

Testing

CS 5010 Program Design Paradigms
“Bootcamp”
Lesson 2.4



Outline

- What do we test?
- How do we test them?
- How do we choose and write test cases?
- How do we go about debugging using tests?

Learning Objectives

- At the end of this lesson, the student should be able to:
 - examine a test and see what it is testing for
 - use the concept of equivalence classes to get 100% expression coverage
 - use the rackunit framework to write test suites for simple programming problems
 - use the rackunit framework to help in debugging simple programs

What do we want to test?

- Lots of things we could want to test, but for now we'll keep it simple:
- We want to make sure the answers from our functions are correct.
- But what makes an answer correct?
- And how do you convince a reader that your tests are testing the right thing?

Qualification Testing

- Does the program provide the functions that are specified in the problem?
- Do they take the right number and type of arguments? Do they return the right type of result?

If they don't, then the program is not ready for further testing...

Acceptance Testing

- Does the program's behavior satisfy the requirements given in the problem?
- This is our primary focus
- The requirements probably give some examples. Be sure to test them!
- Sometimes the requirements are more complicated, so you'll have to make up examples to check the requirements.

Mechanics of Testing

- In the Examples folder, you will find a file called **extras.rkt** that you should put in the folder with your work.
- Copy it into the folder where your code is.
- Near the top of your file, write

```
(require rackunit)
(require "extras.rkt")
```

to load our testing framework.
- Tests live in the file with the code
- That way they get run every time the code is loaded
 - This accomplishes regression testing.
- Wrap your tests in (**begin-for-test**)
 - that way you can put the tests anywhere in your file, and they will be run at the end of the file

Regression testing: Did I just break something?

The simplest test cases

- Compute the right answer by hand, and make up test cases to match

```
(begin-for-test
  (check-equal? (f2c 32) 0
    "32 Fahrenheit should be 0 Celsius")
  (check-equal? (f2c 212) 100
    "212 Fahrenheit should be 100 Celsius"))
```


This may not be enough

- Did I do the hand-computation right?
 - If you're lucky, the problem set said what the answer should be.
 - Otherwise, how does the reader know I didn't just take the result of my function and paste it into the test?
- Maybe there's more than one correct answer!

Property Testing

- Test a *property* of the answer
- Sometimes all you care about is that the answer has some property
- There could be more than one acceptable answer.
- Maybe the answer is something really complicated, so **check-equal?** won't do the job, and the best we can do is check to see whether the answer is "good enough."

Example of Property Testing

```
;; zero-of-quadratic? : Real^4 -> Boolean
;; RETURNS: whether  $\text{abs}(ax^2 + bx + c) < .01$ 
(define (zero-of-quadratic? a b c x)
  (< (magnitude
      (+ (* a x x)
          (* b x)
          c))
     .01))

(define (quadratic-solution1 a b c) ...)
(define (quadratic-solution2 a b c) ...)
```

I didn't have to find the solution to these examples by hand (especially the last one!) I don't care what the solutions are. I only care that they solve the given equations.

```
(begin-for-test
  (check-true (zero-of-quadratic? 1 0 4 (quadratic-solution1 1 0 4)))
  (check-true (zero-of-quadratic? 1 0 4 (quadratic-solution2 1 0 4)))
  (check-true (zero-of-quadratic? 1 0 1 (quadratic-solution1 1 0 1)))
  (check-true (zero-of-quadratic? 1 0 1 (quadratic-solution2 1 0 1)))
  (check-true (zero-of-quadratic? 13 58 6 (quadratic-solution1 13 58 6))))
```

Test Coverage

- How much of the possible behaviors have we tested?
- Want every line in the program exercised. This is called *100% expression coverage*.
- This is our **minimum** testing requirement(*).
- But this doesn't necessarily test all the desired behaviors of our program.
- To get a better handle on this, we introduce the idea of *equivalence partitioning*.

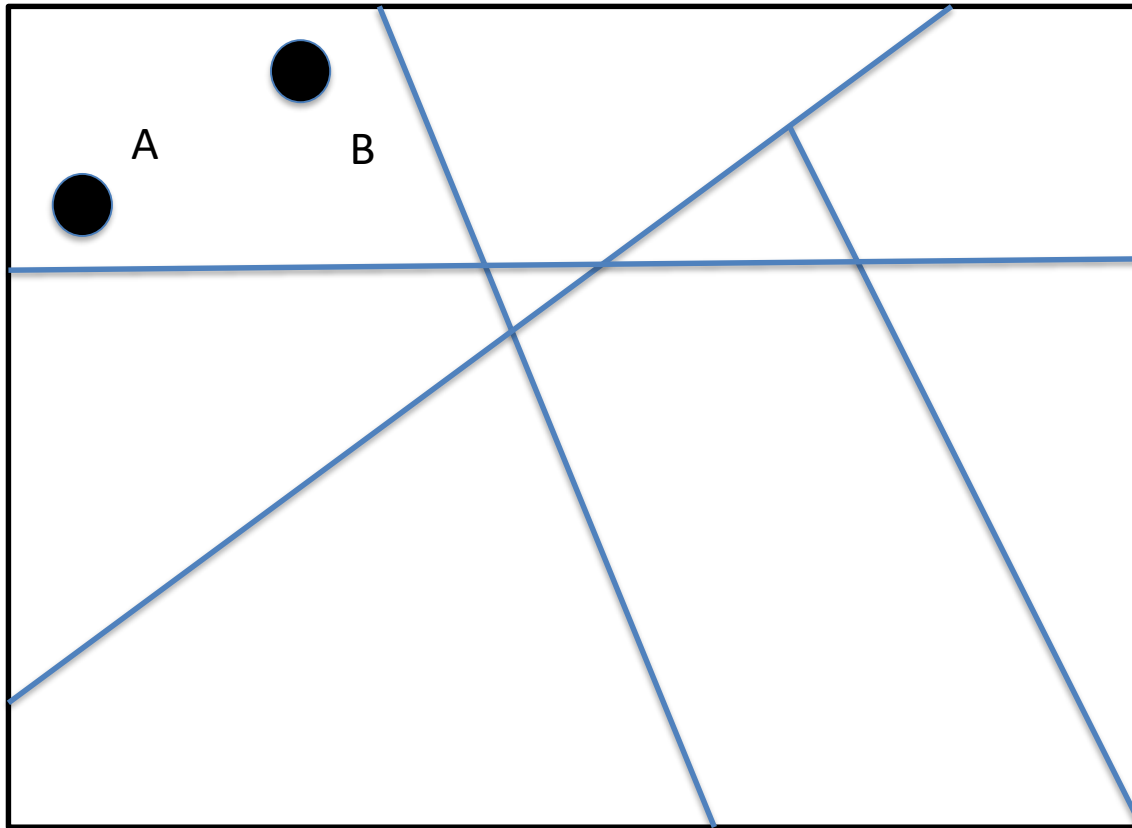
(*) except for functions that call **big-bang**

Equivalence Partitioning

- Possible arguments to your function typically fall into classes for which the program yields similar results.
- Example: f2c had only 1 partition.
- Example: ball-after-mouse depends on
 - Which mouse event we're dealing with
 - Whether the mouse event is inside or outside the ball
 - Whether the ball is selected
- So we need $3 \times 2 \times 2 = 12$ tests to cover all these combinations.

Equivalence Partitioning

Regions of similar behavior



If the program works for input A, it will probably work for input B

Choosing test cases

- The first step in choosing test cases is to divide your program into equivalence partitions.
- Pick some input and output values for each partition. Give mnemonic names to each of them. You can put these definitions with your data definitions, so you can use the names in your examples.
- Then write your tests using the mnemonic names.

Testing ball-after-mouse

- For ball-after-mouse, we decided there were 12 partitions: 3 mouse events, 2 points (inside or outside the ball), and 2 balls (selected or unselected).
- So we create two balls at position (20,30), with radius 10, one unselected and one selected, and define two points, one inside the ball and one outside.

Example (1)

```
;; two balls at (20,30), one unselected and one selected  
(define ball-unselected (make-ball 20 30 10 false))  
(define ball-selected (make-ball 20 30 10 true))
```

```
;; (22,28) is inside the ball at (20,30)  
(define point-inside-x 22)  
(define point-inside-y 28)
```

```
;; (31,19) is outside the ball at (20,30)  
(define point-outside-x 31) ;; 20+10 = 30, so 31 is outside  
(define point-outside-y 19) ;; 30-10 = 20, so 19 is outside
```

The names of these values must be descriptive. Calling them **ball-1** and **ball-2** is not acceptable.

Example (2)

```
;; next we make two balls, one moved to the inside point  
;; and one moved to the outside point.
```

```
;; When a ball is moved, it will stay selected, so we make  
;; selected? true for both of these.
```

```
(define ball-moved-to-point-inside  
  (make-ball point-inside-x point-inside-y 10 true))
```

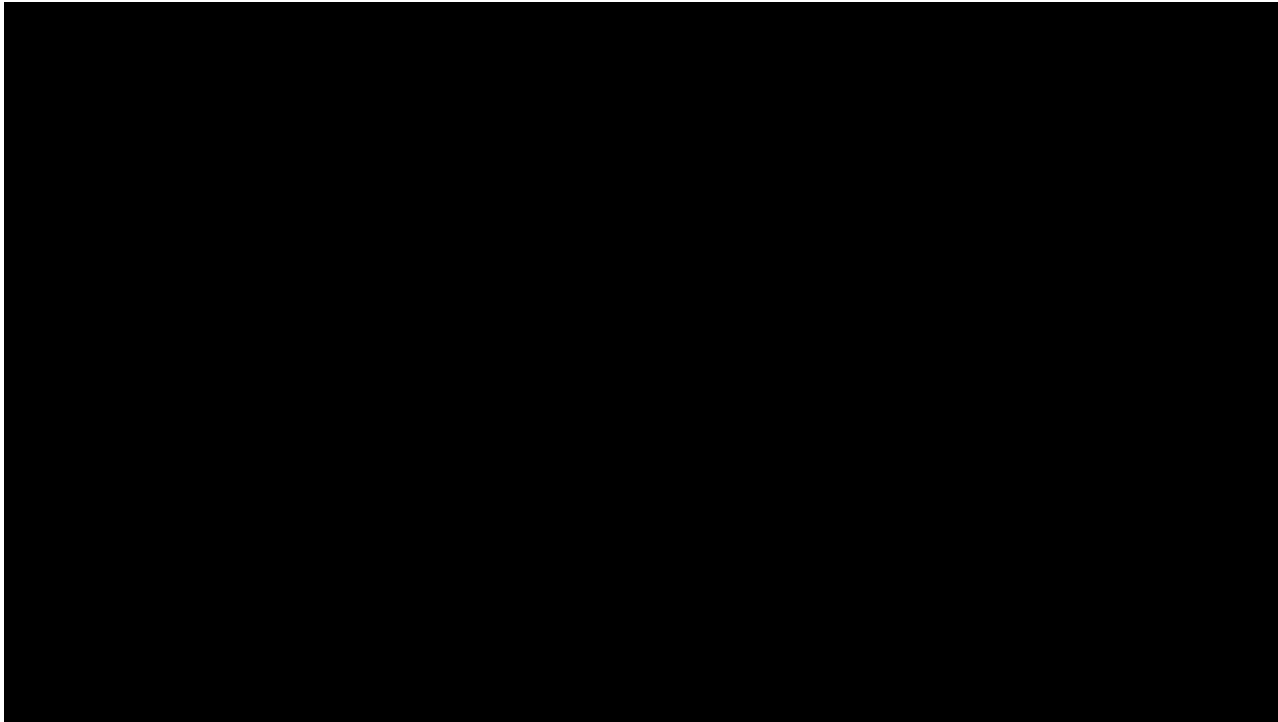
```
(define ball-moved-to-point-outside  
  (make-ball point-outside-x point-outside-y 10 true))
```

Example

```
(check-equal?  
  (ball-after-mouse ball-unselected point-inside-x point-inside-y "button-down")  
  ball-selected  
  "button-down inside the ball should select it")  
  
(check-equal?  
  (ball-after-mouse ball-unselected point-outside-x point-outside-y "button-down")  
  ball-unselected  
  "button-down outside the ball should leave it unchanged")
```

- **check-equal?** takes 3 arguments: the expression to be tested, the value we believe is the correct answer, and an optional string that is printed if the test fails.
- Supply an informative error message if you can. An uninformative error message, like “wrong answer” is worse than no message at all.

Video: ball-after-mouse-with-tests



[YouTube link](#)

Note: this video uses an older version of our testing technology. We use **begin-for-test** instead of **define-test-suite** and **run-test**. The details are a little different, but the principles are the same.

Using Tests

- Run your program with its tests
- Debug so that all your tests pass
- If you didn't achieve 100% expression coverage, go back and add more tests.
 - Just because your tests pass with 100% coverage doesn't mean your program is right!
 - But 100% expression coverage is our standard for this course.
 - Your workplace may have different standards.

except for functions
that call **big-bang**

Testing Pitfalls

- DON'T just paste in the actual results of your function.
- Some functions may have more than one correct answer.
 - your tests should accept *any* correct answer, not just the one your solution happens to produce
 - use property testing to handle this situation.

Testing Pitfalls (2)

- Avoid coincidences in your tests, just as you did in your examples

- Bad:

```
(check-equal?  
  (book-profit-margin  
    (make-book "Little Lisper" "Friedman" 2.00 4.00))  
  2.00)
```

- Better:

```
(check-equal?  
  (book-profit-margin  
    (make-book "Little Lisper" "Friedman" 2.00 5.00))  
  3.00)
```

Tests Written?

- Once you've written the deliverables for the first five steps of the design recipe, it's time to run the program. (Program Review will come later)
- What could possibly go wrong?
- Let's make a short list...

What could go wrong?

- Program fails to load
 - unbalanced parens? The unmatched paren is highlighted in the interaction window.
 - missing function?
 - forgot to write definition
 - misspelled function name
 - forgot to **require** the library module
 - misspelled library name
 - *the error message Racket gives you in this case is especially scary. But don't be frightened. It just means that it couldn't find the library you told it to look for.*

What could go wrong? (2)

- You could get an error calling a Racket primitive.
 - eg: "can't apply string=? to 1"
 - this may be something simple, like the wrong test,
 - or it may be more subtle-- "how did I manage to pass a 1 to string=?"
 - Write more tests to see how you got to this.

What could go wrong (3)

- A test fails
 1. Identify the test that failed
 - Racket will highlight the test that failed. Having an informative error message will also help you identify the test
 2. Check the test: is the answer that it asked for really the right one?
 - If not, fix the test
 - DON'T just paste in the actual results of your function.
 3. If the test is right, play detective by adding new tests.
 - Add a test to see if your function called the right helper. Did it?
 - yes: the helper is the one giving the wrong answer. Test the helper and fix it.
 - no: your original function didn't call the helper as it should.
 - » The call to the helper is probably guarded by a predicate. Test the predicate to see if it is returning the right value.
 - » Did it pass the right arguments? Write some more tests to see.

Debugging by Testing: Example

Code:

```
(define (ball-after-mouse b mx my mev)
  (cond
    [(mouse=? mev "button-down")
     (ball-after-button-down b mx my)]
    [(mouse=? mev "drag") (ball-after-drag b mx my)]
    [(mouse=? mev "button-up") (ball-after-button-up b mx my)]
    [else b]))
```

Imagine we have this function definition and this failing test.

Failing Test:

```
(check-equal?
  (ball-after-mouse
    ball-unselected point-inside-x point-inside-y "button-down")
  ball-selected
  "button-down on an unselected ball should make the ball
  selected")
```

This test checks the *combination* of **ball-after-mouse** and **ball-after-button-down**. If it fails, either procedure might be at fault.

Debugging by Testing (2)

```
(check-equal?  
  (ball-after-mouse  
    ball-unselected  
    point-inside-x point-inside-y  
    "button-down")  
  (ball-after-button-down  
    ball-unselected  
    point-inside-x point-inside-y))
```

On a button-down, we were supposed to call **ball-after-button-down**. So let's create a test to see if that happened.

Test fails: problem is in **ball-after-mouse**

Test succeeds: problem is in **ball-after-button-down**

We know that **ball-after-button-down** was supposed to be called, so these two expressions should return the same thing, even if it's the wrong thing. So if this test fails, we know that **ball-after-mouse** didn't call **ball-after-button-down** correctly. If the test succeeds, we know that **ball-after-button-down** was called, but it is returning the wrong thing, because the test on the previous slide is still failing.

Tracking down your bug

```
(define (ball-after-button-down b mx my)
  (if (inside-ball? mx my b)
      (ball-make-selected b)
      b))
```

```
(check-equal?
 (ball-after-button-down
  ball-unselected
  point-inside-x point-inside-y)
 (ball-make-selected ball-unselected))
```

Let's imagine we've identified **ball-after-button-down** as the likely culprit. We could write another test to see whether **ball-after-button-down** is calling **ball-make-selected** correctly.

Test succeeds: problem is in **ball-make-selected**

Test fails: problem is in **inside-ball?**

Keep your bug from re-appearing

- Leave the extra tests in your file
- That way if your bug reappears you will have the detective work all set up.

Disclaimer

- Our presentation has been specific to Racket and to this course, but the ideas and techniques are adaptable to other settings and other languages.
- Your employer may have different conventions for managing tests.
- If your employer does not have a convention or process for systematic testing, you should urge him (or her) to introduce one.

Summary

- You should now be able to:
 - examine a test and see what it is testing for
 - use the concept of equivalence classes to get 100% expression coverage
 - use the rackunit framework to write test suites for simple programming problems
 - use the rackunit framework to help in debugging simple programs

Next Steps

- Study 02-4-1-test-quadratics.rkt and 02-4-2-ball-after-mouse-with-tests.rkt .
- Remove some of the tests from 02-4-2-ball-after-mouse-with-tests.rkt and compare the code coloring with and without 100% test coverage.
- If you have questions or comments about this lesson, post them on the discussion board.
- Go on to the next lesson.