A Survey of Object Oriented Programming Languages

Maya Hristakeva, RadhaKrishna Vuppala
Univ. of California, Santa Cruz
{mayah,vrk}@soe.ucsc.edu

1 Abstract

Object-oriented programming has become a very important programming paradigm of our times. From the time it was brought into existence by Simula, object-oriented programming has seen wide acceptance. Object-oriented programming languages (OOPLs) directly support the object notions of classes, inheritance, information hiding (encapsulation), and dynamic binding (polymorphism). There is a wide 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 some of the concepts considered fundamental to object-orientation, namely inheritance and polymorphism. Different aspects of inheritance and polymorphism are implemented in various popular OOPLs. We conclude with the observation that there is still lot of work to be done to reach a common ground for these crucial features of OOPLs. This survey presents a detailed 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 earlier surveys [1, 27].

2 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 quite 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 paradigm provides a more intuitive way of programming, it is also has complexities. This is due to the various complex features that the paradigm offers. 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 languages consider it a bad feature. In this survey we discuss the various features of object-oriented programs and how the languages we considered
(Java, C++, C#, Eiffel, Smalltalk, Ruby and Python) differ in implementing these features.

The survey is organized as follows. The Section 3 describes in detail the key concepts of OOPLs. Section 4 presents a brief historical view of the OOPLs and gives a brief overview of each of the languages we studied. Section 5 describes the comparison between various languages in terms of their support for inheritance and polymorphism. Section 6 summarizes the key learnings in the paper.

3 Key Object-Oriented Concepts

While OOPLs came into existence in 1960s (through the Simula [19] language), there is considerable disagreement on what characterizes object-oriented programming. As recent as 2006, Armstrong [1] suggests that consensus does not exist on what the key concepts of object-oriented programming are. This certainly makes it very hard to describe and define OOPLs, since there is not an agreement on a universal definition of what object-orientedness is. Nierstrasz [27] suggests that all languages have a degree of object-orientation, which can be assessed by considering the support the languages provide for encapsulation. Thus, he considers 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 question 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 to extract the key concepts of object-oriented programming. In this survey we select a subset of the “quark” that Armstrong identified, and we discuss how these “quarks” are implemented in the object-oriented programming languages we studied.

3.1 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.

3.2 Abstraction

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

Inheritance [5] is the basic 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 messages, and therefore, all the behavior of the original class. Inheritance promotes code reuse and structured code development. A class with specialized behavior can be implemented by extending the generic superclass by only modifying the methods dealing with specialization. The reuse occurs as the methods unmodified by a subclass are provided by the superclass. A subclass can extend all the aspects of its superclass and can modify (or override) any behavior as its sees fit (as long as the design is maintained). Multiple inheritance allows for a class to inherit traits from multiple classes and is usually considered a dangerous design mechanism.

3.4 Encapsulation

Encapsulation [5] is defined as hiding the details of implementation within a class. The clients are not allowed to peek down the class other than through the standard interface. (In C++ for example, any one can use any field of a structure if they can see it (i.e. in a header file.) Encapsulation can be enforced by making certain fields private (in Java for example) 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 of changing the internal workings of a class.

3.5 Polymorphism

Polymorphism [8] allows for significant programming convenience by providing similar looking structure for handling a variety of objects. For example, methods doing similar job (e.g. addition) may have the same name with different signature (overloading polymorphism); the same class/method (e.g. sorting) may work with multiple types of objects (parametric polymorphism); a subclass can be substituted for a parent class (inclusion polymorphism) [8].

We would cover the different types of polymorphism and inheritance in the subsequent section.

4 Object-Oriented Programming Languages

4.1 A Brief History

As discussed in the previous section, an object-oriented programming language is one that allows object-oriented programming techniques such as encapsulation, inheritance, modularity, and polymorphism. Simula (1967) is generally accepted as the first language to have the primary features of an object-oriented language. It 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 embraced the concepts of class and message of Simula. Smalltalk is the language with which much of the theory of object-oriented programming was developed. In the early 1980s, Bjorn Stroustrup integrated object-oriented programming into the C language. The resulting language was called C++ and it became the first object-oriented language to be widely used commercially. Then in the 1990s a group at sun led by James Gosling developed a similar version of C++ called Java that was developed to let devices, peripherals and appliances posses a common programming interface [7]. In 2000, Microsoft
announced both the .NET platform and a new programming language called C#. C# is similar in many respects to C++ and Java. Ruby and Python are scripting languages which support the object-oriented paradigm, and we thought it would be interesting to include some scripting OOPLs in this survey.

Properties of Smalltalk, Java, C++, C#, Eiffel, Ruby and Python are common OOPLs are outlined in this section.

<table>
<thead>
<tr>
<th></th>
<th>Smalltalk</th>
<th>Java</th>
<th>C++</th>
<th>C#</th>
<th>Eiffel</th>
<th>Ruby</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-Orientation</td>
<td>Pure</td>
<td>Hybrid</td>
<td>Hybrid</td>
<td>Hybrid</td>
<td>Pure</td>
<td>Pure</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Static/Dynamic Typing</td>
<td>Dynamic</td>
<td>Static</td>
<td>Static</td>
<td>Static</td>
<td>Static</td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Inheritance</td>
<td>Single</td>
<td>Single (class)</td>
<td>Multiple</td>
<td>Single (class)</td>
<td>Multiple</td>
<td>Single (class)</td>
<td>Multiple</td>
</tr>
<tr>
<td>Method Overloading</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Operator Overloading</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Generic Classes</td>
<td>N.A.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N.A.</td>
<td>N.A.</td>
<td>N.A.</td>
</tr>
<tr>
<td>Dynamic Binding</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

In pure OOPLs everything is treated consistently as an object, from primitives such as integers, all the way up to whole classes, prototypes, modules, etc. They are designed to facilitate, even enforce, 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 mainly for object-oriented programming, but they also have some procedural elements. This is why they fall into the hybrid OOPLs category.

4.2 Smalltalk

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

4.3 C++

C++ [17] was developed at Bell Labs by Bjarne Stroustrup (1979). It was designed mainly for systems programming, extending the C programming language. C++ is an object-oriented version of C. It 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 language, because both procedural and object-oriented development. It provides multiple inheritance and exception handling, but it does not provide garbage collection. C++ uses compile-time binding, which
means that the programmer must specify the specific class of an object, or at the very least, the most
general class that an object can belong to. This makes for high run-time efficiency and small code size,
but it trades off some of the power of reuse classes [13]. Unlike Java, it has optional bounds checking, it
provides access to low-level system facilities. C++ pointers can be used to manipulate specific memory
locations, a task necessary for writing low-level operating system components.

4.4 Java

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

C#  

C# [6] is an OOP language part of the .NET framework. C# is not a pure OOPLs since it encompasses functional, imperative and component-oriented programming in addition to the object-oriented paradigm. It has an object-oriented syntax based on C++ and is heavily influenced by Java. In some communities it is thought of as Microsoft’s version of Java. Like 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 (CLR) which is similar to the JVM. The C# conception of class and instances, as well as inheritance and polymorphism, are relatively standard. Methods are somewhat more interesting because of the introduction of so-called properties and delegates [9].

4.5 Eiffel

Eiffel [26] is a proprietary language, which was developed in 1985 as a pure object-oriented language.
The design is based on classes. All messages are directed to a class. A class has ability to export some of
its attributes for client visibility and keep others hidden. Eiffel enables use of assertions which express
formal properties of member methods in terms of preconditions, post conditions, and class invariants.
Multiple inheritance is permitted in Eiffel. The name conflict issue is resolved by providing ability to
rename the inherited names. Duplicate names are illegal. Several other feature adaptation clauses are
available to make multiple inheritance safe. To avoid wrong definitions of the inherited functions, all
the assertions defined in parent classes are inherited too. Thus, class designers can choose to define tight
constraints that ensure their subclasses do not deviate much from the expected semantics. The ability to
assign a subclass object to a superclass pointer is provided with static checking ensuring that such is the case.

Encapsulation is well supported in Eiffel. The class designer controls the visibility of class features. A method can be explicitly made available to all or some or none of the 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. However there is no control on the inherited attributes. A subclass inherits all the attributes and can change the visibility of the attributes.

4.6 Ruby

Ruby [14] is an object-oriented scripting language developed in 1993 by Matsumiko Yukihiro. It is similar in purpose to python or Perl. Ruby is designed to be an object-oriented programming language based on Perl, and which borrows features from several OO languages like Eiffel and Smalltalk. Ruby has a pure object-oriented support as it does not allow functions. All methods must belong to some class. Ruby only supports single inheritance, though multiple inheritance functionality is indirectly supported through the modules. A module provides a partial class definition.

4.7 Python

Python [30] is an object-oriented scripting language developed in 1990 by Guido Van Rossum. It has become very popular in recent years due to its application in the internet domain. Python allows both procedural and object-oriented development. Developers can write classes and define methods in them and can also write functions (routines not in a class). There is a different syntax for invoking methods as opposed to invoking functions and this brings out the heterogeneous nature of python programming. Since a programmer can define his own classes, abstraction is supported by python. The encapsulation however is not fully supported as access control is primitive in Python. There are no public, private methods and the only protection is by name mangling. If a programmer knows how name mangling is performed (which is very simple and known mechanism) he could invoke any class method. Python allows multiple inheritance. The issue of name clashes in multiple inheritance is resolved by letting programmer define the order of super classes by the order in which they are declared.

5 Inheritance and Polymorphism in OOPLs

5.1 Inheritance

Inheritance is a fundamental object-oriented technique for software reuse. This also has been the most controversial feature of OOPLs. Inheritance is the language feature that allows code reuse at the level of software modules called “classes”. There are many aspects to the inheritance. We will discuss the uses of inheritance, the controversies surrounding it, various aspects of its implementation and recent developments in this section. Much of the material has been obtained from various sources cited and primarily from Craig Iain’s text [9].

There is no agreement on the various roles that inheritance plays [12, 34, 4]. Inheritance can be used for
many purposes: to represent a subtype, to generalize, to specialize, to add restriction, etc. It is suggested that it is not a good idea to mix various uses of inheritance in a project [12].

5.1.1 Class Hierarchy

Inheritance is a mechanism that brings hierarchical relationship into the class model. Without this hierarchical relationships, having a set of unrelated classes would be a bit too hard to manage. The hierarchical relationship can be extended to all classes as done by languages like Java, C#, Eiffel and Smalltalk. These languages define the most generic class that is an ancestor for all the classes in the language. For Java and Smalltalk this is “Object”, while for Eiffel this is “Any” [20]. The presence of a single ancestor implies that all classes can have certain minimal functionality that they inherit from this ancestor. Another important 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 a notion of a single ancestor class. In C++, classes are modeled as a forest of class hierarchies. The advantage of this approach is that an application does not need to link with the entire object hierarchy for its operation. For a Java executable to run, all the classes in the hierarchy must be present. A C++ application can just link with a subset of classes.

5.1.2 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 base class. The subclass may choose to extend or specialize the inherited code and this depends on the class use. However, the changes for extension or specialization 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. This is the reason why inheritance is such a controversial feature. For example, 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 so, it could change some of the semantics of the inherited draw() (e.g. it could interact with the user or even compute some unrelated computation). The programming language cannot ensure that rectangle’s draw() conforms to certain semantics that Shape’s draw defines.

Eiffel programming language associates pre/post conditions with each class method, and these are inherited by the subclasses. 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, however, only the invariants from the inherited class are applied to the new definition. This could potentially weaken the ability of a superclass designer to ensure that subclass methods follow semantics, but it still eaves some scope for restricting what a subclass method can change. Other languages like C++ and Java leave this upto the programmer’s discretion.

5.1.3 Violation of Encapsulation

In addition to supporting reuse, inheritance allows developer to program generalization/specialization relationships. For example, a vehicle is a general 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 [31]. Encapsulation requires a well defined interface between a class and
its clients. Clients are allowed access only to certain services and no other. However, inheritance introduces another kind of clients for a class services. The descendants are allowed access to almost all the base class attributes/methods violating encapsulation. Violations to encapsulation can have significant impact to code maintainability and also to 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 allow control of interface to the descendants. Keywords like protected, private and public can be applied to methods and attributes to control their visibility to the descendants. A public feature is visible to all classes. A private feature is visible to no other class. (However there is an exception practiced by C++. It allows “friend” classes to access even the private attributes and methods. This is useful for implementing a separate iterator class, for example. However, this lets another class to access internals of a class, and thus, makes it hard for the designers to modify the internal details.) A protected feature is visible to a class and its descendants. There is no method by which C++ or Java can make features public only to certain classes (i.e there is no selective export). When a class is inherited the visibility constraints are inherited by the subclass. The subclass can access the protected variables, but cannot access the private features. The subclass can decrease the visibility by making public features protected or private. The feature visibility cannot be increased. From the maintenance perspective, protected features do not help much. As protected features are visible to subclasses, making any changes to the features would be hard. A private feature is not visible even to the subclass and may be the best way for future extensibility of the class.

Eiffel takes a different approach in handling the feature visibility. 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 also 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 by it. However, the subclass is allowed to redefine the constraints arbitrarily. In contrast, a subclass in C++ and Java can never weaken the visibility constraint set by the parent class. Hence, Eiffel subclasses have power to adjust visibility of each feature selectively unlike in C++ or Java. Smalltalk has no features for encapsulation. All its features are public and are visible to all classes.

5.1.4 Calling Abstract Methods

When an inherited method is being redefined in a subclass, the method definition could be of entirely new meaning that only the specification of the inherited method is preserved. However the new method can be defined in terms of the more abstract methods in the class hierarchy. This can occur when the new method extends the behavior by adding subclass specific behavior to the inherited behavior. In such circumstances, the ability to invoke method in a parent class is very useful. All the languages we considered provide this ability in one form or the other.

C++ uses its scope resolution operator to provide ability for invoking methods in a super class. When method names in multiple applicable scopes are the same, the conflict can be resolved by applying the scope resolution operator as $C_i :: f_i$ where $C_i$ is the name of a class (or a namespace in general) and $f_i$ is the method name. Same mechanism can be used by a method to invoke the methods from a superclass.
This method is general enough for a method to invoke any of the ancestor methods.

Smalltalk and Java provide a special keyword “super()” for invoking the methods in the immediate parent class. When called, super() invokes the method with the same name and signature in the parent class. This is often used in the class constructors. The “super” keyword can be used as an alias for the parent object in Java. Thus, super.add() for example refers to the add() method in the parent class. Eiffel has a similar mechanism that uses a “Precursor” call to invoke a parent method.

Another mechanism that has been used for this purpose is the “next-method” used in the languages like Dylan and CLOS [10, 33]. While these languages are not part of the survey, the mechanism is interesting enough to merit a mention here. The next-method does not take any arguments and is a pseudo-call that involves a call to another method. When next-method is invoked by a method, the next-method mechanism searches the class hierarchy for a method that matches the name and the signature of the caller. The most specific implementation of the method is found and executed.

There is a top-down approach to this ability implemented in languages like Beta [22]. The process of a method definition is inverted in this mechanism which uses the “inner” keyword. This keyword is placed in the superclass method and acts as a place holder for calls to more specialized methods. When the superclass method is directly invoked (i.e. method is invoked through a superclass object) the inner keyword has no effect. When a subclass method is being redefined, the method can include “inner statements”, which means that the statements should be executed in the inner context. Now, when the method is called on the subclass object, it will execute the superclass method with the inner statements filled. The inner statement thus provides a way by which extension code (code provided by the subclass) can be executed in the place marked by the superclass. The superclass code must be written carefully so that it can work for all possible inner statements. This is a way of combining statements from multiple points in the class hierarchy. CLOS allows for more general method combining mechanisms.

5.1.5 Restricting Redefinition

There are some situations where the method provided by a superclass is essential enough to be exported to all descendants, but it is also critical that it should not be redefined. Same may be said about some the data attributes. Many languages provide ability to control the redefinition behavior by the subclasses. Java and C# use the “final” keyword to indicate that a method or class cannot be redefined. Eiffel uses “frozen” to indicate that the class cannot be redefined. C++, Smalltalk do not have this ability, allowing for all methods to be redefined.

5.1.6 Abstract Superclass

There are some situations when only specification is to be inherited without any attached implementation. A subclass is forced to provide implementation, but also is forced to adhere to the specification. This mechanism can be used for ensuring that the subclass conforms to some design. Java and C# use “abstract” keyword to indicate that the method or class has no implementation, and hence, subclasses must provide the definition. Such abstract classes however cannot be instantiated. Eiffel uses the “deferred” keyword to indicate that a method is abstract. These languages can mark the whole class or just some methods. Even if just one method is abstract, the class cannot be instantiated. Java makes this
explicit by forcing the whole class to be marked abstract when a routine is marked. C++ uses a slightly
cumbersome notation involving the “virtual” keyword for achieving the same. When the “virtual” keyword
as applied to a parent class, it implies virtual inheritance and is unrelated to this. C++ overloads the
meaning of “virtual” in this respect. An example of an abstract method in C++ is:

\[ \text{virtual int func(int x) = 0} \]

The “= 0” notation indicates that the method has no implementation. There is no mechanism for marking
a class as abstract in C++. Moreover, there is no notion of abstract classes or methods in Smalltalk.

5.1.7 Undefining Features

There are some situations when a class inherits some undesirable features are from its superclass. For
example, if a “Stack” is implemented by inheriting from a “Queue”, methods like insertLast(), delete-
Last() are undesirable. One option to handle these unwanted methods is to redefine them to return an
error, but they still are part of the class specification and may cause confusion. Ability to undefine these
unwanted methods is very useful in this situation. Eiffel provides such an ability. It should be noted that
when such renaming is done, the subclass is not a ‘subtype’ of the parent class.

5.1.8 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].

5.2 Multiple Inheritance

Multiple inheritance allows a class to inherit from multiple parent classes. There are many situations
where multiple inheritance is required to clarify a design. A person that is both a doctor and an author,
for example, can be properly modeled by designing a class that inherits from both the “doctor” and the
“author” classes. This model more closely parallels inheritance as observed in the biological beings.
However, 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 routinely 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 consi-
derably. In addition to this philosophical objection, there are other interesting reasons why multiple
inheritance is a problem. These problems occur due to the fact that multiple inheritance defines an inher-
itage graph that is structured as a directed acyclic graph (DAG) and not a tree. In a tree there is single
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.

Order of Search: When a feature is referred to by a derived class, the implementation needs to traverse
the inheritance graph to find the parent node which defines that particular feature. Many of the
parents may provide implementations for the feature and the specific method selected depends on the order of traversal. The implementation may have to chose from multiple definitions, and the choice could very well affect the correctness of the program. This aspect impacts the portability of the program, since programs that utilizes multiple inheritance may behave differently on different language implementations.

**Name Collision:** When multiple parents (or ancestors) of a derived class define a method with the same name, it is ambiguous which particular version should be inherited by the derived class. If the derived class does refer to the common method name, it is not clear which one should be invoked. This ambiguity can occur only in a multiple inheritance scenario where the inheritance tree is a DAG and not a tree.

**Repeated Inheritance** occurs if the multiple parent classes of a derived class share same ancestor class. This shared ancestor appears in the multiple paths from the derived class. A subclass may inherit from the same class multiple times, once for each path. This creates consistency problems as any method may update state in only one of these many copies and other methods may work on different copy and compromise the object consistency. Even if a single copy of the ancestor data is maintained somehow, different paths may export methods that update the data multiple times. The order in which parents are listed may become important depending on the order of search done.

Various mechanisms have been used for implementing multiple inheritance in the modern languages.

### 5.2.1 Tree-Based Inheritance

In this method, the DAG structure of an inheritance graph is transformed into a tree by duplicating nodes in common paths. If a feature can be inherited in multiple paths, there are multiple copies of
that feature in the tree. This method deliberately introduces the name clashes and will work only if the implementation supports feature renaming. Also since the nodes are duplicated, this method will not work in cases where only one copy of the node must be used. (A singleton class for example). Moreover, tree-based multiple inheritance cannot effectively solve the duplicated parent node problem.

5.2.2 Graph-Based Inheritance

In graph-based inheritance, the inheritance structure of a class is represented as a graph. The usual problems of name clashes and repeated inheritance are present in this solution and require that languages use different techniques to solve those issues. One method to solve name clashes is to simply make them illegal. This is the approach used in Smalltalk where only the actual names are different (i.e. not inherited from the same node). Another method is to apply renaming of subclass methods to uniquely distinguish the features in different paths. C++ and Eiffel both support renaming of methods. Repeated inheritance problem is solved by use of virtual inheritance in C++. When a class is defined to be virtually inherited, C++ ensures that there is single copy of the classes in that path. This solves the repeated inheritance problem, but there is still danger that multiple updates are done on the state of the common node.

Another problem with this implementation is that it exposes the inheritance structure. Assume that two classes B and C inherit an operator f() from class A. Now let D inherit from both B and C. Since both B and C do not redefine f(), it is not an error to inherit f() from both parents and D will retain the single common definition. If at a later point, class B is redesigned to not inherit from A but still have same behavior, suddenly we have a name clash. This happens because it is no longer possible for the system to verify that semantics of B.f() and C.f() are the same.

5.2.3 Linearized Inheritance

In this approach, the graph structure is flattened into a linear order using some traversal mechanisms. This process involves first putting the superclasses into a single linear order (using a depth first search for example) and then removing the duplicate nodes either from the front or the back, both methods leading to different semantics. The implementation may in addition try to honor the order defined by programmer. This method avoids the name collisions and repeated inheritance issues but introduces some new issues. When the linearization is done, there are some situations when an unrelated class is inserted between two related classes. The consequence is that the programmer may never assume knowledge of a superclass. The communication with a real parent becomes hard to establish and the programmer may have to settle with an “effective parent”. The ordering cannot take into the relative optimality of methods and may result in a suboptimal implementation. Python and Ruby both use the linearisation method [14, 30].

5.2.4 Various Implementations

Of the languages that we considered, multiple inheritance is implemented in languages like C++, Eiffel, and Python. Smalltalk, Java, C# and Ruby do not implement multiple inheritance, opting for multiple interfaces instead. Eiffel’s implementation of multiple inheritance differs from others in several respects. Eiffel tries to provide programmer with option of using either linearized or tree inheritance. A mechanism for renaming features is supported and is used for defining what is known as the “Repeated
Inheritance Rule”. This rule allows a single copy of a feature or multiple copies depending on situation. It provides a “select” construct that can be used to remove ambiguity in inheritance declarations. When subclass encounters common names from its multiple parents, it has option of renaming them to remove ambiguity or to use select operation to select one of them.

5.2.5 Alternatives to Multiple Inheritance

Due to drawbacks in multiple inheritance implementations various alternatives to this otherwise useful ability have come into being such as interfaces (Java), mixins (Ruby), and delegation.

**Interfaces** Languages like Java, C# have completely abandoned multiple inheritance and have adopted multiple interfaces in its place. A interface can contain only the specification without any implementation. There is no implementation and hence several conflicting situations common to multiple inheritance are avoided.

**Mixins** Mixin-based inheritance [3] was developed from a programming technique used in CLOS programming language. A Mixin is an abstract subclass that can be used to redefine behavior of parent classes. Since it is an abstract class, there can be no instantiated object for a mixin. The mixin class plays a role only when it appears in the hierarchical chain of a class. This can happen only when linearized inheritance is used as mixins are not typically subclassed. Linearized inheritance was said to have the drawback that it can introduce unrelated classes between a class and its parent. Mixin programming takes advantage of this feature to provide the behavior specialization. A mixin class is introduced into linear chain of a subclass and its methods are invoked when a inherited methods are invoked in the subclass. The mixin class then performs its own specialization and then invokes classes higher in the chain. Mixins can be used as a sole mechanism for performing multiple inheritance and can be used to introduce multiple inheritance safely.

**Delegation in prototype based languages** Class based languages are only one way of implementing object oriented concepts. Prototype based languages [23] use a different way of reuse. Prototype based languages create objects without resorting to defining classes. Prototype based languages are based on defining prototypes and manipulating them. When a new object is needed an existing object is cloned and then modified. There is no notion of type but rather similarity of objects. Delegation is used for implementing behavior reuse in these systems. When a new object is created, the new object may define some new behavior and retain some old behavior (i.e. retain old methods). There is no need to copy the implementation of these methods into the new object. Rather the delegation is used so that when unmodified method is invoked, the job is delegated to the parent object. Delegation is more powerful concept than inheritance in that delegation is dynamic in nature while inheritance is based on static definition of a class. The specific parent to delegated can be changed at any time giving rise to different object behavior at different times. This is referred to as computed delegation [9].

6 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].
Cardelli and Wegner (1985) [8] distinguish between four main kinds of polymorphism (Overloading, Coercion, Inclusion Polymorphism, and Parametric Polymorphism) grouped into two categories Ad-hoc Polymorphism and Universal Polymorphism. 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 (i.e. for all admissible types).

There are two major kinds of ad-hoc polymorphism: 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.

The two major categories of universal polymorphism are parametric polymorphism and inclusion polymorphism. Parametric polymorphism, aka generic programming, uses type parameters to determine the type of the argument for each application of the function or the class. In other words, it allows abstracting over types. Inclusion polymorphism models inheritance in object-oriented languages. It can be viewed as inclusion of classes, where an object can belong to many different types that need not to be disjoint. In other words, inclusion polymorphism is the ability of different classes to respond to the same message and each implement the method appropriately. For object-oriented programmers polymorphism almost always means inclusion polymorphism, whereas for functional programmers, it means parametric polymorphism [18].

In the following sections, we would give a short overview of overloading, coercion and parametric polymorphism, and we would go into more detail for inclusion polymorphism since it models inheritance in object-oriented languages. We would also talk about how the four different types of polymorphisms are implemented in the different languages that we reviewed: C++, Java, C#, Ruby, Python, Smalltalk, and Eiffel.

### 6.1 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. For example in C#, the addition operator can be overloaded to allow addition on a user-defined type 'Matrix' using the operator keyword:

```csharp
public static Matrix operator+ (Matrix M1, Matrix M2)
.
.
.
double[,] a1 = {{1,2,3},{4,5,6},{7,8,9}};
double[,] a2 = {{9,8,7},{6,5,4},{3,2,1}};
Matrix m1 = new Matrix (a1);
Matrix m2 = new Matrix (a2);
Matrix m3 = m1 + m2;
```

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 existing operators on objects can be modified. C# and C++ provide the operator keyword for this purpose [6, 13]. Operator overloading has often been criticized because it allows programmers to give operators completely different semantics depending on the types of their operands [29]. Changing the semantics can even happen by accident, as a common pitfall in C++ operator overloading is confusion between the arithmetic operators (‘+’, ‘*’) and assignment operators (‘+=’, ‘*=’). An inexperienced programmer could overload the addition (‘+’) operator by giving it the semantics of the assignment by addition (‘+=’) operator. This would result in simple expressions like ‘a + b’ unexpectedly modifying ‘a’ [9].

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 [30], 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 [32]. Moreover, an unique feature of Eiffel is that users can defined arbitrary operators, rather than being limited to redefining a set of predefined operators [35].

**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. The designers of Eiffel have felt that the general case of method overloading is full of potential ambiguities, type-checking failures and complications, and hence it is not worthwhile implementing it [20]. Moreover, in method overloading either the number of arguments or the type of arguments can be different. Since Ruby and Python are dynamically-typed languages (i.e. the variables do not have predefined types), for these languages method overloading is only relevant when the number of the arguments differs. Ruby and Python provides limited method overloading behavior by allowing default arguments values in the methods [21].

In summary, overloading is just a convenient syntactic abbreviation for using the same name for different semantic objects. The compiler resolves this ambiguity statically (i.e. at compile time). Some people consider overloading not to be true polymorphism, because instead of a value having many types, we allow a symbol to have many types, but the values denoted by the symbol have distinct and possibly incompatible types [15].

6.2 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 may reduce program size and improve program readability, but 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.

Coercion is implemented differently in the languages we studied. C++, for example, not only allows implicit conversions between many of its built-in types (e.g. from integer to double), but it also allows programmers a similar privilege for their own types, through converting constructors and user defined conversion operators [17]. Java, on the other hand, only widening conversions between native types are implicit; other conversions require explicit cast syntax [32]. Since coercion polymorphism is not the primary focus of this paper, we will not dedicate more of this paper on it.

6.3 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. For example, in the following 'Stack' class declaration:

```cpp
template <class T> Stack(T *a)
```

allows for stack objects to be instantiated for different types of parameters (i.e. a single class implementation works for infinite number of types). A generic class is one which can work for arguments of many
different types, generally doing the same kind of work independently of the type of the argument. In other words, parametric polymorphism guarantees uniform behavior on a range of types by abstracting over these types.

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 run-time, and thus, dynamically typed languages 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 as 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].

Generic programming is achieved in C++ with the use of templates [13]. C++ templates are implemented using a macro engine that expands templates to generate specialized code. When the parameter is instantiated, the compiler checks and generates the resulting code. It does not allow any qualifications on the types used as parameters in the templates. As a consequence compilers cannot check parametric code for correctness until the template is instantiated. Type checking is thus deferred to the moment of instantiation. Whenever, a template is instantiated, the actual definition of the template is specialized by replacing the template parameter with the actual instance. The specialized code is then checked by the compiler [28]. Moreover, the compiler constructs a special case class for each different use of type parameters. For example, from the above Stack template class, one can construct Stack<int> and Stack<double>. Two distinct classes are generated for the above cases: one with the type parameter replaced by int and one with it replaced by double. This approach duplicates the code of the Stack generic class and produces different specialized compiled forms.

Parametric polymorphism is implemented in Java by using generic classes which were added in Java 5.0. Generic programming in Java performs static type checking on the type parameters and then erases the type information of the type parameters. After the erasure, the specialized behavior is obtained by the subclassing polymorphism [28]. In Java generic class can be instantiated with reference types only. All type checking is done statically during compilation and ClassCastException will never be thrown. Type variables in Java parametric polymorphism cannot be instantiated with primitive types. This limitation is not serious in presence of autoboxing/unboxing mechanism. For comparison, templates in C++ are generally not type checked until they are instantiated. They are typically compiled into disjoint code for each instantiation rather than a single code, hence problems arise with code bloat, but type variables can be instantiated with primitive types [32].

The JVM uses the same instance of code for all uses of the type parameter. Specialized behavior is achieved through inheritance polymorphism of variables belonging to the parameter type. Java uses this approach by “erasing” type information and using the Object class instead of the specialized form. Instances of Object are cast back to the target class whenever necessary. This method has an overhead comparable to that of subclassing, and the code size is comparable to the non-generic version. For example, Stack<Integer> will be transformed to a Stack that contains Object type values, and the compiler will check if the type of the elements inserted into the collection are objects of type Integer
(or a subclass). This approach allows the same code to be used for all instances of Stack, therefore Stack<Double> is also just stack [28].

In C# there are two type of collections: generic and non-generic. Non-generic collections store all the elements as references with type object. This can lead to heterogeneous collections that are not type safe. The alternative, generic collections, introduced in version 2.0 of .NET, uses parametric polymorphism. They are type safe and exhibit better performance than non-generic collections. Like Java, C# is compiled to an intermediate language, which is executed by the runtime environment known as Common Language Runtime (CLR) which is similar to the JVM. C# generics are implemented at the CLR level. The advantage of the C# implementation of generics, compared to Java’s implementation, is that type information is retained at runtime, making optimization by a just-in-time compiler possible [28, 24]. For example, in C#, unlike Java, it is possible to create instances whose type is of the type parameter.

Some authors claim that parametric polymorphism is the purest form of polymorphism since the same class or function can be used uniformly in different type contexts without changes, coercions or any kind of run-time tests or special encodings of representation. However, it should be noted that this uniformity of behavior requires that all data be represented, or somehow dealt with, uniformly (e.g. by pointers as in C++).

### 6.4 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 polymorphism 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.

In subtyping, the type of an object is just the set of names and types of its methods. We say type 'Car' is a subtype of 'Vehicle', if a value of type 'Car' can be used in any context in which a value of type 'Vehicle' is expected. Subtyping is an example of true polymorphism, since objects of a subtype can be uniformly manipulated as if belonging to their supertypes [5]. In the implementation, the representations are chosen very carefully, so that no coercions are necessary when using an object of a subtype in place of an object of the supertype. In this sense the same object has many types. Similarly, operations are careful to interpret the representation uniformly so that they can work uniformly on elements of subtypes and supertypes [8].

In subclassing (inheritance), classes are used not only to create objects, but also to create richer sub-
In most class-based object-oriented languages, subclasses give rise to subtypes, again if 'Car' is a subclass of 'Vehicle', then an instance of class 'Car' may be used in any context where an instance of type 'Vehicle' is expected; thus we say 'Car' is a subtype of 'Vehicle'. A consequence of this is that any variable or parameter declared as having type 'Vehicle' might, at run time, hold a value of class 'Car'; in this situation many object-oriented programmers would say that 'Vehicle' is the variable's static type and 'Car' is its dynamic type [5]. Exceptions to this rule include private inheritance in C++ (which does not create subtypes), and certain operations on derived types in Eiffel, wherein features inherited from a base class can be removed or modified in a way which violates the subtyping rule [2].

Inclusion polymorphism is implemented through dynamic binding (aka late binding). In the context of object-oriented languages, dynamic binding refers to runtime binding of methods to messages. 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. The search for a method to bind to a message starts from the class to which the receiver currently (i.e. at runtime) is an instance of, and then proceeds up the class inheritance hierarchy [9]. In Smalltalk, for example, it may be necessary to search through the class hierarchy at run-time to find the method of an inherited operation. The cost of dynamic binding can be much lower, of course, with the result that there will be more work involved when modifying a method inherited by many subclasses. In C++, the designer of an object class may decide that dynamic binding is not permitted, and thus declare certain operations as virtual functions. Subclasses can specify implementations of virtual functions, and invocation of these functions on instances will be resolved at runtime depending on the class to which the instance belongs.

**Difference between static-binding and dynamic-binding**

Polymorphism may or may not impose a run-time overhead depending on whether dynamic binding is permitted by the programming language. If all objects are statically bound to variables, we can determine the methods to be executed at compile time. For example, instances of ad-hoc polymorphism and parametric polymorphism are normally detected by the compiler which then generates the appropriate code to respond to a given message. On the other hand, if variables can be dynamically bound to instances of different object classes, some form of run-time method lookup must be performed. For example, if we have a class named 'Car' which is a subclass of class 'Vehicle':

```java
Vehicle myVehicle = new Car();
myVehicle.stop();
```

and both classes define a method stop(), then depending on the binding (i.e. static or dynamic) a different version of the stop() method will be executed. In dynamic binding, the type of the object assigned to Vehicle determines the method (i.e. binds to the most specific version of the method). Hence, the Car version of the stop() method will be executed. Static binding, on the other hand, would bind to the base class version of the method (i.e. to the Vehicle implementation of stop()). In dynamic binding, the search for a method to bind to a message starts from the class to which the receiver currently (i.e. at runtime) is an instance of, and then proceeds up the class inheritance hierarchy. On the other hand, static binding initiates the search from the class to which the reference variable pointing to the receiver was declared to be an instance of. Hence, dynamic binding uses the most specific version of a method, whereas, static
binding uses the base class version of a method to bind to a given message [1].

Dynamic binding enables greater flexibility in code development and maintenance. New object types that can respond to existing messages can be incorporated in software systems without changing the base system. Only the client code that instantiates new objects must be modified to accommodate new types. Dynamic binding involves some overhead: space for extra information inside objects, run time method lookup must be performed, and runtime type checking is also necessary. This is why some languages (C++ and C#) make you specify which classes and methods need dynamic binding (i.e. static by default) [17, 6], and in other languages (such as Java, Eiffel, Smalltalk, Ruby and Python) all classes use dynamic binding (i.e. dynamic by default) [2, 32, 21]. In C++ and C#, a user can preface methods with virtual qualifier in the superclass to allow dynamic binding for that method. In C#, it is also required to use the override qualifier to specify the method which should override a virtual method. The search for the method to bind to the message starts in the class which defines the method prefaced with the virtual qualifier, and proceeds down the inheritance hierarchy from there to the class which the receiver object belongs to. In Java, a user can preface a method with final qualifier to prevent any subclass from overriding it. Similarly, in Eiffel a user can use the frozen qualifier to prevent the overriding of a method in subclasses.

Polymorphism is an important feature of object-oriented languages. It enhances software reusability by making it possible to implement generic software that will work not only for a range of existing objects, but also for objects to be added in the future. The languages that we considered vary in the way that they implement the different types of polymorphism (overloading, coercion, parametric and inclusion polymorphism). However, they all have dynamic binding which is what polymorphism means to object-oriented programmers. The ability to bind objects at run-time to the most specific implementation available.

7 Conclusion

Object-oriented programming languages have been used worldwide 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 safe guards 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
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.

References


