Barış Tan Ünal | 22003617 | Section: 2

CS 319 DESIGN PATTERN TAKE HOME EXAM REPORT

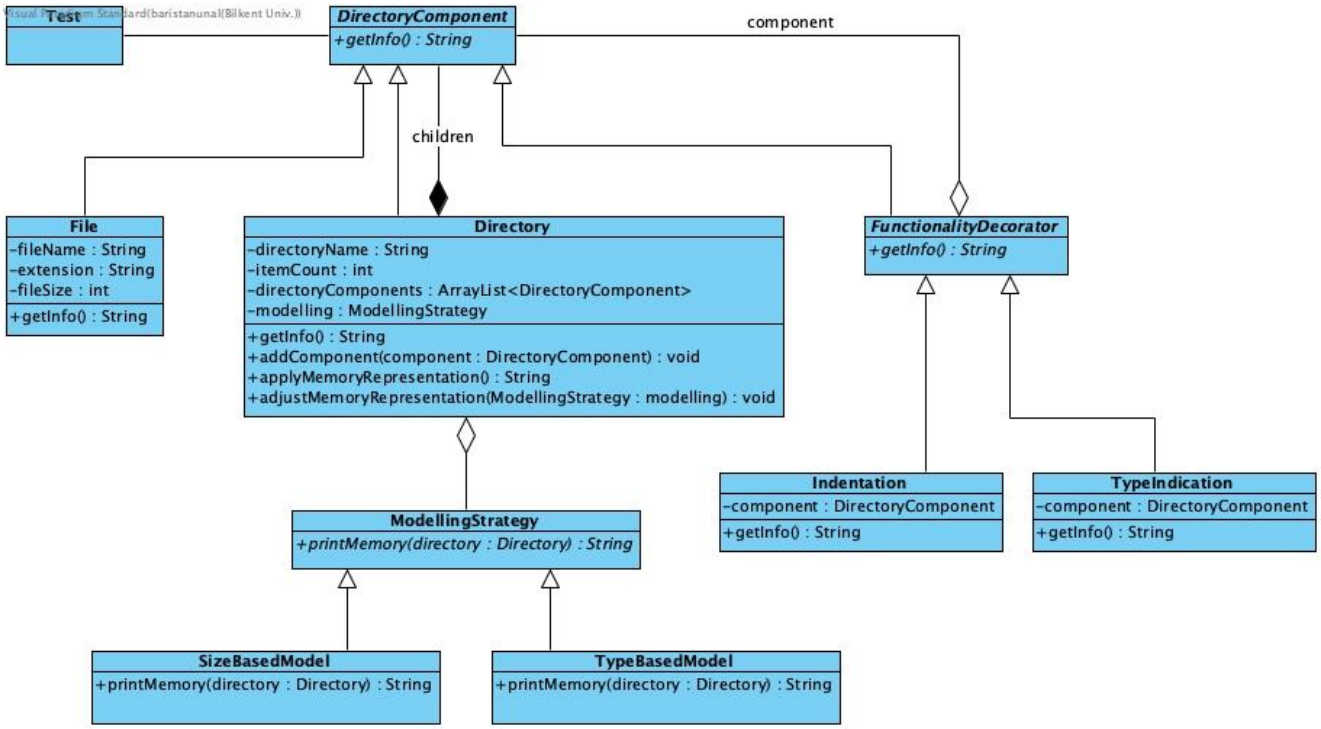

Fig.1. UML Class Diagram of the Implementation

PART 1: Composition

In the first part, we have a File and a Directory class as it was stated in the problem definition. It was also given that Directory objects can include both File objects and other Directory objects. For this to happen, it is clear that Directory objects should hold an array-like structure of both Directory and File objects. I have used an Array List for ease of use. However, it can lead to other complications if we allow this Array List to have any object type (Object) as elements. In order to prevent those complications and come up with a clearer implementation, I have used the *Composition* design pattern. I have created an abstract DirectoryComponent class which is inherited by File and Directory classes. This DirectoryComponent object can either be a File or a Directory. So, our Array List can only hold elements of type DirectoryComponent. That class has an abstract version of getInfo method with an empty body. When the getInfo method is called for

one of the elements in the Array List, the appropriate class's (File or Directory) function is called. This way, I was able to print the whole memory with only one function without knowing whether the components are File or Directory. Instead, this structure handles that in runtime.

PART 2: Decorator

In the second part, the problem definition says it needs two features where one of them should put an indentation in front of each line according to their depth and the other one puts a type indication before each line in parentheses. However, it wants the possibility of these two features being used both individually and combined. This implies that we need two features that can be applied to a Directory object. For these two features, I have created three new classes called Indentation and TypeIndication along with an abstract decorator class called FunctionalityDecorator. Indentation and TypeIndication inherits the FunctionalityDecorator class, which inherits DirectoryComponent class. Both Indentation and TypeIndication classes have a DirectoryComponent object as their property so that it can take another DirectoryComponent in its constructor as a parameter to make it a DirectoryComponent with Indentation feature. After giving that feature to a DirectoryComponent, we can give TypeIndication feature too without giving up on the Indentation feature. I have used *Decorator* design pattern because the problem definition needed the program to be able to add different features dynamically to a Directory object.

PART 3: Strategy

In the third part, it was said that the company wants a "memory representation" tool for the file system which has two alternatives: size-based and type-based. Even though it seems similar to the second part, in this part these representations cannot be combined. These are two separate representations. As the problem definition states, I have added adjust and apply memory representation methods to the Directory class. Additionally, I have added a new property to Directory which is a ModellingStrategy. This property holds which representation will be used when apply method is called. The two new classes SizeBasedModel and TypeBasedModel inherits another new class, ModellingStrategy. These two classes implement a method with the same name, printMemory, which contains the implementation of the two representation styles. I have used *Strategy* design pattern because it enabled me to decide on which algorithm to use at runtime with the help of an abstract strategy class, which is ModellingStrategy in this case.