Refactoring for High Cohesiveness in designing Robust Python Modules

J. Vamshi Vijay Krishna

Assistant Professor, Dept. of IT, CVR College of Engineering, Telangana, India

---***---

Abstract – *A great software must satisfy the customer. With good use-cases, the existing system/software 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*

Key Words: Robust Programming, Design Patterns, Python, Object Oriented Programming, Cohesion, Refactoring

1. 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 programming is defensive and this defensive nature protects the program not only from those who use our application, but also from ourselves. 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].

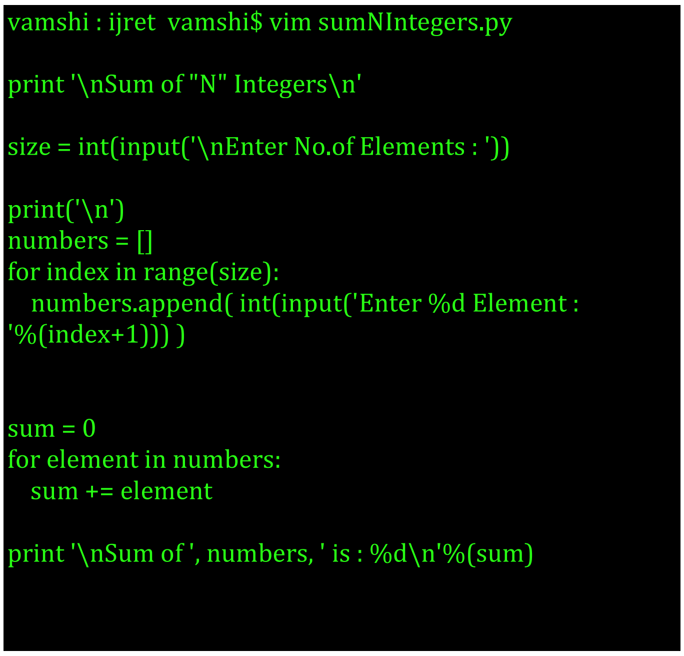
Design patterns directly doesn't go into code. They first go into our brain and with a good working knowledge of patterns, we can then start to apply them to our new designs, and rework our old code which is fragile [4].

A cohesive module does one thing well and doesn't try to do or be something else. 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.

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].

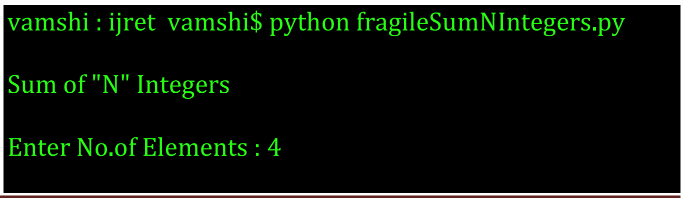
2. CASE STUDY

It's not always possible to design modules which are highly cohesive in nature. For example, consider a scenario of reading 'N' integer numbers from keyboard and then displaying them on screen instead of writing to a file (to make the example easily understandable). sumNIntegers.py script in Fig–1 reads 'N' integers from keyboard and displays the output on screen.

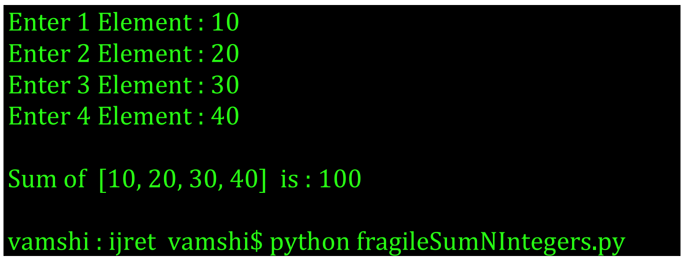


```
vamshi : ijret  vamshi$ vim sumNIntegers.py

print '\nSum of "N" Integers\n'

size = int(input('\nEnter No.of Elements : '))

print('\n')
numbers = []
for index in range(size):
   numbers.append( int(input('Enter %d Element :
'%(index+1))) )


sum = 0
for element in numbers:
   sum += element

print '\nSum of ', numbers, ' is : %d\n'%(sum)
```

Fig – 1: Python Script to calculate Sum of 'N' Integers



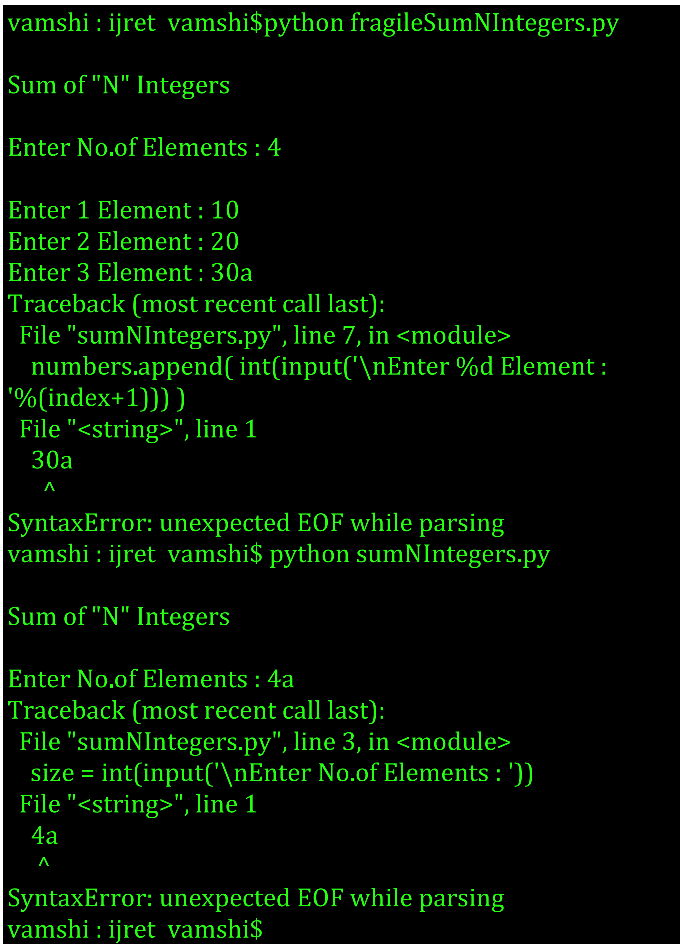
```
vamshi : ijret  vamshi$ python fragileSumNIntegers.py

Sum of "N" Integers

Enter No.of Elements : 4
```



```
Enter 1 Element : 10
Enter 2 Element : 20
Enter 3 Element : 30
Enter 4 Element : 40

Sum of  [10, 20, 30, 40]  is : 100

vamshi : ijret  vamshi$ python fragileSumNIntegers.py
```

Fig – 2: Robust Python Script to add 'N' Integers

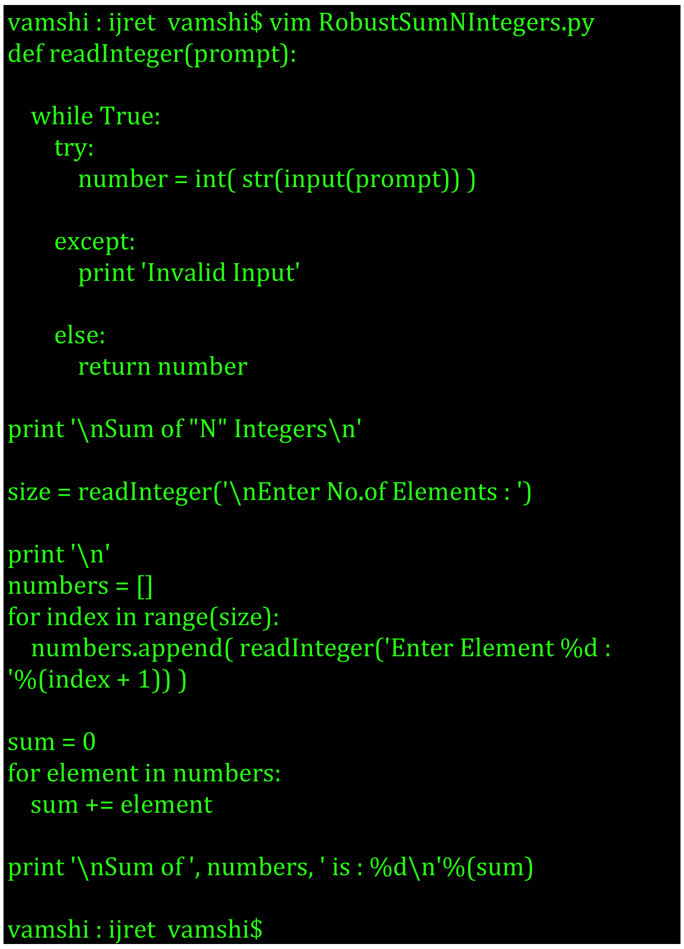
Fig – 2 shows the output of fragileSumNIntegers.py without typos and Fig – 3 shows the output of the script with typos. Typos defeat the purpose of the application. The script fragileSumNIntegers.py is non-robust in nature. It works well with intended input, but mis-behaves with erroneous input



```
vamshi : ijret  vamshi$python fragileSumNIntegers.py

Sum of "N" Integers

Enter No.of Elements : 4

Enter 1 Element : 10
Enter 2 Element : 20
Enter 3 Element : 30a
Traceback (most recent call last):
  File "sumNIntegers.py", line 7, in <module>
    numbers.append( int(input('\nEnter %d Element :
'%(index+1))) )
  File "<string>", line 1
    30a
      ^
SyntaxError: unexpected EOF while parsing
vamshi : ijret  vamshi$ python sumNIntegers.py

Sum of "N" Integers

Enter No.of Elements : 4a
Traceback (most recent call last):
  File "sumNIntegers.py", line 3, in <module>
    size = int(input('\nEnter No.of Elements : '))
  File "<string>", line 1
    4a
     ^
SyntaxError: unexpected EOF while parsing
vamshi : ijret  vamshi$
```

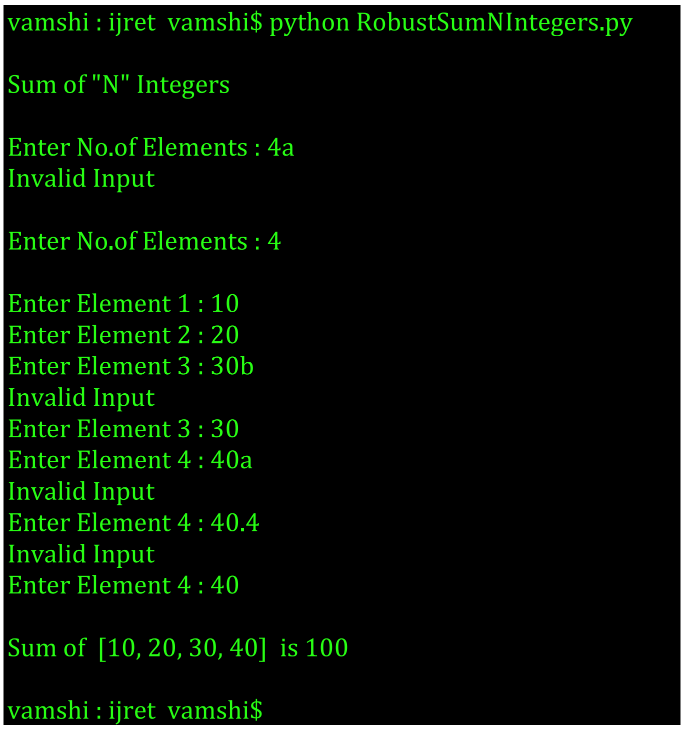
Fig – 3: Output of Script in Fig – 1 with typos

Fig – 4 shows improved version of sumNIntegers.py which is robust in nature. RobustSumNIntegers.py in Fig – 4 handles typos as well as other form of errors like handling float inputs when an integer is expected. The output of the script is shown below in Fig – 5.



```
vamshi : ijret  vamshi$ vim RobustSumNIntegers.py
def readInteger(prompt):

  while True:
    try:
      number = int( str(input(prompt)) )

    except:
      print 'Invalid Input'

    else:
      return number

print '\nSum of "N" Integers\n'

size = readInteger('\nEnter No.of Elements : ')

print '\n'
numbers = []
for index in range(size):
  numbers.append( readInteger('Enter Element %d :
'%(index + 1)) )

sum = 0
for element in numbers:
  sum += element

print '\nSum of ', numbers, ' is : %d\n'%(sum)

vamshi : ijret  vamshi$
```

Fig – 4: Robust Python Script to add 'N' Integers



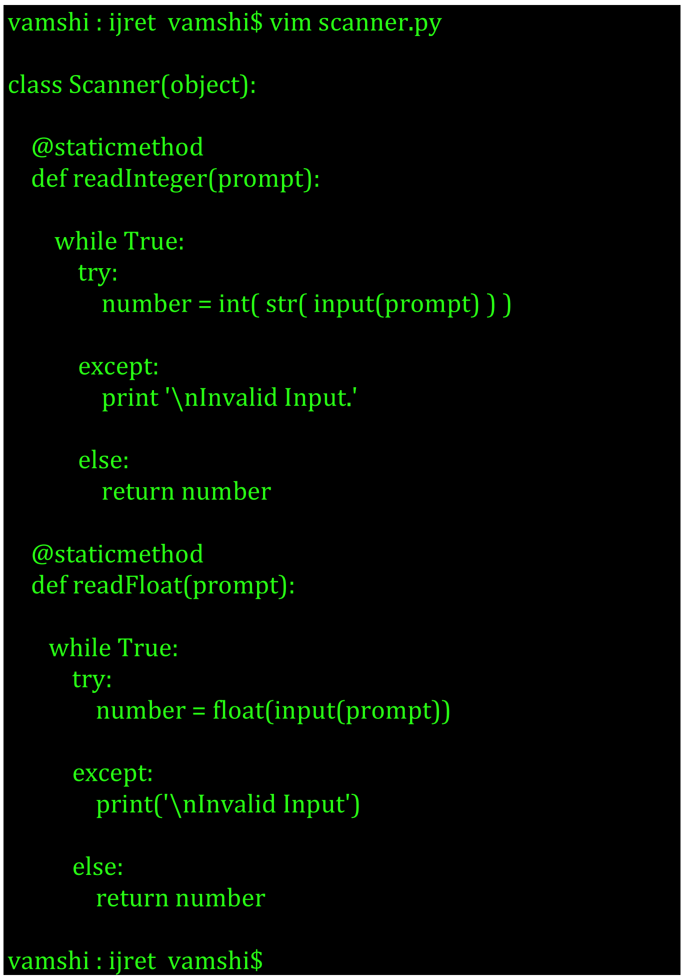
```
vamshi : ijret  vamshi$ python RobustSumNIntegers.py

Sum of "N" Integers

Enter No.of Elements : 4a
Invalid Input

Enter No.of Elements : 4

Enter Element 1 : 10
Enter Element 2 : 20
Enter Element 3 : 30b
Invalid Input
Enter Element 3 : 30
Enter Element 4 : 40a
Invalid Input
Enter Element 4 : 40.4
Invalid Input
Enter Element 4 : 40

Sum of  [10, 20, 30, 40]  is 100

vamshi : ijret  vamshi$
```

Fig – 5: Output of Script in Fig – 4 which handles errors

RobustSumNIntegers.py in Fig – 4 is robust in nature but fails when the user comes up with a new use-case. Apart from being robust, the module should be reusable, extensible. Modules can't be reusable or extensible unless they are cohesive in nature. The module is non-cohesive in nature as it could not implement the Single Responsibility Principle and makes it fragile with different use case.

3. PROPOSED STRATEGY

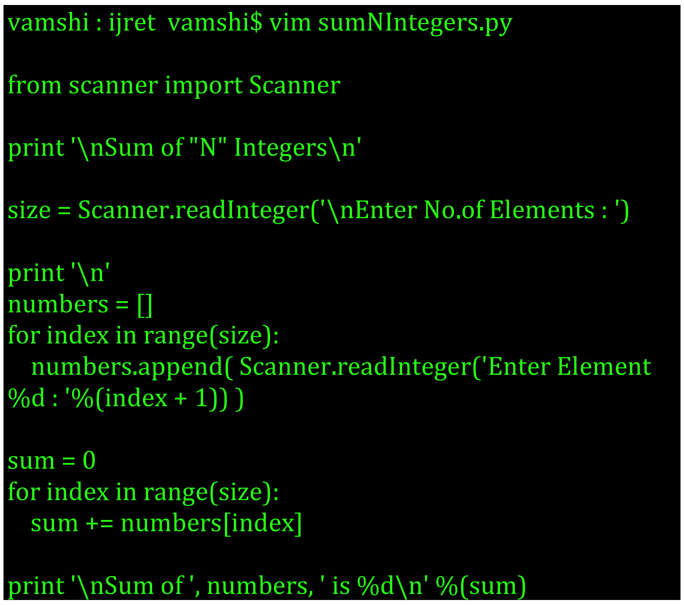
A great software always does what the customer wants it to do. It is well-designed, well-coded, flexible, re-usable, extensible and maintainable. A good Object-Oriented design always ensures that modules are cohesive in nature. Robustness is induced into program in Fig – 4 by emphasizing on High Cohesion and Refactoring principles.



```
vamshi : ijret  vamshi$ vim scanner.py

class Scanner(object):

  @staticmethod
  def readInteger(prompt):

    while True:
      try:
        number = int( str( input(prompt) ) )

      except:
        print '\nInvalid Input.'

      else:
        return number

  @staticmethod
  def readFloat(prompt):

    while True:
      try:
        number = float(input(prompt))

      except:
        print('\nInvalid Input')

      else:
        return number

vamshi : ijret  vamshi$
```

Fig – 6: Cohesive and Robust Scanner Class

We define cohesive and robust module scanner.py in Fig – 6. The module takes the Single Responsibility of reading values of different data types from keyboard in a context.

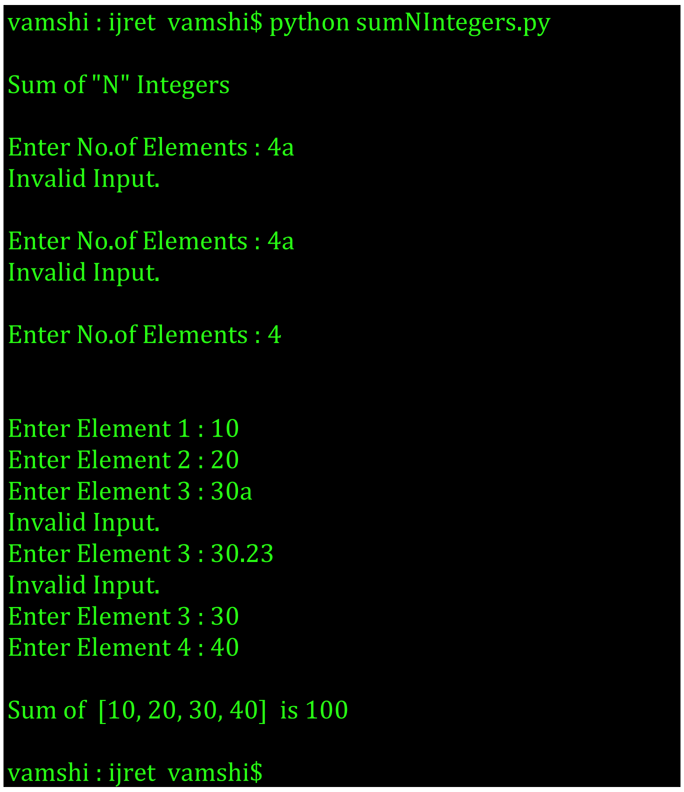


```
vamshi : ijret  vamshi$ vim sumNIntegers.py

from scanner import Scanner

print '\nSum of "N" Integers\n'

size = Scanner.readInteger('\nEnter No.of Elements : ')

print '\n'
numbers = []
for index in range(size):
  numbers.append( Scanner.readInteger('Enter Element
%d : '%(index + 1)) )

sum = 0
for index in range(size):
  sum += numbers[index]

print '\nSum of ', numbers, ' is %d\n' %(sum)
```

Fig – 7: Sum of 'N' Integers using Robust Cohesive module

Fig – 7 shows the usage of scanner.py module in calculating the sum of 'N' Integers. Fig – 8 shows the output of the script sumNIntegers.py which uses scanner.py module which is robust and cohesive in nature.

The output shows that the scanner.py modules handles typos as well as other forms of errors like handling a float value when an integer is expected.



```
vamshi : ijret  vamshi$ python sumNIntegers.py

Sum of "N" Integers

Enter No.of Elements : 4a
Invalid Input.

Enter No.of Elements : 4a
Invalid Input.

Enter No.of Elements : 4


Enter Element 1 : 10
Enter Element 2 : 20
Enter Element 3 : 30a
Invalid Input.
Enter Element 3 : 30.23
Invalid Input.
Enter Element 3 : 30
Enter Element 4 : 40

Sum of  [10, 20, 30, 40]  is 100

vamshi : ijret  vamshi$
```

Fig – 8: Output of Script in Fig – 7

4. CONCLUSIONS

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.