CISC 327
Software Quality Assurance

Lecture “review2”

Review for Mini-Exam #2

Announcements

- Mini-Exam #2 accommodations:
  - You should have received an email from Accommodation @ cs.queensu.ca at some point

Likely topics on mini-exam #2

- From Lectures 8–9:
  - Systematic testing
    - What makes a test method systematic?
    - For a given systematic test method:

      What is the system for choosing test cases?

      What is the completeness criterion?

Likely topics on mini-exam #2

- From Lectures 10–13:
  - Difference between black box and white box
  - Black box method: **Functionality coverage**
    - Requirements partitioning
  - Black box method: **Input coverage**
    - When is exhaustive input coverage practical?
  - Black box method: **Output coverage**
    - When is exhaustive output coverage practical?
  - **Handling multiple inputs and outputs**
  - **Assertions** and **class invariants**

Assertions, pre-/post-conditions, class invariants

- Method preconditions and class invariants **restrict** the possible inputs to test cases
- Class invariant: assertion that holds for all instance variables before and after every method call
- (**not** specific to object-oriented languages, but terminology differs, e.g. *data structure invariants*)
- Postcondition should hold when method returns
  - ideally, checked at run time;
    if not, becomes part of unit test cases

Likely topics on mini-exam #2

- From Lectures 14–16:
  - White box testing: Code injection (in source code)
  - White box methods:
    - Statement coverage
    - Basic block coverage
    - Decision coverage
    - Condition coverage
    - Loop coverage
    - **Path coverage** ⇐ if you only have time to study one...
    - Data coverage

Likely topics on mini-exam #2

- From Lectures 17–19:
  - ~~Mutation testing~~ on mini-exam #3
  - ~~Regression testing~~ on mini-exam #3

But I mean *specifically*...

Likely kinds of questions

1. Is method ____ white box or black box?
2. For a given systematic test method, identify system and completeness criterion
3. For a requirements specification (system **or** unit level), **identify the inputs and outputs** and then
   a. write requirements tests
   b. write input coverage tests
   c. write output coverage tests
4. For a program, identify paths and write covering path tests
   a. NOTE: paths could be impossible! (do example)

Paths

```
1 if (x < 0)
2   y -= 1;

3 if (x < 0)
4   y -= 3;
  else
5   y += 5;

6 return y;
```

Paths

```
1 if (x < 0)
2   y -= 1;

3 if (x < 0)
4   y -= 3;
  else
5   y += 5;

6 return y;
```

P1: 1, 2, 3, 4, 6

P2: 1, 2, 3, 5, 6

P3: 1, 3, 4, 6

P4: 1, 3, 5, 6

Identifying Paths

```
1 if (x < 0)
2   y -= 1;

3 if (x < 0)
4   y -= 3;
  else
5   y += 5;

6 return y;
```

P1: 1, 2, 3, 4, 6

P2: 1, 2, 3, 5, 6

P3: 1, 3, 4, 6

P4: 1, 3, 5, 6

This step is only based on the **flow graph**; doesn’t care what the conditions/statements are

Path coverage: find x

```
1 if (x < 0)
2   y -= 1;

3 if (x < 0)
4   y -= 3;
  else
5   y += 5;

6 return y;
```

P1: 1, 2, 3, 4, 6 • __________

P2: 1, 2, 3, 5, 6 • __________

P3: 1, 3, 4, 6 • __________

P4: 1, 3, 5, 6 • __________

To come up with **covering inputs,**
we definitely care what the code is

Path coverage: find x

```
1 if (x < 0)
2   y -= 1;

3 if (x < 0)
4   y -= 3;
  else
5   y += 5;

6 return y;
```

P1: 1, 2, 3, 4, 6
- −1

P2: 1, 2, 3, 5, 6
- impossible

P3: 1, 3, 4, 6
- impossible

P4: 1, 3, 5, 6
- 0

To come up with **covering inputs,** we definitely care what the code is

Path coverage: find x

```
1 if (x < 0)
2   y -= 1;

3 if (x < 0)
4   y -= 3;
  else
5   y += 5;

6 return y;
```

- P1: 1, 2, 3, 4, 6 • (any x < 0)
- P2: 1, 2, 3, 5, 6 • impossible
- P3: 1, 3, 4, 6 • impossible
- P4: 1, 3, 5, 6 • (any x ≥ 0)

To come up with **covering inputs,** we definitely care what the code is

To save you time…

- In the game *Counterfeit Monkey*, the player solves puzzles by using *lexical manipulation devices* to transform physical objects.
  For example, a device called a *letter remover* can be used on a *cart* to make a *car* (after setting the machine's dial to 't').
- Your employer Bogosys has bought the intellectual property rights to *Counterfeit Monkey* and is re-implementing the game from scratch.

To save you time...

The class Remover has one instance variable, the character `dial`.
The class invariant is
`('a' <= dial) && (dial <= 'z').`
The method Remover.apply will take one parameter (a string of lowercase letters) and return a string of lowercase letters, removing the letter that corresponds to the dial setting.
If that letter does not occur, then return the string unchanged.

(This specification is intentionally ambiguous and will be clarified partway through the question...)

Bonus question

- involves the CTO of Bogosys, who wrote programs to solve several undecidable problems