Mini-Lecture 10

Integrated Development

Stepwise Refinement: Basic Principles

- **Write Specifications First**
  Write a function specification before writing its body
- **Take Small Steps**
  Do a little at a time; make use of **placeholders**
- **Run as Often as You Can**
  This can catch syntax errors
- **Separate Concerns**
  Focus on one step at a time
- **Intersperse Programming and Testing**
  When you finish a step, test it immediately

Stepwise Refinement: Basic Principles

- **Write Specifications First**
  Write a function specification before writing its body
- **Take Small Steps**
  Do a little at a time; make use of **placeholders**
- **Run as Often as You Can**
  This can catch syntax errors
- **Separate Concerns**
  Focus on one step at a time
- **Intersperse Programming and Testing**
  When you finish a step, test it immediately

Integrated Development

Using Placeholders in Design

- Delay do anything not immediately relevant
  - Use comments to write steps in English
  - Add "stubs" to allow you to run program often
  - Slowly replace stubs/comments with real code
- Only create new local variables if you have to
- Sometimes results in creation of more functions
  - Replace the step with a function call
  - But leave the ***function definition*** empty for now
  - This is called **top-down design**

Function Stubs

Procedure Stubs

- Single statement: pass
  - Body cannot be empty
  - This command does nothing
- **Example**:

```
def foo():
    pass
```

Fruitful Stubs

- Single return statement
  - Type should match spec.
  - Return a "default value"
- **Example**:

```
def first_four_letters(s):
    return '' # empty string
```

Purpose of Stubs

Create a program that may not be correct, but does not crash.

Example: Reordering a String

- last_name_first('Walker White') is 'White, Walker'

```
def last_name_first(s):
    """Returns: copy of s in form <last-name>, <first-name>

    Precondition: s is in the form <first-name> <last-name>
    with one blank between the two names"""
    # Find the first name
    # Find the last name
    # Put them together with a comma
    return ' ' # Currently a stub
```

Example: Reordering a String

- last_name_first('Walker White') is 'White, Walker'

```
def last_name_first(s):
    """Returns: copy of s in form <last-name>, <first-name>

    Precondition: s is in the form <first-name> <last-name>
    with one blank between the two names"""
    end_first = s.find(' ')
    first_name = s[:end_first]
    # Find the last name
    # Put them together with a comma
    return first_name # Still a stub
```

Refinement: Creating Helper Functions

```
def  last_name_first(s):
    """Returns: copy of s in the form
    <last-name>, <first-name>
    Precondition: s is in the form
    <first-name> <last-name> with
    with one blank between names"""
    first = first_name(s)
    # Find the last name
    # Put together with comma
    return first # Stub
```

```
def first_name(s):
    """Returns: first name in s
    Precondition: s is in the form
    <first-name> <last-name> with
    one blank between names"""
    end = s.find(' ')
    return s[:end]
```

Refinement: Creating Helper Functions

```
def  last_name_first(s):
    """Returns: copy of s in the form
    <last-name>, <first-name>
    Precondition: s is in the form
    <first-name> <last-name> with
    with one blank between names"""
    first = first_name(s)
    # Find the last name
    # Put together with comma
    return first # Stub
```

```
def first_name(s):
    """Returns: first name in s
    Precondition: s is in the form
    <first-name> <last-name> with
    one blank between names"""
    end = s.find(' ')
    return s[:end]
```

Do This Sparingly

- If you might use this step in **another** function later
- If implementation is rather long and complicated

Example: Reordering a String

- last_name_first('Walker White') is 'White, Walker'

```
def last_name_first(s):
    """Returns: copy of s in form <last-name>, <first-name>

    Precondition: s is in the form <first-name> <last-name>
    with one or more blanks between the two names"""
    # Find the first name
    # Find the last name
    # Put them together with a comma
    return ' ' # Currently a stub
```

Testing last_name_first(n)

```
import name                  # The module we want to test
import introcs               # Includes the test procedures


# First test case
result = name.last_name_first('Walker White')
introcs.assert_equals('White, Walker', result)

# Second test case
result = name.last_name_first('Walker          White')
introcs.assert_equals('White, Walker', result)


print('Module name is working correctly')
```

Quits Python if not equal

Message will print out only if no errors.

Using Test Procedures

- In the real world, we have a lot of test cases
  - I wrote 20000+ test cases for a C++ game library
  - You need a way to cleanly organize them
- **Idea**: Put test cases inside another procedure
  - Each function tested gets its own procedure
  - Procedure has test cases for that function
  - Also some print statements (to verify tests work)
- Turn tests on/off by calling the test procedure

Test Procedure

```
def test_last_name_first():
    """Test procedure for last_name_first(n)"""
    print('Testing function last_name_first')
    result = name.last_name_first('Walker White')
    introcs.assert_equals('White, Walker', result)
    result = name.last_name_first('Walker             White')
    introcs.assert_equals('White, Walker', result)

# Execution of the testing code
test_last_name_first()
print('Module name is working correctly')
```

Test Procedure

```
def test_last_name_first():
    """Test procedure for last_name_first(n)"""
    print('Testing function last_name_first')
    result = name.last_name_first('Walker White')
    introcs.assert_equals('White, Walker', result)
    result = name.last_name_first('Walker            White')
    introcs.assert_equals('White, Walker', result)

# Execution of the testing code
test_last_name_first()
print('Module name is working correctly')
```

No tests happen if you forget this