RESEARCH ARTICLE OPEN ACCESS

A Case Study on Developing Software Using OOPS

Yashika Goyal[1], Himanshu Kakkar[2], Gursheen Kaur[3]
Student[1&3], Assistant Professor[2]
Department of Computer Science & Engineering,
Chandigarh Engineering College,
Landran, India

ABSTRACT
Today, most of the software is designed on an Object-Oriented Programming (OOP) paradigm. Most programs including word processors, spreadsheets, database applications, and e-mail programs are object-oriented. Understanding the basics of object-oriented design is a prerequisite for extending an application to provide some user defined behaviour. An object-oriented programming language is a programming language that supports the precepts of object-oriented programming. Today platform independent languages exist providing various benefits including C++, Java, and Python, in addition to languages specifically for the Microsoft Windows platform including Visual Basic and C#. The object-oriented concepts are the same in all of the languages; only the syntax and presentation are different. Object-oriented approaches to commercial software have been prevalent for more than a decade. Sadly, most undergraduate and graduate engineering education still primarily focuses on procedural based implementation using languages such as FORTRAN, which are usually no longer practical for development of modern large-scale applications and their extension by the end user. Switching from a procedural to an object-oriented paradigm typically requires some effort since the thought process is different.
Keyword's – Oops, ADT, Objects, Class

I. INTRODUCTION

Object oriented programming is recognized for its ability to inherit and reuse the code. A high degree of reusability makes it especially suited for developing and maintaining large software. But as programs get larger, static type-checking becomes imperative, and this has been a concern of the OOP community from the beginning.
The representation of engineering systems in a manner suitable for computer processing is an important aspect of software development for computer aided engineering. The process of abstraction is a well-known technique for developing data representations. Objects are a mechanism for representing data using abstraction, and object-oriented languages are languages for writing programs to manipulate objects.
The widely recognized value of such software engineering techniques as information hiding, encapsulation, strict enforcement of interfaces, and layering were important. The language features that address these issues are those of objects, classes, inheritance, polymorphism, templates, and design patterns.

II. REUSABILITY, EXTENSIBILITY & FLEXIBILITY

Reusability is an important issue in software engineering for at least two major reasons. Firstly, reusability is one means to cope with the pressures of producing ever larger and more functional systems in a short period of time. Reusability allows developers to be more efficient because the same code can be developed once and used in many different applications. Secondly, reliability can be improved by reusing previously developed and previously tested components. The development of new code entails the additional costs in time and money of testing, validation, and verification of the new code. Much of these expenses can be avoided by using "off-the-shelf" components.
Software reuse is certainly not a goal unique to object-oriented programming. While libraries of procedures proved this approach to be useful, in practice procedures were too primitive a unit to promote extensive reuse. Objects and classes are more sophisticated mechanisms for achieving software reuse because they bind together more completely all the aspects of an entire abstraction. Therefore, the abstraction can more easily be transported across applications. Any of the forms of

generalization also contribute to reuse. A class in an inheritance hierarchy can be reused directly when it serves as a generalized base class from which a new class is derived by specialization. Templates can be reused by supplying different parameters for the template arguments. Design patterns allow design experience and success to be reused across designers.

Extensibility in software is important because software systems are long-lived and are subject to user's demands for new features and added capability. Object-oriented programming can help to satisfy this need through inheritance. Inheritance is a generalization/specialization hierarchy. Extensibility is possible in two ways. The first way in which a generalization/specialization hierarchy supports extensibility is that any new attributes or behaviour that is added to a more generalized concept will automatically become part of the attributes and behaviour of its specializations. Flexibility in software systems means, in part, that additions, variations or modification can be made without the need to modify numerous places in the system's code. Initially, many software systems were very brittle in that the addition of a small change could only be accommodated by making modifications in many, and often apparently unrelated, parts of the existing system. This brittle property stood in marked contrast to the prevailing notion that, like hardware systems, software system were supposed to be extremely malleable and changes can be made easily.

Object oriented programming offers flexibility in two ways. Firstly, the separation of an interface from its implementation allows the user of the interface to remain unaffected by changes in the implementation. Thus, a modification can be made to the implementation (to improve its efficiency or reliability) without requiring any changes in the code that uses the interface. Second, polymorphism allows variations and additions to be made to the set of classes over which the polymorphism applies.
Major Concepts in OOP:

- Encapsulation
- Inheritance
- Polymorphism

Encapsulation in this context means putting together the things that should be together, in particular attributes and operations (data and methods).
The fundamental underlying notion is that of "abstraction", and in particular the Abstract Data Type (ADT). An ADT is usually implemented by something called a "class". Data (attributes) and operations (behaviours, and also called functions, or methods) are combined in the class definition. Both are also referred to as "members" of the class. A program consists of a collection of "objects" of various classes that make things happen by communicating with each other by "sending messages"; i.e., one object can send a message to another object by asking that object to perform one of its methods (Java syntax: object. method (parameter_ list) The Principle of Information Hiding insists on a "contract" between the implementer and the user. This establishes a "division of responsibility" in which the implement or should be told only what is necessary for implementing the class, and the client should be told only what is necessary to use the class. In each case it's a question of "minding one's own business". This may be enforced, or at least encouraged by "access control" (public, private, protected) of class members.
Inheritance is a hierarchical relationship in which the members of one class are all passed down to any class that descends from (extends) that class, directly or indirectly. It is in this sense that we appear to be "getting something for nothing". This is just one of a number of different relationships that may exist between different classes.
Thus, inheritance is a mechanism for defining a new class based on the definition of a pre-existing class, in such a way that all the members of the "old" class (superclass, or parent class, or base class) are present in the "new" class (subclass, or child class, or derived class), and an object of the new class may be substituted anywhere for an object of the old class. This is the Principle of Substitutability. An inherited class may simply use the members that are already there, may add new members, or may "override" members that pre-existed in the parent or ancestor class (by giving those "overridden"

member's new definitions). An inheritance hierarchy is a tree-like mapping of the relationships that form between classes as the result of inheritance. (C++ allows multiple inheritance). Inheritance hierarchies should develop naturally as you program, and not be "forced" in any way. Polymorphism literally means "many forms". In OOP, polymorphism refers to the fact that a single name (like open, above) can be used to represent or select different code at different times, depending on the situation, and according to some automatic mechanism. So, for example, the method call object. DoIt() can mean different things at different times in the same program.

III. CONCLUSION

Object-oriented programming is becoming an important technique in the construction of large software systems. Compelling arguments, like reduced maintenance costs, are advanced to encourage its use. To maximise the advantages of such methods, object-oriented programming languages need to be well-designed. When selecting the main features of a programming language, or choosing between alternative designs, formal methods of semantic analysis are invaluable. To date little attention has been given to the formal description of object-oriented languages.

ACKNOWLEDGEMENT

This research paper is made possible through the help and support from everyone, including parents, teachers, family, friends, and in essence, all sentient beings. Especially, please allow us to dedicate our acknowledgment of gratitude toward the following significant advisors and contributors:
First and foremost, we would like to thank Mr. Jitendra Yadav for his most support and encouragement. He kindly read our paper and offered invaluable advices on organization and the theme of the paper.
Second, we would like to thank Ms. Karanpreet Kaur for teaching me about Object Oriented Programming over the past two years of my Bachelor's degree.
Finally, we sincerely thank to our parents, family, and friends, who provided the advice. The product of this research paper would not be possible without all of them.

REFERENCES

[1] Raghav Yadav, Seminar on software development and programming languages. (2013)

[2] G. Booch, "Object-*Oriented Analysis and Design with Applications*," Cummings, 1991.

[3] Jacobson, M. Griss, P. Johnsson, "*Software Reuse*," Addison-Wesley 1997.

[4] Stroustrup, B., *The C++ Programming Language*, Third Edition, Addison-Wesley, Reading, MA, 1997.

[5] Comparison of Object Oriented Programming to Procedural Programming Language, Journal by Sallie Henry.