International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 7, Issue 6, June 2017)

Refactoring for High Cohesiveness in Robust Software Development

J. Vamshi Vijay Krishna
CVR College of Engineering, Dept. of IT, Hyderabad, India

***Abstract*— Great software must satisfy the customer. The one constant in software is *change*. With good use-cases, the existing system can be changed to accommodate the new requirements. Software that is not well-designed falls apart at the first sign of change, but great software can change easily. Objected Oriented Principles helps in writing robust software that is well-designed, well-coded and easy to maintain, reuse and extend. Robust programming prevents abnormal termination or unexpected actions. Striving for high cohesion is one of the key principles in designing a robust software. Favoring Delegation, Composition and Aggregation over Inheritance makes software more robust. Refactoring in a context is sometimes more favorable than delegation, composition and aggregation.**

***Index Terms*—Robust Programming, Design Patterns, Cohesion, Object Oriented Programming, Refactoring**

I. Introduction

Great software always does what the customer wants it to. It is well-designed, well-coded and easy to maintain, reuse and extend [1]. So even if customers think of new ways to use the software, it doesn't break or give them unexpected results. The key ingredient of Robust Programming is that the program doesn't break or give unexpected results [2][3]. Object Oriented principles helps in designing software that is more flexible and extensible. Some key Object Oriented design principles are - encapsulate what varies; code to an interface rather than an implementation; classes should be open for extension and closed for modification; every object should have a single responsibility and all the object's responsibility should be focused on carrying on that single responsibility [1]. Single Responsibility Principle facilitates high cohesion. Modules which are highly cohesive in nature makes a great software.

Robust programming is a style of programming that prevents abnormal termination or unexpected actions. It requires code to handle bad (invalid or absurd) inputs in a reasonable way. Robust programs generally need to deal with 3 kinds of exceptional conditions: user errors - when invalid input is passed to the program; exhaustion - when program tries to acquire shared resources; internal errors - due to bugs.

Robust programming is defensive and this defensive nature protects the program not only from those who use our application, but also from ourselves. Good programming assumes this, and take steps to detect and report those errors, internal as well as external. A robust program differs from a non-robust program by adherence to the following four principles – paranoia, stupidity, dangerous implements, and can't happen principle [2][3].

Design patterns directly doesn't go into code. They first go into our brain. Once we have loaded our brain with a good working knowledge of patterns, we can then start to apply them to our new designs, and rework our old code which is fragile [4]. Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps prevent subtle issues that can cause major problems and improve code readability for coders and architects familiar with the patterns. Common design patterns can be improved over time, making them more robust than ad-hoc designs. Three categories of design patterns are Creational, Structural and Behavioral. Creational design patterns can be divided into class- creation and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done. Structural design patterns are about class and object composition which defines ways to compose objects to obtain new functionality. Behavioral design patterns are more specifically concerned with communication between objects.

A cohesive module does one thing well and doesn't try to do or be something else. It measures the degree of connectivity among the elements of a single module, class or object. The higher the cohesion in software, the more well-defined and related the responsibilities of each individual module/class/object in application. Each module has a very specific set of closely related actions it performs. Single Responsibility Principle facilitates high cohesion which in turn makes software more flexible, extensible and re-usable.

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 7, Issue 6, June 2017)

Refactoring is the process of modifying the structure of code without modifying code's behavior [5][6]. Refactoring is done to increase the cleanness, flexibility of code and usually is related to a specific improvement in software design [7].

II. Case Study

It's not always possible to design modules which are highly cohesive in nature. For example, consider a scenario of reading 10 integer numbers from keyboard and then displaying them on screen instead of writing to a file (to make the example easily understandable). Fig–1 has IntegerDisplayTestDrive.java program which reads 5 integers from keyboard using java.util.Scanner and displays the output on screen.

```
apple:ijetae apple$vim IntegerDisplayTestDrive.java

import java.util.Scanner;

public class IntegerDisplayTestDrive{

   public static void main(String[] args){

      int[] numbers = new int[5];
      Scanner sc = new Scanner(System.in);

      System.out.println( "¥nTo Read 5 Integers & Display
on Screen¥n" );

      System.out.println( "¥nEnter 5 Integer Elements¥n" );
      for(int index = 0; index < 5; ++index)
         numbers[index] = sc.nextInt();

      System.out.printf("¥n5 Integers are¥n");
      for(int index = 0; index < 5; ++index)
         System.out.printf( "%4d" ,numbers[index]);
   }
}
```

Fig – 1: Java program to read 5 Integers using java.util.Scanner

Fig – 2 shows the compilation process and the output of the program without typos.

```
apple:ijetae apple$javac IntegerDisplayTestDrive.java
apple:ijetae apple$ java IntegerDisplayTestDrive


To Read 5 Integers and Display on screen

Enter 5 Integer Elements

5
4
3
2
1

5 Integers are
   5   4   3   2   1

apple:ijetae apple$
```

Fig – 2: Output of program without typos

In Fig – 3, we can see that, java.util.InputMismatchException is thrown when there are typos which defeats the purpose of the application

```
apple:ijetae apple$ java IntegerDisplayTestDrive

To Read 5 Integers and Display on screen

Enter 5 Integer Elements

5
4a
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:864)
at java.util.Scanner.next(Scanner.java:1485)
at java.util.Scanner.nextInt(Scanner.java:2117)
at java.util.Scanner.nextInt(Scanner.java:2076)
at IntegerStoreTestDrive.main(
IntegerStoreTestDrive.java:14)

apple:ijetae apple$
```

Fig – 3: Output of program without typos

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 7, Issue 6, June 2017)

```
apple:ijetae apple$vim RobustIntegerDisplayTestDrive.java

import java.util.Scanner;

public class RobustIntegerDisplayTestDrive{

  public static void main(String[] args){

      int[] numbers = new int[5];
      Scanner sc = new Scanner(System.in);

      System.out.println( "¥nTo Read 5 Integers & Display
on Screen¥n¥n" );

      System.out.println( "¥nEnter 5 Integer Elements¥n" );

      for(int index = 0; index < 5; ++index){
         try{
            numbers[index] = sc.nextInt();
         }catch(Exception ex){

            System.out.println ( "¥nEnter Valid Integer :
");
            sc.next();
            --index;
       }

       System.out.printf( "¥n5 Integers are¥n" );
       for(int index = 0; index < 5; ++index)
         System.out.printf( "%4d" ,numbers[index]);

         System.out.println( "¥n" );
         return;
    }
}
```

Fig – 4: A Robust java program to read 5 Integers.

Fig – 4 has RobustIntegerDisplayTestDrive.java program which is improved version of program in Fig – 1 which handles typos and other exceptional scenarios by using a catch phrase with java.lang.Exception as parameter. Program in Fig – 4 is robust in nature but is not purely based on Object Oriented design principles. Fig – 5 shows how the enhanced program is more robust in nature by handling typos and other forms of errors.

The module is not cohesive in nature as it doesn't clearly implement the principle of Single Responsibility which makes the code fragile with different use case.

```
apple:ijetae apple$ javac RobustIntegerDisplayTestDrive.java
apple:ijetae apple$ java RobustIntegerDisplayTestDrive

To Read 5 Integers and Display on screen


Enter 5 Integer Elements

5
4a

Enter Valid Integer:
ab

Enter Valid Integer:
4
3
2
1a

Enter Valid Integer:
1

5 Integers are
   5   4   3   2   1

apple:ijetae apple$
```

Fig – 5: Output of Java Program in Fig - 4

III. Proposed Strategy

A great software always does what the customer wants it to do. It is well-designed, well-coded, flexible, re-usable, extensible and maintainable. RobustIntegerDisplayTestDrive.java program is monolithic but robust in nature. It violates some Object-Oriented design principles which in turn makes the code fragile.

A good Object-Oriented design always ensures that modules are highly cohesive in nature. Robustness is induced into program in Fig – 4 by emphasizing on High Cohesion and Refactoring principles. Fig – 6 contains user-defined Scanner.java class which is highly cohesive in nature.

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 7, Issue 6, June 2017)

```
apple: ijetae apple$ vim Scanner.java

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Scanner{

   static BufferedReader keyboard;

   static{

     keyboard = new BufferedReader(
           new InputStreamReader(System.in));
   }

   public static int readInteger(String prompt){

     int number;
     String line = null;

     while(true){
        try{
          System.out.print(prompt);
          line = keyboard.readLine();
          number = Integer.parseInt(line);
          break;
        }catch(Exception ex){

         System.out.println("¥nInvalid Integer Input");
        }
     }

     return number;
   }

   public static String readLine(String prompt){

   String line = null;
   while(true){

      try{
         System.out.print(prompt);
         line = keyboard.readLine();

        if(line.trim().equals("") || line.equals("¥n")){
           System.out.println("¥nInvalid String Input");
           continue;
         }
         else
            break;

      }catch(Exception ex){}
    }
   return line;
  }
}
```

Fig – 6: Scanner.java class

```
apple:ijetae apple$ vim CohesiveScannerTestDrive.java

public class CohesiveScannerTestDrive{

   public static void main(String[] args){

      int[] numbers = new int[5];
      int index;

      System.out.println("¥n¥nTo Read 5 integers and
display on screen¥n¥n");

      System.out.println("¥nEnter 5 Integers¥n");
      for(index = 0; index < 5; ++index)
         numbers[index] = Scanner.readInteger ("Enter an
Integer : ");


      System.out.println("¥nEntered 5 Integers are¥n");
      for(index = 0; index < 5; ++index)
        System.out.printf("%4d", numbers[index]);

      System.out.println("¥n");
      return;
   }
}
```

Fig – 7: CohesiveScannerTestDrive.java

```
apple:ijetae apple$ javac CohesiveScannerTestDrive.java

apple:ijetae apple$ java CohesiveScannerTestDrive

To Read 5 integers and display on screen


Enter 5 Integers

Enter an Integer : 5
Enter an Integer : 4a

Invalid Integer Input
Enter an Integer : 3a4

Invalid Integer Input
Enter an Integer : 4
Enter an Integer : 3
Enter an Integer : 2
Enter an Integer : 1

Entered 5 Integers are

   5   4   3   2   1

apple:ijetae apple$
```

Fig – 8: Output of program in Fig – 7

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 7, Issue 6, June 2017)

In Scanner.java program of Fig – 6, we used the concept of refactoring for high cohesion. Instead of writing our Scanner.java class from scratch, or instead of extending java.util.Scanner class, we took help of java.util.Scanner class to achieve the intended functionality of being robust and being cohesive in nature. Scanner.java class in Fig-6 is robust and highly cohesive. It fulfils the Object-Oriented design principle of Single Responsibility. Fig – 8 shows the output which proves the robustness of the module.

IV. Conclusion

As *change* is the only constant in software, we need to use strategies which makes the development of software to be more flexible, re-usable, extensible and maintainable. Preferring delegation, composition and aggregation over inheritance makes module highly cohesive in nature. Refactoring re-usable modules is needed in certain contexts to make modules highly cohesive in nature which is a key factor in development of robust modules.

REFERENCES

[1] McLaughlin, Brett, Gary Pollice, and David West. Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D. " O'Reilly Media, Inc.", 2006.

[2] Bishop, Matt, and Deborah Frincke. "Teaching robust programming." IEEE Security & Privacy 2.2 (2004): 54-57.

[3] Bishop, Matt, and Chip Elliott. "Robust programming by example." IFIP World Conference on Information Security Education. Springer Berlin Heidelberg, 2009.

[4] Freeman, Eric, et al. Head First Design Patterns: A Brain-Friendly Guide. " O'Reilly Media, Inc.", 2004.

[5] Fowler, Martin, and Kent Beck. Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

[6] Polsani, Pithamber R. "Use and abuse of reusable learning objects." Journal of Digital information 3.4 (2006).

[7] Bennedsen, Jens, and Michael E. Caspersen. "Programming in context: a model-first approach to CS1." ACM SIGCSE Bulletin. Vol. 36. No. 1. ACM, 2004.