CS1632, Lecture 8: Writing Testable Code

Bill Laboon


LET'S DEFINE OUR TERMS, GENTLEMEN.

Testable Code

Code for which it is easy to write and perform tests, automated and manual, at various levels of abstraction, and track down errors when tests fail.

Key Ideas for Testable Code

- Segment code - make it modular
- Give yourself something to test
- Make it repeatable
- DRY (Don’t repeat yourself)
- Write code with seams

Segment Code

- Methods should be SMALL and SPECIFIC - “Do one thing and do it well” (UNIX philosophy)

```
# Bad - not small, not specific
def get_num_monkeys_and_set_database monkey_list, database
  if database.nil?
    set_database DEFAULT_DATABASE
  else
    set_database database
  end
  num_monkeys = get_num_monkeys monkey_list
  num_monkeys ||= 0 # sets num_monkeys to 0 if it is nil
  num_monkeys = normalize_monkey_number num_monkeys
  if numMonkeys < 0
     numMonkeys = 0
  end
  num_monkeys
end
```

Refactor

```
# Better
def set_database database
  if database.nil?
    set_database DEFAULT_DATABASE
  else
    set_database database
  end
end
```

```
def get_num_monkeys monkey_list
  num_monkeys = get_num_monkeys monkey_list
  num_monkeys ||= 0 # sets num_monkeys to 0 if it is nil
  num_monkeys = normalize_monkey_number num_monkeys
  if numMonkeys < 0
     numMonkeys = 0
  end
  num_monkeys
end
```

Give Yourself Something to Test

- Return values are worth their weight in gold!
  Easy to assert against
  Guaranteed to exist in Ruby, may as well return something good

- Exceptions, modified state, modified attributes, etc…

```
def add_monkey m
   unless m.nil?
     @monkey_list << m
   end
end
```

Refactor

```
# Better
def add_monkey m
  raise “You passed a null monkey!” if m.nil?
  raise “You passed an invalid monkey!” if m.invalid?
  @monkey_list << m
  @monkey_list.count
end
```

`

- Randomness or dependence on external data should be minimized

- Try to segregate PURE FUNCTIONS from IMPURE FUNCTIONS

- Pure functions = output depends ONLY on input, do nothing else except return a value (no side effects)
  Side effects = write to database, read a global variable, write to file system, etc.

Pure function

```
# Why pure?
# Output only depends on input
# No side effects
# Completely deterministic - for some value x, will
#   always return exact same value
# Referential transparency - for all intents and purposes,
#   4 could be replaced in any code with square(2)
def square x
  x * x
end
```

Impure function

```
# Why impure?
# Result depends on things other than arguments
# Has side effects (writes to file)
# Not deterministic (depends on what time it is)
# Not referentially transparent
def log_message msg
  if @logging == true
    # returns string version of current time
    time = Time.new.to_s
    write_to_file @logging_file, “#{time}: #{msg}”
  end
end
```

This Is Hard to Test

```
def come_out
  roll_1 = (Die::new).roll()
  roll_2 = (Die::new).roll()
  total = dieRoll1 + dieRoll2
  @come_out_total = total
  case total
  when 2, 3, 12
    CRAPS_LOSE
  when 7, 11
    CRAPS_WIN
  else
    CRAPS_PLAY
  end
end
```

Easier to test

```
def come_out_roll die_val_1, die_val_2
 total = dieRoll1 + dieRoll2
 case total
  when 2, 3, 12
    CRAPS_LOSE
  when 7, 11
    CRAPS_WIN
  else
    CRAPS_PLAY
  end
end

def roll_dice die_1, die_2
  die_roll_1 = die_1.roll
  die_roll_2 = die_2.roll
  return die_roll_1, die_roll_2
end
```

DRY - Don’t Repeat Yourself

- Don’t copy and paste code from one section of your program to another

- Don’t have multiple methods with the same or similar functionality

- Try to have “generic” methods that can be applied in as many places as possible

Bad

```
def add_monkey m
   return nil if m.nil?
   @animal_list << m
   @animal_list.count
}

def add_lion l
   return nil if l.nil?
   @animal_list << l
   @animal_list.count
}

def add_parrot p
   return nil if p.nil?
   @animal_list << p
   @animal_list.count
}
```

Refactor

```
def add_animal a
    return nil if a.nil?
    @animal_list << a
    @animal_list.count
}
```

Ensure That You Don't Have Multiple Methods Doing The Same Thing

```
def add_up_array a
  return 0 unless a.is_a?(Array)
  to_return = 0
  a.each { |x| to_return += x }
  to_return
}

# elsewhere in codebase..
def total_array arr
  arr.reduce(:+)
}
```

Why?

- Twice as much room for error
- Bloated codebase
- Perhaps slightly different behavior
- Harder to find errors
- Which one to use?

Replicated Code Could Be Internal To Methods!

```
# In one method…
name = db.where("user_id = " + id_num).get_names[0]

# Elsewhere, in another method…
name = db.find(id).get_names.first
```

You Can DRY This Up, Too

```
def get_name database, id
  # Add in guard code here
  db.find(id).get_names().first()

# In one method…
name = get_name db, id

# Elsewhere, in another method…
name = get_name db, id
```

Be a code anti-natalist!

Code that exists is code that can have defects.

Provide (or Look For) Seams

Seams are places where behavior can be modified without modifying code

Make these common!

The rule of “making more methods” can be considered a special subclass of this rule.

Dependency Injection

- Passing in (“injecting”) any objects that a method relies on (dependencies)

- In other words, favor arguments to methods as opposed to instance variables or internally generated objects

    - Having a ::new in the middle of a method is a code smell

- Will allow you to easily double/stub/mock objects

- Helps make methods pure, too!

Example

```
// SEAM - dependency injection
def printDoc printer, document, arguments
  printer.setArgs arguments
  printer.print document
end
```

```
// NO SEAM - no dependency injection
def print_doc_
  document = generate_doc
  p = Printer::new [:doublesided]
  p.print document
end
```