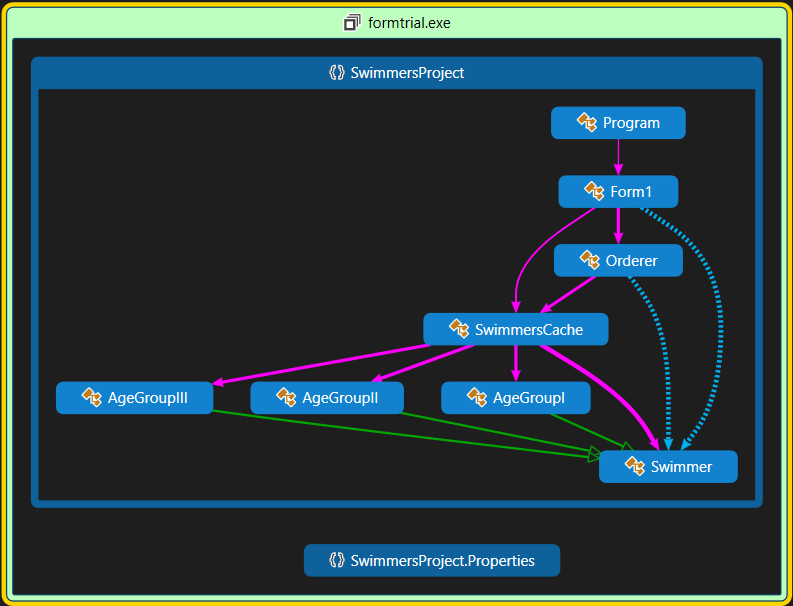
Answer to Design Questions:

1. **Swimmers Question**

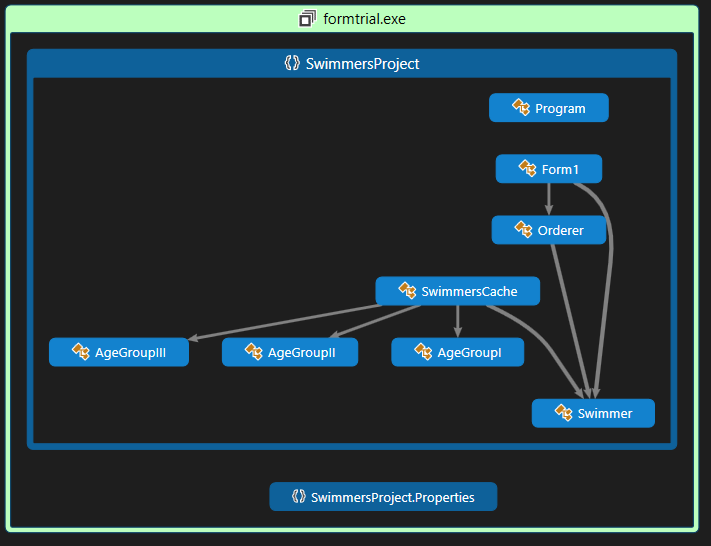
**Class Diagrams:**

**Keys** :

* + - Inherits From - ***Green***
    - Calls - ***Pink***
    - Field Read - ***Blue***
    - Reference – ***Grey***



**Diagram 1. Class Diagram of SwimmersProject Detailing Inheritance, Calls and Field Reads**



**Diagram 2. Class Diagram of SwimmersProject Detailing Reference**

**Description:**

**Pattern Used**: *Prototype Pattern*

**Reason Behind Using the Pattern:**

* *The question stated that the program should be flexible enough that it can be sorted using different parameters without destroying the original data order, in order to fulfil this requirement and by doing we can use cloned objects of the swimmers and can easily do our ordering work without affecting either the data and the order of the swimmers.*

**3.Employee Subordinate Question**

The pattern we used for question 3 is the composite pattern. The reason we selected composite pattern is because we needed to treat this group of objects (in this case, employees) in a similar way as a single object. Using this pattern we create an employee object that contains groups of its own objects (employee objects or subordinates). Every object in this hierarchy is similar save for the fact that it may or may not have subordinates. Therefore, composite pattern is appropriate to solve this problem.

Using this nature of composition we solved the cost control calculation by recursively processing the employee objects.

static void CostControl (TreeNode<EmployeeNode> employee)

{

//Add the salary of the considered employee

if (EmployeeCostControl == 0)

{

EmployeeCostControl += employee.Data.Salary;

}

//If employee has no subordinates, stop there

if (!employee.Children.Any())

{

return;

}

//Add the salary. Then recursively pass the employee to

//traverse through its subordinates

foreach (TreeNode<EmployeeNode> emp in employee.Children)

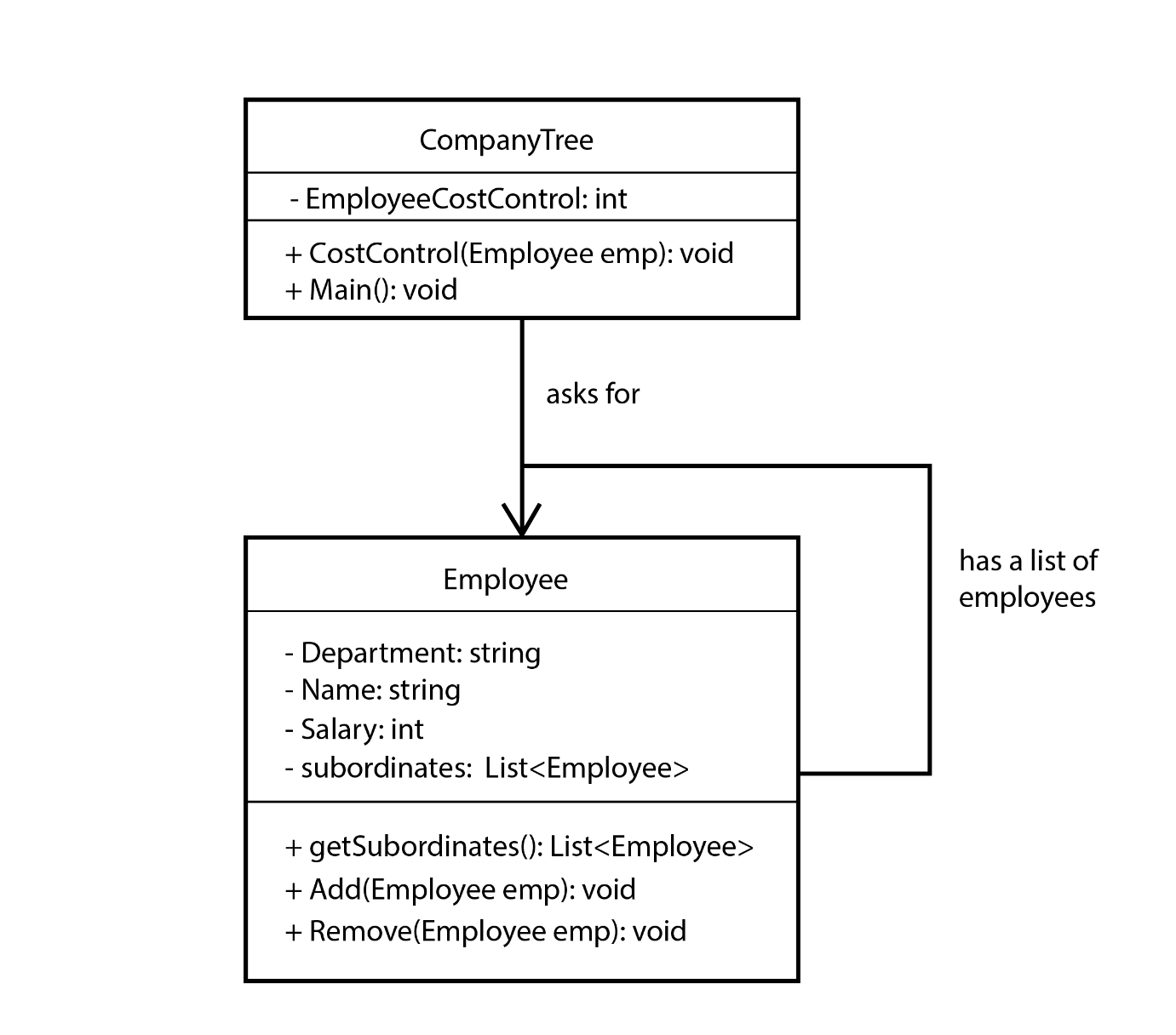
{

EmployeeCostControl += emp.Data.Salary;

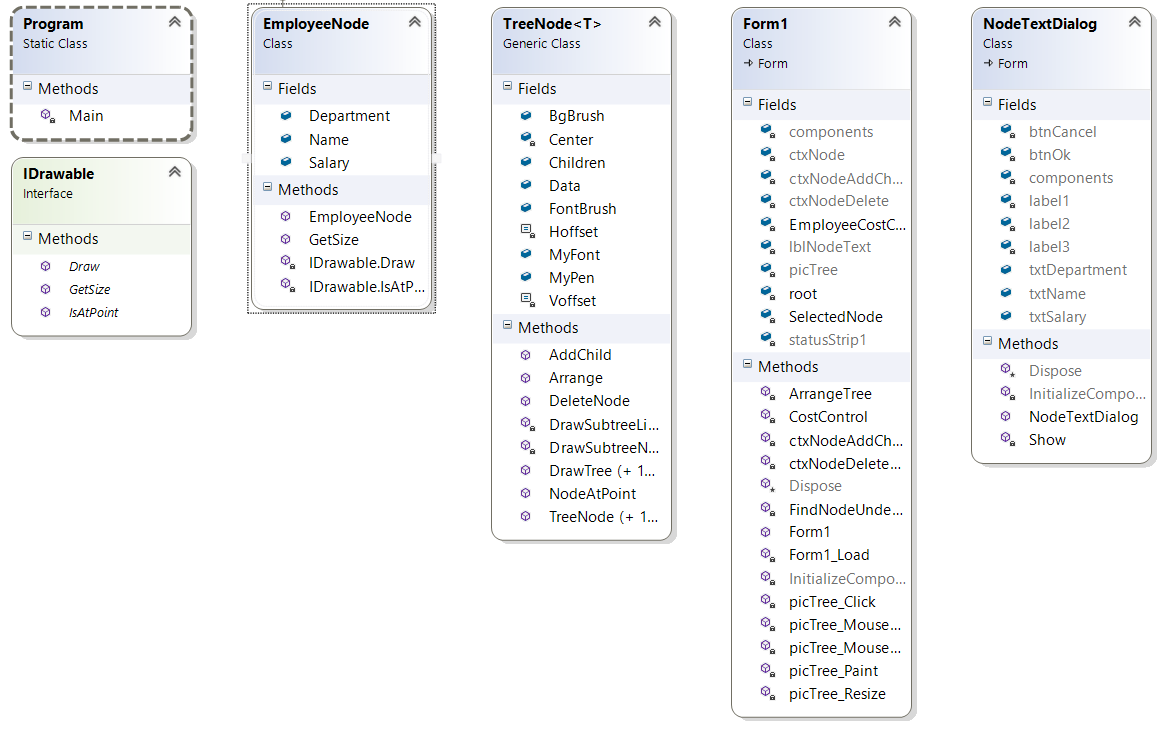
CostControl(emp);

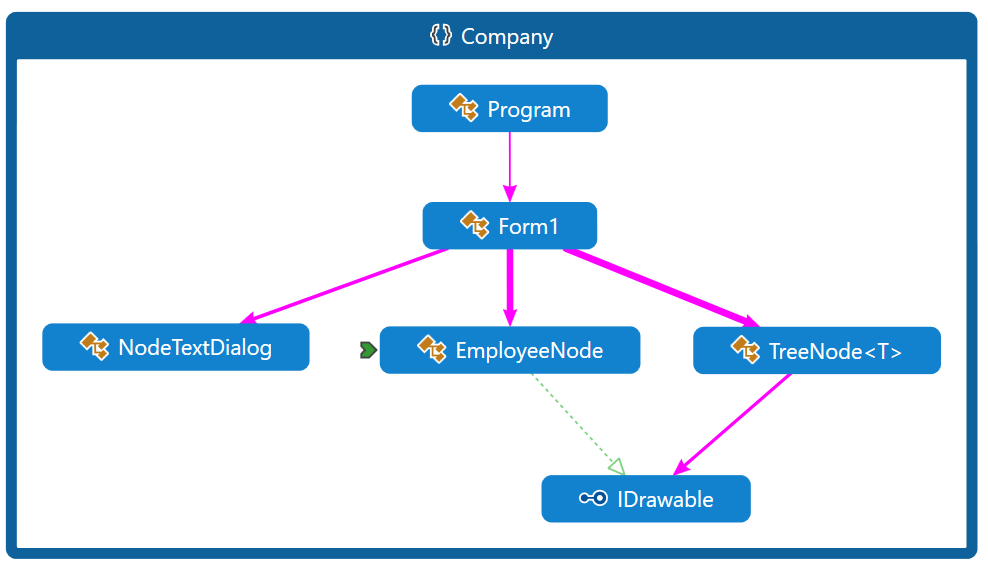
}

}

The basic structure of our class diagram is as follows:

However in order to accommodate the classes needed to construct the graphical application some additional classes are needed. Again a little amount of restructuring is needed to add some details needed to construct textboxes and such.



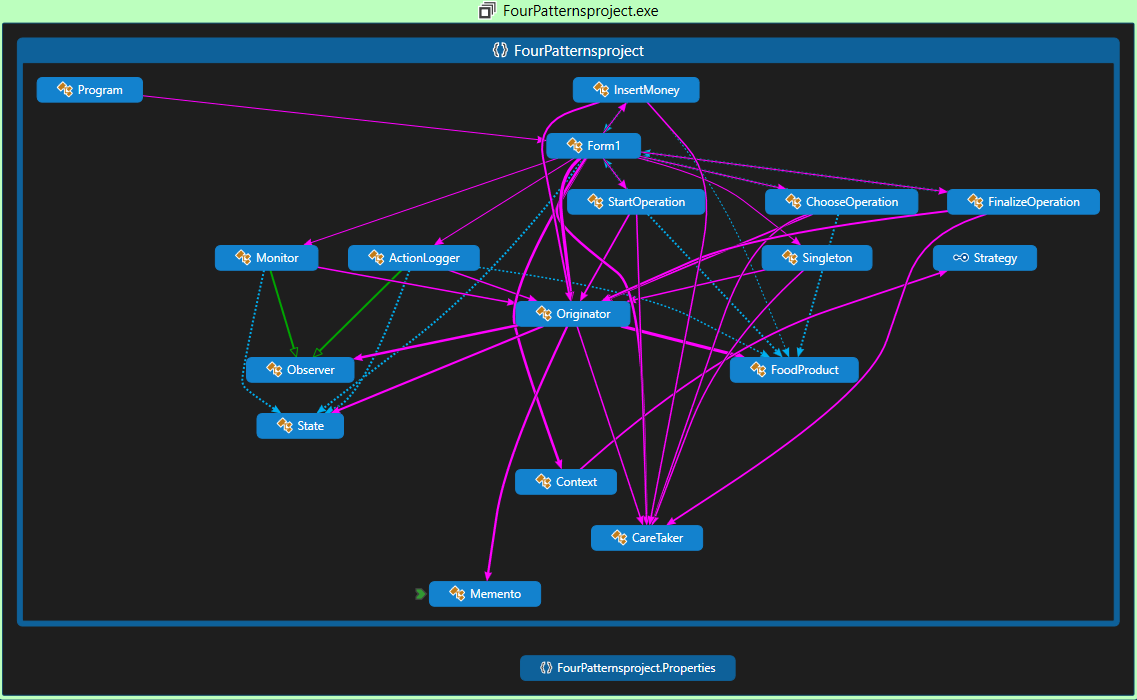


1. **Four Patterns Question**

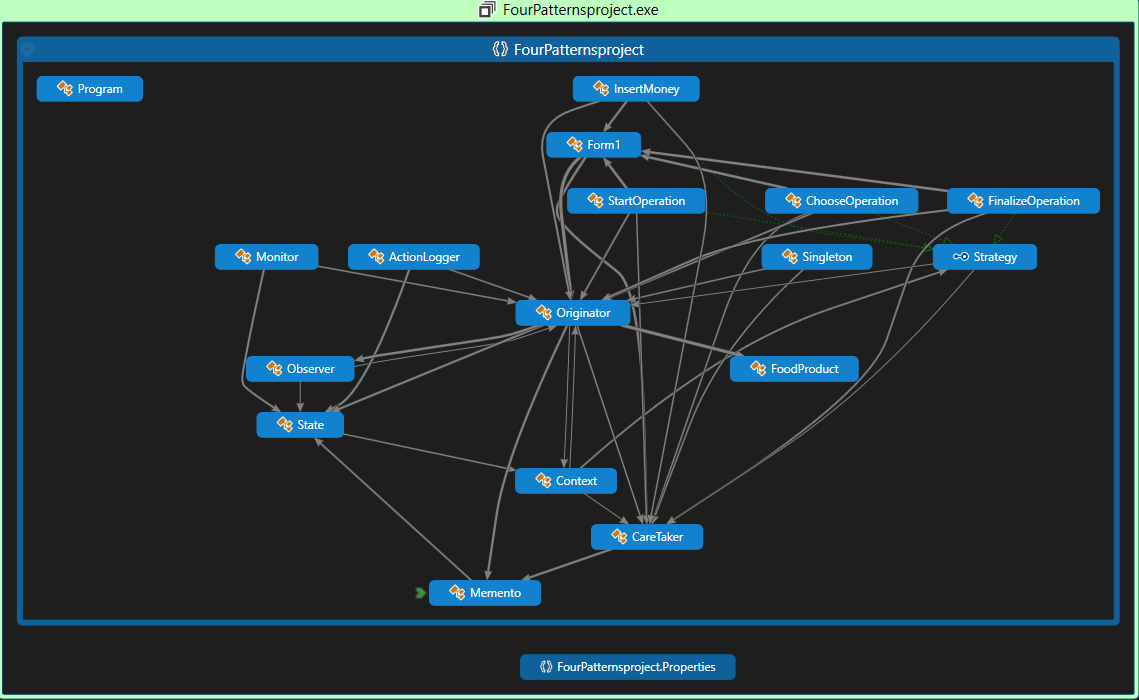
**Class Diagrams:**

**Keys** :

* + - Inherits From - ***Green***
    - Calls - ***Pink***
    - Field Read or Field Writes - ***Blue***
    - Reference – ***Grey***



**Diagram 3. Class Diagram of FourPatternsProject Detailing Inheritance, Calls and Field Reads or Field Writes**



**Diagram 4. Class Diagram of FourPatternsProject Detailing Reference and Implementation**

**Description:**

**Problem at Hand:** Currently in Ethiopia there are no vending machines other than ATMs (as far as we know). So perhaps we taught of doing the software side of a practical vending machine.

**Patterns Used**: *Singleton Pattern, Observer pattern, Memento Pattern and Strategy Pattern*

**Reason Behind Using the Patterns:**

* *Starting with the singleton pattern, it was necessary that we used this pattern to only allow one instance of the serving class to be instantiated which where the originator class and the caretaker class.*
* *The Observer pattern was necessary because the vending machine uses a logging and monitoring service. In order to implement this services we used this pattern in which actionlogger and monitor are the observer classes and the originator is the subject.*
* *The Memento pattern was also used to save and initialize states that the machine was able to undergo.*
* *The Strategy pattern was also used to store and define actions for start, select, back and pay buttons in the UI. They where initialised and part of the memento pattern being referred as fields.*