Object-oriented programming languages, like many programming languages, grew out of the never ending battle to conquer the high cost, poor performance, and difficult task of software development and maintenance. It has been estimated that 70 percent of the cost of software is spent on software maintenance. Where 49 percent is due to changes in user requirements, 17 percent is due to changes in data format, and 13 percent is spent on error correction. It is reasonable to infer from these statistics that it would be highly desirable to have a language that provides mechanisms that would allow for a software system to grow and evolve as the user requirements change, and to have the ability to withstand, with only minor need of local repair, changes to the data formats. It would also seem reasonable to expect from our language a rigorous, mathematically precise specification of the semantics of the language, as well as a mathematically clear and coherent relationship between related entities, to better enable one to determine the correctness of a software system without having to totally rely on the “art” of debugging.

Bertrand Meyer speaks of the five External Quality Factors of a software system, three of which are:

- Correctness
- Extendibility
- Reusability

Correctness deals with the issue of whether or not your software system implements its specification. Ideally this is determined by mathematically proving the implementation correct. This however is extremely difficult and depends on having both a formal specification and a rigorous description of the constructs in your language. It is thus extremely important that a precise mathematical description of the concept of class, the primary building block in an object-oriented language, be given.

Extendibility is characterized by a software system that adapts easily to small changes in the specification. We might want to place stronger conditions on the type of input we have to deal with, or perhaps we might want to introduce a new operation on a particular data type. How do our language constructs support such changes? One possible solution to this problem is through the technique of subclassing, where classes can inherit properties from their "parent" class and modifications can be made, and new data objects and operations can be added to the subclass. If we are interested in rigorously showing that a class meets its specification and if we are also interested in our subclasses preserving some of the
characteristics of the parent classes (so that it will not be necessary to completely redo our proofs of correctness) then it is important that this subclassing feature be rigorously defined in such a way that all the characteristics of the parent class are not lost. This would allow us to partly utilize any correctness proofs concerning the parent class to prove the correctness of the new subclass.

Reusability is concerned with reusing existing software for new developments. Given an existing class, or classes, are there program constructs that will allow for the reuse of the classes in developing other classes. One of the primary tools in object-oriented programming for dealing with reusability is inheritance. This includes inheritance of semantics of class methods as well as their signatures.

Another construct for dealing with reusability is the concept of client classes. This particular programming tool utilizes a class as an abstract data type to define a new class, not necessarily related to the class being used. Here we use features defined in the existing classes to define features in the new class. If a language is to have a chance of solving the “Software Crisis” then it must have constructs that will allow one to generate software systems that possess these external quality factors. Object-oriented languages are an attempt at a realization of a language possessing these qualities.

The concept of object-oriented programming fits naturally in the evolution of programming language paradigms, as a direct descendant of those languages that support the concept of “Abstract Data Types” -languages like Simula, Clu, Ada, and Modula-2. In these languages Abstract Data Types” are the primary building blocks of large software systems, whereas in pure object-oriented languages, ADT’s are in essence the only building blocks. It is thus the case that much of the work in this paper is influenced by the work on ADT’s by the ADJ group and Wagner. However, these languages, as well as the work of the ADJ group, do not include features related to the concept of inheritance. Wirth has proposed a concept of type extension that would introduce a form of single inheritance into languages like Modula-2 and Ada. However the form of inheritance introduced only deals with the extension of the data objects in an ADT and does not deal with data object respecifications or method extensions and redefinitions. Cardelli, and Mitchell, have described a semantics of multiple inheritance for record objects, and although their models are quite satisfying mathematically, they deal only with method “signatures” when defining their class hierarchies and do not take into consideration a class method’s “behavior” or “functionality” when determining subclasses. As such, these models are not complete enough to be used as a model of a language like Eiffel.

The informal model presented in my work is based upon the models given by Cardelli and Mitchell, in the sense that a class is in essence a record type with components of varying types, including functional types (Cardelli also deals with higher order functional types). However, my primary focus is on including method specifications within the model, including how class method functionality is affected by inheritance, including multiple inheritance. I also allow relative constraints to exist among attributes. This allows me to define classes in terms of existing classes by placing constraints on the class attributes. Thus we can define a new class in terms of an existing class in which a particular attribute possesses a particular value as discussed by Attardi.

The language Eiffel developed by Meyer, which had a large influence on my work, has the necessary features to accommodate the above mentioned software criteria. However, in Eiffel a class is defined by its implementation and as such it is very difficult to study the concepts of classes and inheritance from a formal, mathematically precise perspective. Eiffel is also not suitable as a general tool for the specification of software systems based on the paradigm of object-oriented design. Eiffel does have some features for specifying assertions
concerning the invariant of a class as well as the pre- and post-conditions of a class method, but these features are not powerful enough to allow for the complete specification of a class nor are they presented in a complete formal manner to allow for their rigorous mathematical study. Eiffel does impose restrictions on the behavior of a method redefined in a subclass of an existing class. These restrictions are given in terms of the pre- and post-condition specification of the method. The models in do not take into consideration these restrictions.

In my work I investigate the form of classes that appear in a variety of existing programming languages, such as Eiffel, C++, and Smalltalk and look at ways of constructing new classes using existing classes, including constructing new classes via inheritance, which includes the possibilities of “Renaming”, “Name Clashes” and “Redefinition”. Here we explore some of the various interpretations of inheritance that have manifested in a variety of implementations of object-oriented languages. For example, the issue of what should be inherited, specification or implementation, is presently being debated within the computer science community with no clear winner. Languages such as Smalltalk clearly belong within the paradigm of languages that support inheritance of implementation, whereas Eiffel falls within the paradigm of languages that support inheritance of specification. Even amongst these two camps there is still debate over just how much, and what type of, modification should be allowed. In the paper we have tried not to take sides and have described forms of classes and inheritance that describe object-oriented languages satisfying a variety of inheritance properties.

A feature which is present in most object-oriented programming languages is the ability to indiscriminately redefine class methods in a descendant of an existing class. This characteristic of an object-oriented language has many opponents. However, since this ability to totally redefine a method specification is present in most existing object-oriented languages, and also since it has been argued that just such an ability is useful in the software development process, we have allowed for the inclusion of this feature in my description.

References


