SOLID Principles in Software Architecture and Introduction to RESM Concept in OOP

Article *in* Journal of Engineering Science and Technology · March 2015

CITATIONS
0

READS
90

3 authors:

Vamsi Madasu
University of Bridgeport
4 PUBLICATIONS **1** CITATION
SEE PROFILE

Trinadh Venkata Swamy Naidu Venna
University of Bridgeport
2 PUBLICATIONS **3** CITATIONS
SEE PROFILE

Tarik Eltaeib
University of Bridgeport
25 PUBLICATIONS **7** CITATIONS
SEE PROFILE

Journal of Multidisciplinary Engineering Science and Technology (JMEST)
ISSN: 3159-0040
Vol. 2 Issue 2, February - 2015

SOLID Principles in Software Architecture and Introduction to RESM Concept in OOP

Vamsi Krishna Madasu
Department of Computer Science
University of Bridgeport
Bridgeport, CT-06604, USA
vmadasu@my.bridgeport.edu

Trinadh Venkata Swamy Naidu Venna
Department of Computer Science
University of Bridgeport
Bridgeport, CT-06604, USA
tvenna@my.bridgeport.edu

Tarik Eltaeib
Department of Computer Science
University of Bridgeport
teltaeib@my.bridgeport.edu

Abstract- **The article SOLID Principles in Software Architecture and Introduction to RESM concept in OOP gives an outlook of the SOLID principles and to the concepts of Reusability, Extensibility, Simplicity and Maintainability (RESM) in Object Oriented Programming.**

Keywords- SOLID, OOP, Reusability, Extensibility, Simplicity , Maintainability

I. INTRODUCTION

SOLID are the five basic principles which help in creating good software architecture. SOLID is an acronym where

S stands for SRP (Single Responsibility Principle)

O stands for OCP (Open Closed Principle)

L stands for LSP (Liskov Substitution Principle)

I stand for ISP (Interface Segregation Principle)

D stands for DIP (Dependency Inversion Principle).

Using these SOLID principles we can build efficient, reusable and non-fragile software which is sustainable and maintainable for the long term needs. Reusability, Extensibility, Sustainability and Maintainability (**RESM**) are the major issues concerned with the functional programming. To overcome these issues and to build dynamic software we need SOLID principles. SOLID principles gave a suitable answer to develop an efficient Software architecture that can overcome **RESM** problems.

II. EXPLINATION

S-SRP (Single Responsibility Principl

```
class Customer
{
public:
    void Add()
    {
        try
        {
            //Code to add customer to the Database
        }
        catch (exception ex)
        {
            //Code to Catch the exception and add to the Log file
        }
    }
};
```

Fig.1

In the code shown in figure.1 the Customer class has an Add function which adds customers to the Database. There is a problem in this code. The Add function adds customers to the database and at the same time it logs the exception in a log file, which the Customer class is not supposed to do. So SRP says that the class should have only one responsibility and not multiple.

```
class Customer:Logfile
{
    Logfile obj;
public:
    void Add()
    {
        try
        {
            //Code to add customer to the Database
        }
        catch (exception ex)
        {
            obj.Handle("excpetion");
        }
    }
};
```

Fig.2

```
class Logfile
{
public:
    void Handle(string error)
    {
        //Code to Catch the exception and add to the Log file
    }
};
```

Fig.3

The code shown in Fig.2 and Fig.3 give the solution where we have created a separate Logfile class to record the exceptions and we have inherited the customer class from Logfile.

O- OCP (Open Closed Principle)

Open Close Principle says that a class should be closed for modification and should be open for Extension. For example if the customer class has Gold customers, silver customers. In case in future if we want to add a new customer type then we have to modify the customer class and we have to test the functionality again. Instead we can make the class should be closed for modification and should be open for extension. In other words rather than modifying the class we go for extension.

```
class Shape
{
  public:
    virtual void Draw() const = 0;
};

class Square : public Shape
{
  public:
    virtual void Draw() const;
};

class Circle : public Shape
{
  public:
    virtual void Draw() const;
};

void DrawAllShapes(Set<Shape*>& list)
{
  for (Iterator<Shape*>i(list); i; i++)
     (*i)->Draw();
}
```

Fig.4

The code shown in the Fig.4 gives the solution of the square/circle problem, in which square and circle are both child classes derived from Shape class. But they have pure virtual function called Draw() which is open to use but closed for modification.

L-LSP (Liskov Substitution Principle)

This is simply an extension to OCP. Liskov Substitution Principle says that parent should easily replace the child. A parent class object should be able to refer the child class objects. A child class can be derived to use the functionality of the parent class but it should be a replacement of its parent class. If the child class also does the same function of a parent class. Then we say that it is violating LSP

I-ISP (Interface Segregation Principle)

Interface Segregation Principle states that client should not be forced to use an interface if it does not need it. Clients should only know about the methods or interface which are interested to them. Robert Martin formulated this principle when creating software for a Xerox. While developing the software, even a minor change would take redeployment of an hour. Because, the classes that are connected to the print class could see the methods of it.

D-DIP (Dependency Inversion Principle)

Dependency Inversion Principle states that

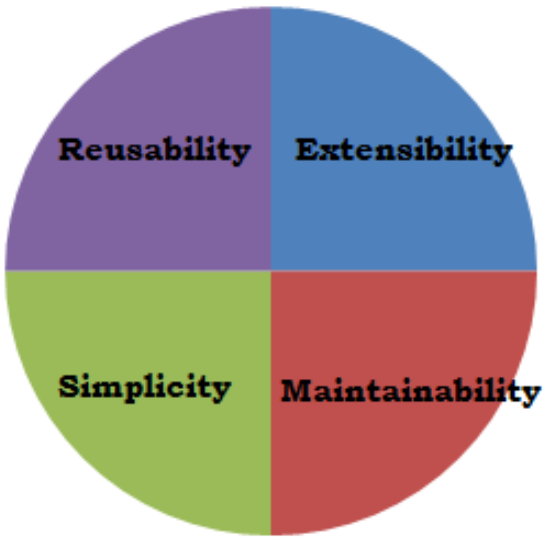
1) High level modules should not depend on low level modules. Both should depend on abstractions

2) Abstractions should not depend on details. Details should depend on abstractions.

The principle of dependency inversion is at the root of many of the benefits claimed for object-oriented technology. Its proper application is necessary for the creation of reusable frameworks.

RESM Concept in OOP

The main problem with the functional programming is Reusability, Extensibility, Simplicity and Maintainability. To overcome these problems we have introduced Reusability, Extensibility, Simplicity and Maintainability of the code so that we can develop software that is sustained for longer period of time.



Reusability: In functional programming the code of a function can't be reused in different parts of the program. To over this problem object oriented programming uses classes and objects. Rather than using functions we can use make use of objects.

Extensibility: The other major problem with functional programming is Extensibility. Whenever we declare a class the scope of the functions is limited to that class only. To over this issue object oriented uses inheritance concept, where it uses parent child relationship.

Simplicity: Object Oriented Programming uses polymorphism concept to main the simplicity of the code.

Maintainability: In software Industry, good software is which can sustain and maintain for a long time. Combining Reusability, Expansibility and Simplicity we can build good software that can sustain and maintain for a longer period of time.

Conclusion:

Finally, We want to conclude that using SOLID principles and RESM concept we can build a Software which can be sustainable in the present competitive business world.

References:

[1]Shivprasad Koirala's article on SOLID Principles in codeproject.com