# Contextual Customization of Reusable Components for High Cohesiveness in Robust Software Development

J. Vamshi Vijay Krishna
Asst. Professor, CVR College of Engineering/ IT Department , Hyderabad, India
Email: j.vamshi@cvr.ac.in

***Abstract:*** **Developers generally prefer reusable code in software development. Writing reusable code is not about developing generic, monolithic modules. It is about writing focused, com-posable modules with high cohesion and loose coupling. Reusing generic, monolithic modules which are non-cohesive makes a fragile software which in turn increases development and maintenance costs. Proper customisation of a generic monolithic module in a context enhances cohesiveness of the module. Customisation for cohesiveness makes the module reliable and robust which in turn reduces the development and maintenance cost.**

***Index Terms*—robust programming, cohesion, monolithic design, generic components, contextual customisation**

## I. Introduction

Being generic is a basic requirement for a module to be reusable [1]. In the quest for writing reusable software, it's a common mistake to make software too generic. Making software too generic causes its usability to suffer. Reusable module is not about designing a module to be a single, flexible, monolithic component which is applicable in an extremely broad range of use cases and environments[2]. It's about designing a cohesive com-posable component.

Preferring reusability over cohesiveness makes a module less robust which in turn affects the development and maintenance cost. Robustness is the ability of computer to cope up with errors during execution and cope with erroneous input[3]. 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.

A robust program ideally detects or prevents these conditions and deals with them in a safe and intelligent way.

Let's take a c code-snippet and elaborate upon the concepts of reusability and cohesiveness. Let's take a simple c program to calculate sum of 2 real numbers

**vamshi-macbook:researchvamshi$mate sum2Floats.c**

```
#include <stdio.h>

float readFloat(void);
float sumFloat(float, float);

int main(void){
        float number1;
        float number2;
        float result;

        number1 = readFloat();
        number2 = readFloat();

        result  =sumFloat(number1, number2);

        printf("\n\nSum  : %f", result);

        return 0;
}

float readFloat(){

        float number;

        printf("\nEnter a Real Number : ");
        scanf(" %f", &number);

        return number;
}

float sumFloat(float number1, float number2){
        return number1 + number2;
}
```

**vamshi-macbook:researchvamshi$gcc sum2Floats.c**
**vamshi-macbook:researchvamshi$./a.out**

Enter a Real Number : 123
Enter a Real Number : 234

**Sum  : 357.000000**
**vamshi-macbook:researchvamshi$./a.out**
Enter a Real Number : 123a45
Enter a Real Number :

**Sum  : 246.000000**
**vamshi-macbook:researchvamshi$**

We get in-consistent result because *readFloat* is not cohesive in nature. It is making use of *scanf* function which is reusable module but not cohesive. There should be clear

DOI:10.32377/cvrjst1219

## II. Proposed Strategy

We are proposing a design methodology where we neither sacrifice reusability aspect nor neglect cohesive aspect of the module being developed. We customize reusable components to enhance its cohesive nature which enhances its robustness [4].

Certain key-points in framing our cohesive modules

- Maximizing genericity complicates use
- Excessive reusability breaks abstraction.
- Favor Composability over monolithic design
- Favor Composability over reusability
- Customise Reusable Components to enhance cohesion.

For example, generic *scanf* function is used to read data of different types. It's very generic nature creates the problem of robustness. Typo in entering an integer using *scanf* forces the integer variable to carry garbage value.

Because of excessive genericity of *scanf* function, it's difficult to modify the existing functionality to facilitate robustness.

It's better to design our own module using existing *scanf* function to deal with the context appropriately.

A developer should neither code from scratch, nor use existing re-usable components blindly. He should customise the existing components to enhance cohesiveness.

Let's understand the above aspects through the case studies given below

## III. Case Study

Let's modify *sum2Floats.c* to enhance cohesive aspect of *readFloat* function which in turn makes the module robust.

**vamshi-macbook:researchvamshi$mate cohesiveSum2Floats.c**

```
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h>
#include <stdlib.h>

float readFloat(const char* prompt);
float sumFloat(float, float);

int main(void){

float number1;
float number2;
float result;

number1 = readFloat("\nEnter Number1 : ");
number2 = readFloat("\nEnter Number2 : ");

result  =sumFloat(number1, number2);

printf("\nResult : %f", result);

printf("\n\n");
return 0;
}
float readFloat(char* prompt){

float number;
bool successFlag = false;

do{
        char ch;
        intextraCharacterCount = 0;

        printf("\n%s", prompt);
        if(scanf(" %f", &number) != 0){
        while((ch = getchar())!='\n')
                extraCharacterCount++;

        successFlag = extraCharacterCount?false :
                                        true;
        }
        else
                successFlag = false;

        if(!successFlag&&extraCharacterCount){
        printf("\nInvalid Input");
        continue;
        }


        else if(!successFlag){
        while(getchar()!='\n');
                printf("\nInvalid Input");
        }

}while(!successFlag);

return number;
}

float sumFloat(float number1, float number2){
        return number1 + number2;
}
```

**vamshi-macbook:researchvamshi$gcc cohesiveSum2Floats.c**
**vamshi-macbook:researchvamshi$./a.out**

```
Enter Number1 : 123a45
Invalid Input
Enter Number1 :abc
Invalid Input
Enter Number1 : 123

Enter Number2 : 234.56a
Invalid Input
Enter Number2 : 234.56

Sum  : 357.559998
vamshi-macbook:researchvamshi$
```

DOI:10.32377/cvrjst1219

Although, we can write *readFloat* module from scratch to be cohesive, we brought about cohesiveness by customising the reusable component *scanf* function. Along with cohesiveness, we enhanced reusable aspect of *readFloat* module by prompting user to provide a contextual prompt.

We can better appreciate the robustness brought about by *readFloat*module if it's being frequently used as part of another module which dramatically makes the new module under development very robust. The code-snippet given below communicates the power of customised cohesive module design.

**vamshi-macbook:researchvamshi$matecohesiveSumStudentMarks.c**

```
intreadInteger(const char* prompt);
intreadArraySize(const char* prompt, int capacity);
float readFloat(char* prompt);

void readFloatArray(const char*, float* numbers, int size);
void displayFloatArray(const char*, float* numbers, int size);

int main(void){

int capacity = 10;
float subjects[capacity];
intnoOfSubjects;

noOfSubjects = readArraySize("\nEnter the No.of Subjects : ", capacity);
readFloatArray("\nEnter %d Subject Marks\n", subjects, noOfSubjects);
displayFloatArray("\n%d Subject Marks are\n", subjects, noOfSubjects);

printf("\n\n");
return 0;
}

void readFloatArray(const char* prompt, float* numbers, int size){

int index;

printf(prompt, size);
for(index = 0; index < size; ++index)
        numbers[index] = readInteger("\n");

return;
}

intreadArraySize(const char* prompt, int capacity){

intarraySize;
bool successFlag = true;
do{
        arraySize = readInteger(prompt);
        if(arraySize> capacity)
        successFlag = false;
        else
        successFlag = true;
}while(!successFlag);

return arraySize;
}

void displayFloatArray(const char* prompt, float* numbers, int size){

int index;

printf(prompt,size);
for(index = 0; index < size; ++index)
printf(" %f", numbers[index]);

return;
}
```

**vamshi-macbook:researchvamshi$gcccohesiveSumStudentMarks.c**
**vamshi-macbook:researchvamshi$./a.out**

```
Enter the No. of Subjects : 5

Enter 5 Subject Marks
85
abc
Invalid Input
83
82
83a
Invalid Input
84
86

5 Subject Marks are
85.00000 83.00000 82.00000 84.00000 86.000000
```

**vamshi-macbook:researchvamshi$**

In the above application, we need to read 5 subject marks. If their is at least 1 typo, the entire data becomes invalid. The *readFloatArray* function internally invokes*readFloat* cohesive function which in turn makes the *readFloatArray* cohesive. This in turn makes the *readFloatArray* module very robust.

DOI:10.32377/cvrjst1219

## IV. Conclusions

Although robust programming is an established programming methodology, developers tend to design cohesive modules from scratch or tend to re-use existing components overlooking cohesion.

Customising Re-usable components for cohesion increases productivity of developer along with robustness of the module. Proper customisation of modules dramatically reduces development and maintenance costs.

## References

[1] Eric Freeman, Elisabeth Freeman, "Head First Design Patterns", O-Reilly Publications

[2] R. Zippel, contributors, 2011. KConfig Documentation. Website: http://www.kernel.org/doc/Documentation/kbuild/kconfiglanguage.txt; visited on November 24th, 2011.

[3] 1. B. Miller, L. Fredriksen, and B. So,"An Empirical Study of the Reliabilityof UNIX Utilities," CommACM, vol. 33, no. 12, Dec. 1990,pp. 32–44

[4] M. Acher, P. Collet, P. Lahire, R.B. France, A domain-specific language for managing feature models, in: Proc. ACM Symposium on Applied Computing,SAC, ACM, New York, NY, USA, 2011, pp. 1333–1340.


DOI:10.32377/cvrjst1219