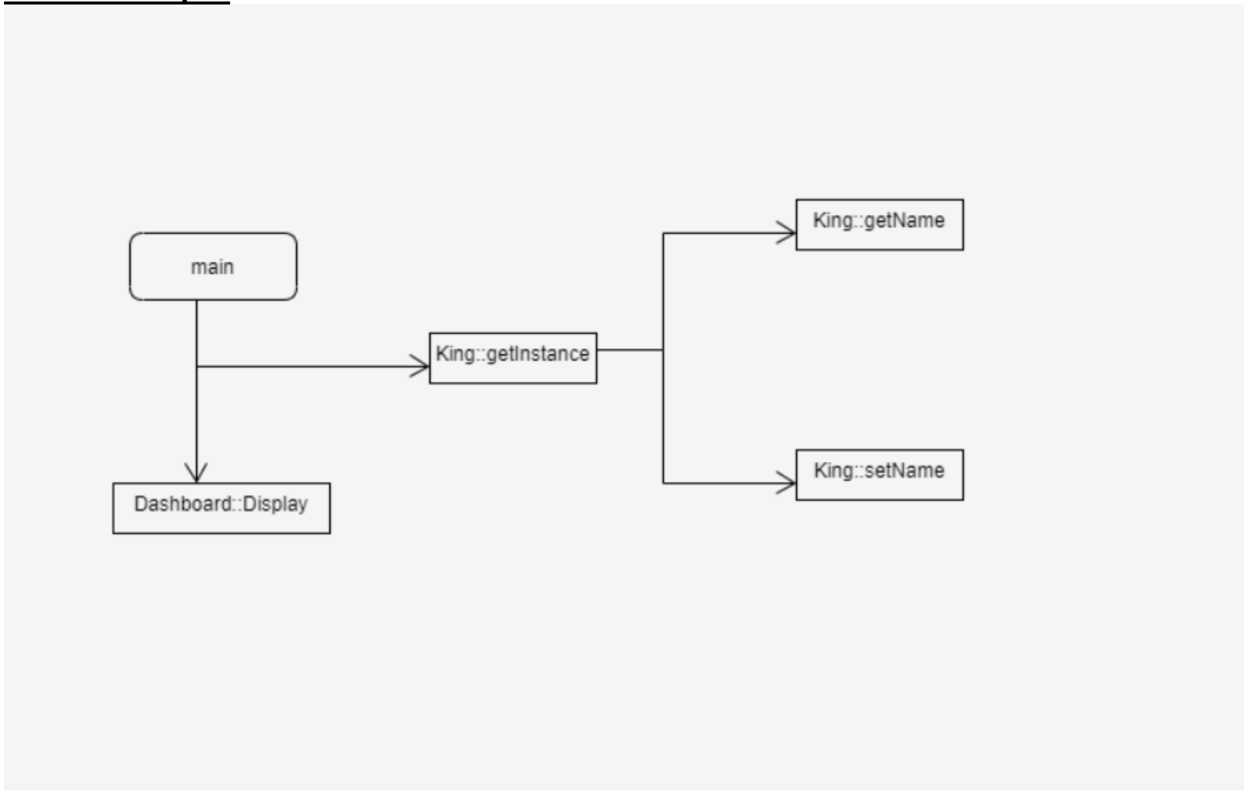
Report-1

Section 1 (Specifications):

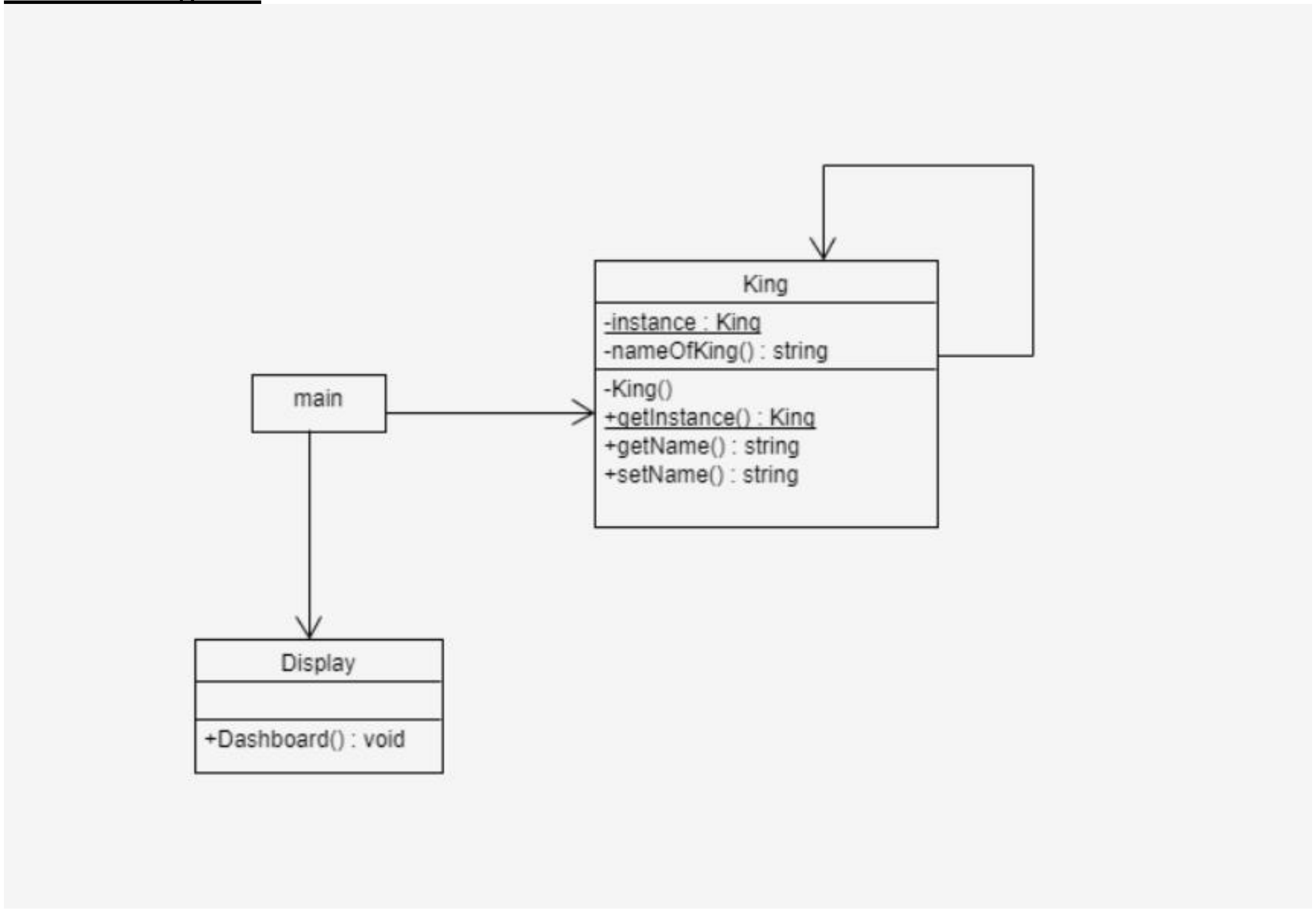
In 1st Scenario, I select the 2nd case. In this case when people promoted themselves as king chaos created. I had to reprogram a virtual kingdom where only one king could exist at any point of time even if people promoted and called themselves Kings. And a new King can be in place only if the existing King died or lost his kingdom to another king in a war. For this case I selected Singleton design pattern. The way I have used this design pattern is, since this pattern involves a class which is responsible for creating an object. I created a class called King which creates an object while making sure that only a single object gets created. This way there will exist only one king in this program. This class also provides a way to access its only object which can be accessed directly without need to instantiate the object of the class. This object can be accessed directly under some conditions. Conditions are if the previous king died or lost his kingdom at war. If the condition is fulfilled then the king object can be instantiated again. That means the king will be replaced.

Section 2 (Design):

2.1 Call Graph:

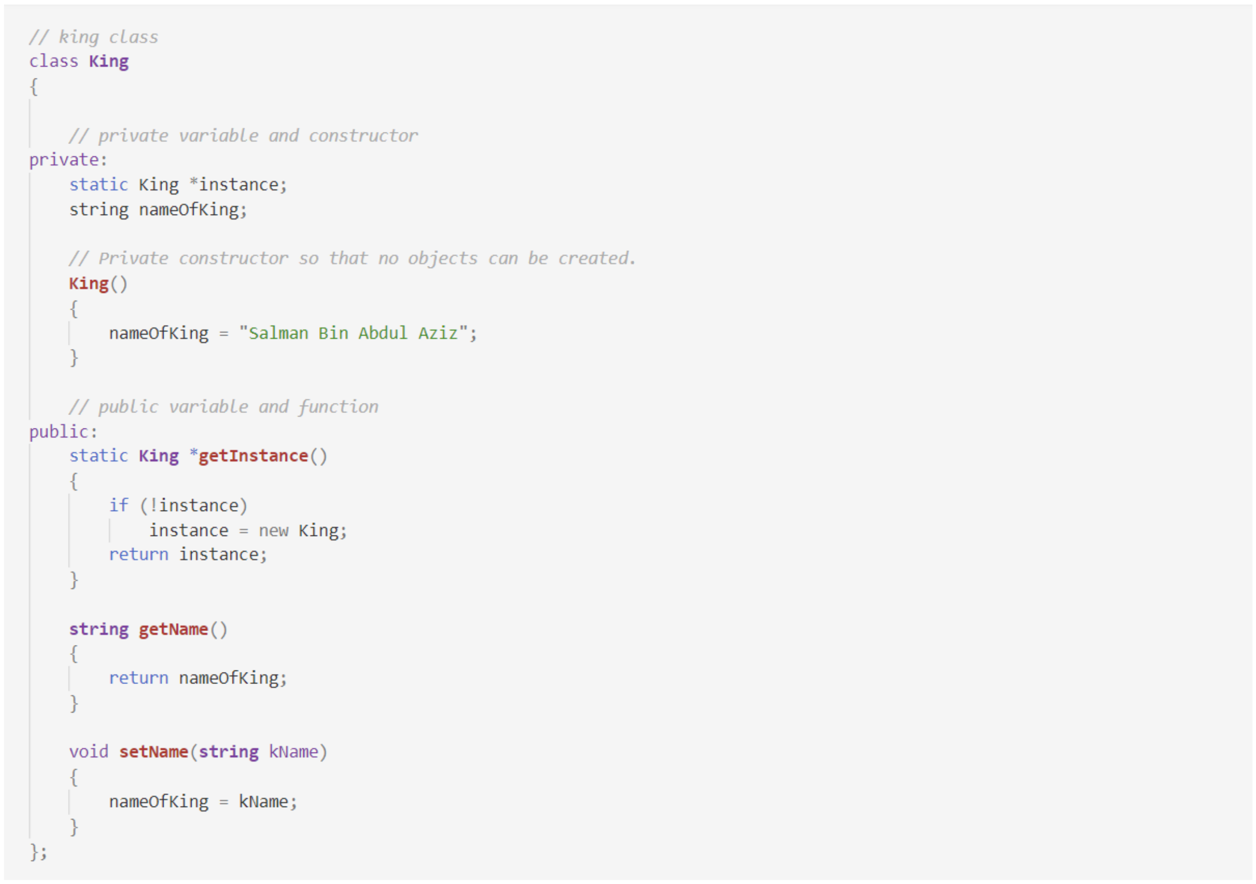


2.2 UML Diagram:

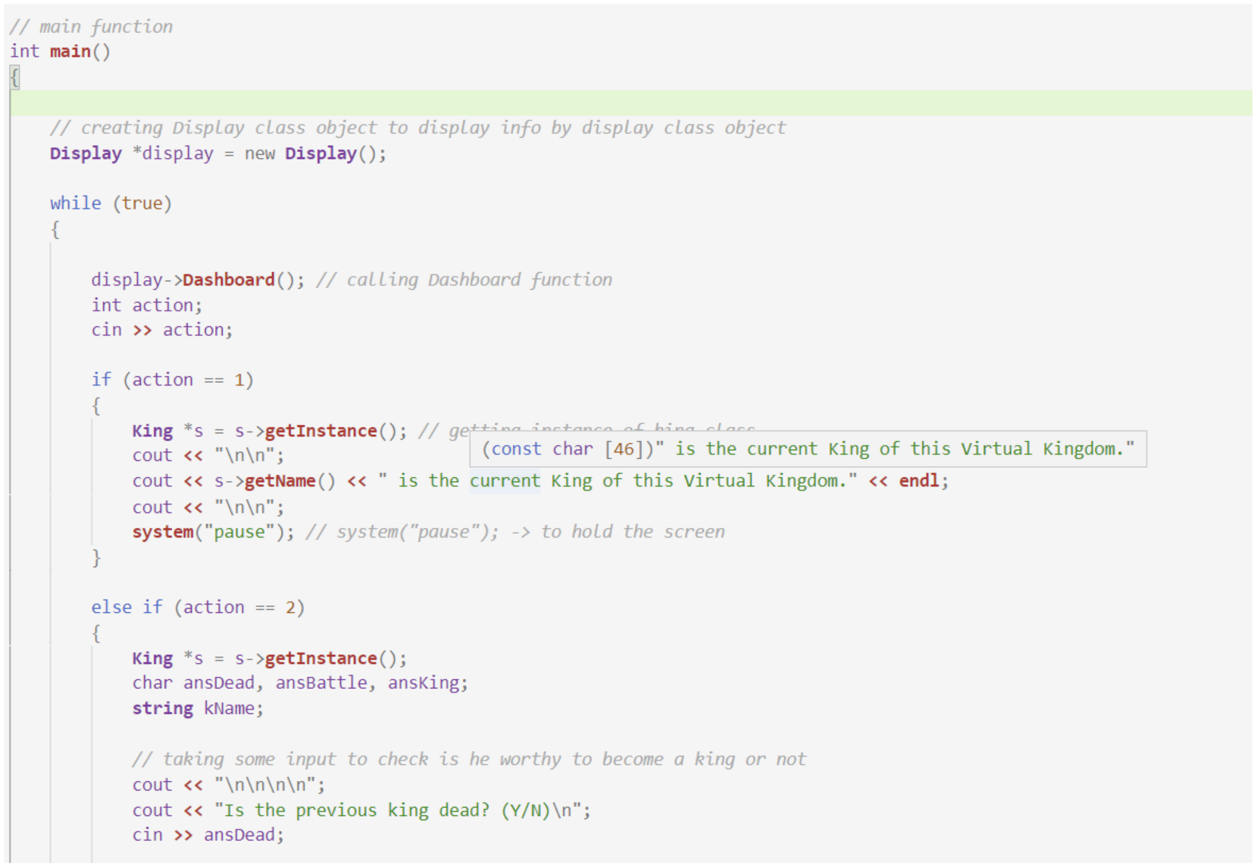


2.3 Implementation:

```
// Display class to print some info
class Display
{

public:
    // dashboard function
    void Dashboard()
    {
        system("CLS");
        cout << "--------------------VIRTUAL KINGDOM--------------------\n\n";
        cout << "1. Who is the current king ?" << endl;
        cout << "2. Promote yourself to become a King " << endl;
        cout << "3. Exit" << endl;
        cout << "Choose your action: ";
    }
};
```

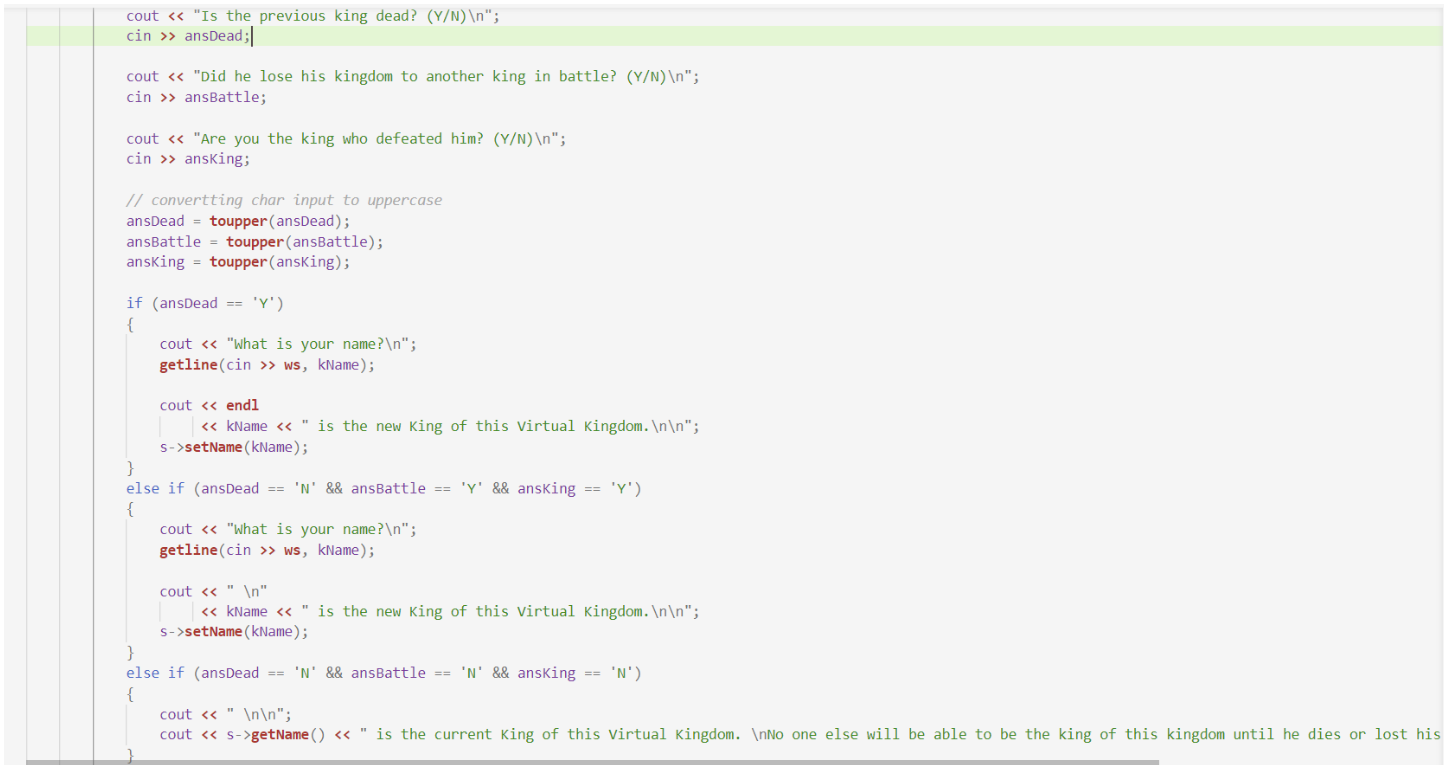


```cpp
// king class
class King
{

    // private variable and constructor
private:
    static King *instance;
    string nameOfKing;

    // Private constructor so that no objects can be created.
    King()
    {
        nameOfKing = "Salman Bin Abdul Aziz";
    }

    // public variable and function
public:
    static King *getInstance()
    {
        if (!instance)
            instance = new King;
        return instance;
    }

    string getName()
    {
        return nameOfKing;
    }

    void setName(string kName)
    {
        nameOfKing = kName;
    }
};
```

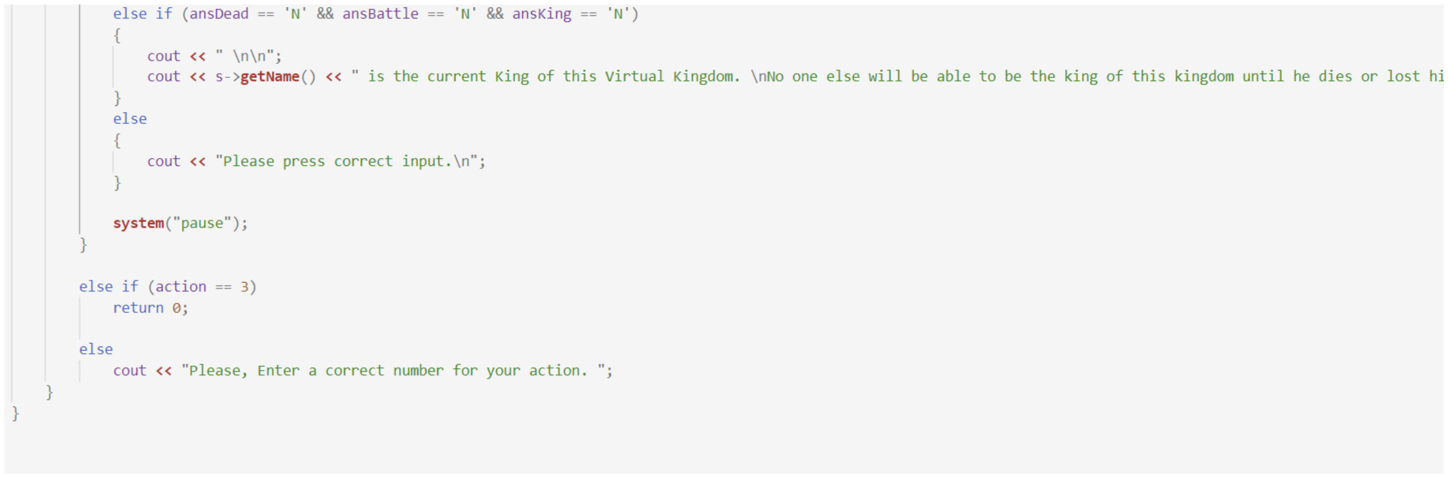


```cpp
// main function
int main()
{

    // creating Display class object to display info by display class object
    Display *display = new Display();

    while (true)
    {

        display->Dashboard(); // calling Dashboard function
        int action;
        cin >> action;

        if (action == 1)
        {
            King *s = s->getInstance(); // getting instance of king class
            cout << "\n\n";
            cout << s->getName() << " is the current King of this Virtual Kingdom." << endl;
            cout << "\n\n";
            system("pause"); // system("pause"); -> to hold the screen
        }

        else if (action == 2)
        {
            King *s = s->getInstance();
            char ansDead, ansBattle, ansKing;
            string kName;

            // taking some input to check is he worthy to become a king or not
            cout << "\n\n\n\n";
            cout << "Is the previous king dead? (Y/N)\n";
            cin >> ansDead;
```

```
(const char [46])" is the current King of this Virtual Kingdom."
```



```
        cout << "Is the previous king dead? (Y/N)\n";
        cin >> ansDead;

        cout << "Did he lose his kingdom to another king in battle? (Y/N)\n";
        cin >> ansBattle;

        cout << "Are you the king who defeated him? (Y/N)\n";
        cin >> ansKing;

        // convertting char input to uppercase
        ansDead = toupper(ansDead);
        ansBattle = toupper(ansBattle);
        ansKing = toupper(ansKing);

        if (ansDead == 'Y')
        {
            cout << "What is your name?\n";
            getline(cin >> ws, kName);

            cout << endl
                 << kName << " is the new King of this Virtual Kingdom.\n\n";
            s->setName(kName);
        }
        else if (ansDead == 'N' && ansBattle == 'Y' && ansKing == 'Y')
        {
            cout << "What is your name?\n";
            getline(cin >> ws, kName);

            cout << " \n"
                 << kName << " is the new King of this Virtual Kingdom.\n\n";
            s->setName(kName);
        }
        else if (ansDead == 'N' && ansBattle == 'N' && ansKing == 'N')
        {
            cout << " \n\n";
            cout << s->getName() << " is the current King of this Virtual Kingdom. \nNo one else will be able to be the king of this kingdom until he dies or lost his
        }
```



```
            else if (ansDead == 'N' && ansBattle == 'N' && ansKing == 'N')
            {
                cout << " \n\n";
                cout << s->getName() << " is the current King of this Virtual Kingdom. \nNo one else will be able to be the king of this kingdom until he dies or lost hi
            }
            else
            {
                cout << "Please press correct input.\n";
            }

            system("pause");
        }

        else if (action == 3)
            return 0;

        else
            cout << "Please, Enter a correct number for your action. ";
    }
}
```

Section 3 (Discussion):

The chosen design pattern is the Singleton design pattern. Which is a creational design pattern. This singleton design pattern is best for this case among all other design patterns. The Singleton design pattern solves two problems at the same time. One is it Ensure that a class has just a single instance. And another is it Provide a global access point to that instance. In this case, since the virtual kingdom has to be allowed only one king to exist at any point of time even if people promoted and called themselves Kings. That means there needs to be such a design pattern where a class could have only one instance. And another requirement is that people should know the name of the current king, whether he is dead or not or if he lost his kingdom to another king in a war. For that requirement there also needs such a design pattern whose instance of a class could be accessed globally. The singleton design pattern meets both requirements. It has a class which creates an object while making sure that only a single object gets created, also this object can be accessed globally. There are also some features. This singleton design pattern makes the default constructor private, to prevent other objects from using the new operator with the singleton class. It creates a static method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object. If your code has access to the singleton class, then it’s able to call the singletons static method. So, whenever that method is called, the same object is always returned. This singleton design pattern also carries singleton object around for the lifetime of the application.

The solid principles of OOP are Single-responsibility-Principle, Open-closed-principle, Liskov-Substitution-principle, Interface-segregation-principle and Dependency-Inversion-Principle. Though the singleton design pattern meets with the Liskov-Substitution-principle and Interface-segregation-principle, it does not fulfill the Open-closed principle and Dependency-Inversion-Principle.