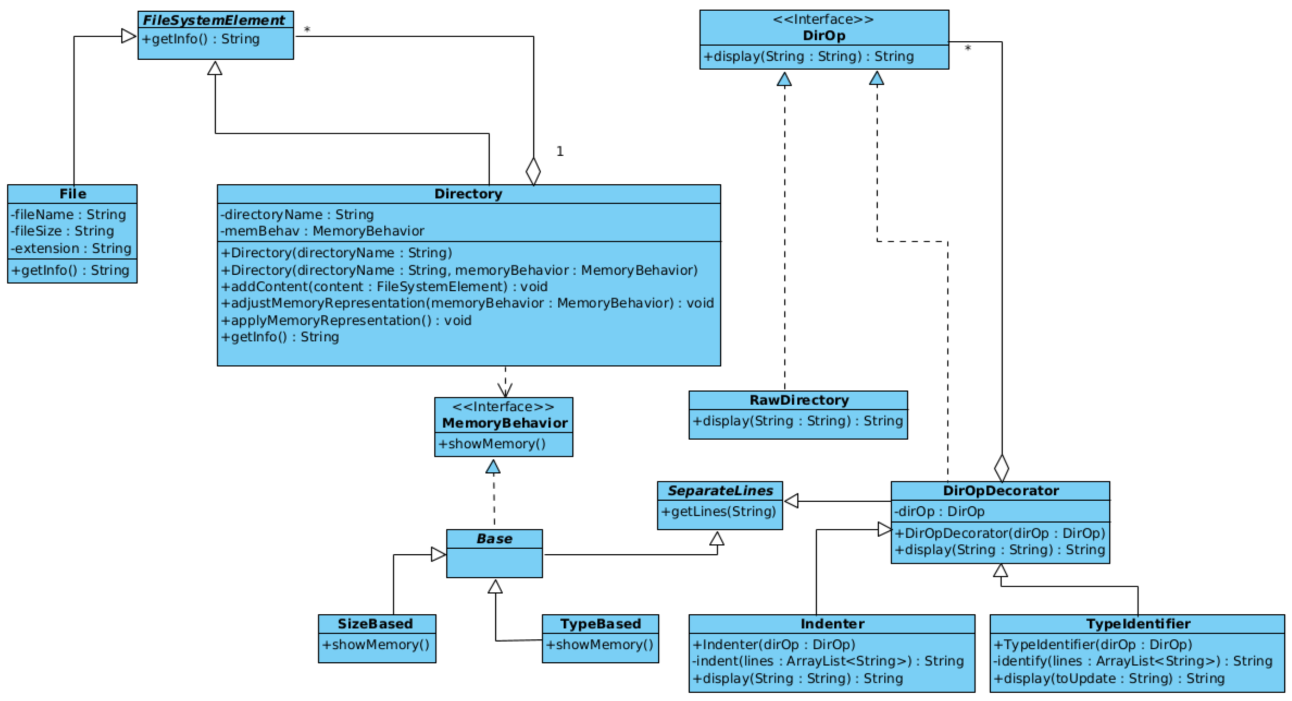
Class Diagram:


FileSystemElement
+getInfo() : String
*
1
<<Interface>>
DirOp
+display(String : String) : String
*
File
-fileName : String
-fileSize : String
-extension : String
+getInfo() : String
Directory
-directoryName : String
-memBehav : MemoryBehavior
+Directory(directoryName : String)
+Directory(directoryName : String, memoryBehavior : MemoryBehavior)
+addContent(content : FileSystemElement) : void
+adjustMemoryRepresentation(memoryBehavior : MemoryBehavior) : void
+applyMemoryRepresentation() : void
+getInfo() : String
<<Interface>>
MemoryBehavior
+showMemory()
RawDirectory
+display(String : String) : String
SeparateLines
+getLines(String)
DirOpDecorator
-dirOp : DirOp
+DirOpDecorator(dirOp : DirOp)
+display(String : String) : String
Base
SizeBased
+showMemory()
TypeBased
+showMemory()
Indenter
+Indenter(dirOp : DirOp)
-indent(lines : ArrayList<String>) : String
+display(String : String) : String
TypeIdentifier
+TypeIdentifier(dirOp : DirOp)
-identify(lines : ArrayList<String>) : String
+display(toUpdate : String) : String

An Explanation of abstract class SeparateLines:

In my Strategy and Decoder Design Patterns, I needed to separate lines of the String returned by getInfo() for the ease of my algorithms. This was a common feature on both sides, so I created an abstract class for it and made necessary classes extend it. I believe I avoided writing duplicate code in this way.

Design Patterns That I Used:

1)Composite Design Pattern:

I used this design pattern in FileSystemElement, File, and Directory. I didn't change any properties of this design pattern. I used this design pattern for part 1, because;

1)The same example exists in the book. If it is in the book, then this means it is working(if at least one of the authors and editors didn't make a blunder). I compared the example in the book with the one in the assignment and decided it would fit here too.

2)As explained in the previous explanation, it fits very well to the situation: it allows particular objects of the pattern to make the pattern continue infinitely in theory. That is, a directory can contain Files, which can't contain any FileSystemElements inside, and directories. And those subdirectories also contain subdirectories and files, and so on.

2)Decoder Design Pattern:

I used this design pattern in DirOp, RawDirectory, DirOpDecorator, Indenter, and TypeIdentifier. I didn't change any properties of this design pattern. I used this design pattern for part 2 because it allows modifying the representation of the contents in a combinatorial way. It is possible to use multiple decorations on the representation with this pattern.

3)**Strategy Design Pattern:**

I used this design pattern in MemoryBehavior, Base, SizeBased, and TypeBased. I made a slight change in the design pattern by adding the abstract class Base. I did it because it helped me to inherit MemoryBehavior and extend SeperateLines at the same time: If a new memory representation feature were to be added, it is not needed to implement MemoryBehavior and extend SeparateLines. Extending Base is sufficient. I used this design pattern for part 3 because it allowed Directory objects to have different memory representations in runtime. Hence, it was possible to change this behavior of Directory objects over time. This pattern fit my needs.