Software Engineering (Part 3)

Robert M. Dondero, Ph.D.
Princeton University

Objectives

- You will learn/review these software engineering topics:
  - Requirements analysis
  - Design
  - Implementation
  - Debugging
  - Testing
  - Evaluation
  - Maintenance
  - Process models

Stages of SW dev

How to order the stages

Objectives

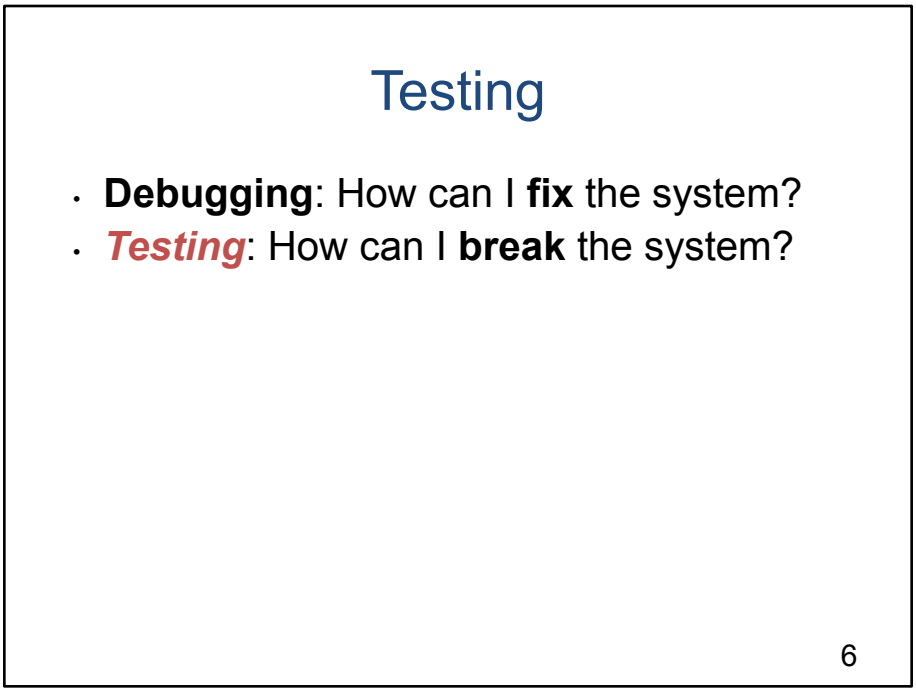
Software Engineering lecture slide decks:

Part 1	Requirements analysis Design (general)
Part 2	Design (object-oriented) Implementation Debugging
Part 3	**Testing** **Evaluation**
Part 4	Maintenance Process models

You're reasonably sure that your code is bug-free. What's next?

Agenda

- Requirements analysis
- Design
- Implementation
- Debugging
- **Testing**
- Evaluation
- Maintenance
- Process models



Testing

Debugging: How can I **fix** the system?

Testing: How can I **break** the system?

A good programmer spends at least as much time composing test code as he/she spends composing production code

Testing: Taxonomy

- Testing taxonomy
  - Internal testing
  - External testing
    - White box
    - Black box
    - Field
  - General strategies

7

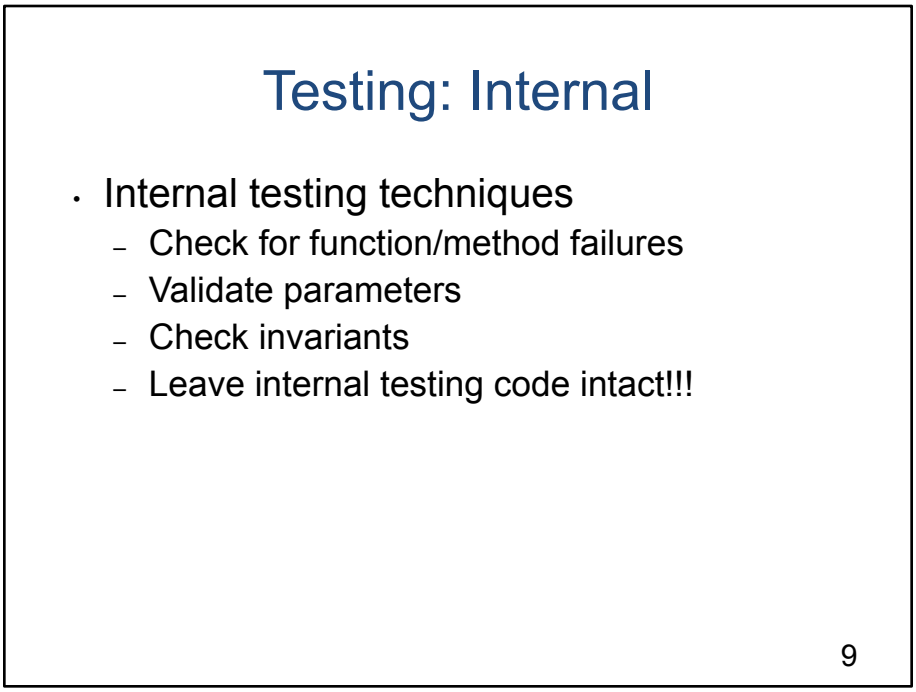
Testing: Taxonomy

[see slide]

That same taxonomy is referenced by every assignment specification
I hope you've had it in mind throughout the entire semester

Testing: Internal

- ***Internal testing***
  - Designing your code to test itself
  - Done by *programmers*



Testing: Internal

Check for function/method failures

C: Check function return values

Java & Python & JavaScript: Consider exceptions thrown by functions/methods

Validate parameters

At leading edge of each function/method, make sure parameter values are valid

Practice “defensive” programming

Check invariants

At leading (and trailing?) edge of each function/method, check aspects of data structures that should not vary

Leave internal testing code intact!!!

Testing: Internal

- **Problem**:
  - You want internal testing code **present** during development
  - Internal testing code can be expensive
  - You might want internal testing code **absent** from production code

Testing: Internal

- **Solution**:
  - ***Asserts***
    - Allow you to **enable** testing code during development
    - Allow you easily to **disable** testing code in production product

Testing: Internal

C: `assert` macro

```
assert(count >= 0);
```

Essentially same as:

```
if (count < 0)
{  fprintf(stderr,
      "assertion failed: (count >= 0),");
   fprintf(stderr,
      "function XXX, file YYY, line ZZZ.");
   exit(134);
}
```

Asserts are **enabled** by default; to **disable** asserts:

```
gcc -D NDEBUG somefile.c
```

Testing: Internal

Python: `assert` statement

```
assert count >= 0, 'count is < 0'
```

Essentially same as:

```
if count < 0:
    raise AssertionError('count is < 0')
```

Asserts are **enabled** by default; to **disable** asserts:

```
python -O somefile.py
```

Testing: Internal

Java: `assert` statement (since JDK 1.4)

```
assert count >= 0 : "count is < 0";
```

Essentially same as:

```
if (count < 0)
   throw new AssertionError("count is < 0");
```

Asserts are **disabled** by default; to **enable** asserts:

```
java -ea SomeFile.java
```

Testing: Internal

JavaScript (browsers): `assert` function

```
console.assert(count >= 0, 'count is < 0';
```

Essentially same as:

```
if (count < 0)
   console.log('count is < 0');
```

Cannot be disabled???

Testing: Internal

JavaScript (Node.js): `assert` function

```
const assert = require('assert');
…
assert(count >= 0);
```

Essentially same as:

```
if (count < 0)
   throw new Error(
      'The expression evaluated to a falsy value');
```

Cannot be disabled!

Testing: Internal

- Assert controversy: enable or disable asserts in production code?
  - If asserts are expensive, then disable them
  - If asserts are inexpensive, then ???

Testing: External

- ***External testing***
  - Designing code or data to test your code

Testing: External: White Box

- ***White box*** **external testing**
  - External testing with knowledge of structure of tested code
  - Done by **programmers**
    - Pro: Know the code => can test all statements/paths/boundaries
    - Con: Know the code => biased by code design

Testing: External: White Box

- White box external testing techniques
  - ***Statement testing***
    - Testing to make sure each **statement** is executed at least once
  - ***Path testing***
    - Testing to make sure each **logical path** is followed at least once
    - Combinatorial explosion => path test small code units

Testing: External: White Box

- White box external testing techniques
  - ***Boundary (corner case) testing***
    - Testing with input values at, just below, and just above limits of input domain
    - Testing with input values causing output values to be at, just below, and just above the limits of the output domain

Glossary of Computerized System and Software Development Terminology

Testing: External: White Box

- Tool support for statement testing
  - **Python**: ***coverage***
    - See Assignments 1-5
    - Another example...

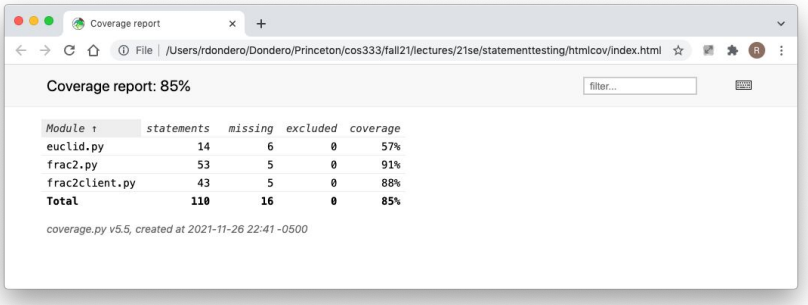
Testing: External: White Box

- See **statementtesting/euclid.py**
- See **statementtesting/frac2.py**
- See **statementtesting/frac2client.py**
- See **statementtesting/buildandrun**
  - Bash shell script (Mac and Linux)
- See **statementtesting/buildandrun.bat**
  - DOS script (MS Windows)

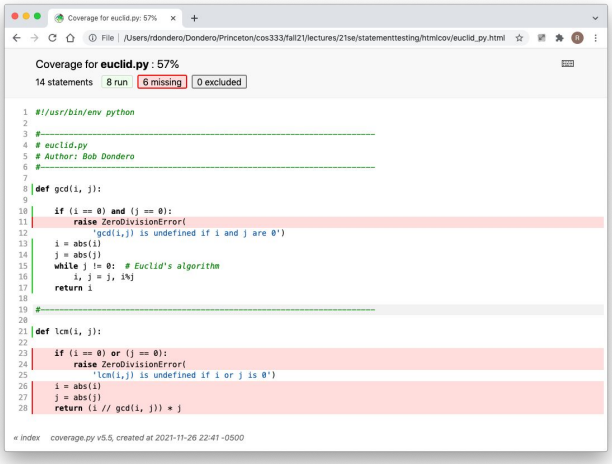
Testing: External: White Box

```
$ ./buildandrun

# Create file .coverage
python -m coverage run frac2client.py
Numerator 1: 1
Denominator 1: 2
Numerator 2: 3
Denominator 2: 4
frac1: 1/2
frac2: 3/4
frac1 hashcode: -3550055125485641917
frac1 does not equal frac2
frac1 is less than frac2
frac1 is less than or equal to frac2
-frac1: -1/2
frac1 + frac2: 5/4
frac1 - frac2: -1/4
frac1 * frac2: 3/8
frac1 / frac2: 2/3

# Create directory htmlcov
python -m coverage html

# View the results, htmlcov/index.html, using a browser
$
```

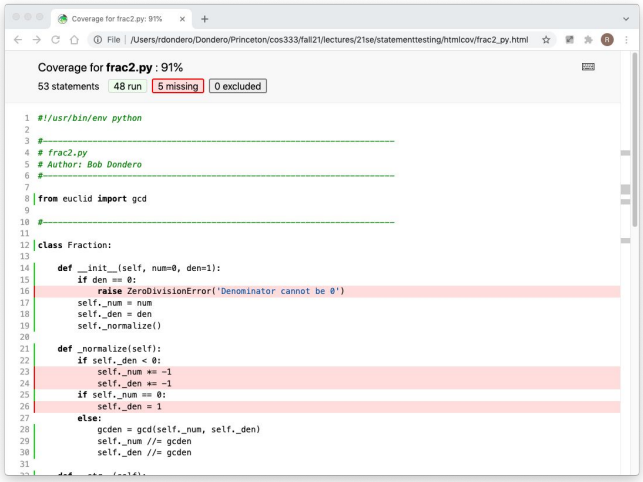
Testing: External: White Box



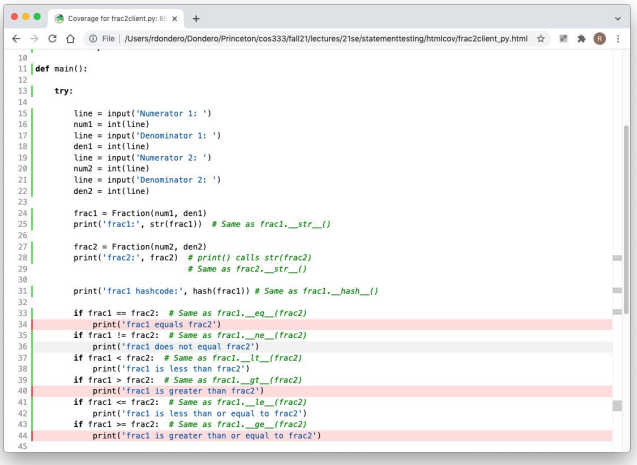
Testing: External: White Box



Testing: External: White Box



Testing: External: White Box



Testing: External: White Box

Language	Statement Testing Tool
Python	*coverage*
Java	*JaCoCo* *
C	*gcov* *
JavaScript (browser)	*istanbul*
JavaScript (Node.js)	*istanbul*

* See me if you want an example

Testing: External: Black Box

- ***Black box*** **external testing**
  - External testing without knowledge of structure of tested code
  - Done by *quality assurance (QA) engineers*
    - Pro: Do not know the code => unbiased by code design
    - Con: Do not know the code => unlikely to test all statements/paths/boundaries

Testing: External: Black Box

- Black box external testing techniques
  - ***Use case testing***
    - Testing driven by use cases developed during design
  - ***Stress testing***
    - Test with a large quantity of data
    - Testing with a large variety of (random?) data

Testing: External: Black Box

- ***Field*** **external testing**
  - External testing, often in unexpected ways!
  - Done by *customers*
    - Pros: Often use code in unexpected ways
    - Con: Often don't like participating!

Testing: General Strategies

- **General testing strategies**
  - Automate the testing
    - Create **scripts** to test your **programs**
    - Create software **clients** to test your **modules**
    - Compare implementations when possible

Testing: General Strategies

- Tool support for automating testing
  - **Python**: ***PyUnit***
    - Example...

Testing: General Strategies

- See **testautomationgood/euclid.py**
  - From Python Language (Part 4) lecture
- See **testautomationgood/frac2.py**
  - From Python Language (Part 4) lecture
- See **testautomationgood/testfraction.py**
  - Instead of frac2client.py
  - Uses PyUnit
- See **testautomationgood/buildandrun**
  - Bash shell script (Mac and Linux)
- See **testautomationgood/buildandrun.bat**
  - DOS script (MS Windows)

Testing: General Strategies

```
$ ./buildandrun

# Run unit tests
python testfraction.py
.....
----------------------------------------------------------
--------------------
Ran 5 tests in 0.000s

OK
$
```

Testing: General Strategies

- See testautomationbad/euclid.py
  - Same as previous
- See **testautomationbad/frac2.py**
  - Contains a logic error
- See testautomationbad/testfraction.py
  - Same as previous
- See testautomationbad/buildandrun
  - Same as previous
- See testautomationbad/buildandrun.bat
  - DOS script (MS Windows)

Testing: General Strategies

```
$ ./buildandrun

# Run unit tests
python testfraction.py
..F..
======================================================================
FAIL: runTest (__main__.MulTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "testfraction.py", line 35, in runTest
    self.assertEqual(prod, expected, 'Incorrect product')
AssertionError: <frac2.Fraction object at 0x103be6940> !=
<frac2.Fraction object at 0x103be6f40> : Incorrect product

----------------------------------------------------------------------
Ran 5 tests in 0.001s

FAILED (failures=1)
$
```

Testing: General Strategies

Language	Test Automation Tool
Python	***PyUnit***
Python (PyQt5)	***pytestqt*** *
Java	***JUnit*** *
C	***CUnit*** *
JavaScript (browser)	***Mocha***
JavaScript (Node.js)	***Mocha***
Web apps	***Selenium*** *

* See me if you want an example

39

Testing: General Strategies

- General testing strategies (cont.)
  - Test incrementally
    - Use scaffolds and stubs
    - Do ***regression testing***
  - Let debugging drive testing
    - Reactive mode
    - Proactive mode: do ***fault injection***

Aside: COS 333 Project Testing

- In *Product Eval* document:
  - Internal testing?
  - White box external testing?
    - Statement testing?
    - Boundary testing?
  - Black box external testing?
    - Use case testing?
    - Stress testing?
  - Test automation?

You've tested your code to make sure it meets your expectations. What's next?

Agenda

- Requirements analysis
- Design
- Implementation
- Debugging
- Testing
- **Evaluation**
- Maintenance
- Process models

Evaluation

- **Testing**
  - Does the system meet your (the programmer's) expectations?
- ***Evaluation***
  - Does the system fulfill the needs of its users?

Evaluation

- Kinds of evaluation
  - By users
    - Actually, by software engineers in collaboration with users
  - By evaluation experts

Evaluation: Users

- Questionnaires
- **Interviews**
- Focus groups
- Direct observation

Recall requirements gathering techniques

Evaluation: Users

- Conducting interviews
  - Recruit a set of users
  - (If necessary) compose a short written intro to your system
  - Compose a written task sequence
    - Maybe abstracted from use cases developed during design
  - Then…

Evaluation: Users

- Conducting interviews (cont)
  - For each user:
    - (If necessary) give the user the short intro
    - Ask the user to read the short intro
    - Confirm that the user understands the short intro
    - Give the user the task sequence
    - For each task:
      - Ask the user to read the task
      - Confirm that the user understands the task
      - Ask the user to use your system to perform the task
      - Ask (force!!!) the user to talk aloud while performing the task
  - Take copious notes
  - Audio/video record?

Evaluation: Users

- Repeat for each kind of user

Evaluation: Experts

- **Who**: Jakob Nielsen
- **When**: 1990s
- **What**: Techniques that help experts to evaluate systems composed by others

Evaluation: Experts

- ***Heuristic Evaluation***
  - From Jakob Nielsen
  - For evaluating the **whole** system **generally**
  - Using these 10 heuristics…

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(1) Visibility of system status**
    - **The system should always keep users informed** about what is going on, through appropriate feedback within reasonable time.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(2) Match between system and the real world**
    - **The system should speak the user's language**, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(3) User control and freedom**
    - **Users often choose system functions by mistake and will need a clearly marked "emergency exit"** to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(4) Consistency and standards**
    - Users should not have to wonder whether different words, situations, or actions mean the same thing. **Follow platform conventions.**

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(5) Error prevention**
    - **Even better than good error messages is a careful design which prevents a problem from occurring** in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(6) Recognition rather than recall**
    - Minimize the user's memory load by **making objects, actions, and options visible**. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(7) Flexibility and efficiency of use**
    - Accelerators—unseen by the novice user—may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. **Allow users to tailor frequent actions.**

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(8) Aesthetic and minimalist design**
    - **Dialogues should not contain information which is irrelevant** or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(9) Help users recognize, diagnose, and recover from errors**
    - **Error messages should be expressed in plain language** (no codes), precisely indicate the problem, and constructively suggest a solution.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(10) Help and documentation**
    - Even though it is better if the system can be used without documentation, it may be necessary to **provide help and documentation**. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- For more info on heuristic evaluation:
  - Wikipedia article: https://en.wikipedia.org/wiki/Heuristic_evaluation
  - Helen Sharp, Jenny Preece, Yvonne Rogers. *Interaction Design: Beyond Human-Computer Interaction*.
  - Nielsen, Jakob. *Usability Engineering*.

Evaluation: Experts

- ***Cognitive Walkthrough***
  - From Cathleen Wharton, Jakob Nielsen
  - For evaluating **part** of the system in **detail**

Repeatedly:
Will the correct action be sufficiently evident to the user?
Will the user know what to do to achieve the task?
Will the user notice that the correct action is available?
Can users see the button or menu item that they should use for the next action?
Will the user associate and interpret the response from the action correctly?
Will users know from the feedback that they have made the correct or incorrect choice of action?

Yvonne Rogers, Helen Sharp, Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction (3rd Edition)*. Wiley, 2011.

Aside: COS 333 Project Eval

- In *Product Eval* document
  - **User eval (interviews)**
    - Required
    - Task list and detailed interview notes in document appendix
    - Summary in document body
  - **User eval (questionnaires/surveys)**
    - Optional, but commendable

Aside: COS 333 Project Eval

- In *Product Eval* document
  - **Expert evaluation (heuristic evaluation)**
    - Required
    - You play the role of the expert
  - **Expert evaluation (cognitive walkthrough)**
    - Optional, but commendable
    - You play the role of the expert
  - Discuss with your adviser

So the system is finished. Or is it?

Continued in
Software Engineering (Part 4)…