Comparative Study on Main and Sub-code Reusabilities

A.N.Swamynathan[*] and Dr. K.Nirmala[#]

[*]Research Scholar, Research & Development Centre, Bharathiar University, Coimbatore-641 046, India
[#]Associate Professor, Dept. of Computer Science, Quaid-E-Millath Govt. College(w), Chennai-600 002, India

Summary:
Most of the coding and reused coding of south Indian IT companies are based on object oriented programming environment (OOP). In OOPs, it is possible to use certain code again for different modules through inheritance. While calling class member function in objects of a particular class, interface and dependency related problems are encountered. To overcome these kinds of problems, we propose a general purpose code reusable model that analyzes language structure through two possible reusing environments. The common and traditional approach is the main to sub-coding. For different but similar projects sub-codes can call main with appropriate logical checks. The paper justifies model based approach for code reusability under OOPs for these two approaches. Unless a comparative study is performed on these two approaches, a generalized model would not be possible to construct. However the descriptive presentation of reusability model is beyond the scope of this paper. The paper is a part of another whole research. The whole research attempts to investigate different levels of programmers that form pairs in pair programming on reusable coding. However this paper limits its scope only to the comparative study on these two coding approaches. The paper presents comparison studied through experiment on a few application areas. Risk factors are obtained through trials while attempting reused coding.

Keywords:
Interface conformance, Reusability, Inheritance, Segment dependency.

1. Introduction

The current trend in software development towards reuse of coding has indicated the need for quality reusable code. In particular, the intent of reusability guidelines is to treat source code components as isolated, encapsulated, modular units that are totally independent from other units [7][9]. That is each unit or module is designed to be independent of how or when it can be used or reused while correctly implementing it. Software is unique in its own way and may differ from each other in its coding or other interfaces. Based on its coding or interface, the software development complexity may also vary. While developing new software, some previously developed modules of old software may be adapted to the new one. In general software packages are developed by various developers by adapting different development life cycles [4]. In that situation it is necessary to analyze the interface risk and access rights.

The primary goal of software development life cycle model is to develop software in a methodical manner. The organization should therefore prepare accurate document based life cycle models for the development.

Large projects are usually splitting into many modules based upon divide and conquer technique. Finally all the modules are grouped together and checked for their efficiency. Many problems come to pass in interfacing components and managing the software development.

In traditional programming library functions or archival functions known as sub-codes are reused for different applications, say through one main coding. However ANSI C allows sub-codes to call main function in a recursive fashion [3]. It is very important to use appropriate logical checks in such a case.

The paper presents comparative study performed on the above two approaches through trial runs for different projects. In both the approaches, the number of trials required for every error detected is counted as a risk factor in this experiment. A generalized model is proposed from this study.

1.1 Inheritance concept in OOPS

Inheritance concept is the most important property of OOPS. It is a technique that creates a new class from an already defined class. The new class contains all the attributes of the old class [1][9] in addition to some of its own attributes. Additionally it can override some of the attributes and features of old class. When there is a need for some functionality, user can inherit those related functions of the base class and use it. Once the class is defined then it can also be reused several times by other applications after being inherited into the class which suits that particular application. Through inheritance all the applications can be made to inherit the designed class into a new class and can be used in new class. The object class should be at the zenith of the class hierarchy. Every class should descend from it in a direct or indirect manner. In general the derived class inherits some or all of the traits from the base class and a class may inherit properties of more than one class in a single or more than one level [2]. As described earlier two types of calls are popular. Each type is described below.

Manuscript received July 5, 2012
Manuscript revised July 20, 2012

In type I, a sub code is used by a main code. The same sub code may be reused by another main code also. In such situations, we call the invoking code a main code and the invoked code one sub code. For each invocation of a sub code, a main code represents first use or reuse of the sub code, and the reusability of a sub code is related to the number of main codes in which it can sensibly be used (see Fig 1.0).

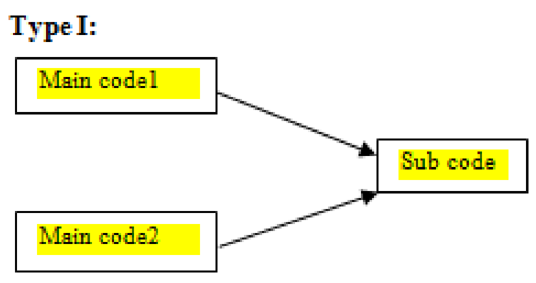

Fig 1.0 Several main codes reuse a sub code

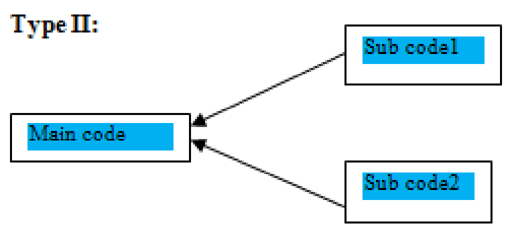

Fig 2.0 Main code reuse: one main code invokes different

Another aspect of this structure is that it can also work in the reverse order. The first case is as shown in fig 1.0. However, sometimes we work the other way around: we have a main code, and use it with different sub code similar to recursive calls, as shown in fig. 2.0. For example, we might have main code that calls a procedure for a particular application, but in a different setting we may need the same sub code but this time to a different procedure say another application. We call this as main code reuse. In general, mechanisms that support reusability allow main code reuse, but some mechanisms specifically support main code reusability [8]. Main code and sub code are really roles played by sections of code. Sometimes a section of code can play both the roles.

In OOP, this organizational technique is known as aggregation. Use of aggregation allows the definition of the new class by reusing existing classes. The new class, as main code, provides functionality in part by using the services of the existing classes, as sub codes.

The interface as shown in Figs 1.0 & 2.0 is the most critical part for reusing the codes. The main and sub code can affect each other's behavior only through the interface dependency. Different times at different programs runs, two segments can affect each dependency, representing the fact that each segment depends on the other during the execution. We describe the interface between the sub code and main code by enumerating all the dependencies between them. Note that not only one can be the main code that depends on the sub code, but the sub code can also be dependent on the main code. Finally, the interface of a single segment can also be described as the dependencies that must be satisfied. The dependencies of OOP are elaborated below.

```
Main code 1.
int main( )    //Application I
{
   int z;
   z=addition(5,4);
   cout <<"the  result is"<<z;
   return 0;
 }
                                          Sub code:
                                          int addition(int a,int b)
                                          { int r;
                                             r=a+b;
                                             return ( r );}

Main code 2.   //Application II
int main( ) {
   int x;
   x=addition(5,4);
   cout <<"the  result is"<<x;
   return 0;}
 }
```

Figure 3.0 an illustration of type I

In the above coding it is illustrated that the sub-code more or less acts like a library function, while the main codes are meant for different applications.

```
Main code:                          Sub code 1:
  int main( )                         int sub(int x, int y)
  {                                       {
   int a;                                   int z;
   a=sub(10,5);                             z=x-y;
   cout<<"the result is"<<a;           //call main with
appropriate   a=mul(          a=mul(10,5); //inheritance
(Application I)
cout<<"the result is"<<a;               }
   //inheritance are checked            Sub code 2:
   // for appropriate                   int mul(int x, int y)
   // applications                      {
   return 0 }                                 int z;
                                          z=x*y;
  //call main with appropriate inheritance
                                        //(Application II)
                                         }
```

Figure 4.0 an illustration of type II

1.2 Segment dependencies

We classify segment dependencies in two ways: contract or noncontract, and explicit, implicit, or informal. Contract dependencies [3] are those that have been intentionally introduced by the programmer, this dependency sometimes voluntarily required for the code segment. Whereas non-contract dependencies are taken accidentally. Code that relies on non-contract dependencies is less likely to be reusable. Encapsulation [2][9] can be seen as reducing such dependencies in object oriented programs. Explicit dependencies are those that are described directly in the language. Implicit dependencies are those for which there is no language support for describing them, but which can nevertheless be checked in some way or other. Informal dependencies cannot be described in the language, nor can they be checked. Informal dependencies are not as helpful as implicit dependencies because there is no way to ensure they have been met. Implicit dependencies are not as helpful as explicit dependencies because, it is not obvious what must be done to meet them. We give examples for each of the resulting categories in Table 1.0

Table 1.0 Categories of dependencies

	contract	Non-Contract
Explicit	Public interface of a class	Modula-2 interface (exposes type declarations)
Implicit	Use of Extern functions in C++	Non-static global variables that should be static
Informal	A list that keeps the items in order	The order that an iterate produces items from a set

We can describe how language features affect the reusability of code by focusing on dependencies. For example, in most languages, the use of an actual parameter in a sub code represents a dependency by the sub code on the name from the main code. This is an informal dependency. By introducing a parameter to replace the use of the global variable, we replace the implicit dependency by an explicit dependency now the sub code depends on the main code to supply a value to the parameter. As another example, pass-by-value can be seen as a dependency by the sub code on the values supplied by the main code, but note that the main code in no way depends on the formal parameter used by the sub code: the dependency is one-way. On the other hand, pass-by reference also introduces a dependency by the main code on the sub code the main code now relies on the sub-code changing the value of the formal parameter in the “expected” way.
Removing non-contract dependencies and making the rest explicit does not necessarily mean we make the code more reusable. We need to know which dependencies should be made explicit. For this we need the concepts of safety and generality of code. Safety represents how and when the obligations introduced by the existence of dependencies are met. Generality consists of flexibility, how to relax any checking while considering safety, and customizability, how to introduce useful dependencies. The creation of reusable code can then be described as increasing generality while maintaining safety.

2. Experiment

Our model is useful to us in improving our understanding of reusability particularly on the comparative performance of these two types. In particular, we have gained a new perspective on mechanisms involved in OOP. The discussion above shows how the model encompasses several key concepts of OOP: classes, encapsulation, and composition. All these affect the two types of approaches. Our next step was to consider the role of inheritance and related mechanisms in the two types for comparison.
As with composition, inheritance allows definition of the new class (derived) by reusing an existing class (old). What is different about inheritance is that it can affect the interface of the new class: the interface to the derived class can include the interface to the old class. For reusability, this is the important aspect of inheritance: interface conformance. The new class interface conforms to the old class interface if it includes all the parts of the old class interface. This implies that instances of the derived class may be used anywhere instances of the old class may be used.

Table 2.0 Experiments of selected types

S.No	Applications	No. of risks in trial runs	
		Type I	Type II
1	Finance/Accounting	2	4
2	Banking Applications	3	7
3	Travel Management	6	8

This is an example of main code reuse, which we discussed in the previous section. Of course, main code reuse is possible using the class mechanism alone: we can take an existing main code and implement the class it uses differently. With inheritance, however, we can use a main code with several different classes, even in the same program. This is the primary connection between inheritance and software reusability. This observation has important consequences: it provides guidance about when to use inheritance, and guidance about how to use it.
An important form of inheritance involves abstract classes. The advantage of abstract classes is that context code can be written in terms of the abstract class, and then used with any inheriting concrete class. In this way

the main code will be reusable with any implementation of the class, even if several implementations are used within one program.
If the derived class interface has extra features, this is fine: any main code will still work with the derived class. If the derived interface conforms to the old class interface, but with different behavior, this also fine, because the main code will also work with the derived class. This is an alternative explanation of polymorphism: where main code is used with different classes that conform to one interface, but where each has different behavior. Polymorphism reduces the strictness of the type checking, and so makes type dependencies more flexible. In a similar way, propagation patterns [2] can be seen as making dependencies more flexible.
Abstract classes are also the basis of object-oriented frameworks. In this approach, a high-level design is written as a program that consists only of abstract classes, and the design is applied to particular situations by providing implementations of the abstract classes. Frameworks can be seen as providing reusable main code. Just as reusable macros enable macro libraries, and reusable procedures enable procedure libraries, we speculate that in a similar way abstract class and frameworks could lead to "context libraries".
We prepared coding on three applications, namely 1.Finance/Accounting 2.Banking applications & 3.Travel management. For these three experimental applications we coded in the selected two types. The main code of 1 & 2 are similar. The number of trial runs (risk factors) is presented in table2.0

3. Conclusions

In this paper we have performed a comparative study aimed to understand the nature of software reusability and presented an outline for adapting the selected two types. Our model consists of two main roles for code: main code and sub code, where reuse each way is of interest using the same interface. Our trial runs on different applications has assisted us in understanding the connection between OOP and reusability by clarifying the effects of inheritance and some other related techniques particularly for the two approaches. We also use our study to assist analyze various strategies that support reusability. We hope to develop a model further and are interested in applying for adaptability analysis. Finally we feel that the main concept is how dependencies govern reusability, and believe better understanding of this is important to develop more reusable codes particularly for the two approaches. We conclude that risk factors are more in type II approach.

References

[1] M.Coram, and S.Bohner,"The Impact of Agile Methods on Software project Management", Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer based Systems (ECBS'05), IEEE Computer Society, 2005, Pp 363-370.

[2] Robert Biddle and Ewan Tempero. Understanding OOP language support for reusability. In Seventh Annual orkshop on Institutionalizing Software Reuse (WISR7),August 1995.

[3] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph based customization. Communications of the ACM, pages 94–101,May 1994.

[4] IEEE Software. Special issue on systematic reuse, September 1994.

[5] G.Chin, "Agile Project Management: How to Succeed in the Face of Changing Project Requirements" American Management Association (AMACOM), New York, USA, 2004.

[6] L.Angelis, and I.Stamelos, "Investigating the Extreme Programming System-An Empirical Study", Empir Software Eng,Springer Science+Business Media, 2006, Pp 269-301.

[7] William B. Frakes and Kyo Kang"Software Reuse Research: Status and Future" IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 31, NO. 7, JULY 2005.

[8] Boehm B. W. (1991), 'Software Risk Management:Principles and Practices', IEEE Software, vol.8, no 1, pp.32-41.

[9] Fairley R (1994), 'Risk management for Software Projects' IEEE Software, vol. 11, No.3, pp.57-67.

A.N.Swamynathan received his B.Sc., Degree in Computer Science from Adhiparasakthi College of Science, Kalavai, University of Madras, Chennai and M.Sc., Degree in Computer Science from Thanthai Hans Roever College, Perambalur, Bharathidasan University, Tiruchirappalli. He also received his M.Phil. Degree in Computer Science from Manonmaniam Sundaranar University, Tirunelveli. He is now doing his Ph.D in Computer Science at Research and Development Centre, Bharathiar University, Coimbatore. His field of interest is Software Engineering, Operating System and Computer Architecture.

Dr. K.Nirmala received her Ph.D Degree in Computer Science from NITTTR, Taramani, University of Madras, Chennai. She has fifteen years of teaching experience in the field of Computer Science education at College level. Since 1997 she has been working in various capacities in the Department of higher education, Tamilnadu. She is now working as Associate Professor , Computer Science in Quaid-e-millath Govt. College for women, Chennai. Her field of interest is Data mining, Software Engineering and Neural Networks. She has been published many technical papers at various national and international conferences and journals.