variable that seems to be causing the problem.

2.Use a kind of “binary chop”

Tracking Variables

Debugging Programs - Binary Chop

Example of binary chop: think of a number from 1 and 100.

Eg: run the code below:

Run RUBY

```ruby
MIN = 1
MAX = 100

def read_string prompt
  puts prompt
  value = gets.chomp
end

def read_integer prompt
  value = read_string(prompt)
  while (!is_numeric?(value))
    puts("Please enter a number!")
    value = read_string(prompt)
  end
```

It is the same with programming – start at the beginning of the code and the end, and print out the state of the variables.

Work towards the error point – trying to find out when the variables get set with incorrect data.

Binary Chop

Summary

- We have looked at testing and debugging.
- Ideally modules (functions and procedures) should be tested using a bottom-up approach.
- Bottom-up testing can be done using Test Harnesses which contain the test data.
- Test data and test harnesses should be created before writing the module.
- If there are errors in the code we need to inspect the values of variables at key points.
- Two approaches to finding errors are:
    - Tracking the changes to selected variables in sequence through the program.
    - Checking the state of variables using a systematic search approach, like the binary chop.

Further Reading: Agarwal, B. Gupta, M. & Tayal, S.P 2009, Software Engineering and Testing, Jones & Bartlett Learning Chapt 7.