This is the evaluation of the code with respect to principles and design of OOP:

1. **Encapsulate what varies**: Encapsulation is like bundling of methods and variables. It is also a method of restricting direct access of variables to the user. In our program, we require direct access of variables. We could have provided encapsulation using the getter function to prevent variable access to other classes directly. In the Grid class, we have used encapsulation. The variables mentioned in Grid Class isn't accessible to any other class. The methods are mostly public or default, since they are required by other classes.

2. **Program to an interface not implementation**: We have not used interfaces in our program since it was more than what we wanted not needed. We used loops to control the multiple execution of our program. The balance in our program is achieved without use of interfaces. Since our program is a game where we Generate and Solve Sudoku, the interface isn't required here. By fiddling with variables, the Sudoku 9X9 matrix can be changes to 16X16 or 25X25, but it will require some in depth changes also.

3. **Depend on abstraction, do not depend on concrete classes:** We are using "concrete" classes in our program and not using abstraction. basically abstraction serves as a layer to reduce the complexity of the code such that irrelevant information is hidden. But we have used concrete classes since we had to use objects of these classes in other classes for execution. Hence changing one of the component in any class will also change the behaviour of other class that uses that component.

4. **Favour Composition over inheritance**

   We have used objects of VerifySudoku class, grid class, etc in the program to obtain the features of these classes without having the need to inherit them. This has helped the program to use the features of these classes without being confined to inherit them. The Generator class has used the object of grid class to make the grid in the console. We also used the VerifySudoku object to check if sudoku is valid or not.

5. **Strive of Loose Coupling between Objects that interact**.
   The objects of different classes have very divergent properties and functions. The only thing they share in the common is the basic sudoku matrix required by all the classes and the order of the matrix. Despite that, none of the classes have conflicting functions or tend to interfere with the other class variables.

6. **Classes should be open for extension but closed for modification**
   The required number of variables have been kept as minimum as possible and all the methods are tightly integrated with each other. Any changes to the methods have a very high chance of breaking the program. On the other hand, the program has the flexibility to scale up or down can be extended to solve sudoku of various sizes by making small changes to the program code.

DESIGN PATTERNS:

1. We haven't used interfaces for our program and have decided the class object as per our needs, hence the **strategy design pattern** is not used by us. Instead, every time the user choses to play a new game a new Generator, VerifySudoku, Solver, GUIFrame and Grid object is created.

2. We have extended the VerifySudoku class to Generator class and Solver class since both Solver and Generator uses some functions of VerifySudoku class. This is a case of inheritance. But, we have also used Grid class without extending it to Solver or Generator class. For the Grid class, it's object is created inside the object of other classes wherever it uses. Also, in SudokuBoard class, we have used NineSquare object. Hence we, have followed **the decorator design pattern with respect to certain classes.**

3. Though out classes our loosely coupled with most cases, only the Input or the half-Sudoku matrix and the order of that matrix is passed on to other objects. As the input matrix or it's order changes, other classes behaviour also changes with it. Like, if we change the order of matrix in the Main function, the Grid class also adjusts itself to make the matrix with respect to it. Hence the Grid class observes the variable order of matrix in the main class. Hence, **the observer design pattern is applicable for certain cases**.

4. We are well aware of the type of object that we are creating, since we are creating them only when needed. Hence, **the factory design patterns are not applicable**.

5. We have not used the builders design pattern in the project. All the classes are built on their own with at most one level of inheritance. We could have used the design approach to create the sudoku matrix in more comprehensive way with the main matrix that inheriting the sub matrices.

6. In command design pattern, encapsulate the request as an object and passes it to the invoker object to invoke the appropriate object according to the query. Since the queries in the program were rather simple and did not required multiple types of inputs, this design pattern was not used in the program.