Testing

Interface and Implementation

One of the most important ideas in computing is the difference between *interface* and *implementation*

One of the most important ideas in computing is the difference between *interface* and *implementation*

Interface: how something interacts with the world

One of the most important ideas in computing is

the difference between *interface* and *implementation*

Interface: how something interacts with the world

Implementation: how it does what it does

One of the most important ideas in computing is the difference between *interface* and *implementation*

Interface: how something interacts with the world

Implementation: how it does what it does

```
def integrate(func, x1, x2):
  ...math goes here...
  return result
```

One of the most important ideas in computing is the difference between *interface* and *implementation*

Interface: how something interacts with the world

Implementation: how it does what it does

```
def integrate(func, x1, x2):
  ...math goes here...
  return result
```

Interface: (f, x_1, x_2) -> integral

One of the most important ideas in computing is the difference between *interface* and *implementation*

Interface: how something interacts with the world

Implementation: how it does what it does

```
def integrate(func, x1, x2):
  ...math goes here...
  return result
```

Interface: (f, x_1, x_2) -> integral

Implementation: we don't (have to) care

Often use this idea to simplify unit testing

Often use this idea to simplify unit testing

Want to test components in program one by one

Often use this idea to simplify unit testing

Want to test components in program one by one

But components depend on each other

Often use this idea to simplify unit testing

Want to test components in program one by one

But components depend on each other

How to isolate the component under test from other components?

Often use this idea to simplify unit testing

Want to test components in program one by one

But components depend on each other

How to isolate the component under test from other components?

Replace the other components with things that have the same interfaces, but simpler implementations

Often use this idea to simplify unit testing

Want to test components in program one by one

But components depend on each other

How to isolate the component under test from other components?

Replace the other components with things that have the same interfaces, but simpler implementations

Sometimes requires *refactoring*

Often use this idea to simplify unit testing

Want to test components in program one by one

But components depend on each other

How to isolate the component under test from other components?

Replace the other components with things that have the same interfaces, but simpler implementations

Sometimes requires *refactoring*

Or some up-front design

Back to those fields in Saskatchewan...

Test function that reads a photo from file

Test function that reads a photo from file

```
def read_photo(filename):
  result = set()
  reader = open(filename, ’r’)
  …fill result with rectangles in file…
  reader.close()
  return result
```

Test function that reads a photo from file

```
def read_photo(filename):
  result = set()
  reader = open(filename, 'r')
  …fill result with rectangles in file…
  reader.close()
  return result

def test_photo_containing_only_unit():
  assert read_photo('unit.pht') == { ((0, 0), (1, 1)) }
```

Experience teaches that this is a bad idea

Experience teaches that this is a bad idea

1. External files can easily be misplaced

Experience teaches that this is a bad idea

1. External files can easily be misplaced
2. Hard to understand test if fixture stored elsewhere

Experience teaches that this is a bad idea

1. External files can easily be misplaced
2. Hard to understand test if fixture stored elsewhere
3. File I/O is much slower than memory operations

Experience teaches that this is a bad idea

1. External files can easily be misplaced
2. Hard to understand test if fixture stored elsewhere
3. File I/O is much slower than memory operations

The longer tests take to run, the less often they will be run

Experience teaches that this is a bad idea

1. External files can easily be misplaced
2. Hard to understand test if fixture stored elsewhere
3. File I/O is much slower than memory operations

The longer tests take to run, the less often they will be run

And the more often developers will have to backtrack to find and fix bugs

Original function

```
def count_rect(filename):
  reader = open(filename, ’r’)
  count = 0
  for line in reader:
    count += 1
  reader.close()
  return count
```

Original function

```
def count_rect(filename):
  reader = open(filename, ’r’)
  count = 0
  for line in reader:
    count += 1
  reader.close()
  return count
```

One rectangle per line, no comments or blank lines

Original function

```
def count_rect(filename):
  reader = open(filename, ’r’)
  count = 0
  for line in reader:
    count += 1
  reader.close()
  return count
```

One rectangle per line, no comments or blank lines

Real counter would be more sophisticated

Refactored

```
def count_rect_in(reader):
  count = 0
  for line in reader:
    count += 1
  return count

def count_rect(filename):
  reader = open(filename, 'r')
  result = count_rect_in(reader)
  reader.close()
  return result
```

Refactored

```
def count_rect_in(reader):
  count = 0
  for line in reader:
    count += 1
  return count

def count_rect(filename):
  reader = open(filename, 'r')
  result = count_rect_in(reader)
  reader.close()
  return result
```

Does the work, but does *not* open the file

Refactored

```
def count_rect_in(reader):
  count = 0
  for line in reader:
    count += 1
  return count

def count_rect(filename):
  reader = open(filename, 'r')
  result = count_rect_in(reader)
  reader.close()
  return result
```

Opens the file

Refactored

```
def count_rect_in(reader):
  count = 0
  for line in reader:
    count += 1
  return count

def count_rect(filename):
  reader = open(filename, 'r')
  result = count_rect_in(reader)
  reader.close()
  return result
```

Opens the file

Keeps name of original function

Now write tests

Now write tests

```
from StringIO import StringIO

Data = ’’’0 0 1 1
1 0 2 1
2 0 3 1’’’

def test_num_rect():
  reader = StringIO(Data)
  assert count_rect(reader) == 3
```

Now write tests

```
from StringIO import StringIO

Data = ’’’0 0 1 1
1 0 2 1
2 0 3 1’’’

def test_num_rect():
  reader = StringIO(Data)
  assert count_rect(reader) == 3
```

A "file" that tests can be run on

Now write tests

```
from StringIO import StringIO

Data = '''0 0 1 1
1 0 2 1
2 0 3 1'''

def test_num_rect():
  reader = StringIO(Data)
  assert count_rect(reader) == 3
```

Acts like a file, but uses a string in memory for storage

Now write tests

```
from StringIO import StringIO

Data = ’’’0 0 1 1
1 0 2 1
2 0 3 1’’’

def test_num_rect():
  reader = StringIO(Data)
  assert count_rect(reader) == 3
```

Doesn't know it isn't reading from a real file

Use the same method to test output

Use the same method to test output

Write to a `StringIO`

Use the same method to test output

Write to a `StringIO`

Use `getvalue` to get and check its final contents

Use the same method to test output

Write to a `StringIO`

Use `getvalue` to get and check its final contents

```
def test_write_unit_only():
  fixture = { ((0, 0), (1, 1)) }
  writer = StringIO()
  photo_write(fixture, writer)
  result = writer.getvalue()
  assert result == '0 0 1 1\n'
```

Use the same method to test output

Write to a `StringIO`

Use `getvalue` to get and check its final contents

```
def test_write_unit_only():
  fixture = { ((0, 0), (1, 1)) }
  writer = StringIO()
  photo_write(fixture, writer)
  result = writer.getvalue()
  assert result == '0 0 1 1\n'
```

Doesn't know it isn't reading from a real file

Use the same method to test output

Write to a `StringIO`

Use `getvalue` to get and check its final contents

```
def test_write_unit_only():
  fixture = { ((0, 0), (1, 1)) }
  writer = StringIO()
  photo_write(fixture, writer)
  result = writer.getvalue()
  assert result == ’0 0 1 1\n’
```

Get everything written to the `StringIO` as a string

One more task

One more task

```
def photo_write(photo, writer):
  contents = list(photo)
  contents.sort()
  for rect in contents:
    print >> writer, rect[0][0], rect[0][1],
                     rect[1][0], rect[1][1]
```

One more task

```
def photo_write(photo, writer):
  contents = list(photo)
  contents.sort()
  for rect in contents:
    print >> writer, rect[0][0], rect[0][1],
                     rect[1][0], rect[1][1]
```

Why do the extra work of sorting?

One more task

```
def photo_write(photo, writer):
  contents = list(photo)
  contents.sort()
  for rect in contents:
    print >> writer, rect[0][0], rect[0][1],
                     rect[1][0], rect[1][1]
```

Why do the extra work of sorting?

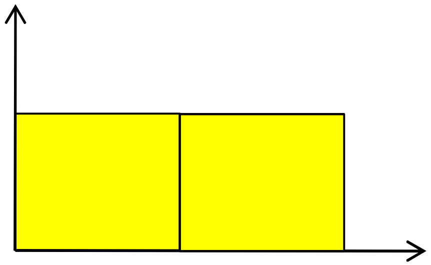
This version is simpler and faster

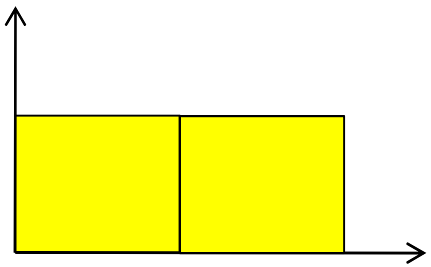
```
def photo_write(photo, writer):
  for rect in photo:
    print >> writer, rect[0][0], rect[0][1],
                     rect[1][0], rect[1][1]
```

This version is simpler and faster

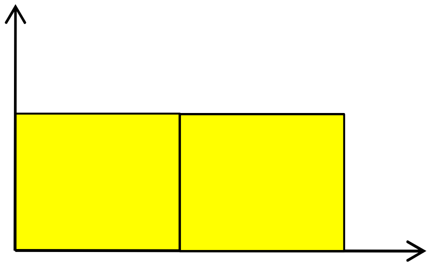
```
def photo_write(photo, writer):
  for rect in photo:
    print >> writer, rect[0][0], rect[0][1],
                     rect[1][0], rect[1][1]
```

But there is no way to predict its output!



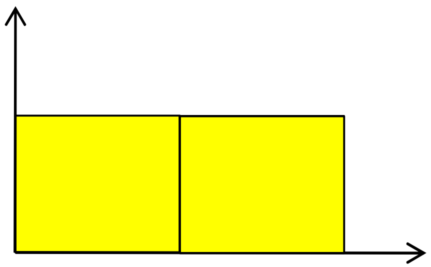


```
two_fields = { ((0, 0), (1, 1)), ((1, 0), (2, 1)) }
photo_write(two_fields, …)
```



```
two_fields = { ((0, 0), (1, 1)), ((1, 0), (2, 1)) }
photo_write(two_fields, …)
```

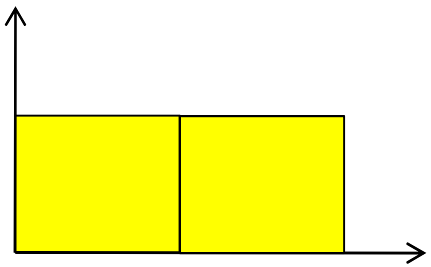
```
0 0 1 1
1 0 2 1
```



```
two_fields = { ((0, 0), (1, 1)), ((1, 0), (2, 1)) }
photo_write(two_fields, …)
```

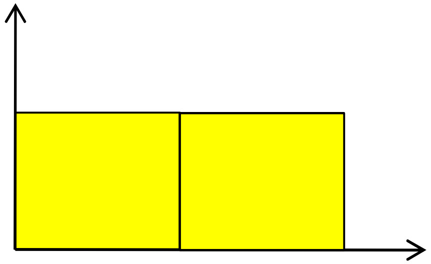
```
0 0 1 1
1 0 2 1
```

```
1 0 2 1
0 0 1 1
```



```
two_fields = { ((0, 0), (1, 1)), ((1, 0), (2, 1)) }
photo_write(two_fields, …)
```

$$\boxed{\begin{matrix}0&0&1&1\\1&0&2&1\end{matrix}} \neq \boxed{\begin{matrix}1&0&2&1\\0&0&1&1\end{matrix}}$$



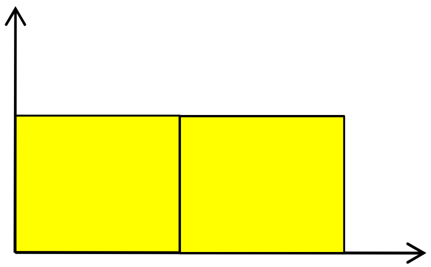
```
two_fields = { ((0, 0), (1, 1)), ((1, 0), (2, 1)) }
photo_write(two_fields, …)
```

Sets are unordered

```
0 0 1 1
1 0 2 1
```

≠

```
1 0 2 1
0 0 1 1
```

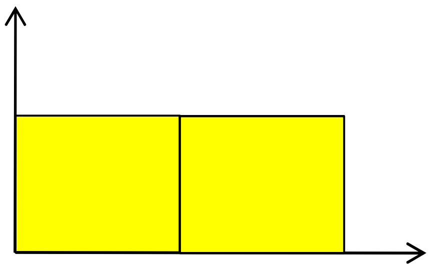


```
two_fields = { ((0, 0), (1, 1)), ((1, 0), (2, 1)) }
photo_write(two_fields, …)
```

$$\begin{array}{|cccc|} 0 & 0 & 1 & 1 \\ 1 & 0 & 2 & 1 \end{array} \neq \begin{array}{|cccc|} 1 & 0 & 2 & 1 \\ 0 & 0 & 1 & 1 \end{array}$$

~~Sets are unordered~~

Set elements are stored in an arbitrary order



```
two_fields = { ((0, 0), (1, 1)), ((1, 0), (2, 1)) }
photo_write(two_fields, …)
```

0	0	1	1
1	0	2	1

≠

1	0	2	1
0	0	1	1

~~Sets are unordered~~

Set elements are stored in an arbitrary order

We can't test if we can't predict the result

Our existing tests are inconsistent

Our existing tests are inconsistent

```
# From input test
Data = ’’’0 0 1 1
1 0 2 1
2 0 3 1’’’
```

Our existing tests are inconsistent

```
# From input test
Data = ’’’0 0 1 1
1 0 2 1
2 0 3 1’’’


# From output test
def test_write_unit_only():
  fixture = { ((0, 0), (1, 1)) }
  …
  assert result == ’0 0 1 1\n’
```

Our existing tests are inconsistent

```
# From input test
Data = '''0 0 1 1
1 0 2 1
2 0 3 1'''
```

```
# From output test
def test_write_unit_only():
  fixture = { ((0, 0), (1, 1)) }
  …
  assert result == '0 0 1 1\n'
```

Our existing tests are inconsistent

```
# From input test
Data = ’’’0 0 1 1
1 0 2 1
2 0 3 1’’’

# From output test
def test_write_unit_only():
  fixture = { ((0, 0), (1, 1)) }
  …
  assert result == ’0 0 1 1\n’
```

Do photo files have
a newline at the end
of the last line or not?

Our existing tests are inconsistent

```
# From input test
Data = '''0 0 1 1
1 0 2 1
2 0 3 1'''

# From output test
def test_write_unit_only():
  fixture = { ((0, 0), (1, 1)) }
  …
  assert result == '0 0 1 1\n'
```

Do photo files have a newline at the end of the last line or not?

Either answer is better than "maybe"

Have to *design for test*

Have to *design for test*

Depend on interface, not implementation

Have to *design for test*

Depend on interface, not implementation

– So it's easy to replace other components for testing

Have to *design for test*

Depend on interface, not implementation

– So it's easy to replace other components for testing

– And tests don't have to be rewritten over and over

Have to *design for test*

Depend on interface, not implementation

– So it's easy to replace other components for testing

– And tests don't have to be rewritten over and over

Isolate interactions with outside world

Have to *design for test*

Depend on interface, not implementation

– So it's easy to replace other components for testing

– And tests don't have to be rewritten over and over

Isolate interactions with outside world

– Like opening files

Have to *design for test*

Depend on interface, not implementation

– So it's easy to replace other components for testing

– And tests don't have to be rewritten over and over

Isolate interactions with outside world

– Like opening files

Make things you are going to examine deterministic software carpentry

created by

Greg Wilson

August 2010