CS 314 Black Box Testing Pre-Class Assignment

Pre-Class Tasks:

1. Complete pre-class readings. (Estimated time: 20 min)
2. Complete the pre-class preparation. (Estimated time: 20 min)
3. Take the on-line quiz (Q3 – Black Box Testing) before 11:55pm on Tues Feb 23. You may take the quiz multiple times.

Overview

This pre-class work is meant to help you complete the following user stories:

- As a student in CS 314, I want to understand what Black Box testing is so that I can decide when BB testing is useful.
- As a student in CS 314, I want to understand what I need to know about a possible BB test so that I can systematically go about creating one.
- As a student in CS 314, I want to practice creating a small set of BB tests so that when I get to class I can apply what I learned to developing BB tests for the Adventure Game.

Pre-Class Readings and Video:

- Black Box testing in the 'CS314SP2016-BBTesting.pdf' file.
- You Tube Black Box testing video: https://www.youtube.com/watch?v=Wi75S5TTfQ0

Pre-Class Preparation:

In Assignment 3, you will be adding unit tests to the Adventure Game to achieve high test coverage. You will use a tool to measure test coverage that is described in the assignment web page. A very common strategy is write black box tests as much as possible, then add what are called white box tests until the coverage goal is achieved. The rationale is that black box tests written from a user guide, program guide, or class API can test multiple methods and potentially classes in the same test, whereas white box tests are strictly applicable to a particular control flow in a particular method.

One of the key things needed for writing black box tests is to determine the proper input domain for a test, and to break it into equivalence classes. According to Ammann and Offutt[1], the input domain covers method input parameters, global variables, instances (objects) of a class, and user inputs to a program. If we are developing black box tests for a class, one input domain is all possible states of objects of that class. If we are developing black box tests for a program, one input domain is all possible states of that program. Note that all states include valid and invalid states, however, it may not be possible to create an invalid state to test.

Consider the Java Stack class described in the slides 'CS314SP2016-BBTesting.pdf'. If we create a Stack<Integer> object, then try to push a string onto the stack, the compiler prevents this; there is an invariant imposed for all instances of this class that it can only contain items of type Integer since we created it for Integers. This is a situation where we can't create an invalid state. However, no matter what type of items a Stack contains, if we try to perform a pop operation on an empty stack, the specification states we should get an EmptyStackException. See the Java 8 reference API for Stack at:

https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html

A good way to systematically think about black box tests for an API (note that all the method signatures of a class constitute its API) is to separate them into categories. Again consider the Stack class and the categorization of methods discussed in the slides. The categories are constructors/destructors, state

[1] Introduction to Software Testing, Paul Ammann and Jeff Offutt, Cambridge Press, 2008

changing operations, and non-state changing operations. Next think about the input domains that make sense for these categories. An example is for the constructor – is it possible to create a test where you try to make a new Stack using a variable that is already a Stack? According to the API, is this possible? What is supposed to happen if you try this?

Consider the non-state changing method *empty()*. According to the reference, this should return true if and only if the stack contains no items, and otherwise it returns false.

1. What is the input domain for this method?
2. What are the equivalence classes for the input domain when talking about this method?

The input domain is the set of states of a stack. Since we're thinking about the *empty()* method, and it only has 2 outputs, this nicely divides the input domain into stacks that are empty and stacks that aren't empty. These are the equivalence classes for the input domain. Equivalence classes all together must cover the entire input domain (this is called completeness), and they must not overlap.

1. Are completeness and non-overlapping requirements met with this partitioning of the input domain?

The great thing about equivalence classes is that since all inputs in one class are equivalent, we only have to test one of them. We just need to define one test for every equivalence class.

There is a big HOWEVER to this idea though. It turns out that many bugs occur around the boundaries between equivalence classes. For the *empty()* method, this means that a Stack with 1 item in it is a good boundary test, and since boundaries may have defects, a Stack with 2 items and maybe one with 5 items are good ideas too. What about the empty Stack? It seems like we have an empty Stack and that is all. But what does 'empty' mean? What about pushing a *null* on the Stack? Is it empty or not? What if we try to use the method on a non-existent Stack? For example, declare a Stack variable, then try to run the empty method before using a constructor to create the Stack.

In the case of a method that has different types of arguments, you are faced with the situation where the state of class objects are one input domain, and there are input domains for each of the arguments. You might therefore have several different input domains you have to think about.

Consider the chooseDropItem() method in AdventureGame.java. Its signature is:

```
private int chooseDropItem(Player p, BufferedReader keyB)
```

The input domains are the sets of: (1) states of an AdventureGame object, (2) states of a Player object, and (3) states of a BufferedReader. We can argue that the sets of states of a Player object and a BufferedReader are actually parts of the set of states of an AdventureGame object. However, it may be easier to think of them as 3 different input domains while we are developing BB tests, so that is what we will do for the remainder of this discussion. Look at the reference for BufferedReader at:

https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html.

This class can be used to read a character of some sort (int, char, part of a char array, …) or a string (a line of text or a bunch of lines).

1. What are the input domains of the *chooseDropItem()* method?
2. What are the equivalence classes for the input domains when talking about this method?
3. For the equivalence classes of each domain, are the equivalence classes complete and non-overlapping?
4. What are some possible boundary conditions on the equivalence classes?

For this method of the AdventureGame class, one input domain is the states of an AdventureGame object. One partitioning of this domain yields the equivalence classes that an AdventureGame object exists, or it doesn't exist. The compiler won't let us call the method on a non-existent class, so in this partitioning the only equivalence class is the domain of existing AdventureGame objects. For the *chooseDropItem()* method, possible argument input domains are the states of Player objects, and the states of BufferedRead objects. If we partition based on existing and non-existent objects, we are again left with a single partition since the compiler doesn't allow non-existent objects as arguments.

If we think about the method, and read the game description, it seems that the part of the Player object state that is relevant is the set of Items that are part of the Player state. So, possible equivalence classes for the Player state set of Items are the number of items a player has: 0 and greater than 0. Possible boundary conditions for the number of items are 0, 1, 2, and "lots" (maybe 5 or 10). The user guide doesn't indicate that the type of Item matters (i. e. Key or Treasure) to this method, but we might try some test cases that are all one type and some that are multiple types.

The BufferedRead object takes some input from the keyboard (according to the user guide the number of the Item to drop). One partitioning of the keyboard input can be divided into equivalence classes of alpha, numeric, special characters, different language characters (e.g. character codes beyond the ASCII set), etc. We should also think about multiple character strings. In this case no characters, 1 character, 2 characters, and lots of characters would be reasonable boundary tests.

One important thing about testing with multiple input domains is that the test should only contain 1 input that might cause a failure. Thus, initial tests might include states and inputs that we know work, then we vary one input to something that we think may be invalid.

The in-class activity is to use the excerpt from the game description below to develop black box tests.

Game Description Excerpt

Some rooms have doors that are locked. To open locked doors, you must find the key to locked doors, which are placed somewhere in the labyrinth. As you go from room to room, you can

- look at the room,
- go into an adjacent room or through an adjacent door,
- pick up an object in the room, or
- drop an object that you are carrying.

All of these capabilities have to do with rooms. For example, 'look at the room' means that you have come into a room. The input domain is the states of Room objects. A partition of this domain is that a Room object exists and another is that it doesn't. Another partitioning has to do with structure – the Room has a Door, and it doesn't have a Door. What about walls? Partitioning could be no walls, less than 6 sides, more than 6 sides. Another partitioning has to do with contents – no contents, one thing, one thing of each type, several things of all one type, and several things of mixed types.

1. Think about how you will develop BB tests for the 2nd items in the list above: 'go into an adjacent room or through an adjacent door'. Specifically, what are the input domains, how can you partition them into equivalence classes that are complete and non-overlapping, and what are boundary conditions for each partition.
2. Think about how you will develop BB tests for the Room.getRoomContents() signature.:

   ```
   public Item[] getRoomContents()
   ```

   Specifically, what are the input domains, how can you partition them for completeness and non-overlapping, and what are boundary conditions. You can also look at the class diagram for the game if this helps (hint: you can identify the specializations of the Item class from this diagram).