International Journal of Scientific Research in Computer Science, Engineering and Information Technology
© 2019 IJSRCSEIT | Volume 5 | Issue 2 | ISSN : 2456-3307
DOI : 10.32628/CSEIT195248

A Survey of Object-Oriented Programming Languages

M. Surya[1], S. Padmavathi[2]
[1]Computer Science and engineering, Sri Krishna College of Technology, Coimbatore, Tamilnadu, India
[2]Assistant Professor, Computer Science and engineering, Sri Krishna College of Technology, Coimbatore, Tamilnadu, India

ABSTRACT

Object oriented programming has become a very important programming CONCEPT of our times.The time it was brought into existence by Simula. It directly support the object notions of classes, inheritance, information hiding, and dynamic binding. There is a variety of implementations for each of these concepts, and there is no general agreement as to how a particular concept must be interpreted. This survey takes a detailed look at the concepts which are fundamental to object-orientation, namely inheritance and polymorphism. Different aspects of inheritance and polymorphism are implemented in various popular Object oriented program language. We conclude that there is still lot of work to be done to reach a common ground for these to achieve features of OOPs. This survey presents a comparison of Java, C++, C# , Eiffel, Smalltalk, Ruby and Python in terms of their inheritance and polymorphism implementations. The paper also presents a compilation of the observations made by several surveys [1].

Keywords : OOPL, C++, C#

I. INTRODUCTION

There is a big variety of programming languages catering to various kinds of development requirements. Three of the main categories are procedural languages (e.g. C, Pascal, etc.), functional languages (e.g. Haskel, Ocaml, etc.), and object-oriented programming languages (e.g. C++, Java, etc.). The object-oriented design paradigm has been popular for some time owing its success to the powerful features it offers for making program development easy and robust. OOPLs, such as C++ and Java, offer an intuitive way of developing programs and provide powerful features for supporting the program development. While languages like C can be used to develop programs that follow an object-oriented design, the support of features such as inheritance, encapsulation, strong type support, exception handling, etc. in the OOPLs make them more suitable for such development. While the object oriented programming concepts provides a more intuitive way of programming, it is also has complexities. This is due to the various complex features that the paradigm provides. OOPLs differ widely in the way they implement features that are associated with the object design. For example, some languages support multiple inheritance while some other languages consider it a bad feature. In this survey we discuss the various features of object oriented programs and how the languages we considered differ in implementing these features. The survey is organized as follows.

II. KEY OBJECT-ORIENTED CONCEPTS

While OOPLs came into existence in 1960s , there is considerable disagreement on what characterizes object oriented programming. As recent as 2006, Armstrong [1] suggests that the key concepts of

object-oriented programming are not exist . This certainly makes it very hard to describe OOPLs, since there is not an agreement on a universal definition of what object orientedness is. Nierstrasz [27] suggests that languages have a part of object orientation that can be assessed by considering the support the languages provide for encapsulation. Thus, he assign encapsulation to be the fundamental building block of the object oriented paradigm. While Encapsulation is certainly a fundamental concept, it is not sufficient to define what the object oriented programming is. Armstrong [1] approaches this by taking a quantitative approach of considering the most commonly occurring concepts among various documents in the object-oriented programming literature. In other words, he performed a feature selection exercise over a corpus of documents which is used to extract the key concepts of object oriented programming. In this survey we select a subset of the "quark" which is identified by Armstrong , and we discuss how these "quarks" are implemented in the object oriented programming languages we studied.

2.1 Class

A class [5] provides the basic mechanism by which attributes and methods common to a concept are grouped together. It provides a description of runtime behavior of the objects instantiated from it. The object-oriented paradigm implies that the methods in a class are not based on some common algorithms. Instead, they are based on the intuitive understanding of what methods the modeled object is allowed to hold. The methods also depend on the level of detail at which the object is being modeled at. Thus, a class defines a logical grouping of methods and attributes. It acts as a means by which abstraction and encapsulation are achieved. The complex details of implementation are hidden within the abstraction (i.e. are implemented with the class workings), which aids in dealing with complexity. A well designed class will have an expected interface, which is considered as an immutable contract between the class and its client.

2.2 Abstraction

The abstraction is the simplified view of reality. The level depends on the object being abstracted and on the requirements of the problem domain. The abstraction is presented by the methods and the attributes that the class exports to the clients.

2.3 Inheritance

Inheritance [5] is the mechanism by which hierarchical class designs can be carried out. Creating a subclass of the original class provides inheritance from the original class properties. The new class inherits all of the existing properties, therefore, all the behavior of the original class. Inheritance promotes code reusability of code . A class with specialized behavior can be implemented by extending the generic superclass by modifying the methods dealing with specialization. The reuse occurs as the methods unmodified by a subclass which is provided by the super class. A subclass can extend all the aspects of its super class and can modify any behavior . Multiple inheritance allows for a class to inherit traits from multiple classes which is usually considered as a dangerous design mechanism.

2.4 Encapsulation

Encapsulation [5] is hiding the details of implementation within a class. The users are not allowed to peek down the class other than the standard interface. For example, any one can use any field of a structure . Encapsulation can be enforced by making certain fields private and clients cannot directly reference these fields though they are aware of the fields. Encapsulation allows for keeping a clearer boundary between a class and the external world, and it gives programmers the freedom to changing the internal workings of a class.

2.5 Polymorphism

Polymorphism [8] allows for significant programming by providing similar looking structure for handling a variety of objects. For example, methods doing similar job may have the same name with different signature the same class/method may work with multiple types of objects a subclass can be substituted for a parent class [8].

III. 3.OBJECT-ORIENTED PROGRAMMING LANGUAGES

3.1 A Brief History

An object oriented programming language is one which allows object oriented programming techniques such as encapsulation, inheritance, modularity, and polymorphism. Simula (1967) is accepted as the first language to have the primary features of an object oriented language which was created for making simulation programs. The idea of object oriented programming gained momentum in the 1970s with the introduction of Smalltalk (1972 to 1980), which has the concepts of class Simula. Smalltalk is the language, with the help of which much of the theory of object oriented programming was developed. Bjorn Stroustrup integrated object oriented programming into the C language by which the language generated called C++ which became the first object oriented language to be widely used. James Gosling developed a version of C++ called Java which was developed to let devices and peripherals and appliances which posses a common programming interface [7]. In 2000, Microsoft announced both the .NET platform and C# is similar in many respects to C++ and Java. Ruby and Python are scripting languages, they support the object oriented concepts, and we thought it would be interesting to scripting OOPLs in this survey.In pure OOPLs everything is treated as an object, from primitives such as integers, are way upto whole classes, prototypes, modules, etc. They are designed to facilitate, the object oriented paradigm. Of the languages that we considered, Smalltalk, Eiffel and Ruby are pure OOPLs. Languages such as C++, Java, C# , and Python were designed only for object oriented programming, but they also have some procedural elements. This is why they fall under the hybrid OOPLs category.

3.2 Smalltalk

Small talk [19] was the general purpose object oriented programming language. It is a pure dynamically object oriented language. Small talk supports a uniform object model. Everything a programmer deals with the object including primitive types and user-defined types. Clients can access the functionality of a class by invoking well defined methods. Hence, all operations are performed by sending messages to the objects. Small talk supports the ability to instantiate objects. Small talk supports full inheritance, where all the aspects of the parent class are available to the subclass. Small talk does not support multiple inheritance because Multiple inheritance can cause significant maintenance burden, as changes in any parent class will affect multiple paths in the inheritance hierarchy. Initial implementations of Small talk support reference counting for automatic memory management. The main idea is to reduce the programming burden. Moreover, encapsulation is not in Smalltalk but it allows direct access to the instance slots, and it also allows complete visibility of the slots.

3.3 C++

C++ [17] was developed by Bjarne Stroustrup (1979). It was designed for systems programming, extending the C programming language. C++ is an object-oriented version of C which has added support for statically typed object oriented programming, exception handling, virtual functions, and generic programming to the C programming language. C++ is not a pure object oriented languages, because are both procedural and objected oriented development. It give the concept of multiple inheritance and exception handling, but it does not provide garbage collection. C++ uses compile-time binding, means that the programmer must specify the specific class of

an object. This makes for high run time efficiency , but it trades off some of the power of reuse classes [13]. Unlike Java, it has bounds checking, it provides access to low-level system facilities. C++ pointers can be used to manipulate specific memory locations, which is a task necessary for writing low-level operating system components.

3.4 Java

Java [16] is an object oriented language. It has similar syntax to C++ which make it easier to learn. However, Java is not compatible with C++ which does not allow low level programming constructs, which ensures type safety and security. Java does not support C/C++ pointer arithmetic, which allows the garbage collector to relocate referenced objects, and ensures type safety and security. similar to Small talk, Java has garbage collection which runs on a protected java virtual machine. Java is a portable language that can run on any web-enabled computer via that computer's web browser. A major benefit of Java byte code is portability, since the byte code can be executed independent of the operating system on a given computer. However, running interpreted programs is always slower than running programs compiled to native executables [16].Java has class hierarchy with class Object at the root and provides single inheritance of classes. Java provides interfaces along with multiple inheritance. Java is considered an impure object oriented language because its built-in types are not objects , it has implemented basic arithmetic operations as built-in operators, rather than messages to objects.

3.5 C#

C# [6] is an OOP language part of the .NET framework and it is not a pure OOPLs since it encompasses functional, imperative and component-oriented programming in addition to the object-oriented concepts. It has an object oriented concepts based on C++ and is heavily influenced by Java. In some communities it is has been assigned as Microsoft's version of Java. similar to Java, it has garbage collection and it is compiled to an intermediate language, which is executed by the runtime environment known as Common Language Runtime which is similar to the JVM. The C# conception of class and instances, inheritance and polymorphism, are relatively standard. Methods are more interesting because of the introduction of so-called properties and delegates [9].

3.6 Eiffel

Eiffel is a language, which was developed in 1985. Eiffel is a pure object-oriented language. The design is based on classes and All messages are directed to a class. A class has ability to export some of its attributes for user visibility and keep others hidden. Eiffel enables the use of assertions which express formal properties of member methods in terms of preconditions, post conditions, and class in variants. Multiple inheritance is permitted in Eiffel. The name conflict issue is solved by providing ability to rename the inherited names. Duplicate names are not allowed. Several other feature adaptation are available to make multiple inheritance safe. To avoid wrong definitions all the assertions defined in parent classes are inherited. Thus class designers can choose to define tight constraints that ensure their subclasses do not deviate much . The ability to assign a subclass object to a superclass pointer is provided with static checking. Encapsulation is supported in Eiffel. The class designer controls the visibility of class features. A method can be explicitly made available to all subclasses. The data can be exported in a read only fashion. There is no syntactic difference between a attribute access and access to a method with no parameters.There is no control on the inherited attributes. A subclass inherits all the attributes of the parent class and can change the visibility of the attributes.

3.7 Ruby

Ruby [14] is an object-oriented scripting language developed by Matsumiko Yukihiro. It is similar in purpose to python. Ruby is designed to be an object-oriented programming language based on Perl, and which borrows features from several Object Oriented languages like Eiffel and Small talk. Ruby has a pure object oriented which does not allow functions. All methods must belong to few class. Ruby only supports single inheritance, though multiple inheritance functionality is indirectly supported. A module provides a partial class definition.

3.8 Python

Python is an object oriented scripting language developed by Guido Van Rossum. It has become very popular in recent years because its application in the internet domain. Python allows both procedural and objected oriented development. Developers can write classes and methods in them and can also write functions . There is a different syntax for invoking methods which is opposed to invoking functions and this brings out the heterogeneous nature of python programming. A programmer can define his own classes, abstraction which is supported by python. The encapsulation however is not fully supported in which access control is primitive in Python. There are no public methods , private methods and the only protection is by name mangling. If a programmer knows how name mangling is performed he could invoke any class method. Python allows multiple inheritance. The issue of name clashes in multiple inheritance is resolved by l programmer define the order of superclasses by the order in which they are declared.

IV. 4 INHERITANCE AND POLYMORPHISM IN OOPL

4.1 Inheritance

Inheritance is a fundamental object oriented technique. This also has been the most controversial feature of OOPLs. Inheritance is the language feature which allows code reuse at the level of software modules called “classes”. Inheritance can be used for many purposes which is used to represent a subtype, to generalize, to specialize, to add restriction, etc. It is suggested that it is not a good to mix various uses of inheritance in a project [12].

4.1.1 Class Hierarchy

Inheritance is a mechanism which brings hierarchical relationship into the class model. Without this hierarchical relationships, having a set of unrelated classes would be too hard to manage. The hierarchical relationship can be extended to all classes which is done by languages like Java, C#, Eiffel and Small talk. These languages define the most generic class that are ancestor for all the classes in the language. For Java and Small talk this is “Object”, while for Eiffel this is “Any” [20]. The presence of a single ancestor implies that all classes can have minimal functionality in which they inherit from this ancestor. The advantage is that object of any class can be downcast to the pointer of the ancestor. This allows C’s void pointer like functionality even in strongly types languages. C++ does not have any notion of a single ancestor class. In C++, classes are designed as a forest of class hierarchies. The advantage of this approach is the application does not need to link with the entire object hierarchy for its operation. For a Java , all the classes in the hierarchy must be present. In C++ application can just link with a subset of classes.

4.1.2 Control of Superclass

Inheritance provides ability to represent an “is a” relationship in software. The fundamental way of using inheritance is to define a subclass which inherits all the attributes of a parent class. The subclass may extend or specialize the inherited code and this depends on the class use. The changes for extension may happen in a way that violates the “is a” relationship. It is very hard for a programming language for ensure that inheritance is properly

applied or not. This is the reason why inheritance is a controversial feature. For example, let us consider a draw() method inherited from a shape object into a rectangle object. The rectangle's draw() could be coded to draw itself. But while doing , it could change some of the semantics of the inherited draw(). The programming language cannot ensure that rectangle's draw() conforms to certain semantics that Shape's draw defines. Eiffel programming language associates some conditions with each class method, and these are inherited by the sub classes. This can be used to control the amount by which a subclass method conforms to the base method specification. When a subclass chooses to redefine a method,only the in variants from the inherited class are applied to the new definition. This could potentially weaken the ability of a super class designer to ensure that subclass methods follow semantics, but it still leaves some scope for restricting what a subclass method can change. Other languages like C++ and Java leave this upto the programmer's discretion.

4.1.3 Violation of Encapsulation

To supporting reuse, inheritance allows developer to program generalization relationships. For example, a vehicle is a concept which can be specialized either as a car or as a truck. This is supported by allowing methods to be overridden. This extension causes inheritance to work against the encapsulation. Encapsulation requires a well defined interface between a class and its clients. Clients are allowed to access only to the certain services and no other. However, inheritance introduces another kind of clients for a class services. The descendants are allowed to access the almost all the base class attributes violating encapsulation. Violations for encapsulation can have significant impact to code maintainability and also the freedom of the class designer in modifying the base class. This can be addressed by defining a well defined interface for the descendants. Languages like C++ and Java allows control of interface to the descendants. Keywords like protected, private and public can be applied to methods to control their visibility to the descendants. A public feature is visible to all classes. A private feature is visible to no other class. A protected feature is visible to a class and There is no method by which C++ or Java can make features public only to some classes . When a class is inherited the visibility constraints are also inherited by the subclass. The subclass can access the protected variables, but it cannot access the private features. The subclass can decrease the visibility by making the public features protected or private. The feature visibility cannot be increased. From the maintenance perspective, protected features do not help much. Protected features are visible to subclasses, making any changes to the features would be hard. A private feature is not visible even to the sub class and may be the best for future extensibility of the class. Eiffel takes a different approach in handling the feature visibility and Each feature in a class can be separately exported to any set of classes including the null set. When a feature is visible to a class, it is visible to the descendants. If a feature is visible to "None", then this is considered a private variable. A feature exported to "Any" is visible to all classes. This is because the "Any" class is the root class for the single tree class hierarchy in Eiffel. When a subclass inherits the features, these visibility constraints are also inherited. However, the subclass is allowed to redefine the constraints arbitrarily. In contrast, a subclass in C++ and Java can never weaken the constraint set by the parent class. Hence, Eiffel sub classes have power to adjust visibility of each feature selectively unlike in C++ or Java. Small talk has no features for encapsulation. All its features are public and visible to all classes.

4.1.4 Miscellaneous Issues in Inheritance

Inheritance has adverse effect on synchronization requirements of a concurrent object. The concurrent object-oriented community has named this "inheritance anomaly". When a class with

synchronized code is derived, it necessitates redefinition of the inherited methods in order to preserve the synchronization requirements, and this denies the "reuse" benefit of inheritance. This is seen in COOL languages like Java. Java uses monitors for synchronized methods and exhibits the anomaly described as "history dependent anomaly" [25].

4.2 Multiple Inheritance

Multiple inheritance allows a class to inherit from multiple parent classes. There are many situations in which multiple inheritance is required to clarify a design. A person that is both a doctor and an author, for example, can be properly modelled by designing a class that inherits from both the "doctor" and the "author" classes. This model is more closely parallels inheritance as observed in the biological beings. This introduces new issues and has been subject of controversy. When a class inherits from multiple classes, the class is not likely to be "is-a" version of any of the parent classes. A programmer can use multiple inheritance to reuse from potentially unrelated classes. As discussed before, inheritance weakens encapsulation of the base classes and multiple inheritance increases this risk considerably. In addition to this objection, there are other interesting reasons why multiple inheritance is a problem. These problems occur due to the fact that multiple inheritance defines an inheritance graph that is structured as a directed acyclic graph (DAG) and not a tree. In a tree there is a path from any derived class to any ancestor node. But in a DAG structure, there can be multiple paths between a subclass and its ancestors. Now when subclass refers to an inherited feature name, searching for this feature name poses significant problems.

Types of Inheritance:

- Tree-Based Inheritance
- Graph-Based Inheritance
- Linearized Inheritance
- Alternatives to Multiple Inheritance

V. POLYMORPHISM

Polymorphism allows programmers to write functions and classes that work uniformly with different types, and the different notions of what a type is give rise to different from of polymorphism [18]. Ad-hoc polymorphism is obtained when a function works, or appears to work, on several different types and may behave in unrelated way for each type. Universal polymorphism a class or a function behaves in the same way on infinitely many types. The main difference between ad-hoc polymorphism and universal polymorphism is that ad-hoc polymorphic functions execute distinct code for a small set of potentially unrelated types, while universal polymorphic functions execute the same code for an infinite number of types.

There are two major kinds of ad-hoc polymorphism they are overloading and coercion. Overloading polymorphism refers to the use of a common function name (including symbolic names in the case of operators) to refer to different actual functions that are distinguished with respect to the types and the number of the arguments. Coercion polymorphism allows for a value of one type to be converted to a value of another. The distinction between overloading and coercion polymorphism in many cases is not very clear and depends on the implementation, in particular, when considering untyped languages and interpreted languages.

5.1 Overloading

Overloading polymorphism refers to the use of a common function name (including symbolic names in the case of operators) to refer to different actual functions that are distinguished with respect to the types and the number of the arguments [8]. In our discussion we distinguish between operator overloading and method overloading.

Operator Overloading allows for some operators to have different implementations depending on the

types of the arguments. For example, operators such as '+' and '*' are applicable to both integer and real arguments. Operator overloading is also known as the ability of a programmer to define operators (such as '+', '*', and '==') for user-defined types. Operator overloading is useful because it allows user defined types to behave as the types built-in the language. New operators cannot be created, only the functionality of the existing operators on objects can be modified.

Of the languages under consideration, C++, C#, Eiffel, Ruby and Python support operator overloading. To make the use of operator overloading safer some languages such as Ruby and Eiffel require that all operations must be messages to objects (i.e. all operators are always method calls) and that operators must have an equivalent functional form, so that using the operator as a method call will behave precisely the same as using it in infix, prefix, or postfix form [20, 14]. The support of these two criteria allows for safer use of operator overloading. Python only supports the criteria that all operators must have an equivalent form, whereas C++ and C# do not support either of the above criteria. Java only allows operators for arithmetical operations to be overloaded for all numeric types and string, but both arguments must be of the same type. However, it is not possible to give new meanings to the operators in Java. Moreover, an unique feature of Eiffel is that users can defined arbitrary operators, rather than being limited to redefining a set of predefined operators.

Method Overloading is the ability of a class to have two or more methods with the same name. In other words, a method name is used to represent two or more distinct methods, and calls to these methods are disambiguated by the number and the types of the arguments passed to the method (i.e. by the method signature) [9]. Of the languages under consideration, Java, C++ and C# allow method overloading and they support it in a similar fashion. As long as the methods signatures are different, the compiler treats methods with overloaded names as though they had completely different names. The compiler statically (using early binding) determines what method code is to be executed [18]. Moreover, in Java, C++ and C# the overloading can happen when a method in a superclass is inherited in a subclass that has a method with the same name, but different signature. Then the respective compilers again use early binding to differentiate between the overloaded methods. Ruby, Eiffel, Smalltalk and Python do not support method overloading.

5.2 Coercion

Coercion is a semantic operation which is needed to convert an argument of one type to the type expected by a function [8]. For example, when an integer value can be used where a real is expected, and vice versa. Coercion can be provided statically, by automatically inserting the required type conversion code between arguments and functions at compile time, or the necessary type conversions may have to be determined dynamically by run-time tests on the arguments. Coercions are essentially a form of abbreviation which reduce program size and improve program readability, but it may also cause subtle and sometimes dangerous system errors. Coercion does not achieve true polymorphism [9]. Although an operator may appear to accept values of many types, the values must be converted to some representation before the operator can use them. Hence, the operator really works on only one type.

5.3 Parametric Polymorphism

Parametric polymorphism is obtained when a function works uniformly on a range of types, which normally exhibit some common structure. It uses type parameters to determine the type of the argument for each application of the function or the class. Functions and classes that exhibit parametric polymorphism are also called generic functions and

classes, respectively. The primary benefit of parametric polymorphism is that it allows statically typed languages to retain their compile-time type safety yet remain nearly as flexible as dynamically typed languages. Dynamically typed languages such as Smalltalk, Ruby and Python do not need parameterized types in order to support generic programming. Types are checked at runtime, and thus dynamically typed languages which support generic programming inherently. Hence, of the languages that we considered parametric polymorphism is only relevant for C++, C#, Java and Eiffel. Eiffel in particular uses generics extensively a mechanism for type safe generic containers and algorithms. C++ templates are even more flexible, having many uses apart from simple generic containers, but are also much more complex [11].

5.4 Inclusion Polymorphism

Inclusion polymorphism models subtyping and subclassing (inheritance) in object-oriented languages. It refers to the capability of having an object's specific class/type not known until runtime. Moreover, inclusion polymorpshim can be viewed as inclusion of classes/types, where an object can belong to many different types that need not to be disjoint. In other words, it is the ability of different classes to respond to the same message and each implement the method appropriately [8]. For object-oriented programmers polymorphism almost always means inclusion polymorphism. Inclusion polymorphism is implemented through dynamic binding. In the context of object oriented languages, the dynamic binding refers to runtime binding. Dynamic binding is important whenever a child class has overridden a method of the parent class, and the class of each object in the program is not obvious at compile time.

VI.CONCLUSION

Object-oriented programming languages are used across the world on many different projects and applications. Mastery of the object-oriented paradigm has become an essential part of any programmer's careers. The key features of the object-oriented paradigm (abstraction, encapsulation, inheritance, and polymorphism) have different flavors in the various OOPLs available to the users. For example, inheritance has been implemented in a variety of ways by OOPLs. Some of these implementations allow the inheritance to become a dangerous feature for the software development and some provide useful safeguards against such abuse. There is still lot of work to be done not only to reach a common representation for these crucial features of OOPLs, but also to to find appropriate ways to implement features such as inheritance and polymorphism to avoid misuse. In the survey, we have only considered a snapshot in the time of language evolution. The current languages (Java, Python, Ruby, C# , etc) would most likely add new features, which might favor development of safer programs, or they might bow to the popular pressure and add potentially unsafe features into the languages (generics in Java). Moreover, we believe that Eiffel should be appreciated as a OOPL, which has a good balance of features that provide programmers abundant safe guards for stable design. A language feature by itself is not a bad thing, as long as the language can provide controls against abuse. It is clear that each language has a 1 different priority and different reasons for adding in features. This will keep the field of object-oriented programming ripe for awhile with no clear single best object-oriented language.

VII. REFERENCES

[1]. D.J. Armstrong. The quarks of object-oriented development. Communications of the ACM, 49(2):123-128, 2006.

[2]. Isabelle Attali, Denis Caromel, and Sidi Ould Ehmety. A natural semantics for eiffel dynamic binding. ACM Trans. Program. Lang. Syst., 18(6):711-729, 1996.

[3]. G. Bracha and W. Cook. Mixin-based inheritance. In Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications, pages 303-311. ACM New York, NY, USA, 1990.

[4]. K.B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good match for object-oriented languages. Lecture Notes in Computer Science, 1241:104-127, 1997.

[5]. Kim B. Bruce. Foundations of object-oriented languages: types and semantics. MIT Press, Cambridge, MA, USA, 2002.

[6]. Vinny Cahill and Donal Lafferty. Learning to Program the Object-Oriented Way with C# . Springer-Verlag New York, Inc., 2002.

[7]. Luiz Fernando Capretz. A brief history of the object-oriented approach. SIGSOFT Softw. Eng. Notes, 28(2):6, 2003.

[8]. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. ACM Comput. Surv., 17(4):471-523, 1985.

[9]. Iain Craig. The Interpretation of Object-Oriented Programming Languages. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[10]. I.D. Craig. Programming in Dylan. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1997.

[11]. T. V. Cutsem. Eiffel and c++: A comparison between object oriented languages. http://prog.vub.ac.be/ tvcutsem/publications/oo comparison.pdf.

[12]. S.H. Edwards. Inheritance: One mechanism, many conflicting uses. In Proc. 6th Ann. Workshop on Software Reuse.

[13]. Margaret A. Ellis and Bjarne Stroustrup. The annotated C++ reference manual. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[14]. D. Flanagan and Y. Matsumoto. The ruby programming language. O'Reilly Media, Inc., 2008.

[15]. Jerry Gao, Chris Chen, Y. Toyoshima, David Kung, and Pei Hsia. Identifying polymorphism change and impact in object-orientated software maintenance. Journal of Software Maintenance, 8(6):357-387, 1996.

[16]. J. Gosling, B. Joy, G. Steele, and G. Bracha. Java (TM) Language Specification, The (Java (Addison-Wesley)). Addison-Wesley Professional, 2005.

[17]. H. Schmidt. C/C++ Programmer's Reference. McGraw-Hill, 2003.

[18]. K. Henney. Promoting Polymorphism, 2001.

[19]. D.H.H. Ingalls. The Smalltalk-76 programming system design and implementation. In Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 9- 16. ACM Press New York, NY, USA, 1978.

[20]. Ian Joyner. Objects Unencapsulted: Java, Eiffel, and C++?? Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[21]. D. Kaustub and D. Grimes. A comparison of object oriented scripting languages: Python and Ruby, 2001.

[22]. B.B. Kristensen, O.L. Madsen, B. Moller-Pedersen, and K. Nygaard. The BETA programming language. MIT Press Cambridge, MA, USA, 1987.

[23]. H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented

systems. In Proceedings of the 1986 conference on Object-oriented programming systems, languages, and applications, volume 21, pages 214-223. ACM New York, NY, USA, 1986.

[24]. Juval Lowy. An Introduction to C# Generics. Technical report, Visual Studio 2005 Technical Articles, 2005.

[25]. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. 1993.

Cite this article as :

M. Surya, S. Padmavathi, "A Survey of Object-Oriented Programming Languages", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 5 Issue 2, pp. 187-197, March-April 2019.
Journal URL : http://ijsrcseit.com/CSEIT195248