A Classification of Object-Oriented Development Methodologies

L. F. Capretz and P. A. Lee

Computing Laboratory
University of Newcastle upon Tyne
Newcastle upon Tyne - NE1 7RU - United Kingdom

Abstract

In the last few years, demand for object oriented software systems has increased dramatically, and it is widely accepted that present software engineering methodologies are unable to cope with the needs of that demand. The object oriented paradigm has promised to revolutionize software development; as a result, several methodologies have recently arisen to support software development based on an object oriented approach. This paper is concerned with object oriented methodologies for software systems and proposes a classification scheme for existing object oriented development methodologies. Additionally, the paper presents gaps which could be filled with the development of new object oriented methodologies.

1 INTRODUCTION

For many years, it has been recognised that the use of methodologies has an important role to play in order to accomplish a well-engineered software system. Methodologies provide a set of rules, principles, guidelines and notational conventions which helps software engineers to understand, organise and decompose software systems, and hence manage their complexity. Such methodologies, therefore, facilitate the development of complex and/or large software systems and give the software engineers the feeling that technology is an extension of their capabilities. In the past few years, many methodologies have been proposed to support the engineering of software systems. These methodologies have addressed different aspects of software development ranging from requirements through testing. Many of these methodologies have often appeared in response to new ideas about how to handle software complexity.

A recent idea which has been receiving a great deal of attention from software engineers is the **object-oriented** paradigm. Currently, this paradigm is thought to be an important aspect of software development, so much so that it is now a major research area which is expected to bring significant benefits in the design of software systems. The rapid development of this paradigm during the past ten years can be attributed to several important reasons, which include: better modelling of real-world applications; better structure for software systems based on abstract data type concepts; and the possibility of software reuse during the development of a software system.

Due to the rapid developments in the object-oriented arena, it has become very fashionable to describe many kinds of software system using object oriented terminology, and the term itself has become a buzzword. In the scope of this paper, an **object** embodies an abstraction characterised by an entity in the real-world. A **class** (or type) is a template description which specifies common properties and behaviour for a group of similar objects and an object is an instance of a class. The properties and behaviour of objects, and hence their commonalities, are described in terms of **attributes** and **operations**. An attribute is a named property of an object which holds a value and maintains an abstract state for that object. An operation identifies an action which may be applied to an object. Objects from each class are manipulated by invoking the operations upon the attributes of those objects. The classes themselves can be organised into class hierarchies. Such class hierarchies allow similar classes to be related together in such a way that commonalities of one class can be inherited (reused) rather than duplicated by classes lower in the hierarchy, thus simplifying the design and implementation of those lower level classes. **Inheritance** is a mechanism which permits classes to share attributes and operations based on relationships of specialisation and generalisation between them within a hierarchy of classes.

Because of the perceived importance of an object oriented approach, several methodologies have recently emerged to support object oriented design. Ideally, an object oriented design methodology should allow designers to produce software systems mainly in terms of classes, inheritance and objects. Nevertheless, this point of view is not emphasised by some current methodologies as will be seen later in this paper. Despite all of the progress so far in the object oriented paradigm, there is a gap concerning object oriented design. That is, despite the acknowledged importance of software design methodologies and the increasing popularity of the object-oriented paradigm, there is no generally accepted object oriented design methodology which essentially addresses object oriented design and considers reusability as part of the software development life cycle model. Moreover, reusability should be emphasised as part of the methodology within an alternative software life cycle model.

The remainder of this paper is organised as follows: the second section states the main requirements for an object oriented methodology. A general classification for software development methodologies is discussed in section three. Section four presents the current state of the art in object oriented methodologies as well as a classification in terms of the approaches they mixed with. Final remarks on the proposed classification is outlined in section five. The purpose of the paper is to identify the major similarities and differences between the methodologies, and hence to compare and classify them. To a large extent, the classification and comparison synthesise different directions of thoughts, for instance, the phase of the software life cycle for which a methodology is suitable; whether it is language-dependent; and whether it mixes with other approaches. From this outlook, the paper points out gaps which could be filled by new software development methodologies.

2 REQUIREMENTS FOR AN OBJECT-ORIENTED METHODOLOGY

An important idea brought forward by software engineering is the concept of **software life cycle models**. Such models have been proposed in order to systematise the several stages which a software system goes through (Royce, 1987; Boehm, 1988; Henderson-Sellers, 1990). The software life cycle can be divided into different phases, although in practice some phases may overlap each other. Software development involves a set of transformations starting from requirements and ending after implementation. Between these two points, a number of other abstract representations are described. The aim is to divide a complex software system into more manageable pieces, that is, each new abstract representation gives the designers more details about a software system than the previous one and allows them to make additional refinements in order to move towards the next abstract representation.

The most well-known software life cycle models do not take into account the issue of how to reuse existing software components when the design of a new software system is being undertaken. The main issue in software reuse is the creation of components which can be reused in software systems other than the one for which they were originally created. **Reusability** is seen as a suitable technique for improving software quality and reducing software development costs and time, and it has therefore emerged as an important issue in software engineering. In the past, reusability was primarily concerned with using subroutines from a library during the implementation phase of the software life cycle. Nowadays, however, a great deal of research has been carried out in order to accomplish reusability during the design and analysis phases as well (Capretz and Lee, 1992). The idea of reusability within an object-oriented approach is attractive because it is not just a matter of reusing only the code of a subroutine but also encompasses the reuse of any commonality expressed in class hierarchies. The inheritance mechanism encourages reusability within an object-oriented approach (rather than reinvention) by permitting a class to be used in a modified form by deriving a sub-class from it (Johnson, 1988; Micallef, 1988; Gossain, 1990).

Preferably, there should be specific methodologies suitable to object-oriented software development because there are specific object-oriented concepts involved. The unsuitability of most of the current methodologies for dealing with complexity inherent of software development suggests the use of different techniques followed by an informal change of approach, from a functional decomposition to an object-oriented, during software development. For instance, the designer starts analysis following a functional decomposition point of view and afterwards, during the implementation phase, changes to an object-oriented point of view. This change in approach leads the thought process to follow an object-oriented approach in the middle of software development instead of starting software development based on classes, objects and inheritance from the outset.

One great advantage of using the object-oriented paradigm is the conceptual continuity across all phases of the software development life cycle. The conceptual structure of a software system not only remains the same from system analysis down through implementation, but also remains the same during the refinement of a design. Therefore, when the object-oriented paradigm is used, the design phase is linked more closely to the system analysis and the implementation phases because designers have to deal with similar abstract concepts (such as classes and objects) throughout software development. However, the object-oriented paradigm still needs an organised and disciplined view of software development, and to be extended to cover more phases of the software life cycle model.

Additionally, software system designers hardly ever solve a new problem from scratch. Instead, they try to identify similarities between the new application and previous applications and their solutions. By making suitable transformations from previous experience, designers attempt to solve the new problem. Experts explicitly construct high level abstractions of a software system whereas novices think about low level entities and their behaviour within a software system. The experts tend to think in more abstract and high level terms following a top-down approach whereas the novices usually start thinking about low level abstractions of a software system and software development is thus predominantly bottom-up. Therefore, it can be assumed that the knowledge designers have about an application domain increases the chance of reusing solutions from previous experience (Capretz and Lee, 1992). However, many object-oriented methodologies do not take this human feature into account.

This paper is interested in identify gaps in order to find an approach which yields a single coherent object-oriented design methodology, rather than separate methods to solve specific parts of a design. Such a methodology must pay attention to object-oriented concepts already discussed, for instance, classes, objects and inheritance. The proper use of these concepts can lead to a truly general object-oriented design methodology as independent as possible of any programming language. Moreover, reusability should be emphasised as part of the methodology within an alternative software life cycle model.

3 CLASSIFICATION OF METHODOLOGIES

Many methodologies have been proposed over the last few years. Such methodologies provide some discipline in handling the problem of software complexity because they usually offer a set of rules and guidelines to help software engineers understand, organise, decompose and represent software systems. Such methodologies may be classified into three approaches. Firstly, some methodologies deal with functions; they emphasise refinement through functional decomposition. Typically, software development follows a top-down fashion by successively refining functions, for example, Structured Design (Yourdon, 1979), HIPO (Stay, 1976) and Stepwise Refinement (Wirth, 1971).

In a second approach, there are methodologies which recommend that software systems should be developed with emphasis on data rather than functions. That is, the system architecture is based on the structure of the data to be processed by a software system. The software system should be structured mainly through the identification of data components and their meaning. This sort of style can be noted in the early Jackson Structured Programming methodology (Jackson, 1975), SLAN-4 (Beichter, 1984) and the Entity-Relationship Model (Chen, 1976). The Entity-Relationship Model (ERM) is the most common approach to data modelling. ERM is a graphical technique which is easy to understand, yet powerful enough to model real-world applications, and entity-relationship diagrams are readily translated into a database implementation.

A third fashion consists of methodologies which aim to develop software systems from both functional and data points of view but separately. Examples of such methodologies are SADT (Ross, 1977), Structured Analysis and System Specification (DeMarco, 1979) and Structured System Analysis (Gane, 1979). SADT provides different kinds of diagrams to represent functions and data. As far as Structured System Analysis and Structured Analysis are concerned, designers can represent and refine functions through data flow diagrams and use a data dictionary to describe data. These methodologies organise a specification and design around hierarchies of functions. Structured analysis begins by identifying one or more high level functions which describe the overall purpose of a software system. Then, each high level function is decomposed into smaller, less complex functions.

There are also some software development environments which have automated some of those methodologies. The chief purposes of these environments are to increase productivity and enhance the quality of the developed software system. PSL/PSA (Teichroew, 1977) and EPOS (Lauber, 1982) are good examples of such environments.

A combination of approaches which follow a structured analysis, structured design and structured programming is collectively known as the **structured development** approach. Structured development iteratively divides complex functions into subfunctions. When the resulting subfunctions are simple enough, decomposition stops. This process of decomposition is known as the *functional decomposition* approach. Structured development also includes a variety of notations for representing software systems. During the specification and analysis phases, data flow diagrams, entity-relationship diagrams and a data dictionary are used to logically describe a software system. In the design phase, details are added to the specification model and the data flow diagrams are converted into structure chart diagrams ready to be implemented in a procedural language.

Structured analysis has been suggested as an attractive starting point for object-oriented design primarily because it is well-known, many designers are trained in its techniques, and many tools support its notations. However, structured analysis is not the ideal front-end to object-oriented design, mainly because it can perpetuate a functional decomposition view of the application. Applying a functional decomposition approach first and an object-oriented approach later on the same software system is likely to lead to trouble because functional decomposition can not be properly mapped into object-oriented decomposition. A better trend in analysis is to use an object-oriented analysis method in which there are attempts to identify and model the essential classes and objects of a software system.

The object oriented paradigm organises software systems into classes and establishes relationships between them. Objects model real-world entities and combine both attributes and operation. Each object is an instance of a class which is the building block of a system architecture. A positive benefit of following an object-oriented approach is traceability between software abstractions and reality because organising a software system around classes maps real-world entities into software components, particularly classes and objects. Thus, any methodology which deals with the object-oriented paradigm should have means of representing classes, objects and inheritance. Ideally, object-oriented design and implementation should be part of a software development process in which an object oriented philosophy was used throughout software development, as shown in Figure 1. In this figure, the dashed arrows represent an unnatural mapping between concepts of different approaches as opposed to bold arrows which indicate a smooth transition from one phase to another.

Similar issues are discussed by Loy (1990) who has shown that simply attempting to combine an object-oriented approach with a structured development approach gives rise to some problems. It jeopardises traceability from requirements to implementation because, in early phases, a software system is described in terms of functions and later on the description is changed in terms of object oriented concepts (see Figure 1). Furthermore, structured development methodologies do not localise information around objects but on the data flow between functions, and a software system is composed of data flow and functions. In contrast, the object-oriented paradigm organises a software system around classes and objects which exist in the designer's view of the real world application. A discussion of why new methodologies are still required is presented in the last section of this paper.

4 CLASSIFICATION OF EXISTING OBJECT-ORIENTED METHODOLOGIES

Recently, there has been a profusion of so callled object oriented methodologies for analysis and design influenced by a variety of different backgrounds (Capretz, 1991). Nevertheless, it can be noticed that there are two major directions concerning object oriented methodologies:

a) **adaptation**: this is concerned with the mixing of an object oriented approach to well-known structured development methodologies;

b) **assimilation**: this emphasises the use of an object oriented approach for developing software systems, but following the traditional waterfall software life cycle model.

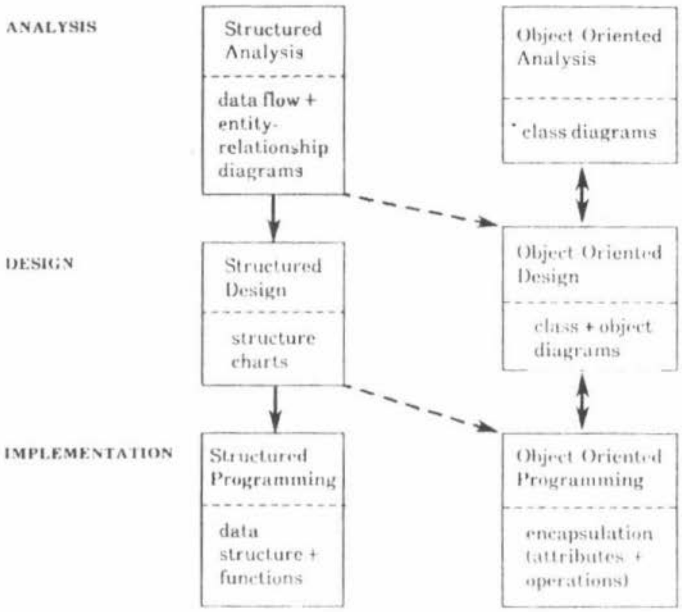

Figure 1: Some Combinations of Approaches

4.1 Adaptation

Adaptation proposes a framework for mixing an object oriented approach with existing methodologies. It has been suggested by Henderson Sellers and Constantine (1991) that a combination of structured development and an object oriented approach helps tackle the problem of software development. Designers use their experience and intuition to derive a specification from an informal description in order to get a high level abstraction for a software system, based on functional decomposition. The adaptation of structured development to an object oriented approach preserves the specification and analysis phases using data flow diagrams and proposes heuristics to convert these diagrams into an object model in such way that subsequent phases can then follow an object oriented approach. However, adaptation approaches are trying to evolve object oriented methodologies from existing ones and as a result bringing their limitations with them.

Nevertheless, some advantages of this adaptive approach are:

- a complementary (though unnatural) coupling between structured development and an object oriented approach;
- structured development methodologies are widely known and used, and support top down functional decomposition, the most common fashion for software development;
- a smoother migration from an old, practised and well known approach to a new one including classes, objects and inheritance;

- a gradual change of tools and environments to an object oriented approach.

Currently, the most widely used software engineering methodologies are those for structured development. Those methodologies are popular because they are applicable to many types of application domains. On account of this popularity, structured development has been mixed up with an object-oriented approach. Those designers who come from a traditional software engineering background, such as functional decomposition and data modelling techniques, will probably find the methodologies of Shlaer and Mellor (1988), and Coad and Yourdon (1990) familiar because these methodologies are clearly adaptations of traditional structured development methodologies and data modelling techniques.

These methodologies have oversimplified the object-oriented paradigm by misusing the concepts of classes and objects during the analysis phase as objects only. Basically, they concentrate on modelling real-world entities as objects, and it can be considered as an extension of the Entity-Relationship Model (Chen, 1976), suggesting that they are incremental improvements over existing approaches to data modelling. Moreover, they have not discussed the impact of their methodology on other phases of the software life cycle. These methodologies may be used during a period of transition from structured development to an object-oriented approach as a compromise. However, they cannot permit the full advantages of an object oriented approach to be gained.

Jackson (1983) has proposed a methodology called the Jackson System Development (JSD). JSD has some features which appear on the surface to be similar to object oriented design. The main task is to model the application and to identify *entities* (which could be viewed as objects), *actions* (i.e. operations) and their interactions. However, JSD is not fully suitable for object oriented design because there is very little to support the object-oriented paradigm. For instance, there is no consideration of inheritance and encapsulation.

Other less known proposals where object-oriented concepts are by products of structured development could also be considered. Some of these methods are merely extensions of structured development methodologies. Masiero and Germano (1988) and Hull *et al.* (1989) put together an object-oriented approach with Jackson (1983) methodology, and the product of a design is implemented in Ada. Bailin (1989) and Bulman (1989) mix up an object oriented approach with Structured System Analysis (Gane, 1979) and the Entity Relationship Model for an object-oriented requirements specification model (Chen, 1976). Lastly, Alabiso (1988) and Ward (1989) combine the object oriented style with Structured Analysis (DeMarco, 1979), Structure Design (Yourdon, 1979) and the Entity-Relationship Model.

The history of object oriented technology dates from the 1970s, but up to the mid-1980s, much of the work in the object oriented arena focused on object oriented programming. However, the application of an object-oriented approach to software design has occurred since that time, mainly for those familiar with Simula 67. From the beginning of the 1980s, some attempts to develop software systems using an object oriented approach have emerged. The first significant step towards an object oriented design methodology, started within the Ada community. Many ideas about object oriented design came out with the work of Abbott (1983) and Booch (1983a), (1983b). Booch rationalised Abbott's method, and referred to it as *Object-Oriented Design* (Booch, 1983b). Both Abbott and Booch have recommended that a design should start with an informal description of the real-world application and from this description designers could identify classes and objects from nouns, and operations from verbs. The work of Booch is significant because it was one of the earliest object-oriented design methodologies to be described in the literature. Booch is also one of the most influential advocates of object oriented design within the Ada community.

Realising the drawbacks of the technique based on identification of classes and objects from narrative descriptions, later Booch methodologies have not advocated the use of a narrative description. Instead, Booch (1986) has combined object oriented design with existing methodologies and called it *Object Oriented Development*. Booch suggested that existing methodologies such as SREM (Alford, 1977), Structured System Analysis (Gane, 1979) or Jackson

Structured Development (Jackson, 1983) could be used during the requirements and specification phases as a step before object-oriented design. Subsequently, Booch (1991) has proposed a new object-oriented development methodology which includes a variety of models that address functional and dynamic aspects of software systems.

As far as Booch's influences are concerned, they can be summarised a follows: what has come to be known as object-oriented design in the context of Ada was first proposed by Booch (1983b), later extended by Booch (1986) and after refined by Seidewitz (1989), Heitz (1989) and Jalote (1989). Berard (1986) and Sincovec and Wierner (1987) also present principles and methods biased by Booch (1983a) with implementation also totally driven towards Ada. These design methodologies concentrate on identifying objects and operations, and are object-oriented in the sense that they view a software system as a collection of objects. Most of these methodologies are based on an informal description or representation of the software requirements, from which object, attributes and operations can be identified. Moreover, all of these methodologies apply hierarchical decomposition, a trend to decompose a software system by breaking it into its components through a series of top-down refinements towards an implementation in Ada.

Wasserman *et al.* (1990) have proposed OOSD, a graphical representation for Object Oriented Structured Design. OOSD provides a standard design notation by supporting concepts of both structured and object-oriented design since the main ideas behind OOSD come from Structured Design (Yourdon, 1979) and Booch (1986) notation for Ada packages. Therefore, OOSD allows designers to gradually shift from structured development to an object oriented approach. However, OOSD focuses mainly on a graphical representation without addressing the method by which a design could be created and gives no explicit technique for diagramming software system decomposition, needed for large software systems. Furthermore, it does not set up guidelines to identify classes and operations, so it is supposed that designers should follow steps recommended by other methodologies. It is expected that designers will use their own methods together with OOSD, which simply provides a notation for object oriented design, not a step-by-step methodology.

Rumbaugh *et al.* (1991) have developed the Object Modelling Technique (OMT) which focuses on object modelling as a software development technique. Basically, OMT proposes three models for software development: the object model, the dynamic model and the functional model. In addition, OMT proposes three stages which should be carried out to produce a design: the analysis stage, the design stage and the object design stage. Some considerations on OMT can be made at this point. The first stage of OMT is equivalent to OORA (Coad 1990), in that it basically models the application in terms of classes and special relationships between them. However, it is confusing that the object model actually deals with classes. Later, during the system design and object design stages, OMT incorporates structured development based on a functional decomposition approach following the traditional waterfall software life cycle model.

4.2 Assimilation

Assimilation is a trend that puts the object oriented paradigm within the traditional waterfall software life cycle model. In recent years several object oriented methodologies have appeared but they cover only partially that software life cycle model. Several authors have tried to fit the object-oriented paradigm into this framework: Booch (1991), Wirfs-Brock *et al.* (1990), Pun and Winder (1989), and Lorensen (1986) can be considered as good examples. So far, these methodologies are not well-known and not generally accepted, but their main ideas encompass object-oriented concepts because they are based on at least classes, objects and inheritance. These methodologies still need to be used in practical contexts to develop large scale software systems in order to be evaluated and improved.

Wirfs-Brock *et al.* (1990) have focussed on the identification of *responsibilities* and *contracts* to build a *responsibility-driven* design. Responsibilities are a way to apportion work among a group of objects which comprise a real-world application. A contract is a set of related responsibilities defined by a class, and describes the ways in which a given client object can interact with a server object. This methodology emphasises the actions that must be accomplished and which objects will perform those actions. The responsibilities of an object are the services it provides for all objects which communicate with it. Both objects must fulfil a contract: the client object by making the requests that the contract specifies, and the server object by responding appropriately to these requests. Objects fulfil their responsibilities either by performing the necessary computation themselves or by collaborating with other objects.

Some observations should be made on responsibility driven design. The methodology is based on the identification of classes by looking at nouns in a natural language description of the system specification. This technique has the same problems as the first versions of Abbott and Booch methodologies. Additionally, the methodology bases inheritance only on responsibilities, and ignores inheritance of attributes. Moreover, the concept of operation is separated from the notion of class. Responsibility-driven design divides a large software system into subsystem only after identifying some classes and their responsibilities. However, the partitioning of a software system into subsystems should be considered at the beginning of software development as a technique to decompose large software systems.

Additionally, the methodology also suggests the use of class cards to describe classes and subsystems. The technique of recording design on cards was introduced by Beck and Cunningham (1989) who have proposed the Class, Responsibility, and Collaboration (CRC) cards. They have found that index cards are a simple technique for teaching object-oriented thinking to newcomers of the object oriented paradigm. Each card contains the name of a class, a description of the responsibilities associated with that class and its collaborators. This technique may work well with simple software systems but its use in the design of large and complex software systems is doubtful because as the number of classes and subsystems grows sharply, the number and arrangements of class cards may become cumbersome.

The state-of-the-art of object oriented methodologies is evolving rapidly. As object-oriented methodologies mature, they are likely to borrow ideas from one another. There are more restrictive and less known object oriented methodologies. Pun and Winder (1989) have proposed a methodology targeted at object oriented design and programming. Lorensen (1986) has described the rudiments of object oriented software development by explaining that it is fundamentally different from traditional structured development, such as those based on data flow diagrams and a functional decomposition approach. Lorensen has described an alternative methodology to object-oriented design which has evolved from software development using Smalltalk, which directly supports classes and inheritance.

Jacobson (1987) has claimed to have a full object oriented development methodology called the ObjectOry which combines a technique to develop large software systems, called the **block design** (Jacobson, 1986), with Conceptual Modelling (Borgida, 1985) and object-oriented concepts. Jacobson has stated that it is quite natural to unite these three approaches since they rely on similar ideas aiming at, among other things, the production of reusable software components. Conceptual modelling emphasises finding concepts relevant to model a real-world application, and it is appropriate for representing the entities of the application and the relationships between these entities. ObjectOry concerns the design of some large scale software systems which have been developed today using techniques such as Structured System Analysis (Gane, 1979).

Cunningham and Beck (1986) have proposed diagrams for representing Smalltalk programs. The diagrams emphasise the representation of the message passing which takes place between objects. The diagrams as proposed are only suitable to small programs because the power of representation is weak. Classes are listed with the most specialised class at the top leading some designers to complain that it is not intuitive to place sub-classes above super-classes. Another limitation concerns the restrictiveness of the graphical elements in the diagrams which are more suitable for representing Smalltalk programs and are more likely to be used during the implementation phase.

As far as C++ is concerned, Ackroyd and Daum (1991) have suggested a basic graphical notation for classes, objects, methods and inheritance. The graphical notation also adds numerous specialised representation for polymorphism, overloading, delegation, static variables and many other properties which can provide graphical representation for object oriented programs. Nevertheless, the proposed notation is tied closely to C++ features, particularly those for private, protected and public classes. Furthermore, it is not apparent how to use the notation for other programming languages nor how to use it as a language-independent graphical notation for object-oriented design. Additionally, the diagrams could depict an extensive representation of implementation details which makes them hard to use and comprehend for large programs.

5 CONCLUSIONS

This paper faces the object-oriented paradigm from a methodology standpoint, rather than from an implementation standpoint. The survey prompted by the perceived inadequacy of existing object-oriented methodologies for designing object-oriented software systems, and has sought to establish a viable and comprehensive object-oriented design methodology which obtains the benefits of the object-oriented paradigm, such as encapsulation and inheritance. From the point of view of programming languages, the road to an object-oriented approach is an evolutionary step, whereas from the point of view of software development methodologies, the differences which exist between traditional structured development methodologies (based on functional decomposition) and those based on an object-oriented approach suggest that a revolution is taking place.

The paper has briefly reviewed many object-oriented methodologies and outlined their oversights and weaknesses. There is some dissatisfaction with current methodologies which seem to place too much emphasis on designing for the task in hand and not enough on designing reusable components nor designing with reusable components. Based on this survey, it is evident that further research on object-oriented methodologies is required in order to overcome the deficiencies and limitation of existing software development methodologies.

Most of the early object-oriented methodologies which have appeared either focus on an implementation in Ada (which does not provide inheritance) or disregard abstraction in terms of classes, and instead focus on object instantiation. Other methodologies with the intention of combining existing structured methodologies together with object oriented concepts have led to the misuse of objects only as data without regarding the operations on that data; the operations are treated separately as functions. Another problem with such combinations is the mapping of concepts from one approach to another. The adoption of new concepts, the change of vocabulary can confuse designers about which one should be used in which phase of the software life cycle.

Although a number of object-oriented design methodologies are becoming available and gaining increasing use in order to answer a broad range of software engineering questions, such methodologies are still at a relatively early stage of growth. It is clear that even more experimentation is required (particularly in developing very large software systems using an object-oriented approach) before this paradigm can claim to be a mature subject. Additionally, new experiments will provide several case studies to evaluate the various claims made about the object-oriented paradigm, which can only be fully tested when applied to developing substantial systems.

Therefore, it can be concluded that so far there has been no generally accepted object-oriented design methodology. Only limited object-oriented methodologies have been found. As a result it should be the intention of further research to create an object-oriented design methodology which allows designers to apply powerful object-oriented principles to the design of a wide range of applications from the beginning of software development. To put it in other words, **revolution** is what is needed to tackle the problems of software development. New methodologies which exploit the benefits of the object-oriented paradigm within a substitute software life cycle model have to be pursued. Capretz (1991) describes a Methodology for Object Oriented Design (**MOOD**) which takes domain analysis and reusability into account as an important aspect of a new software development life cycle model, however, this effort is certainly not enough.

The object-oriented paradigm is such a powerful set of concepts that eventually it will get completely absorbed into the software development culture, in the same way that structured development methodologies and to some extent abstract data types concepts have been. This is evident in the abundance of research looking at various aspects of the paradigm. Consequently, the 1990s are likely to be a period of gradual acceptance of the object-oriented paradigm which will become the main approach to developing software systems in this decade. The future of object-oriented software engineering might well be to accept a hybrid trend with other approaches and mapping of concepts between such approaches. However, the authors believe that the object-oriented paradigm should (and hopefully will!) pervade the entire software life cycle.

REFERENCES

Abbott R. J. (1983) "Programming Design by Informal English Description", *Communications of the ACM*, **26**(11), November 1983, pp. 882-894.

Ackroyd M. and Daum D. (1991) "Graphical Notation for Object-Oriented Design and Programming", *Journal of Object-Oriented Programming*, **3**(5), January 1991, pp. 18-28.

Alabiso B. (1988) "Transformation of Data Flow Analysis Model to Object-Oriented Design", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA'88*, San Diego, California, September 1988, *ACM SIGPLAN Notices*, **23**(11), November 1988, pp. 335-353.

Alford M. W. (1977) "A Requirements Engineering Methodology for Real-Time Processing Requirements", *IEEE Transactions on Software Engineering*, **SE-3**(1), January 1977, pp. 60-69.

Bailin S. C. (1989) "An Object-Oriented Requirements Specification Method", *Communications of the ACM*, **32**(5), May 1989, pp. 608-623.

Beck K. and Cunningham W. (1989) "A Laboratory for Teaching Object Oriented Thinking", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA'89*, New Orleans, Lousiana, *ACM SIGPLAN Notices*, **24**(10), October 1989, pp. 1-6.

Beichter F. W., Herzog O. and Petzsh H. (1984) "SLAN-4 - A Software Specification and Design Language", *IEEE Transactions on Software Engineering*, **SE-10**(3), March 1984, pp. 155-162.

Berard E. (1986) *An Object-Oriented Design Handbook*, EVB Software Engineering Inc., Rockville, Maryland.

Boehm B. W. (1988) "A Spiral Model of Software Development and Enhancement", *Computer*, **21**(5), May 1988, pp. 61-72.

Booch G. (1983a) *Software Engineering with Ada*, Benjamin/Cummings, Menlo Park, California.

Booch G. (1983b) "Object Oriented Design", Freeman P. and Wasserman A. I. (eds.) *Tutorial on Software Design Techniques*, Fourth Edition, IEEE, Silver Spring, Maryland, pp. 420-436.

Booch G. (1986) "Object Oriented Development", *IEEE Transactions on Software Engineering*, **SE-12**(2), February 1986, pp. 211-221.

Booch G. (1991) *Object-Oriented Design with Applications*, Benjamin/Cummings, Redwood City, California.

Borgida A. (1985) "Features of Languages for the Development of Information System at the Conceptual Level", *IEEE Software*, **2**(1), January 1985, pp. 63-72.

Bulman D. M. (1989) "An Object Based Development Model", *Computer Language*, **6**(8), August 1989, pp. 49-59.

Capretz L. F. (1991) *Object-Oriented Design Methodologies for Software Systems*, PhD Thesis, Computing Laboratory, University of Newcastle upon Tyne, U.K. (Technical Report No. 370).

Capretz L.F. and Lee P.A. *Reusability and Life Cycle Issues Within an Object-Oriented Design Methodology*, Ege R., Singh M. and Meyer B. (eds.) *Proceedings of TOOLS USA'92 (Technology of Object-Oriented Languages and Systems)*, University of California, Santa Barbara (CA), USA, Prentice Hall, Englewood Cliffs (NJ), USA, pp. 139-150, August 1992.

Chen P. P. (1976) "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Transactions on Database Systems*, 1(1), March 1976, pp. 9-36.

Coad P. and Yourdon E. (1990) *Object-Oriented-Analysis*, Prentice Hall, Englewood Cliffs, New Jersey.

Cunningham W. and Beck K. (1986) "A Diagram for Object-Oriented Programs", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA'86*, Portland, Oregon, September 1986, *ACM SIGPLAN Notices*, 21(11), November 1986, pp. 361-367.

DeMarco T. (1979) *Structured Analysis and System Specification*, Prentice-Hall, Englewood Cliffs, New Jersey.

Gane C. and Sarson T. (1979) *Structured System Analysis: Tools and Techniques*, Prentice-Hall, Englewood Cliffs, New Jersey.

Gossain S. and Anderson B. (1990) "An Iterative-Design Model for Reusable Object-Oriented Software", Meyrowitz N. (ed.) *Proceeding of the Conference on Object-Oriented Programming: Systems, Languages and Applications and the European Conference on Object-Oriented Programming - OOPSLA'90*, Ottawa, *ACM SIGPLAN Notices*, 25(10), October 1990, pp. 12-27.

Heitz M. (1989) *HOOD Reference Manual*, Issue 3.0, European Space Agency, Noordwijk, The Netherlands, September 1989.

Henderson-Sellers B. and Edwards J. M. (1990) "The Object-Oriented Systems Life Cycle", *Communications of the ACM*, 33(9), September 1990, pp. 142-159.

Henderson-Sellers B. and Constantine L. L. (1991) "Object-Oriented Development and Functional Decomposition", *Journal of Object-Oriented Programming*, 3(5), January 1991, pp. 11-17.

Hull M. E. C., Zarea-Aliabadi A. and Guthrie D. A. (1989) "Object Oriented Design, Jackson System Development (JSD) Specification and Concurrency", *Software Engineering Journal*, 4(2), March 1989, pp. 79-86.

Jackson M. A. (1975) *Principles of Program Design*, Academic Press, New York, New York.

Jackson M. A. (1983) *System Development*, Prentice Hall, London.

Jacobson I. (1986) "Language Support for Changeable Large Real Time System", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA'86*, Portland, Oregon, September 1986, *ACM SIGPLAN Notices*, 21(11), November 1986, pp. 377-384.

Jacobson I. (1987) "Object Oriented Development in an Industrial Environment", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA'87*, Orlando, Florida, October 1987, *ACM SIGPLAN Notices*, 22(12), December 1987, pp. 183-191.

Jalote P. (1989) "Functional Refinement and Nested Objects for Object Oriented Design", *IEEE Transactions on Software Engineering*, SE-15(3), March 1989, pp. 264-270.

Johnson R. E. and Foote B. (1988) "Designing Reusable Classes", *Journal of Object-Oriented Programming*, **1**(2), June/July 1988, pp. 22-35.

Lauber R. J. (1982) "Development Support Systems", *Computer*, **15**(5), May 1982, pp. 36-46.

Lorensen W. (1986) *Object Oriented Design, CRD Software Engineering Guidelines*, General Electric Co., Corporate Research and Development Center.

Loy P. H. (1990) "A Comparison of Object Oriented and Structured Development Methods", *Software Engineering Notes*, **15**(1), January 1990, pp. 44-48.

Masiero P. and Germano F. S. R. (1988) "JSD as an Object Oriented Design Method", *Software Engineering Notes*, **13**(3), July 1988, pp. 22-23.

Micallef J. (1988) "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", *Journal of Object-Oriented Programming*, **1**(1), April 1988, pp. 12-36.

Pun W. W. Y and Winder R. L. (1989) "A Design Method for Object-Oriented Programming", Cook, S. (ed.) *Proceedings of the European Conference on Object Oriented Programming - ECOOP'89*, Nottingham, United Kingdom, July 1989, Cambridge University Press, Cambridge, pp. 225-240.

Ross T. R. and Schoman K. E. (1977) "Structured Analysis for Requirements Definitions", *IEEE Transactions on Software Engineering*, **SE-3**(1), January 1977, pp. 6-15.

Royce W. W. (1987) "Managing the Development of Large Software Systems", *Proceedings of the Ninth International Conference on Software Enginnering*, Monterey, California, March 1987, IEEE Computer Society Press, Washington, D.C., pp. 328-338.

Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W. (1991) *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey.

Seidewitz E. (1989) "General Object Oriented Software Development: Background and Experience", *The Journal of Systems and Software*, **9**(2), February 1989, pp. 95-108.

Shlaer S. and Mellor S. J. (1988) *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice-Hall, Englewood Cliffs, New Jersey.

Sincovec R. F. and Wierner R. S. (1987) "Modular Software Construction and Object Oriented Design Using Ada", Peterson G. E. (ed.) *Tutorial: Object-Oriented Computing*, IEEE Computer Society Press, Washington, D.C., pp. 30-36.

Stay J. F. (1976) "HIPO and Integrated Program Design", *IBM System Journal*, **15**(2), April 1976, pp. 143-154.

Teichroew D. and Hersey E. A. (1977) "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems", *IEEE Transactions on Software Engineering*, **SE-3**(1), January 1977, pp. 41-48.

Ward P. (1989) "How to Integrate Object Orientation with Structured Analysis and Design", *IEEE Software*, **6**(2), March 1989, pp. 74-82.

Wasserman A. I., Pircher P. A. and Muller R. J. (1990) "The Object Oriented Structured Design Notation for Software Design Representation", *Computer*, **23**(3), March 1990, pp. 50-63.

Wirfs-Brock R., Wilkerson B. and Wiener L. (1990) *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, New Jersey.

Wirth N. (1971) "Program Development by Stepwise Refinement", *Communications of the ACM*, **14**(4), April 1971, pp. 221-227.

Yourdon E. and Constantine L. L. (1979) *Structured Design*, Prentice Hall, Englewood Cliffs, NJ.