# Practical Assignment 4

## Task 1

### Question 1.1

1. Use the -g flag
2. Steps:
    a. gdb
    b. file marks
    c. run
3. Error: Program received signal SIGFPE, Arithmetic exception.

   Line: 17

   Arguments: (a = -2, b = 0)

4. List command:

```
14
15 int improve(int a, int b)
16 {
17    return ((double)(a / b)) * 100;
18 }
```

5. Stack trace:

```
#0 0x0000555555555204 in improve (a=-2, b=0) at
marks.cpp:17
#1 0x00005555555551dd in main() at marks.cpp:11
```

- `main()` called the function at line 11 (main() function is the bottom of the stack).
- `improve()` function got called and added to the stack.
- The `improve()` function crashed at line 17

6. up 1 command:

```
7        int main() {
8           int mark = 59, highest = 87;
9           cout << improve(mark, highest);
10          mark = -2; highest = 0;
11          cout << improve(mark, highest);
12          return 0;
13       }
14
15     int improve(int a, int b)
```

7. print highest

```
$1 = 0
```

8. There was a division by 0 error. Mark is divided by highest, but highest is 0.

## Question 1.2

3. ==159==

4. Invalid write of size 4

5. Where the error occurred

6. Integers are 4 bytes each. An array of 10 integers was allocated, but not deallocated. This means that 40 bytes was lost.

7. Delete the array (deallocate the memory)

```
delete marks;
```

# Task 2

Memento will have to be used in both implementations, as to navigate back and forth, one will have to store the directories and files in some way.

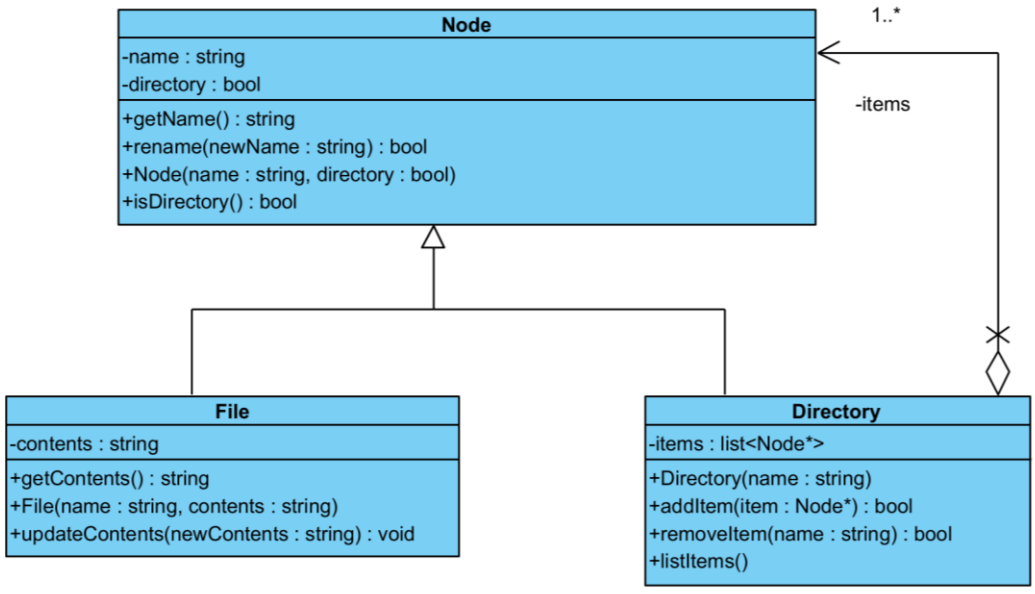
## 1. Memento and Decorator

The decorator design patter will allow both files and folders to be used in a uniform way. In this pattern, a file will be of one concrete decorator type and a directory will be of another type. Navigating the structure will be done using recursion.

## 2. Memento and Composite

The composite design pattern will give the same benefits as the decorator but will also allow the structure to be more easily navigated without using recursion. A directory will have a list of all the files and directories that it contains and will allow for removing and adding files/directories from within each directory.

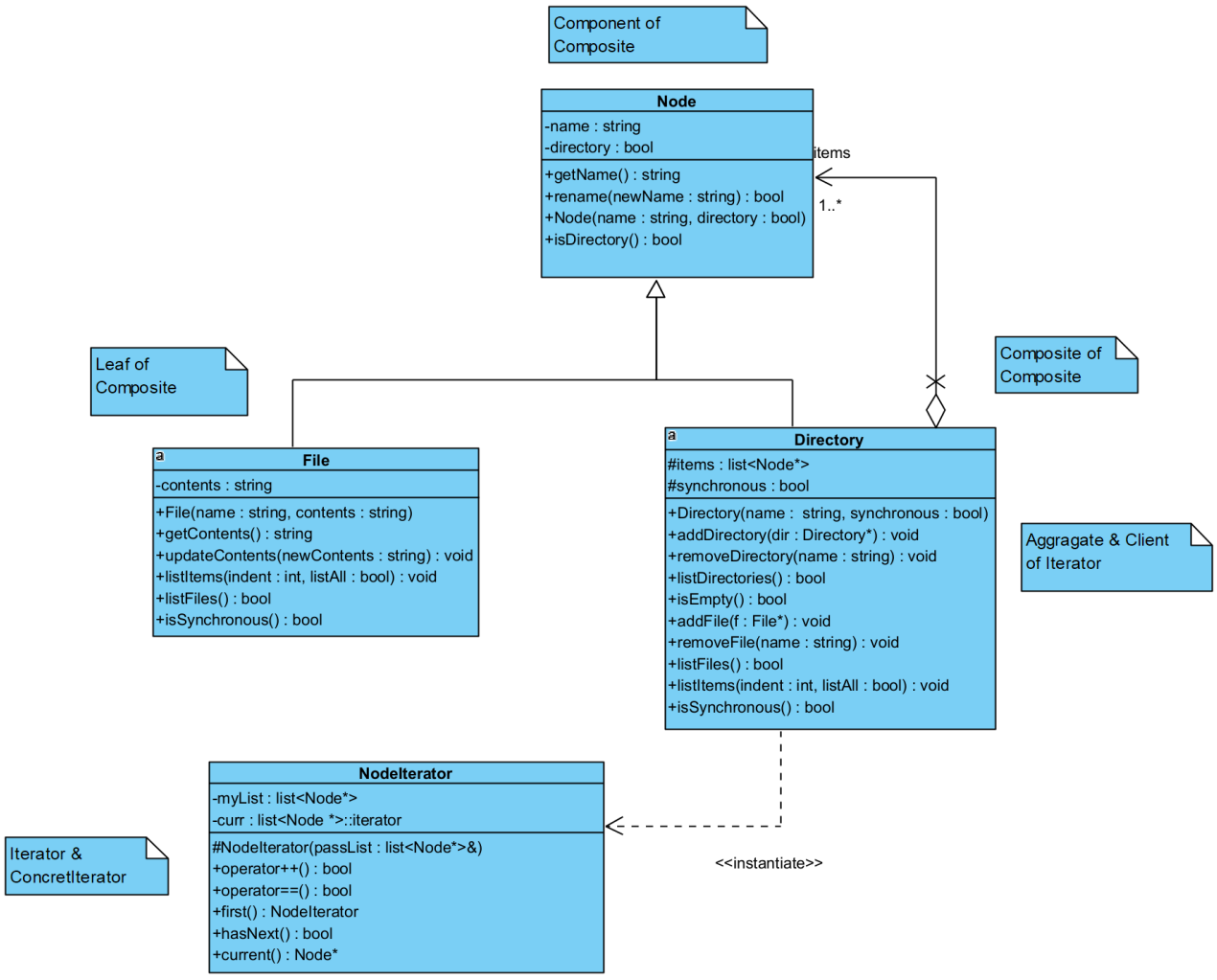
Since the list of components (files/directories) is more explicit, I think the management of nodes will be easier.



# Task 3

3.1) UML Same as above

3.3) One would use the factory method to create an abstract iterator product and two concreate products (Directory and File iterator).

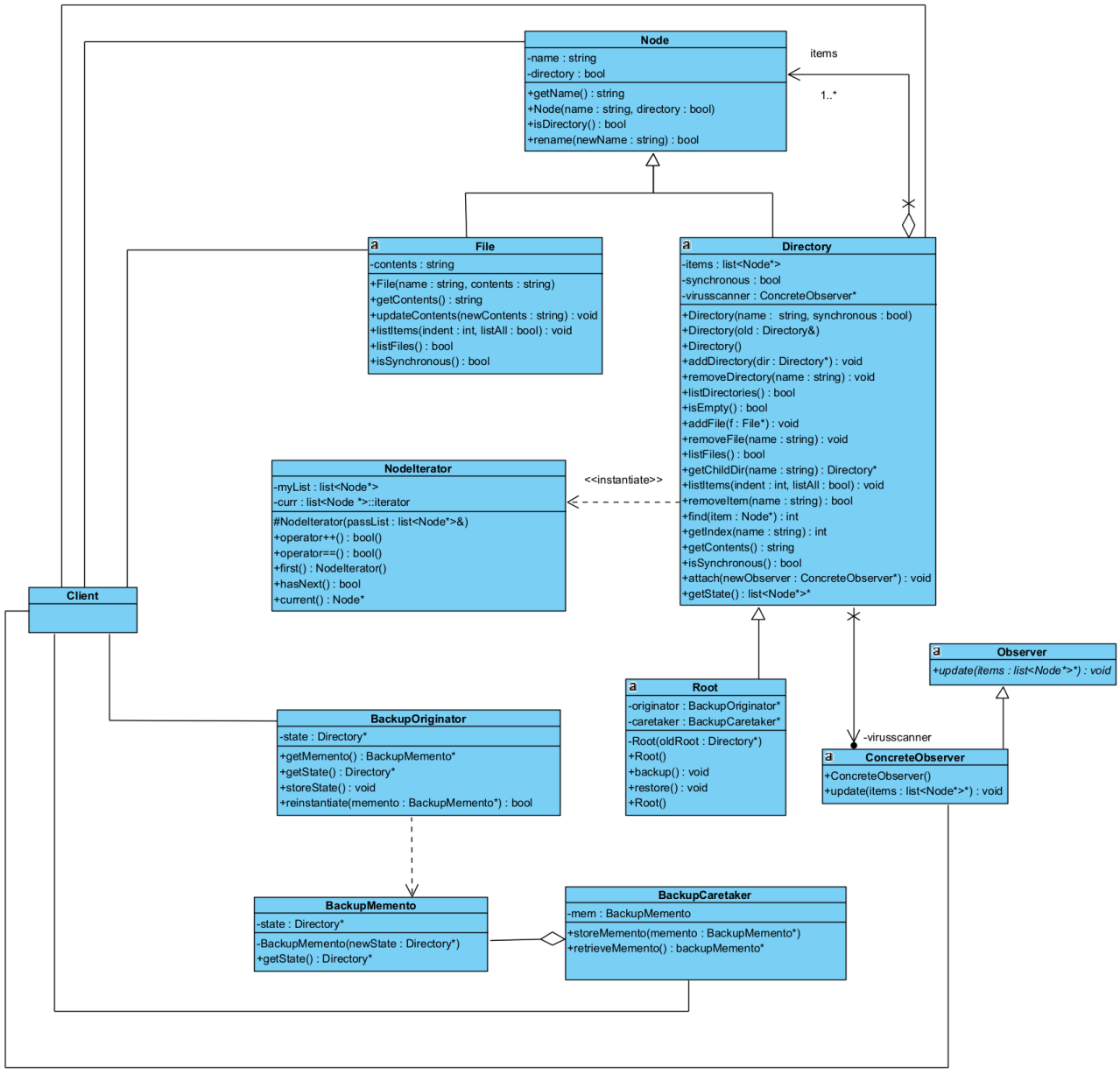



3.4) Having two different iterators means that you need covariant return types. This will make your code very complex, as you will need to check the type of the iterator before calling any operation on it.

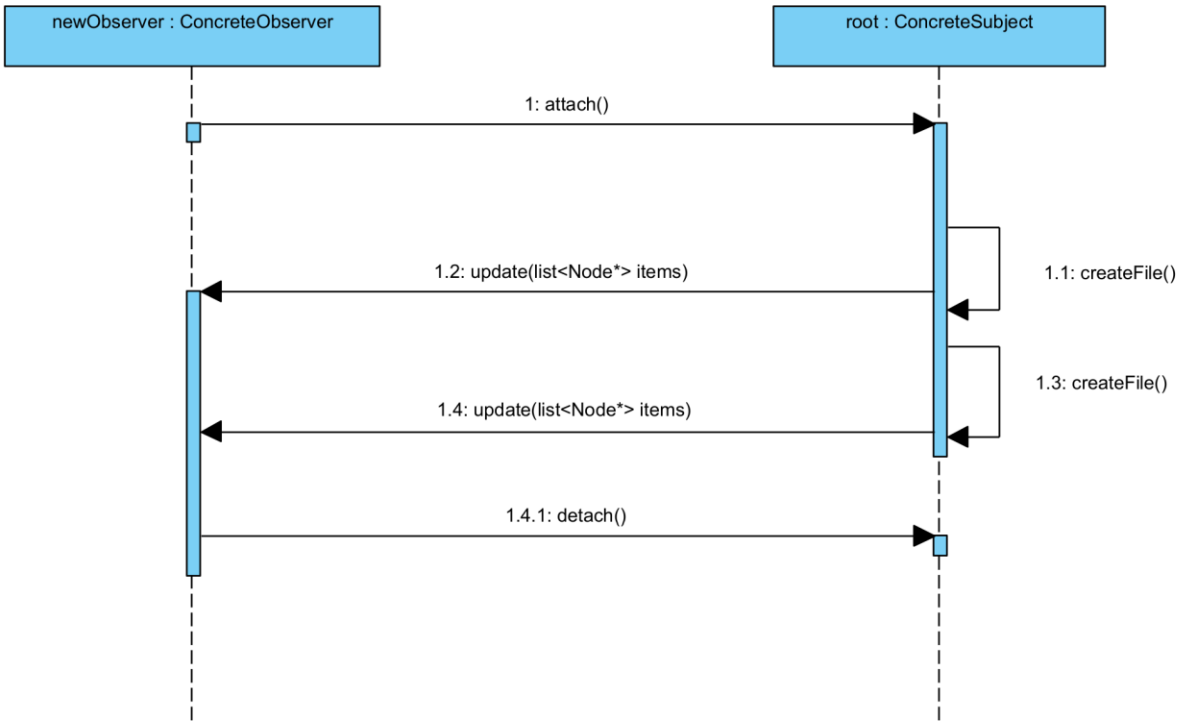
Instead, you can just return a pointer to a NodeIterator and use that instead of having two different iterators.

The approach I used, was having a single iterator for both directories and files (Node iterator), as there was no need to distinguish the two (they would be identical).

# Task 4



# Task 5


newObserver : ConcreteObserver
root : ConcreteSubject
1: attach()
1.2: update(list<Node*> items)
1.1: createFile()
1.3: createFile()
1.4: update(list<Node*> items)
1.4.1: detach()