3S03 - Software Testing
Assignment 1

Mark Hutchison
hutchm6@mcmaster.ca

Jatin Chowdhary
chowdhaj@mcmaster.ca

February 12, 2022

Contents

1 Question 1 - The Dangers of Inheritance from a Testing Viewpoint

Inheritance is an extremely useful concept in object-oriented programming; it allows developers to save time by maximizing code reusability. However, from a testing viewpoint, the use of an inheritance hierarchy can have detrimental effects on the controllability and observability of code. The following is a simple yet effective example that demonstrates this. In addition, the following code utilizes proper modular code practices; file separation, to name a few.

```
class Cat(object):

    def __init__(self, colour: str, size: str, home: str):
        self.colour: str = colour
        self.size: str = size
        self.home: str = home

    def __str__(self) -> str:
        if self.size == "small":
            return "Meow"
        elif self.size == "medium":
            return "Mow"
        elif self.size == "big":
            return "Roar"
        else:
            return "Hiss"

    def play(self, duration: float, toy_weight: float) -> bool:
        if toy_weight > 20:
            raise ToyToHeavy()
        if duration > 1440:
            raise DayWasted()
        return duration < toy_weight
```

```
class Lion(Cat):

    def __init__(self, colour: str):
        super().__init__(colour, "big", "Africa")
    def __str__(self)__ -> str:
        return super().__str__()
    def play(self, duration: float, toy_weight: float) -> bool:
        return super().play(duration, toy_weight)
```

Although this example is relatively small, the following issues (**Observability** and /textbfControllability) are present:

1.1 Observability

From an observability perspective, the example above lacks observability; it is not easy to observe the behaviour of the program. For example, in order to understand the behaviour of the `Lion().__str__()` function, the entire `Cat().__str__()` function, in the parent class, needs to be examined as well. Upon examining the function in the parent class, the variables of the `Lion` class instance need to be examined as well. Evidently, observability has been compromised. Our initial goal of deducing the result of the `Lion().__str__()` function has turned into a tedious analysis of 2 classes, `Lion` and `Cat`, and requires tracing variables from one to the other. Although this example is relatively small, repeating this on a much larger scale - tracing multiple functions across multiple parent classes to see the effects of each function - hinders observability. Furthermore, Python allows multiple parent classes to be passed in to a single child, if establishing an inheritance chain is not required. However, it can be argued that inheritance chaining is the worst option in terms of observability due to the recursive search of the parent functions. From a testing perspective, this is a nightmare.

1.2 Controllability

In terms of controllability, inheritance requires developers to not only worry about how the function parameters for the current function call, but also how those parameters will climb the inheritance tree. The `play()` function demonstrates this. Although the code in the `Lion` class does not show any errors, there are 2 conditions in the parent class (i.e. `Cat`) that can cause errors. As previously stated, developers need to concern themselves with how the current function uses the parameters, in addition to how the parent class uses the parameters; controllability has been compromised. Furthermore, not only does the child class need to be verified as correct, but the parent class as well. Failure to do so can cause errors to propagate from the parent class to the child class.

2 Question 2 - Junit on the `sort()` function

Now, this question is more flawed in the theory of what the test does, since the implementation of `sort()` is not known. However, if there is a fault this test case would not find it:

- **Reachability:** Running the `sort()` function does traverse us to a potential final state that contains the fault. Again, we cannot confirm alone that a fault exists, but we are moving into a new state here, and it is our final state reached by the test case.
- **Infection:** The final state of sorted could be out of order, thus being "infected" by a fault. Our test case, run likely by a test oracle, needs to validate the ENTIRE order of the list to ensure no fault exists.
- **Propagation:** The potentially infected state is the final state as an assertion takes place immediately after our sorting occurs. This means that the fault does indeed have a chance to cause a fatal error or unexpected result.
- **Revealability:** The assert statement on reveals that the first item is correct, leaving 3 items to be potentially unsorted. We, however, cannot see these items and thus cannot see the fault if it exists. We must see and traverse the entire sorted list to be sure there is no fault propagating in this test. Again, this is likely being handled by some sort of test oracle, in which would attempt to force you to review the list contents in its entirety, not just 1 item.

Again, no fault might actually exist, but the test provided does not do enough to say that. As of this moment, this test can contain a fault and allow it to pass without issue.

3 Question 3 - Test Driven Development

3.1 Requirements Analysis

The first issue encountered when preparing to design test cases for this problem is the lack of specificity; no actual requirements are given. Apart from the names of each function, the expected type signature and functionality of each function is not known. Without the ability to concretely determine these requirements, assumptions are going to be made about the functions and their behavior; this creates ambiguity - discussed later.

When coming up with criteria without requirements, we need to think about overall project cost, and the repercussions of our assumptions being wrong. Due to this, minimizing cost/work is essential. The assumptions we make need to reduce the amount of effort/work we put into the (`Calc`) functions. Hypothetically, if a client states further requirements, then it is easier to add more functionality and negotiate extra pay than it is to waste time on work the client did not want; and bare the cost that comes with it.

The three major requirements not detailed in the specification document are:

- All 4 `Calc` functions will be implemented on integers, and return integers. Hence, there will be no issues that arise with floating point arithmetic. Furthermore, we will follow conventions that come with integer division, such as flooring floating point results.
- In the event of $overflow^*$, depending on how Java handles it, we need to do something specific. My original thought was to allow any errors to permeate through and be caught later in exception handling, however, Java may allow overflow.
- The lack of explicit details about the intended scalability of this project in the future. We do not know how this class is intended to interact with others, so we must keep minimalism in mind. We can achieve this by using correct design principles that anticipate change.

3.2 Pre-Coding Test Generation

Coming up with tests prior to implementing the code was potentially one of the harder things to do, even with functions so easy. Of course you have your standard tests like just doing $3 \cdot 4 == 12$ or $5 + 8 == 13$, but then you also have to account for edge cases that come from properties related to domains. Some examples include:

- **Identity Cases:** Nearly every operator has two identity cases: The zero case and the "do nothing" case.
  - Addition: $n + (-n) == 0$ and $n + 0 == n$
  - Subtraction: $n - n == 0$ and $n - 0 == n$
  - Multiplication: $n \cdot 0 == 0$ and $n \cdot 1 == n$
  - Division: $\frac{n}{n} == 1$ and $\frac{n}{1} == n$
- **Equivalency Cases:** Sometimes, these operators become each other. Since a and b must be integers, we do not worry about Multiplication becoming Division or Division becoming Multiplication. However, subtracting a negative value is just adding the positive, while adding a negative value is just subtracting the positive. We can add automated testing for these cases.
- **Boundary tests:** There are some boundary tests that will need to be implemented:
  - Addition: You cannot add anything to the maximum integer, so we expect an error here. Also, attempting to add a negative value to the minimum integer is also an error. *Note: Our initial idea for boundary testing on addition is to check if the result ends up being negative, because 2 positive integers must add up to a positive integer. If the result is negative, then* $overflow^*$ *has occurred. However, this idea is not sound because if the result overflows to a positive number, then an error will not be raised, despite the occurrence of an overflow. Similar logic applies to subtraction.*
  - Subtraction: You cannot subtract a negative value from the minimum integer, so we expect an error here. *Note: Boundary testing for subtraction poses similar problems outlined above, under Addition.*
  - Multiplication: The absolute value of the product of two integers cannot exceed the integer boundaries.
  - Division: You cannot divide by 0.

Once we implement all of the specific properties mentioned in a test case, add some standard unit tests to verify that specific values are equal to what we expect, and verify we are happy with the testing suite - We can move on to the code.

Coding these are all relatively simple; each operator is only one line, after all! However, I wanted to make it explicitly clear that our code can error from performing specific operations, as those errors exist in our test cases. I am not going to do any if statements or anything like that, I am simply going to allow the errors through to the test case and denote the function can throw the expected error.

3.3 Post-Coding Modifications

As feared, Java does allow $overflow^*$ (and $underflow$). Due to this, a decision needs to be made: Either explicitly account for it in all 4 methods (including the given source code), or refactor the test cases to allow the flow edge cases. The former requires more work, but the latter removes a good chunk of our boundary tests, so both are a costly choice. In the end, I refer to the minimalism point brought up in the discussion about requirements: Imposing a harder to test condition on the code is likely a worse option than removing the test cases. If the overflowed is not desired, we can account for it after acquiring more information.

*Note: *The terms overflow and underflow follow normal definitions laid out by the principles in computer science. If a number exceeds the maximum, then overflow has occured. If a number goes below the minimum, then overflow has (also) occured. Our definitions for overflow are based on modular/clockwork arithmetic. Examples are: `MAX_INT + 1` and `MIN_INT - 1`, respectively. The term underflow refers to a number that is too small to accurately represent, so the value ends up being 0. Since we are only dealing with operations on integers, we can ignore underflow and only focus on overflow.*