3S03 - Software Testing
Assignment 1

Mark Hutchison
hutchm6@mcmaster.ca

Jatin Chowdhary
chowdhaj@mcmaster.ca

February 11, 2022

Contents

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

Yes, inheritance can be great for coders, but there are always going to be issues that come with this benefit. For the purposes of this example, let's use a simple example that also uses proper modular code practices like file separation and so on:

```
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)
```

Now, although this example is small scale, we can still see some obvious issues:

1.1 Observability

In order to know what `Lion().__str__()` will do, we need to examine the entirety of the `Cat().__str__()`. When examining the function of the parent class, we will then notice that we need to check the variables of the `Lion` class instance. Now, what should have been a rather straightforward string result has become the analysis of 2 classes and tracing variables from to to the other. Imagine this on a much grander scale; having to trace multiple functions across multiple parent classes to see the effects of one versus the other. Python even allows multiple parent classes to be passed in to a single child if you don't want to establish an inheritance chain. However, I would argue inheritance chaining is the worse option in terms of observability due to the recursive search of the parent functions.

1.2 Controllability

In terms of controllability, you need to not only worry about how the function parameters for the current function call, but how those parameters will climb the inheritance tree. Look at the `play()` function; although the code of the `Lion` class doesn't show the errors, there are 2 conditions in the parent class that can cause errors. You need to worry about how the parent uses these values as well as how your current function uses them. Additionally, your parent function now needs to be verified as correct as your child class, or else errors from the parent will propagate to the child!

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

Now, this question is more flawed in the theory of what the test does, seeing as we can't see how `sort()` was implemented. If there is a fault however, 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 can't 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 doesn't 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 actual requirements 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.

When coming up with criteria without requirements, we need to think about overall project cost, and the repercussions of our assumptions being wrong. Because of this, to minimize cost, the assumptions we make need to be in order to reduce the amount of effort or work put into these functions. If they need to impose further specifications on the work, it is easier to add more and negotiate the extra pay than it is to waste the time on work they didn't want and eat the cost that comes with it.

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

- All 4 Calc functions will be implemented on integers, and return integers. So there will be no issues with floating point arithmetic, and we will follow conventions that come with integer division, such as flooring floating point results.
- In the event of overflow or underflow, 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, but Java may allow the overflow.
- The lack of explicit details about the intended scalability of this project in the future. We don't know how this class is intended to interact with others, so we must keep minimalism in mind.

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 don't 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 can't 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.
  - Subtraction: You can't subtract a negative value from the maximum integer, so we expect an error here. Also, attempting to subtract any value from the minimum integer is also an error.
  - Multiplication: The absolute value of the product of two integers cannot exceed the integer boundaries
  - Division: You can't 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 ladder 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.