*International Journal of Latest Trends in Engineering and Technology*
*Vol.(16)Issue(1), pp.033-036*
*DOI: http://dx.doi.org/10.21172/1.171.05*
*e-ISSN:2278-621X*

# STRATEGIES AND TECHNIQUES TO DESIGN OBJECT-ORIENTED PROGRAMMING

Dr. Manish Kumar[1]

**Abstract- In recent years, there has been a surge of interest in the Object-Oriented programming. The object oriented paradigm, which advocates bottom-up program development, appears at first sight to run counter to the classical, top-down approach of structured programming. Object-oriented analysis and design are popular concepts in today's software development environment. In fact, object oriented programs may have better structure than programs obtained by functional decomposition. The definitions of the basic components of object oriented programming, object, class, and inheritance, are still sufficiently fluid to provide many choices for the designer of an object oriented language.**

**Keywords – Object-Oriented Programming**

## I. INTRODUCTION

We live in a universe of objects. These objects exist in nature, in man-made substances, in business, and in the items that we use. They can be classified, depicted, sorted out, joined, controlled, and made. Along these lines, it is nothing unexpected that an object-oriented view would be proposed for the creation of computer programming.

Since the time Whelan Associates v. Jaslow Dental Laboratory i courts dealing with software have attempted to understand the process of software design. In doing as such, they have concentrated solely on conventional techniques for procedural programming and "top-down" planii. In addition, most observers on programming security present this conventional model of programming configuration before articulating their proposed degree of insurance. While this model precisely reflects programming plan during the 1980s, the conventional methodology isn't appropriate to the exponential increment in size and intricacy that will portray programming ventures during the 1990siii. As one pundit of conventional plan noted: "In the event that manufacturers fabricated structures the manner in which software engineers composed projects, at that point the main woodpecker that tagged along would annihilate human civilization"iv. In request to address the issue of multifaceted nature, software engineers are probably going to go to protest situated structure and examination since it permits developers to receive a completely extraordinary methodology toward critical thinking and unequivocally supports the improvement of libraries of reusable programming "components". Traditional programming dialects depend on the idea of a technique, which permits developers to compose a little area of code which performs one little taskv.

A valuable program may have a life expectancy of numerous years. During that time, it will be kept up, most likely by a wide range of software engineers. Three kinds of maintenance are perceived in programming designing: perfective, versatile, and remedial. Perfective support improves the program without changing its usefulness. Versatile upkeep reacts to changing prerequisites and makes up for changes in the earth in which the program is utilized. Restorative support analyses and corrects already unfamiliar blunders. The most troublesome undertaking facing maintenance developer is to comprehend existing code. By and large, the code was composed by a software

[1]*Principal, Vidya Vihar Institute Of Technology, Purnea, Bihar, India.*

engineer who is not, at this point accessible for meeting. The programming language and the apparatuses that help it must be intended to help the upkeep developer however much as could be expected.

This process has also been described as "functional decomposition," since the "primary question addressed by the systems analysis and design is WHAT does the system do [or] what is its function?" vi The complex function identified at this stage must be further decomposed into smaller functions, a process which is repeated until the problem can be "expressed as some combination of many small, solvable problems."

Object-oriented decomposition yields littler frameworks through the reuse of basic components or mechanisms, thus providing an important economy of expression. Object-oriented systems are also more resilient to change and thus better able to evolve over time, because their design is based upon stable intermediate structures. Indeed, object-oriented decomposition greatly reduces the risk of building complex software systems, because they are designed to evolve incrementally from smaller systems in which we already have confidencevii.

## II. PROCESS OF DESIGNING OBJECT-ORIENTED PROGRAMS

The concepts of objects, classes, inheritance, polymorphism, and dynamic binding, we can formulate an analytical approach for writing software that takes advantage of the object-oriented model. There have been many formal attempts to define an "object- oriented approach" to software designviii. This methodology has a few focal points over conventional programming plan. To start with, as opposed to asking “what tasks must the program perform. “Object-oriented design asks "how might the individuals who will depend on this program depict their concern, and what might they recognize as the significant on-screen characters (both human and lifeless) in the difficult space." This immediate spotlight on the difficult area powers the developer to address the particular needs of clients in the issue space before composing any code. Interestingly, the traditional programming model promotes an early emphasis on the "tasks", that the product must perform and along these lines expels the concentration from the difficult space. At a beginning phase of the structure procedure, the conventional software engineer gets bound to the particular directions that will be utilized to compose the program, frequently before potential clients have recognized the entirety of their prerequisites. Second, the object-oriented approach assists with uncovering shared characteristics that may exist across comparative applications (vertical area investigation) just as shared characteristics that can be reused in various pieces of a similar application (flat space examination).

## III. SHIFTING TOWARDS OBJECT-ORIENTED LANGUAGES

Table 1: Some Common Object Oriented Terminology

| | |
|---|---|
| Attribute | Defines the structural properties of classes, unique within a class, generally a noun. |
| Class | A set of objects that share a common structure and common behavior manifested by a set of methods; the set serves as a template from which object can be Instantiated (created). |
| Object | An instantiation of some class which is able to save a state (information) and which offers a number of operations to examine or affect this state. |
| Message | A request that an object makes of another object to perform an operation. |
| Inheritance | A relationship among classes, wherein an object in a class acquires characteristics from one or more other classes. |
| Coupling | Object X is coupled to object Y if and only if X sends a message to Y. |
| Cohesion | The degree to which the methods within a class are related to one another. |
| Method | An operation upon on object, defined as part of the declaration of a class. |

This article centers on the design strategies used in object oriented programming and the utilization of knowledge. We have focused our investigation on the structure components identified with two focal parts of the item arranged

worldview: the area of the code for an unpredictable arrangement in various classes and the meaning of basic plans at various levels in the class progressive system.

OOP gives data on the procedures followed for creating complex intends to accomplish the primary objectives of the issue; activities which are portions of these mind boggling plans are connected to various items. At the point when initially explaining the class model, the substances evoked and utilized by software engineers are objects, activities and various sorts of connections between them. While portraying the class model as far as the programming language, articles ought to be planned to computational elements, for example, the static depiction of classes, i.e., the class name, its sort and properties. For software engineers who are knowledgeable about OOP, we anticipate that intricate plans should be created in a broadness first and top-down way. The utilization of information structures, for example, blueprints identified with OO dialects gathering activities and articles, ought to permit subjects (1) to develop a plan, at an elevated level of deliberation first and in a reasonable manner, before growing less conceptual levels, and (2) to coordinate the portrayal of activities and the depiction of items in their first drafts of the arrangements. Moreover, one inquiry is whether complex plans are created based on the articles portrayal, i.e., objects are grown first and afterward activities are created in a request identified with the relationship to a solitary class. This would mirror the utilization of compositions which are sorted out around primary objects. In an OO framework, the items are express and are utilized to structure the framework, so configuration based around the articles ought to be a typical technique. In a procedural language, the activity are essential, and any articles are certain in the arrangement. From this points of view, the typical structure path is to seek after the connections among activities, and see what articles show up in transit. For software engineers who are amateurs in OOP yet experienced in Procedural dialects, we anticipate the depiction of activities and the portrayal of items to be discrete n the principal drafts of the arrangements created by these subjects. We anticipate that unpredictable plans should be created based on portrayal of activities, i.e., activities are grown first at that point objects are related to activities. Besides, we expect activities which are segments of complex intends to be created in their execution request. The utilization of information structures, for example, outlines identified with procedural dialects, should lead subjects to create plans from portrayal of activities; these patterns speak to activities and data on their execution request however need data on the connection between plans activities and items. The challenges at that point experienced in partner components of complex intends to articles ought to block a carefully top-down and broadness first method of creating plans. So the distinction in the request wherein plans are created ought to mirror the utilization of various types of information, either outlines identified with OO dialects in which activities are sorted out around articles or patterns identified with procedural dialects in which activities are spoken to in their execution request. To the extent that the utilization of information, for example, outlines identified with procedural dialects, isn't fitting, we ought to expect there to be more mistakes and furthermore more updates of the disintegration of classes by novices in OOP than by developers experienced in OOP.

## IV. CONCLUSION

This paper introduces the basic concept for object-oriented design and programming. Generally, software engineers had a decision of two styles of programming. From one viewpoint, they could have ordering, static investigation, next to zero cooperation with running projects, and efficiency. Then again, they could have deciphering, dynamic investigation, connection with running projects, and some level of inefficiency. Dialects which bolstered the first style of programming, for example, C and Pascal, offered quick compilers. Dialects which upheld the subsequent style, for example, C++ and C#, Smalltalk, JAVA offered rich advancement conditions. Article advances lead to reuse, and reuse prompts quicker programming improvement and more excellent program with simpler to keep up.

Software engineers need fast access to exact and forward-thinking data about the projects that they are creating. On the off chance that the dialects that they use give classes and legacy, they have to recognize what classes can do and how they are identified with each other. The class order program of Smalltalk gives a portion of this data; getting comfortable with it is a fundamental piece of learning Smalltalk. The short and flatten utilities of Eiffel separate data from source code.

We have based on both of these thoughts by giving perspectives, because the views are constructed by the compiler, they can contain information derived by the compiler in addition to information provided to the compiler. For example, the client view of a class contains all of the ancestors and all of the inherited attributes of the class. It excludes information that the programmer does not need, such as private attributes.. For instance, the customer perspective on a class contains the entirety of the predecessors and the entirety of the acquired properties of the class. It avoids data that the software engineer doesn't require, for example, private qualities. There are points of interest in utilizing the compiler to create data that will be utilized by the earth. Since the compiler completes an itemized examination of the source code at any rate, it is common and efficient to hold the consequences of the investigation rather than reproduce them with different devices.

Article arranged projects normally utilize numerous classes, just a couple of which are composed for a specific application. The creators of Smalltalk understood that developers would require help to find their way around the framework. In structuring and actualizing, we have changed the inactive idea of "perusing" into the dynamic idea of producing the data that the software engineer needs and giving simple access to it.

## REFERENCES


[1] i 797 F.2d 1222 (3d Cir. 1986), cert. denied, 479 U.S. 1031 (1987).

[2] ii MELVILLE B. NIMMER & DAVID NIMMER, NIMMER ON COPYRIGHT, 13.03 [F] at 13-78.30 to .32 (1991); Peter S. Menell, An Analysis of the Scope of Copyright Protection for Application Programs, 41 STAN. L. REV. 1045, 1055-56 (1989); David Nimmer et al., A Structured Approach to Analyzing the Substantial Similarity of Computer Software in Copyright Infringement Cases, 20 ARIZ. ST. L.J. 625, 637-38 (1988); Gary L. Reback & David L. Hayes,The Plains Truth: Program Structure, Input Formats and Other Functional Works, COMPUTER LAW., Mar. 1987, at 1, 5.

[3] iii For a general discussion of the inherent complexity of modern software see GRADY BOOCH, OBJECT-ORIENTED DESIGN WITH APPLICATIONS 2-8 (1991) [hereinafter BOOCH, OBJECT-ORIENTED DESIGN].

[4] iv Booch, Reuse of Software Components Could Reduce Costs, GOV'T COMPUTER NEWS, Sept. 25, 1987, at 86.

[5] v ELLIOT B. KOFFMAN, PROBLEM SOLVING AND STRUCTURED PROGRAMMING IN PASCAL 52-59 (1985). The terms "procedure," "module," and "subroutine" are used interchangeably to denote a small number of programming instructions which perform a single task.

[6] vi Brian Henderson-Sellers & Julian M. Edwards, The Object-oriented Systems Life Cycle, COMM. ACM, Sept. 1990, at 142, 145

[7] vii BOOCH, OBJECT-ORIENTED DESIGN

[8] viii BOOCH, OBJECT-ORIENTED DESIGN, supra note 4; Henderson-Sellers & Edwards, supra note 9; Korson & McGregor, supra note 11; Ronald J. Norman, Object-oriented Systems Analysis: A Methodology for the 1990s, J. SYS. MGMT., July 1991, at 32; Rebecca J. Wirfs-Brock & Ralph E. Johnson, Surveying Current Research in Object-Oriented Design, COMM. ACM, Sept. 1990, at 104.