International Journal of Advanced Computer Research (ISSN (print): 2249-7277 ISSN (online): 2277-7970)
Volume-1 Number-1 Issue-1 September 2011

Analysis of Function generation on the basis of Object Oriented paradigm

Kamlesh Gujar[1], Surendra Mishra[2], Pankaj Kawadkar[3]
M.Tech Scholar, Department of Computer Science, SSSIST Sehore, India[1]
Head, PG Department of Computer Science, SSSIST Sehore, India[2]
Head, MCA, SSSIST Sehore, India[3]

Abstract

Object-oriented approaches to software design and implementation have gained enormous popularity over the past two decades. However, whilst models of software systems routinely allow software engineers to express relationships between objects, object-oriented programming languages lack this ability. Instead, relationships must be encoded using complex reference structures. When the model cannot be expressed directly in code, it becomes more difficult for programmers to see a correspondence between design and implementation the model no longer faithfully documents the code. As a result, programmer intuition is lost, and error becomes more likely, particularly during maintenance of an unfamiliar software system. In this paper we discuss how to reduce the program size by fractioning the program based on functions so that the table fragment size of program reduce and the program efficiency is increases.

Keywords

OOP, Program Fraction, Code Segment, Object Oriented Approach

1. Introduction

Today's market much more emphasize on software quality. This has led to an increasingly large body of work being performed in the area of software measurement, particularly for evaluating and predicting the quality of software. In turn, this has led to a large number of new measures being proposed for quality design principles such as coupling. High quality software design, among many other principles, should obey the principle of low coupling. Stevens et al., who first introduced coupling in the context of structured development techniques, define coupling as "the measure of the strength of association established by a connection from one module to another" [1]. Therefore, the stronger the coupling between modules, i.e., the more inter-related they are, the more difficult these modules are to understand, change, and correct and thus the more complex the resulting software system. Some empirical evidence exists to support this theory for structured development techniques; [2], [3].

Test-driven development (TDD) is not, despite its name, a testing technique but rather a development technique in which the tests are written prior to the source code [4]. The tests are added gradually during the implementation process and when the tests are passed, the code is re factored to improve its internal structure. This incremental cycle is repeated until all the functionality is implemented [5]. The idea of TDD was popularized by Beck [6] in the Extreme Programming (XP) method. Therefore, although TDD seems to have just recently emerged, it has existed for decades; an early reference to the use of TDD features in the NASA Project Mercury in the 1960s [7].

Basically there are two different kinds of abstractions namely classes and interfaces. The most important difference is that a class can hold functional logic and an interface is used to organize source code and it will also provide the boundary between the levels of abstraction. According to object oriented programming, the class provides encapsulation and abstraction and the interface provides abstraction and cannot inherit from one class but can implement multiple interfaces. The above said differences are minor and they are very similar in structure, complexity, readability and maintainability of source code [8]. Here, the difference in usage of class inheritance and interface concepts are measured for class diagrams by coupling metrics proposed by Chidamber and Kemrer and Brian.

Complexity of source code directly relates to cost and quality. Many coupling models are presented in the literature to measure the possible interactions between objects and to measure design complexity. High coupling between objects increases complexity and cost. Low coupling is good for designing object oriented software. Inheritance introduces more interactions among classes [9]. This will increase the complexity. This paper presents a comparison between object oriented interfaces and inheritance class diagrams.

The remaining of this paper is organized as follows. We discuss class and object in Section 2. In Section 3 we discuss about Object Oriented Concepts. In section 4 we discuss about Evolution and Recent Scenario. In section 5 we discuss about the Challenges. The conclusions and future directions are given in Section 6. Finally references are given.

2. Class and Object

A class is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors. Java class objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object. Methods are nothing but members of a class that provide a service for an object or perform some business logic. Java fields and member functions names are case sensitive. Current states of a class's corresponding object are stored in the object's instance variables. Methods define the operations that can be performed in java programming.

Syntax:
class classname
{
Methods + variables;
}

An object is an instance of a class created using a new operator. The new operator returns a reference to a new instance of a class. This reference can be assigned to a reference variable of the class. The process of creating objects from a class is called instantiation. An object encapsulates state and behavior.

An object reference provides a handle to an object that is created and stored in memory. In Java, objects can only be manipulated via references, which can be stored in variables.
Creating variables of your class type is similar to creating variables of primitive data types, such as integer or float. Each time you create an object, a new set of instance variables comes into existence which defines the characteristics of that object. If you want to create an object of the class and have the reference variable associated with this object, you must also allocate memory for the object by using the new operator. This process is called instantiating an object or creating an object instance.

The purpose of a class diagram is to depict the classes within a model. In an object oriented application, classes have attributes (member variables), operations (member functions) and relationships with other classes. The UML class diagram can depict all these things quite easily. The fundamental element of the class diagram is an icon the represents a class. This icon is shown in Figure 1.

Figure1 Class Icon

A class icon is simply a rectangle divided into three compartments. The topmost compartment contains the name of the class. The middle compartment contains a list of attributes (member variables), and the bottom compartment contains a list of operations (member functions). In many diagrams, the bottom two compartments are omitted.

Even when they are present, they typically do not show every attribute and operations. The goal is to show only those attributes and operations that are useful for the particular diagram.This ability to abbreviate an icon is one of the hallmarks of UML. Each diagram has a particular purpose. That purpose may be to highlight on particular part of the system, or it may be to illuminate the system in general. The class icons in such diagrams are abbreviated as necessary. There is typically never a need to show every attribute and operation of a class on any diagram. Figure 2 shows a typical UML description of a class that represents a circle.

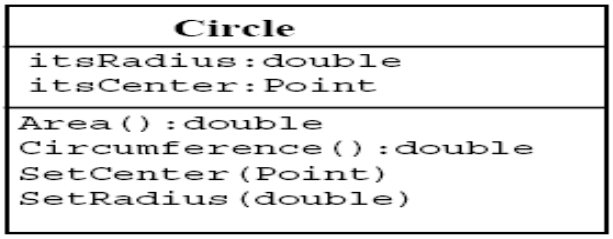

Figure 2 Circle Class

Notice that each member variable is followed by a colon and by the type of the variable. If the type is redundant, or otherwise unnecessary, it can be omitted. Notice also that the return values follow the member functions in a similar fashion. Again, these can be omitted. Finally, notice that the member function arguments are just types. I could have named them too, and used colons to separate them from their types;

If we analyze the above figure then we can deduce that if our program is divided into number of pieces according to their functionality then the accordance of that program produce more flexibility in comparison to the previous one. In C++ we represent this program which is shown in figure 3.

```
class Circle
{
  public:
    void SetCenter(const Point&);
    void SetRadius(double);
    double Area() const;
    double Circumference() const;
  private:
    double itsRadius;
    Point itsCenter;
};
```

Figure 3 Class Circle

So we divide the program in at least four parts according to the function.

3. Object Oriented Concepts

Throughout this evolution, what it means for a programming language to be object-oriented has been the subject of debate: it is not unusual for an object-oriented language to lack a feature declared elsewhere to be indispensable. Simula, for example, lacks dynamic dispatch, but the designer of C++,Bjarne Stroustrup, believes that a language does not support object-oriented programming without in C++ parlance virtual functions .

At the very least, however, an object is a package with a unique identity, some state and some behavior. For the purposes of this work, an object's identity will be its address in memory. An object's state will be formed from a collection
of named fields, which take values including object identities thus, an object may hold a reference to another object, or even to itself. Where a field does not hold such a reference, its value is said to be 'null'.

An object's behaviour will be formed from a collection of named methods, which contain commands that, amongst other actions, operate on the object's fields. An object's fields and methods together form its set of attributes.

Through references, an object method may access the attributes of other objects as well as the attributes of its own object: a method always knows the identity of the object to which it belongs, known as a reference to self. In general, the target of a message invocation is known as the receiver of the method call.

Encapsulation
We have already discussed the history of object-oriented programming languages with respect to their ability to modularize a software system by encapsulating some state and behavior. Depending on the available language features, an object's state can be hidden from the outside world so that the object forms a boundary around some of its fields.

Abstraction
By encapsulating state, an object can ensure that the environment does not manipulate its state in an unexpected way. Where a language supports the specification of hidden attributes, those that remain public form an interface for the object. An object representing a car may, for example, expose methods that allow the driver to switch the car on, turn left and right, change speed and to switch it off. It would not, however, expose methods that allow individual spark plugs to be fired such a method might form part of the car's implementation, but the driver has no need to view the implementation of the car in such detail.

Generalization
It is expected that some objects will share common properties:
for example, vehicles usually have an engine and can carry passengers, regardless of whether they are cars or aeroplanes. Rather than specifying such properties for every vehicle, we can regard 'being a vehicle' as a property that all vehicle objects share.

One might be tempted to conclude that an object-oriented system, once developed, can be reused or extended simply by combining components of existing classes in different ways, by adding operations to existing classes.

Reuse of behavior
A special case of generalization involves the reuse of behavior or, more specifically, the code that implements that behavior. Not only does this help enforce the idea that vehicles behave similarly, but the ability to reuse code to implement the behavior of several objects improves the maintainability of the code: a bug fixed in one object's behavior is fixed for all objects using that code.

Specialization
While groups of objects may be ostensibly the same, slight variations may be accommodated: like other vehicles, a rocket may carry passengers and has an engine, but unlike other vehicles it also has a heat shield, for example. To start from scratch with a new concept of 'being a heat-shielded vehicle would involve the reimplementation of engines and the advantages of generalization would be lost.

Overriding of behavior
A method is overridden where its implementation, derived from some generalization, is replaced. The

attributes possessed by the resulting object will match those of the original object, but the new object's method will behave differently

4. Evolution and Recent Scenario

Object-oriented software is based on the notions of class, encapsulation, inheritance, and polymorphism. These notions make it more challenging to design metrics for the characterization of OO-based software vis-a-vis what it takes to do the same for the purely procedural code [10], [11]. An early work by Coppick and Cheatham [12] attempted to extend the then popular program-complexity metrics, such as the Halstead [13] and the McCabe and Watson complexity measures [14], to OO software. Subsequently, other works on OO software metrics focused mostly on the issue of how to characterize a single class with regard to its own complexity and its linkages with other classes.

In 2010, Bei-Bei Yin et al. [15] proposed two quantitative measures of heterogeneity of software structural profile based on entropy. Three case studies are presented to show the effectiveness of the proposed measures. Different from the perspectives adopted in these works, our previous work found that the networks of software dynamic execution processes may also be scale-free. Scale-free degree distribution demonstrates that during the execution process the methods being invoked only a few times are far more abundant than those being frequently invoked.

In 2010, Juan Luo et al. [16] proposed a combinatorial restructuring algorithm which guarantees learning optimality and furthermore reduces the search space to be polynomial in the size of learning set, but exponential to the number of piece-wise bounds.

In 2011, Shinobu Nagayama et al. [17] proposed a new architectures for numeric function generators (NFGs) using piecewise arithmetic expressions. The proposed architectures are programmable, and they realize a wide range of numeric functions. To design an NFG for a given function, we partition the domain of the function into uniform segments, and transform a sub function in each segment into an arithmetic spectrum. From this arithmetic spectrum, they derive an arithmetic expression, and realize the arithmetic expression with hardware.

5. Challenges

1. Improper understanding of the problem
The users of a software system express their needs to the software professionals. The requirement specification is not precisely conveyed by the users in a form understandable by the software professionals. This is known as impedance mismatch between the users and software professionals.

2. Change of rules during development during the software development process
because of some government policy or any other industrial constraints realized, the users may request the developer to change certain rules of the problem already state.

3. Preservation of existing software
In reality, the existing software is modified or extended to suit the current requirement. If a system had been partially automated, the remaining automation process is done by considering the existing one. It is expensive to preserve the existing software because of the non availability of experts in that field all the time. Also, it results in complexity while integrating newly developed software with the existing one.

4. Management of development process
Since the size of the software becomes larger and larger in the course of time it is difficult to manage, coordinate, and integrate the modules of the software.

5. Flexibility due to lack of standards
There is no single approach to develop software for solving
a problem. Only standards can bring out uniformity. Since only a few standards exist in the software industries, software development is a laborious task resulting in complexity.

6. Behavior of discrete systems
The behavior of a continuous system can be predicted by using the existing laws and theorems. For example, the landing of a satellite can be predicted exactly using some theory even though it is a complex system. But, computers have systems with discrete states during execution of the software.

6. Conclusion and Future Directions

This paper presents discuss several concepts and on how to reduce coupling in object oriented programming. Due to the reduction in coupling, developers can produce quality programs. Classes in object-oriented systems, written in different programming languages, contain identifiers and comments which reflect concepts from the domain of the software system.

Object-oriented approaches to software design and implementation have gained enormous popularity over the past two decades. Instead, relationships must

be encoded using complex reference structures. When the model cannot be expressed directly in code, it becomes more difficult for programmers to see a correspondence between design and implementation the model no longer faithfully documents the code. As a result, programmer intuition is lost, and error becomes more likely, particularly during maintenance of an unfamiliar software system. In this paper we discuss how to reduce the program size by fractioning the program based on functions so that the table fragment size of program reduce and the program efficiency is increases.

References

[1] W. Stevens, G. Myers, and L. Constantine, "Structured Design,"IBM Systems J., vol. 13, no. 2, pp. 115-139, 1974.
[2] R.W. Selby and V.R. Basili, "Analyzing Error-Prone SystemsStructure," IEEE Trans. Software Eng., 1991.
[3] P.A. Troy and S.H. Zweben, "Measuring the Quality of Structured Designs," J. Systems and Software, 1981.
[4] Beck, K., Test-Driven Development by Example, Addison-Wesley, Boston, MA, USA, 2003.
[5] Astels, D., Test-Driven Development: A Practical Guide, Prentice Hall, Upper Saddle River, USA, 2003.
[6] Beck, K., Extreme Programming Explained, Second Edition:Embrace Change, Addison-Wesley, USA, 2004.
[7] G. Larman and V.R. Basili, "Iterative and Incremental Development: A Brief History", 2003, IEEE.
[8] Mathew Cochran,"Coding Better: Using Classes Vs Interfaces",January 18th, 2009.
[9] Mohsen D. Ghassemi and Ronald R. Mourant,"Evaluation of Coupling in the Context of Java Interfaces", Proceedings OOPSLA, ACM 2000.
[10] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Trans., 1994.
[11] N. Churcher and M. Shepperd, "Towards a Conceptual Framework for Object-Oriented Software Metrics," 1995.
[12] C.J. Coppick and T.J. Cheatham, "Software Metrics for Object-Oriented Systems," 1992.
[13] M.H. Halstead, Elements of Software Science. Elsevier, 1977.
[14] T.J. McCabe and A.H. Watson, "Software Complexity," Crosstalk,J. Defense Software Eng. 1994.
[15] Bei-Bei Yin, Ling-Zan Zhu, and Kai-Yuan Cai ," Entropy-based Measures of Heterogeneity of Software Structural Profile ", 2010 34th Annual IEEE Computer Software and Applications Conference Workshops.
[16] Juan Luo and Alexander Brodsky," An Optimal Regression Algorithm for Piecewise Functions Expressed as Object-Oriented Programs", 2010 Ninth International Conference on Machine Learning and Applications.
[17] Shinobu Nagayama, Tsutomu Sasao, Jon T. Butler," Numeric Function Generators Using Piecewise Arithmetic Expressions" 2011 41st IEEE International Symposium on Multiple-Valued Logic.