CS5351 Software Engineering
2021/2022 Semester A
Quiz
120 minutes
18 Oct 2021

Fill this section.

Student Name: ______________________

Student ID: _________________________

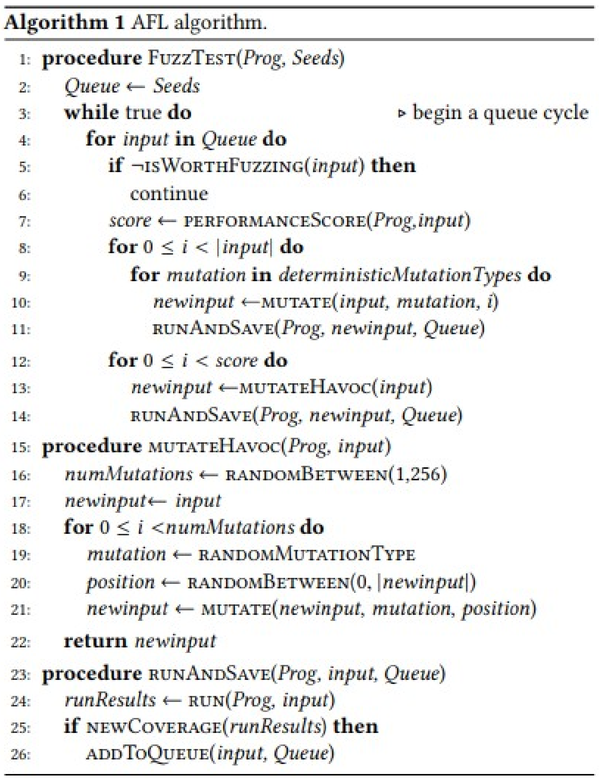
Team Number: ______________________

Seat Number: _______________________

Answer all four questions.

Question	**1**	**2**	**3**	**4**	**Total**
Max Mark	**35**	**25**	**20**	**20**	**100**

Question 1. AFL is a fuzzer to detect security vulnerabilities. It has been described as a **grey-box, feedback-oriented, and mutation-based** technique. Its algorithm is presented in Figure 1. The in-class exercise for AFL is reproduced as Appendix A.



Algorithm 1 AFL algorithm.

```
procedure FuzzTest(Prog, Seeds)
  Queue ← Seeds
  while true do                                    ▹ begin a queue cycle
    for input in Queue do
      if ¬isWorthFuzzing(input) then
        continue
      score ← performanceScore(Prog,input)
      for 0 ≤ i < |input| do
        for mutation in deterministicMutationTypes do
          newinput ←mutate(input, mutation, i)
          runAndSave(Prog, newinput, Queue)
      for 0 ≤ i < score do
        newinput ←mutateHavoc(input)
        runAndSave(Prog, newinput, Queue)
procedure mutateHavoc(Prog, input)
  numMutations ← randomBetween(1,256)
  newinput← input
  for 0 ≤ i <numMutations do
    mutation ← randomMutationType
    position ← randomBetween(0, |newinput|)
    newinput ← mutate(newinput, mutation, position)
  return newinput
procedure runAndSave(Prog, input, Queue)
  runResults ← run(Prog, input)
  if newCoverage(runResults) then
    addToQueue(input, Queue)
```

(a) Use the example in Appendix A to explain the following:

It is a grey-box technique:

It is a grey-box technique: the fuzzer begins with an input seed 13 and measures the branch coverage in g(·).

It is a feedback-oriented technique:

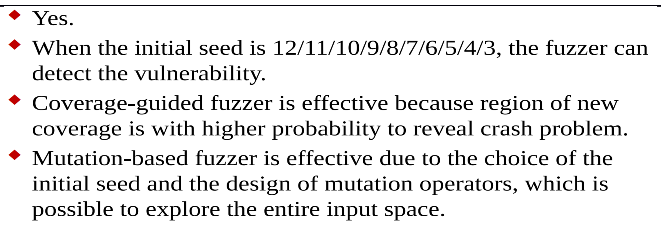
It is a feedback-oriented technique: the fuzzer determines a new test seed (e.g., 11) if it covers new branch compared with the historical coverage. (seed 11 covers B1 and B3)

It is a mutation-based technique:

It is a mutation-based technique: the input to the function under test is generated from the seed. (e.g., seed 12 is generated via subtracting 13 by 1)

(b) In the example shown in Appendix A, there are 13 possible values (0-12), and 2 of them will crash the program. The probability of **random selection** among these 13 test cases to crash the problem is 2 / 13 = 15.38%. However, by following the illustration of the algorithm, the fuzzer needs to generate the test case sequence ‹12, 6, 11, 6, 10, 5, 4, 2, 3〉, which has 9 test cases (or 8 test cases even if we count the two “6” as one) to detect the program crash. **The probability of using the fuzzer to detect program crashes is lower than random selection in this illustrated example.**

Is fuzzing effective in general (e.g., when the seed test case could be any integer in the range [0, 13]? Why?)



- Yes.
- When the initial seed is 12/11/10/9/8/7/6/5/4/3, the fuzzer can detect the vulnerability.
- Coverage-guided fuzzer is effective because region of new coverage is with higher probability to reveal crash problem.
- Mutation-based fuzzer is effective due to the choice of the initial seed and the design of mutation operators, which is possible to explore the entire input space.

(c) There are two mutation operators O1 and O2 in the algorithm. Is the order of applying these two operators important to the effectiveness of the fuzzer? (Effectiveness is defined as the number of test cases to detect the first occurrence of the program crash). You should illustrate your solution with the given seed sequence set ‹13〉.

Explanation

Yes. In this example, the buggy test cases lie in the middle part of the input space such that applying O2(dividing by 2) is faster to approach to the buggy cases.

Illustration

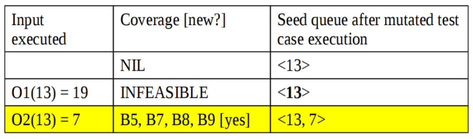
When applying O2 ahead of O1, the executed test cases are <13, 6, 12, 3>, which is shorter than the one from O1-O2 order.

(d) Design a new set of mutation operators (instead of using O1 and O2) and plug it into the algorithm so that it can be more effective than random selection of test cases. (Note that you may have two or more mutation operators)

New operator O1:

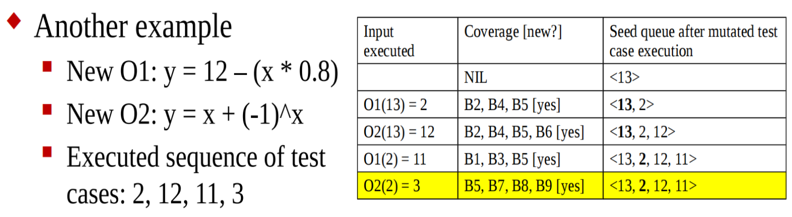
$$y = \lfloor {}^{3}/_{2}\, x \rfloor$$

$$y = \lceil {}^{1}/_{2}\, x \rceil$$



Input executed	Coverage [new?]	Seed queue after mutated test case execution
	NIL	<13>
O1(13) = 19	INFEASIBLE	<**13**>
O2(13) = 7	B5, B7, B8, B9 [yes]	<13, 7>

New operator O2:

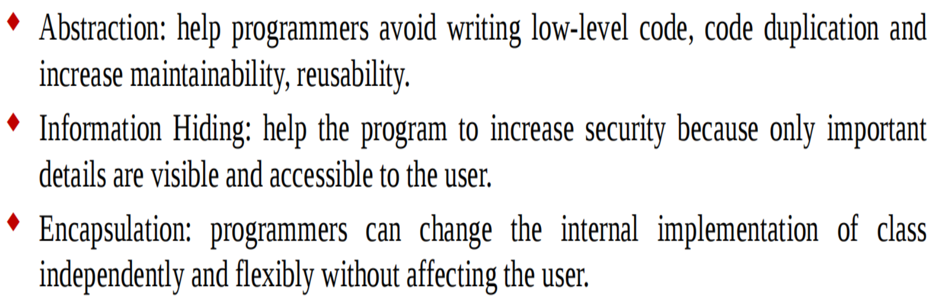


- Another example
  - New O1: y = 12 – (x * 0.8)
  - New O2: y = x + (-1)^x
  - Executed sequence of test cases: 2, 12, 11, 3

Input executed	Coverage [new?]	Seed queue after mutated test case execution
	NIL	<13>
O1(13) = 2	B2, B4, B5 [yes]	<**13**, 2>
O2(13) = 12	B2, B4, B5, B6 [yes]	<**13**, 2, 12>
O1(2) = 11	B1, B3, B5 [yes]	<13, **2**, 12, 11>
O2(2) = 3	B5, B7, B8, B9 [yes]	<13, **2**, 12, 11>

Question 2. (Fundamental Principles for Design)

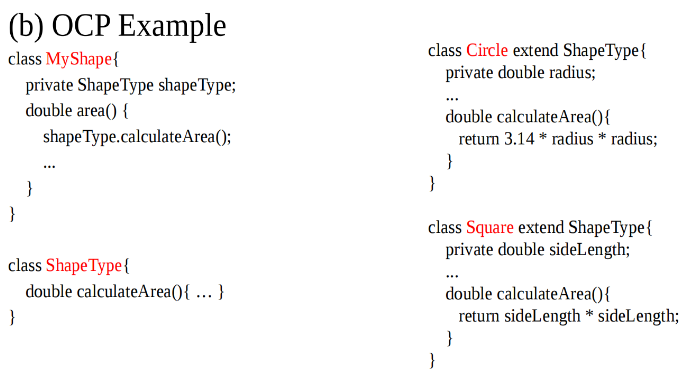
(a) Are there any benefits if we care about some fundamental design principles like abstraction, information hiding, and encapsulation in organizing the code as well as reducing the amount of code smells in it? List out the benefits.



- Abstraction: help programmers avoid writing low-level code, code duplication and increase maintainability, reusability.
- Information Hiding: help the program to increase security because only important details are visible and accessible to the user.
- Encapsulation: programmers can change the internal implementation of class independently and flexibly without affecting the user.

(b) Sketch your original exemplified code of Open-Closed Principle (OCP) and explain how to modify your exemplified code so that OCP is violated. **Your example is not allowed to be <u>taken or modified</u> from any source, including the teaching materials.**

OCP example



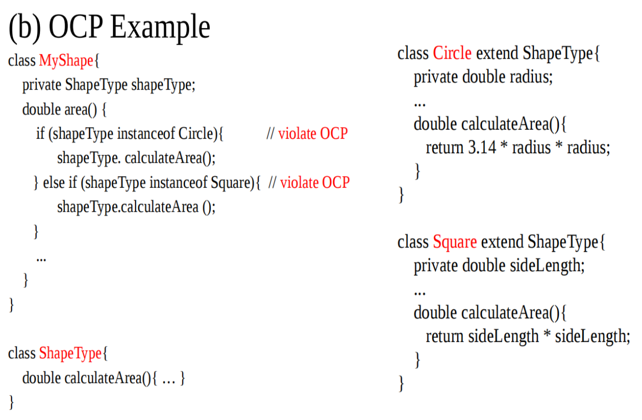
(b) OCP Example

```
class MyShape{
   private ShapeType shapeType;
   double area() {
      shapeType.calculateArea();
      ...
   }
}

class ShapeType{
   double calculateArea(){ … }
}
```

```
class Circle extend ShapeType{
   private double radius;
   ...
   double calculateArea(){
     return 3.14 * radius * radius;
   }
}

class Square extend ShapeType{
   private double sideLength;
   ...
   double calculateArea(){
     return sideLength * sideLength;
   }
}
```



(b) OCP Example

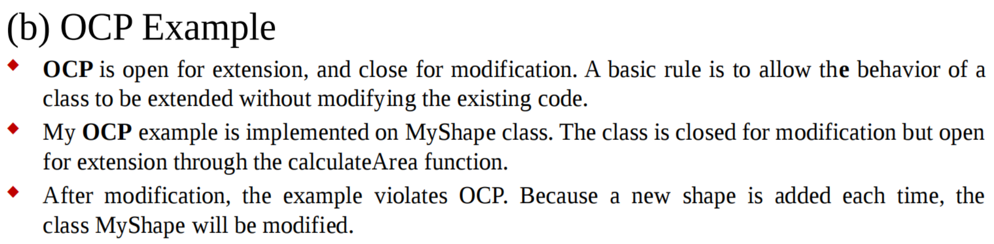
```
class MyShape{
   private ShapeType shapeType;
   double area() {
      if (shapeType instanceof Circle){         // violate OCP
         shapeType. calculateArea();
      } else if (shapeType instanceof Square){  // violate OCP
         shapeType.calculateArea ();
      }
      ...
   }
}

class ShapeType{
   double calculateArea(){ … }
}
```

```
class Circle extend ShapeType{
   private double radius;
   ...
   double calculateArea(){
     return 3.14 * radius * radius;
   }
}

class Square extend ShapeType{
   private double sideLength;
   ...
   double calculateArea(){
     return sideLength * sideLength;
   }
}
```

Explanation



(b) OCP Example

- **OCP** is open for extension, and close for modification. A basic rule is to allow th**e** behavior of a class to be extended without modifying the existing code.
- My **OCP** example is implemented on MyShape class. The class is closed for modification but open for extension through the calculateArea function.
- After modification, the example violates OCP. Because a new shape is added each time, the class MyShape will be modified.

(c) Code Refactoring has several factors that affect developers to conduct it or not. Study the four plots in Appendix C about a survey on Windows development teams in Microsoft. Explain why you will or will not conduct code refactoring to remove the violation of OCP explained in your answer for Question 2(b). List out your reasons.

(c) Reasons

- Figure 1: For all the refactoring types except for "Rename", most of time half of programmers want to do refactoring manually.
- Figure 2: The process of code refactoring is not an easy one since it may result in high number of regression bugs and build breaks.
- Figure 3: Code smells are not the driving factors for refactoring. Figure 3 shows improved readability of code is the highest benefit of refactoring.
- Figure 4: Readability is the highest reason given by developers for initiating refactoring. This does not support the notion of code smells being the driving factors for refactoring.

I will conduct code refactoring to remove the violation of OCP because it will improve readability of code and will not involve significant risks.

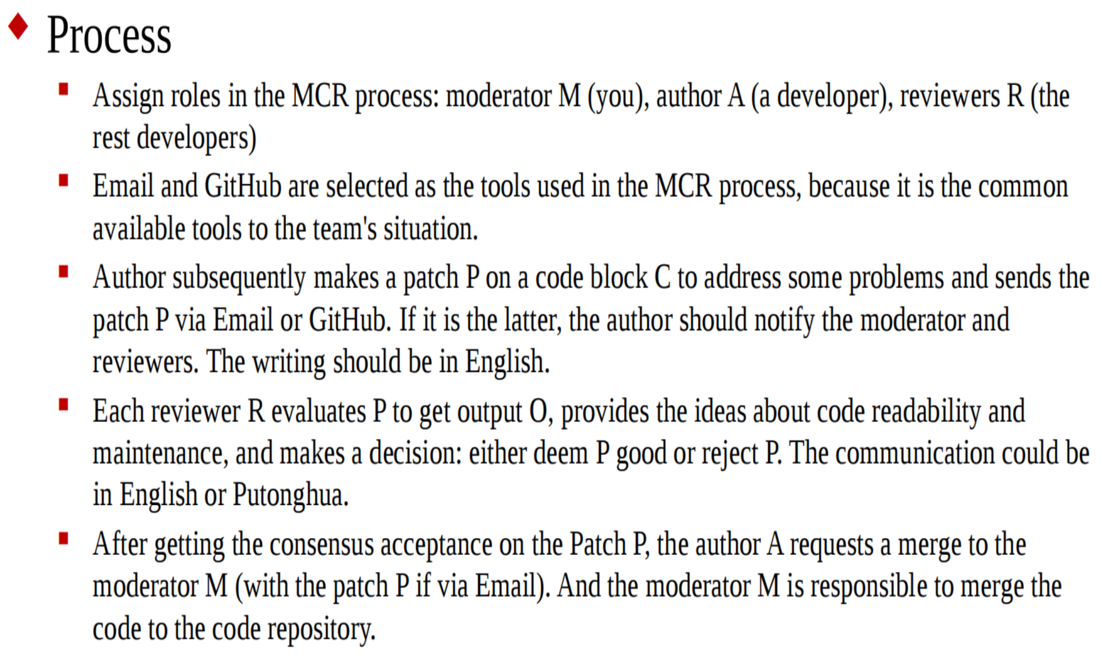
Question 3. **Modern Code Review** is an activity for informal, asynchronous, and tool-based code review of code change. Study the following scenario and answer the questions below.

In a company, a manager heard that modern code review is good. The manager asks you and three other colleagues to form a team to use modern code review for code development in the next two months in a project. You are assigned to lead this trial run of modern code review. After some evaluations of the company's infrastructure and practices, you observe the following:

- The team has two developers in Shanghai, one developer in Tokyo, and you work in Hong Kong. Only one developer in Shanghai and you are senior developers, and the other two are newcomers to the company. The mother tongues for the Shanghai, Tokyo, and Hong Kong developers are Putonghua with a strong Shanghai accent, Japanese, and Cantonese. No one is strong in English, but everyone can write and speak professional English. Communications in English among team members sometimes lead to misunderstandings and may require a few rounds of clarifications.
- The company has outsourced its email system to Microsoft by using Office365.
- For instant messaging tools, the Shanghai developer can only use QQ, the Tokyo developer prefers using Line, and you prefer using Telegram. The Tokyo developer can use Telegram, and you can use all three kinds of instant-messaging tools.
- There is a Git-based code repository like GitHub but private to the company.

State your customized modern code review process for the project to be used in the next two months. Explain how your customization can help to improve the development productivity over the original one. State any assumption explicitly in your answer if necessary.

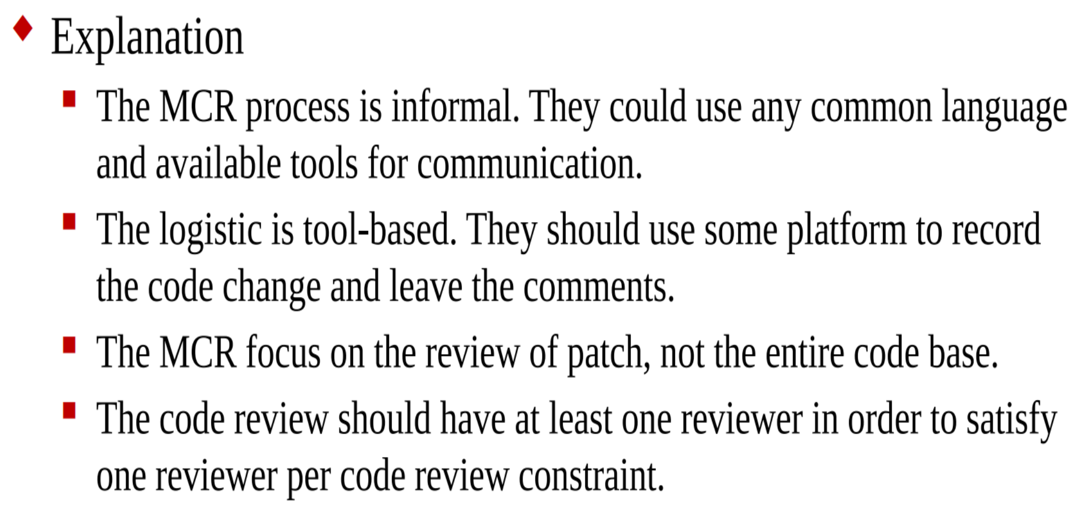
Customized modern code review process:



Process

- Assign roles in the MCR process: moderator M (you), author A (a developer), reviewers R (the rest developers)
- Email and GitHub are selected as the tools used in the MCR process, because it is the common available tools to the team's situation.
- Author subsequently makes a patch P on a code block C to address some problems and sends the patch P via Email or GitHub. If it is the latter, the author should notify the moderator and reviewers. The writing should be in English.
- Each reviewer R evaluates P to get output O, provides the ideas about code readability and maintenance, and makes a decision: either deem P good or reject P. The communication could be in English or Putonghua.
- After getting the consensus acceptance on the Patch P, the author A requests a merge to the moderator M (with the patch P if via Email). And the moderator M is responsible to merge the code to the code repository.

Explanation:



- Explanation
  - The MCR process is informal. They could use any common language and available tools for communication.
  - The logistic is tool-based. They should use some platform to record the code change and leave the comments.
  - The MCR focus on the review of patch, not the entire code base.
  - The code review should have at least one reviewer in order to satisfy one reviewer per code review constraint.

Question 4: Software Process

Your manager hears that Scrum is good in project management, and it can make the delivery of products in a more predictable manner. It also eases the life of software engineers. The manager requests you to use Scrum with the following constraints he sets where you will technically lead the project. He, however, has no knowledge about Scrum or other Agile development methodologies.

- The project is divided into three releases so that the final product can be delivered to the customers incrementally.
  - Release 1: a barebone system supporting the basic use cases
  - Release 2: an enhanced system covering all high-value and basic use cases
  - Release 3: a complete system with all use cases
- The next release should be delivered to the customer two months after the current release. The deadline for Release 1 is two months later (18 Dec 2021).
- The manager tells you to set each release as a single sprint.
- The manager will monitor the burndown chart against the above plan. If the project is delayed, the manager will decide which use cases to be removed from the corresponding release.
- To present a professional image to customers, he requests you not involving customers in the development process except collecting user requirements and conducting user acceptance testing and deployment.

Evaluate the following aspects of your manager's requirements against the Scrum methodology and explain whether you will recommend your manager to make changes.

Customer involvement:

(a) Customer involvement:

- The manager's requirement of customer involvement is improper.
- The manager does not want to involve customers too much in the development process, which violates the rule of Scrum. We should know that the requirements cannot be fully understood or defined at one time. If we don't communicate with customers continuously in the development process, the product may not be accepted by the customer finally.
- Hence, in each sprint, the developer group should be together with customers to discuss the goal of the current Sprint, prioritize functions to be completed and divided into detailed tasks when there is any uncertainty for developers.

One sprint for one release:

(b) One sprint for one release:

- The manager's requirement of one sprint for one release is improper.
- The deadline for Release 1 is two months later, therefore the first sprint will be last for 2 months. For one sprint, the development group should conduct periodic short planning and review meetings for tasks completed and not completed in time and revise the set of backlog tasks accordingly. A too long sprint violates the rule of Scrum and the development group cannot summarize and adjust the current work in time. And also cannot draw the burndown chart to track the project.
- Two months is too long for only one sprint. It is better to divide 2 months into 4 sprints and each sprint is 2 weeks.

Use case prioritization:

(c) Use case prioritization:

- The manager's requirement of use case prioritization is improper.
- The manager is responsible for prioritizing the backlog during Scrum development. In different release, the manager plans for different targets, from the basic use cases to high value, until all use cases. Prioritizing use cases is a good way to guide the development effort and conduct the incremental development. Because customers can know about the development result directly and give feedbacks to the developers to guide the next release. However, the prioritization process should be talked with customers and agreed by the customer at first, but not only decide by the manager.
- The manager should conduct a meeting with customers and prioritize the requirements with customers and the prioritization should be decided by customers.

Project backlog versus sprint backlog:

(d) Project backlog versus sprint backlog:

- The manager's requirement of backlog is improper.
- The project backlog is a prioritized and structured list of deliverables that are a part of the scope of a project, which is often a complete list that breaks down work that needs to be completed. The manager himself cannot decide how to complete use cases in which release and should first talk with customers.
- The sprint backlog lists tasks to complete during a sprint, which is updated once a day. The manager can decide scenarios or stories to be completed in which sprint.
- Both project backlog and sprint backlog should be used in the development process. When the project is delayed, the manager should talk with customers and decide which use cases to be removed from the corresponding release.

Additional Sheet

Additional Sheet

***** END OF QUIZ *****

Appendix A: In-class exercise for AFL performed in Week 5

Study the content on Page 1 and Page 2 of this exercise. **What is the sequence of test cases generated (and thus executed) by AFL until it crashes the function g()? ____________________**

Consider AFL, a coverage-guided fuzzer, to be applied to test a function g(int x) using the following setting:

- The given seed sequence set is a sequence with one element 13, i.e., ‹13〉.
- There are two deterministic mutation operators. The algorithm applies O1 before O2 on the same seed.
  - (O1) decrement the input by 1. 13-01->12
  - (O2) divide the input by the integer 2. Note that the division is an integer arithmetic operator. 13-02->6

Here

- There are no nondeterministic mutation operators.
- Each value for *x* for fuzzing should be limited to the range of 0 to 12.
- The coverage achieved by executing g(x) on each input is shown in the following table. If there is a tick (✓) in a cell, it the test case indicated by the row heading executed the branch statement indicates by the column title. For instance, when x = 5, the test case will execute the branches B5, B7 and B8. Note that not all branches in g() is shown in the table, but you can ignore the other branches in this exercise.

x's value	Branches in g() executed by each test case									Will g() crash?
	B1	B2	B3	B4	B5	B6	B7	B8	B9	
0	✓			✓	✓					No
1				✓	✓					No
2		✓		✓	✓					No
3					✓		✓	✓	✓	**Yes**
4					✓		✓		✓	No
5					✓		✓	✓		No
6		✓		✓	✓	✓				No
7					✓		✓	✓	✓	**Yes**
8					✓		✓	✓	✓	No
9		✓		✓	✓					No
10					✓					No
11	✓		✓		✓					No
12		✓		✓	✓	✓				No

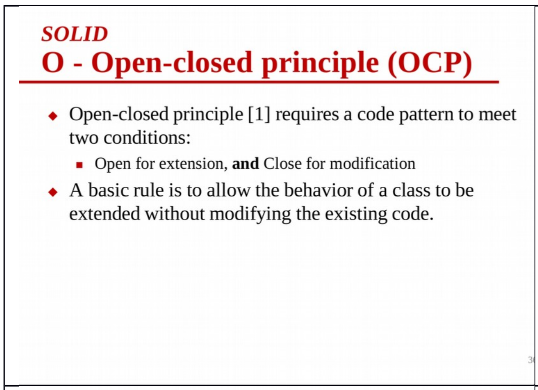
The algorithm previously shown on slide 39 of the Lecture notes on Program Testing (Part 1) is re-printed below for ease of your reference. **Since there are no nondeterministic mutation operators, line 7 and lines 12-22 in the algorithm can be ignored.**

For this exercise, in the algorithm, |*input*|, *mutate*(), *addToQueue*(), *newCoverage*(), *isWorthFuzzing*(), and *Run*(*Prog, input*) are defined as follows:

- **|input|** is defined as 1 for all inputs.
- **addToQueue(x, Y)** appends x to the current sequence Y.
- **newCoverage(result)** returns *true* if the test case used to run the program (at line 24) can execute branches not yet executed by the test cases existing in Queue; otherwise, it returns *false*.
- **isWorthFuzzing(y)** always returns *true*.
- ***Run*(*Prog, input*)** will execute *g*(*input*).
- **The algorithm will stop if running the program at line 24 will cause g() to crash.**
- Since there is no non-deterministic mutation operator, you can assume ***PerformanceScore*(*Prog, input*)** always returns -1.

Executed sequences of test cases: 12, 6, 11, 6, 10, 5, 4, 2, 3

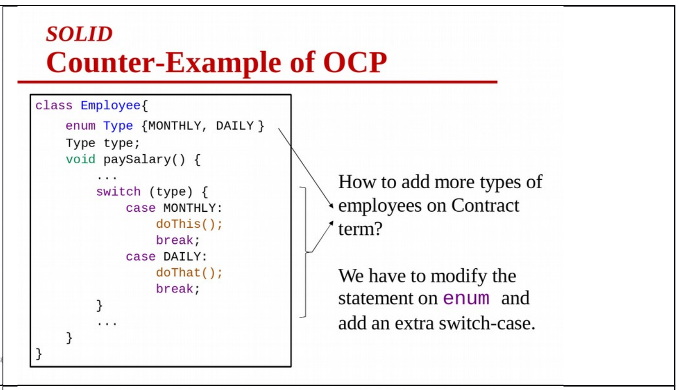
Appendix B: OCP slides in the lecture notes



SOLID

O - Open-closed principle (OCP)

- Open-closed principle [1] requires a code pattern to meet two conditions:
  - Open for extension, **and** Close for modification
- A basic rule is to allow the behavior of a class to be extended without modifying the existing code.



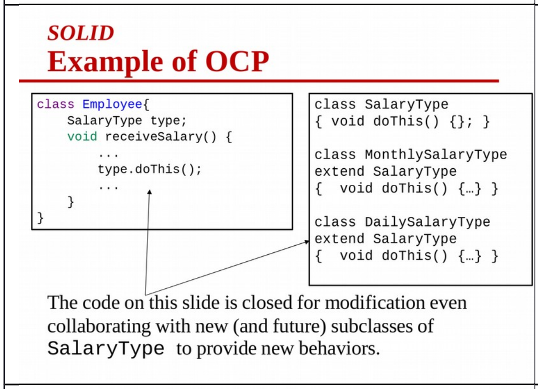
SOLID

Counter-Example of OCP

```
class Employee{
    enum Type {MONTHLY, DAILY }
    Type type;
    void paySalary() {
        ...
        switch (type) {
            case MONTHLY:
                doThis();
                break;
            case DAILY:
                doThat();
                break;
        }
        ...
    }
}
```

How to add more types of employees on Contract term?

We have to modify the statement on `enum` and add an extra switch-case.



SOLID

Example of OCP

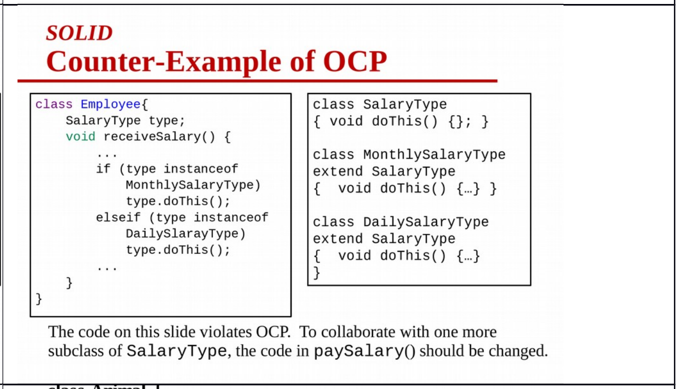
```
class Employee{
    SalaryType type;
    void receiveSalary() {
        ...
        type.doThis();
        ...
    }
}
```

```
class SalaryType
{ void doThis() {}; }

class MonthlySalaryType
extend SalaryType
{   void doThis() {…} }

class DailySalaryType
extend SalaryType
{   void doThis() {…} }
```

The code on this slide is closed for modification even collaborating with new (and future) subclasses of `SalaryType` to provide new behaviors.



SOLID

Counter-Example of OCP

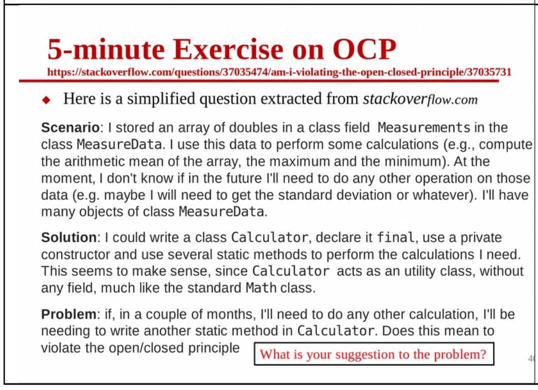
```
class Employee{
    SalaryType type;
    void receiveSalary() {
        ...
        if (type instanceof
            MonthlySalaryType)
            type.doThis();
        elseif (type instanceof
            DailySlarayType)
            type.doThis();
        ...
    }
}
```

```
class SalaryType
{ void doThis() {}; }

class MonthlySalaryType
extend SalaryType
{   void doThis() {…} }

class DailySalaryType
extend SalaryType
{   void doThis() {…}
}
```

The code on this slide violates OCP. To collaborate with one more subclass of `SalaryType`, the code in `paySalary`() should be changed.



5-minute Exercise on OCP

https://stackoverflow.com/questions/37035474/am-i-violating-the-open-closed-principle/37035731

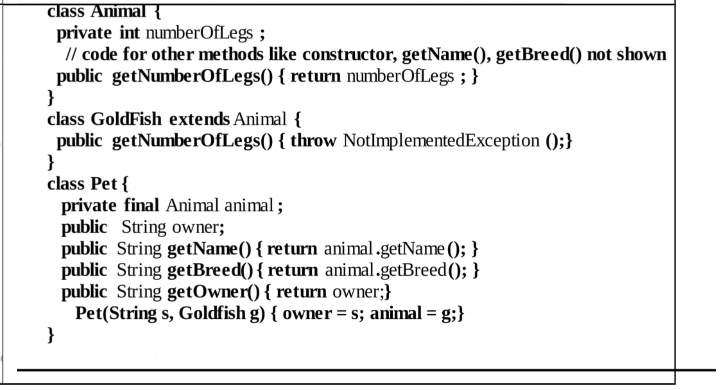
- Here is a simplified question extracted from *stackoverflow.com*

Scenario: I stored an array of doubles in a class field `Measurements` in the class `MeasureData`. I use this data to perform some calculations (e.g., compute the arithmetic mean of the array, the maximum and the minimum). At the moment, I don't know if in the future I'll need to do any other operation on those data (e.g. maybe I will need to get the standard deviation or whatever). I'll have many objects of class `MeasureData`.

Solution: I could write a class `Calculator`, declare it `final`, use a private constructor and use several static methods to perform the calculations I need. This seems to make sense, since `Calculator` acts as an utility class, without any field, much like the standard `Math` class.

Problem: if, in a couple of months, I'll need to do any other calculation, I'll be needing to write another static method in `Calculator`. Does this mean to violate the open/closed principle

What is your suggestion to the problem?



```
class Animal {
  private int numberOfLegs ;
    // code for other methods like constructor, getName(), getBreed() not shown
  public  getNumberOfLegs() { return numberOfLegs ; }
}
class GoldFish extends Animal {
  public  getNumberOfLegs() { throw NotImplementedException ();}
}
class Pet {
   private final Animal animal ;
   public  String owner;
   public  String getName() { return animal.getName(); }
   public  String getBreed() { return animal.getBreed(); }
   public  String getOwner() { return owner;}
     Pet(String s, Goldfish g) { owner = s; animal = g;}
}
```

Appendix C: Descriptive Statistics in a Survey on Refactoring Issues and Benefits

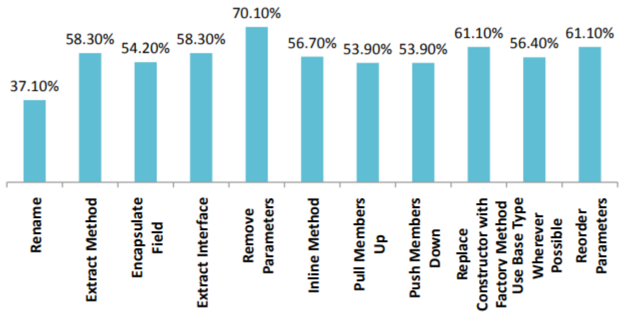


Figure 1: The percentage of survey participants who know individual refactoring types but do those refactorings manually

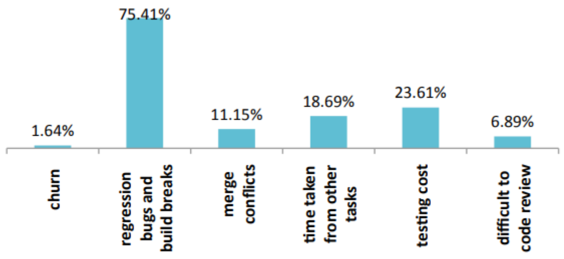


Figure 2: The risk factors associated with refactoring

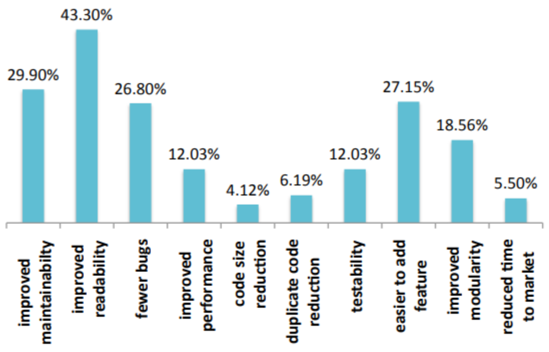


Figure 3: Various types of refactoring benefits that developers experienced

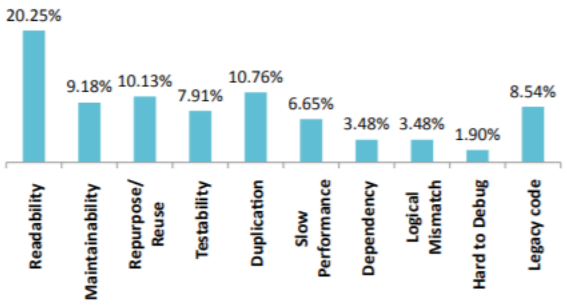


Figure 4: The symptoms of code that help developers initiate refactoring

Source: Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12). ACM, New York, NY, USA, , Article 50 , 11 pages.