CS 563:
Software Maintenance and Evolution

Evaluating Test Suite Efficacy

Oregon State University, Spring 2024

Today's Plan

- Learn about how to measure quality of test-suite
- In-class exercise: software testing

Announcements

- Homework-2 due tonight (with automatic 1-day extension)
- Except Homework-1, all assignment grades are posted
- Next class: final project discussion and feedback on whatever is done by then. Make sure to update your paper!
- Next Wednesday's class: Final project presentations.
    - Remember the feedback you received for paper presentation and reviews while creating your slides and presentation for your final project report.
    - Also remember to refer the lecture and resources on technical writing and presentation.

Measuring Test Suite Quality

- How do we know that our test suite is good?
  - **Too few tests:** may miss catching bugs
  - **Too many tests:** costly to run, redundant, hard to maintain

Measuring Test Suite Quality

- How do we know that our test suite is good?
  - **Too few tests:** may miss catching bugs
  - **Too many tests:** costly to run, redundant, hard to maintain
- Two approaches:
  - **Code coverage** metrics
  - **Mutation analysis** (or mutation testing)

Code Coverage

- Metric to identify the extent to which program's code is tested by a given test suite
- Computed as percentage of some aspect of the program executed by the tests
- 100% coverage is rare in practice e.g., its impossible to cover unreachable/dead code
  - Often required for safety-critical applications

Types of Code Coverage

- **Function coverage:** which **functions** were invoked?
- **Statement coverage:** which **statements** were executed?
- **Branch coverage:** which **branches** were taken at conditionals?
- **Others:** line coverage, condition coverage, basic block coverage, path coverage, etc.

Quiz(1/2): Code Coverage Metrics

statement is executed

conditional statement is partially executed

statement is not executed

- Test Suite: *assert(foo(1,0)==0)*

```
int foo (int x, int y) {
  int z = 0;
  if (x <= y){
    z = x;
  }else{
    z = y;
  }
  return z;
```

Quiz(1/2): Code Coverage Metrics

statement is executed

conditional statement is partially executed

statement is not executed

- Test Suite: *assert(foo(1,0)==0)*
- Statement coverage = %
- Branch coverage = %

```
int foo (int x, int y) {
  int z;
  if (x <= y){
    z = x;
  }else{
    z = y;
  }
  return z;
```

Quiz(1/2): Code Coverage Metrics

statement is executed

conditional statement is partially executed

statement is not executed

```
int foo (int x, int y) {
  int z;
  if (x <= y){
    z = x;
  }else{
    z = y;
  }
  return z;
```

- Test Suite: *assert(foo(1,0)==0)*
- Statement coverage = (4/5)x100 = 80 %
- Branch coverage = (1/2)x100 = 50 %

Quiz(2/2): Code Coverage Metrics

statement is executed

conditional statement is partially executed

statement is not executed

```
int foo (int x, int y) {
    int z;
    if (x <= y){
      z = x;
    }else{
      z = y;
    }
    return z;
```

- Test Suite: *assert(foo(1,0)==0)*
- Statement coverage = 80 %
- Branch coverage = 50 %
- What values of *x, y,* and *z* in Assert(*foo(x,y)==z)* will give 100% for both coverages?

Quiz(2/2): Code Coverage Metrics

statement is executed

conditional statement is partially executed

statement is not executed

```
int foo (int x, int y) {
  int z;
  if (x <= y){
    z = x;
  }else{
    z = y;
  }
  return z;
```

- Test Suite: *assert(foo(1,0)==0)*
- Statement coverage = 80 %
- Branch coverage = 50 %
- What values of *x*, *y*, and *z* in assert(*foo(x,y)==z)* will give 100% for both coverages?
  (any x, y such that x <=y)
  assert(foo(1, 2) == 1)

Mutation Analysis

- Based on “competent programmer assumption”
  - The program is close to correct to start with
- Key Idea: Test variations (mutants) of the program
  - E.g., replace > with <, == with !=, + with –, x with x+1 or x–1 etc. and check if your test suite can *kill the mutants*
- A good test suite would fail at least one test detecting mutants
- If not, add tests that can kill undetected mutants
- What if mutated program is *semantically equivalent* to original program?

Real problem in practice that requires manual intervention to formally prove the equivalence

Quiz(1/2): Mutation Analysis

	Test 1: assert *foo(0,1)* == 0	Test 2: assert *foo(0,0)* == 0
Mutant 1: x <= y ➡ x > y	Pass/Fail?	Pass/Fail?
Mutant 2: x <= y ➡ x != y	Pass/Fail?	Pass/Fail?

```
int foo (int x, int y) {
  int z;
  if (x <= y){
    z = x;
  }else{
    z = y;
  }
  return z;
```

Is the test suite adequate with respect to both the mutants? Yes or No

Quiz(1/2): Mutation Analysis

	Test 1: assert *foo(0,1)* == 0	Test 2: assert *foo(0,0)* == 0
Mutant 1: x <= y ➡ x > y	Fail	Pass
Mutant 2: x <= y ➡ x != y		

```
int foo (int x, int y) {
  int z;
  if (x <= y){
    z = x;
  }else{
    z = y;
  }
  return z;
```

Is the test suite adequate with respect to both the mutants? Yes or No

Quiz(1/2): Mutation Analysis

	Test 1: assert *foo(0,1)* == 0	Test 2: assert *foo(0,0)* == 0
Mutant 1: x <= y ➡ x > y	Fail	Pass
Mutant 2: x <= y ➡ x != y	Pass	Pass

```
int foo (int x, int y) {
  int z;
  if (x <= y){
    z = x;
  }else{
    z = y;
  }
  return z;
```

Is the test suite adequate with respect to both the mutants? Yes or **No**

Quiz(2/2): Mutation Analysis

	Test 1: assert *foo(0,1)* == 0	Test 2: assert *foo(0,0)* == 0
Mutant 1: x <= y ➡ x > y	Fail	Pass
Mutant 2: x <= y ➡ x != y	Pass	Pass

```
int foo (int x, int y) {
  int z;
  if (x <= y){
    z = x;
  }else{
    z = y;
  }
  return z;
```

Can you add a test case such that test suite becomes adequate with respect to both the mutants?

Quiz(2/2): Mutation Analysis

	Test 1: assert *foo(0,1)* == 0	Test 2: assert *foo(0,0)* == 0
Mutant 1: x <= y ➡ x > y	Fail	Pass
Mutant 2: x <= y ➡ x != y	Pass	Pass

```
int foo (int x, int y) {
  int z;
  if (x <= y){
    z = x;
  }else{
    z = y;
  }
  return z;
```

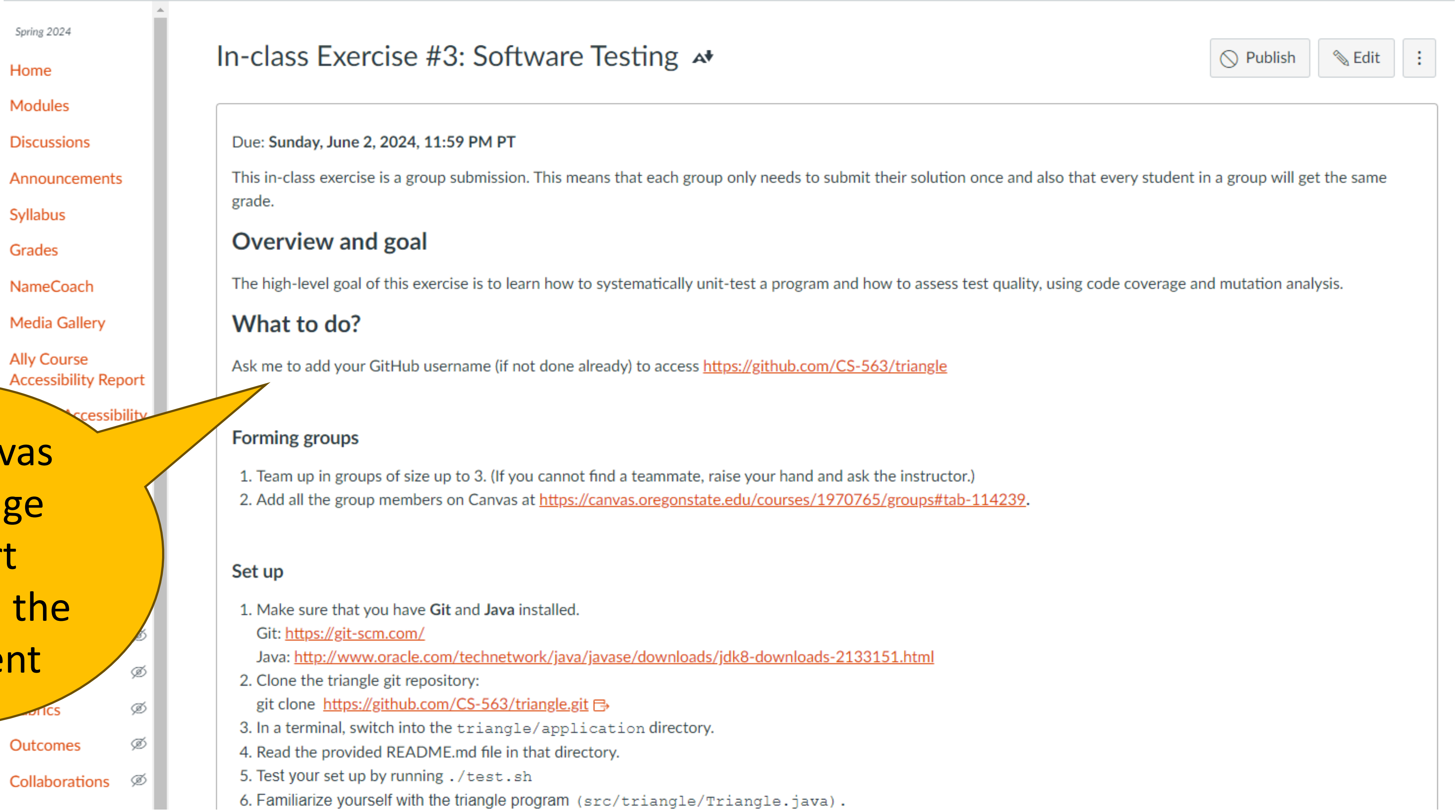
Can you add a test case such that test suite becomes adequate with respect to both the mutants?

Test 3: assert *foo(1,0)* == 0

Short 2 min Survey

rb.gy/c7gj8c

In-Class Exercise#3 Software Testing



Spring 2024

- Home
- Modules
- Discussions
- Announcements
- Syllabus
- Grades
- NameCoach
- Media Gallery
- Ally Course Accessibility Report
- Outcomes
- Collaborations

In-class Exercise #3: Software Testing

Publish | Edit

Due: **Sunday, June 2, 2024, 11:59 PM PT**

This in-class exercise is a group submission. This means that each group only needs to submit their solution once and also that every student in a group will get the same grade.

Overview and goal

The high-level goal of this exercise is to learn how to systematically unit-test a program and how to assess test quality, using code coverage and mutation analysis.

What to do?

Ask me to add your GitHub username (if not done already) to access https://github.com/CS-563/triangle

Forming groups

1. Team up in groups of size up to 3. (If you cannot find a teammate, raise your hand and ask the instructor.)
2. Add all the group members on Canvas at https://canvas.oregonstate.edu/courses/1970765/groups#tab-114239.

Set up

1. Make sure that you have **Git** and **Java** installed.
   Git: https://git-scm.com/
   Java: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
2. Clone the triangle git repository:
   git clone https://github.com/CS-563/triangle.git
3. In a terminal, switch into the `triangle/application` directory.
4. Read the provided README.md file in that directory.
5. Test your set up by running `./test.sh`
6. Familiarize yourself with the triangle program `(src/triangle/Triangle.java)`.

Go to Canvas course page and start working on the Assignment