SWE_261P_ Project_Part_2
Team Members:
· Yuxin Huang: yuxinh20@uci.edu
· Changhao Liu: liuc50@uci.edu
· Ruokun Xu: ruokunx@uci.edu
Team Name: OffersAreHere
Github link: https://github.com/yx-hh/commons-io

Finite State Machine Introduction

Finite State Machine (FSM) is a computational model based on a hypothetical machine consisting of one or more states to simulate the flow of state transition. FSM is a directed graph with each node representing a single state, and each edge is the action that takes the artifact from one state to another(Pezzè and Young, pg 65). Apart from these, each machine only has one active state at a time. Each state has a set of transitions depending on user actions (Cited from [What is a Finite State Machine?]). Finite state machines facilitate functional testing by describing the program as a state-transition model, helping testers understand the characteristics of the program and save their time to write cases to fully test the program(Pezzè and Young, pg 65).

Usage and Advantages of Finite Models

Finite models represent a simpler version of the artifact but keep the main characteristics of the artifact, making it easier for testers to understand the transitions within the feature and to analyze results based on a variety of actions(Pezzè and Young, pg 55). Thanks to this model, testers are able to trace the error states as well as complete states for the artifact. Each time the system only stays at one state, which is controlled by inputs from testers. In other words, we can draw a state transition flow of the program according to the finite model theory. The state transition flow is able to clearly illustrate the state of the current process and help testers save time and to comprehensively examine the program with fewer test cases. Furthermore, with a finite model we can easily observe which branch the bug exists in. (Cited from [Model Based Testing Tutorial: What is, Tools & Example])

Feature Chosen

The feature with the FSM function model we chose is to check whether the directory is empty by analyzing its corresponding path. There are a few valid states and three error states that may occur:

- Return true if the directory of that path is valid and empty.

- Return false if the directory is valid and has no content in it.
- Throw NotDirectoryException if the path doesn't lead to a directory, as such a file cannot be opened.
- Throw IOException if any I/O errors happen.
- Throw SecurityException if the installed security manager finds that the user doesn't have the proper access permission to the directory.

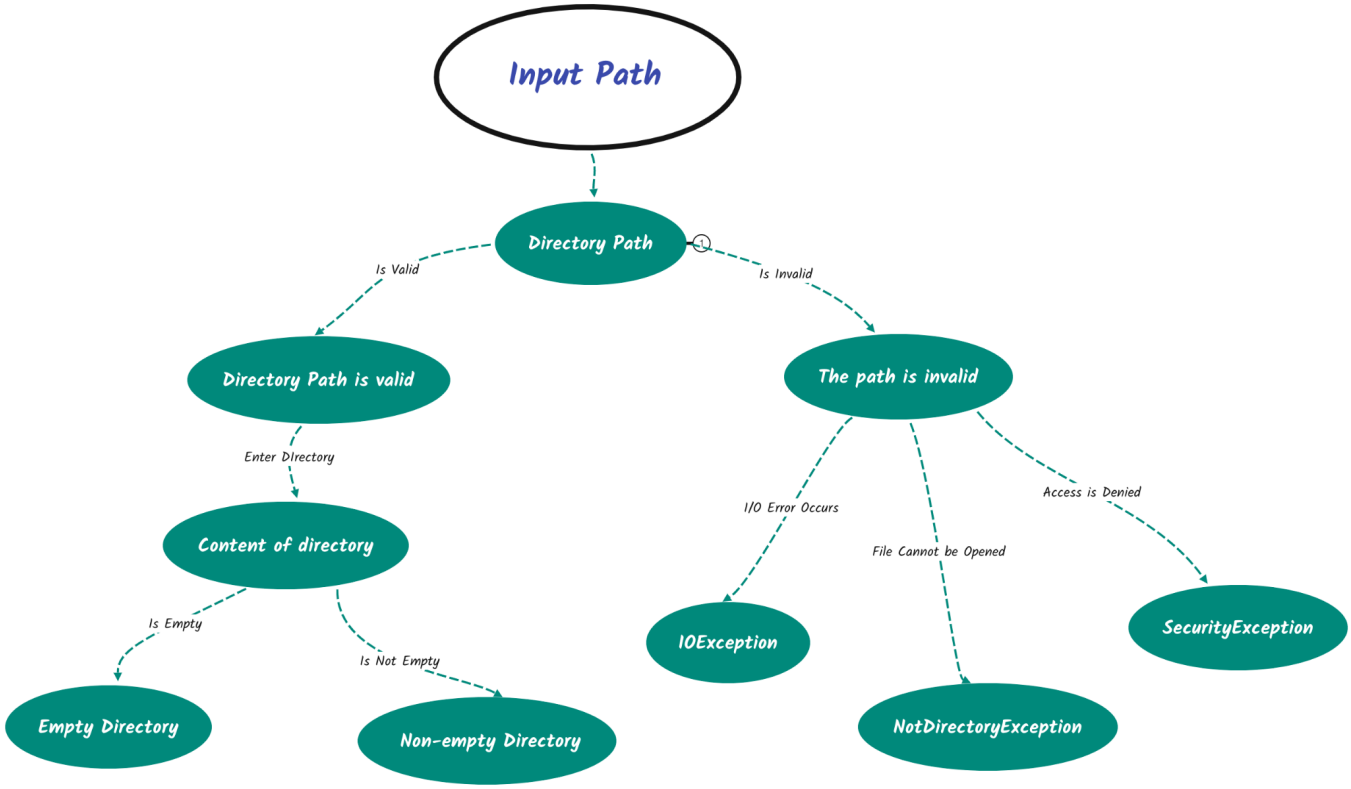
Functional Model of this feature

As you can see from the model we drew below, the feature that we tested can be converted to a functional model. First, the model starts in a directory path state. Depending on whether the path is valid or not, the model will move to a valid path state or an invalid path state. In the valid directory state, we can transition to the "Content of directory" state by entering the directory. After that, by checking the content of the directory, we can move to the "Empty Directory" state or the "Non-empty Directory" state.

If the directory path is invalid, the model moves from the "Directory Path" state to the invalid path state. The next transition depends on the occurrence of each error. For example, if an I/O error occurs, the model will proceed to the IOException state.

In conclusion, building a functional model guides us to gain a deeper understanding of this feature and to write test cases to comprehensively cover some corner cases.

Finite State Flow Graph



Test Cases covers the Functional Model

1. Test case for checking whether directory is empty state or not: expect true with empty content under the directory

```
@Test
  public void testIsEmptyDirectoryStateTrue() throws IOException {
     final Path tempDir = Files.createTempDirectory(getClass().getCanonicalName());
     assertTrue(PathUtils.isEmptyDirectory(tempDir));
     Files.delete(tempDir);
  }
```

2. Test case for checking whether directory is empty state or not: expect false with file under the directory

```
@Test
  public void testIsEmptyDirectoryStateFalse() throws IOException {
     final Path tempDir = Files.createTempDirectory(getClass().getCanonicalName());
     Files.createTempFile(tempDir, "prefix", null);
     assertFalse(PathUtils.isEmptyDirectory(tempDir));
```

```
    FileUtils.deleteDirectory(tempDir.toFile());
  }
```

3. Test case for checking whether directory is empty state or not: expect NotDirectoryException exception with input a file path rather than a directory path

```
@Test
  public void testIsEmptyDirectoryStateNotDirectoryException() throws IOException {
    final Path filePath = Files.createTempFile(tempDirPath, "prefix", null);
    assertThrows(NotDirectoryException.class, () -> PathUtils.isEmptyDirectory(filePath));
    Files.delete(filePath);
  }
```

4. Test case for checking whether directory is empty state or not: expect IOException exception with input a non-exist path rather than a real path

```
@Test
  public void testIsEmptyDirectoryStateIOException() throws IOException {
    final Path tempDir = Files.createTempDirectory(getClass().getCanonicalName());
    Files.delete(tempDir);
    assertThrows(IOException.class, () -> PathUtils.isEmptyDirectory(tempDir));
  }
```

Reference

Pezzè Mauro, and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, 2008.

Bors, Mátyás Lancelot. *"What Is a Finite State Machine?"* Medium, Medium, 10 Mar. 2018, https://medium.com/@mlbors/what-is-a-finite-state-machine-6d8dec727e2c.

Hamilton, T. (2022, January 1). *Model based testing tutorial: What is, Tools & Example. Guru99.* Retrieved February 9, 2022, from https://www.guru99.com/model-based-testing-tutorial.html#:~:text=one%20by%20one%3A-,Finite%20State%20Machines,inputs%20given%20from%20the%20testers