3-2003

A Brief History of the Object-Oriented Approach

Luiz Fernando Capretz
University of Western Ontario, lcapretz@uwo.ca

A Brief History of the Object-Oriented Approach

Luiz Fernando Capretz
University of Western Ontario
Department of Electrical & Computer Engineering
London, ON, CANADA, N6G 1H1
lcapretz@acm.org

ABSTRACT:

Unlike other fads, the object-oriented paradigm is here to stay. The road towards an object-oriented approach is described and several object-oriented programming languages are reviewed. Since the object-oriented paradigm promised to revolutionize software development, in the 1990s, demand for object-oriented software systems increased dramatically; consequently, several methodologies have been proposed to support software development based on that paradigm. Also presented are a survey and a classification scheme for object-oriented methodologies.

1. INTRODUCTION

Over the past three decades, several software development methodologies have appeared. Such methodologies address some or all phases of the software life cycle ranging from requirements to maintenance. These methodologies have often been developed in response to new ideas about how to cope with the inherent complexity of software systems. Due to the increasing popularity of object-oriented programming, in the last twenty years, research on object-oriented methodologies has become a growing field of interest.

There has also been an explosive growth in the number of software systems described as object-oriented. Object-orientation has already been applied to various areas such as programming languages, office information systems, system simulation and artificial intelligence. Some important features of present software systems include:

- Complexity: the internal architecture of current software systems is complex, often including concurrency and parallelism. Abstraction in terms of object-oriented concepts is a technique that helps to deal with complexity. Abstraction involves a selective examination of certain aspects of an application. It has the goal of isolating those aspects that are important for an understanding of the application, and also suppressing those aspects that are irrelevant. Forming abstractions of an application in terms of classes and objects is one of the fundamental tenets of the object-oriented paradigm.
- Friendliness: this is a paramount requirement for software systems in general. Iconic interfaces provide a user-friendly interaction between users and software systems. An icon is a graphical representation of an object on the screen, with a certain meaning to its observer, and is usually manipulated with the use of a mouse, a process that has come to be known as *WYSIWYG* (What You See Is What You Get) interaction. In such interfaces, windows, menus and icons are all viewed as objects. The trend to object-oriented graphical interfaces is evident in many areas of software development; experience suggests that user interfaces are significantly easier to develop when they are written in an object-oriented fashion. Thus the object-oriented nature of the WYSIWYG interfaces maps quite naturally into the concepts of the object-oriented paradigm.
- Reusability: reusing software components already available facilitates rapid software development and promotes the production of additional components that may themselves be reused in future software developments. Taking components created by others is better than creating new ones. If a good library of reusable components exists, browsing components to identify opportunities for reuse should take precedence over writing new ones from scratch. Inheritance is an object-oriented mechanism that boosts software reusability.

The rapid development of this paradigm during the past ten years has important reasons, among which are: better modeling of real-world applications as well as the possibility of software reuse during the development of a software system. The idea of reusability within an object-oriented approach is attractive because it is not just a matter of reusing the code of a subroutine, but it 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 when a sub-class is derived from it [1, 2, 3, 4].

2. THE BACKGROUND OF THE OBJECT-ORIENTED APPROACH

The notion of "object" naturally plays a central role in object-oriented software systems, but this concept has not appeared in the object-oriented paradigm. In fact, it could be said that the object-oriented paradigm was not invented but actually evolved by improving already existing practices. The term "object" emerged almost independently in various branches of computer science. Some areas that influenced the object-oriented paradigm include: system simulation, operating systems, data abstraction and artificial intelligence. Appearing almost simultaneously in the early 1970s, these computer science branches cope with the complexity of software in such a way that objects represent abstract components of a software system. For instance, some notions of "object" that emerged from these research fields are:

- Classes of objects used to simulate real-world applications, in Simula [5]. In this language the execution of a computer program is organized as a combined execution of a collection of objects, and objects sharing common behavior are said to constitute a class.
- Protected resources in operating systems. Hoare [6] proposed the idea of using an enclosed area as a software unit and introduced the concept of monitor, which is concerned with process synchronization and contention for resources among processes.
- Data abstraction in programming languages such as CLU [7]. It refers to a programming style in which instances of abstract

data types (i.e. objects) are manipulated by operations that are exclusively encapsulated within a protected region.

- Units of knowledge called frames, used for knowledge representation. Minsky [8] proposed the notion of frames to capture the idea that behavior goes with the entity whose behavior is being described. Thus a frame can also be represented as an object.

All these influences have been gathered together and the object-oriented paradigm has been seen as a way to converge them, as shown in Figure 1. The common characteristic of these ideas is that an object is a logical or a physical entity that is self-contained. Clearly, other belated items could be added to that list, such as innovations in programming languages, as demonstrated in Ada; and advances in programming methods, including the notion of modularization and encapsulation. Nevertheless, Simula was the first programming language that had objects and classes as central concepts. Simula was initially developed as a language for programming discrete-event simulations, and objects in the language were used to model entities in the real-world application that was being simulated.

Despite the early innovation of Simula, the term "object-oriented" became prominent from Smalltalk [9]. The Smalltalk language, first developed in 1972 in the Learning Research Group at Xerox Palo Alto Research Center, was greatly influenced by Simula as well as by Lisp. Smalltalk was the software half of an ambitious project known as the Dynabook, which was intended to be a powerful personal computer. Research on Smalltalk has continued and the Smalltalk language and the environment were by-products of that project. From Smalltalk, some common concepts and ideas were identified and they gave support, at least informally, to the object-oriented paradigm. Because of the evolution and dissemination of programming languages like Smalltalk, this new paradigm has evolved, and new languages, methodologies, and tools have appeared.

3. CHARACTERISATION OF AN OBJECT-ORIENTED MODEL

Although object-oriented programming has its roots in the 1970s, there were many definitions about what precisely the term object-oriented meant. The term meant different things to different people because it had become very fashionable to describe any software system in terms of object-oriented concepts. To some, the concept of object was merely a new name for abstract data types; each object had its own private variables and local procedures, resulting in modularity and encapsulation. To others, classes and objects were a concrete form of type theory; in this view, each object is considered to be an element of a type which itself can be related through sub-type and super-type relationships.

To others still, object-oriented software systems were a way of organizing and sharing code in large software systems. Individual procedures and the data they manipulate are organized into a tree structure. Objects at any level of this tree structure inherit behavior of higher level objects; inheritance turned out to be the main structuring mechanism which made it possible for similar objects to share program code. Despite many authors being concerned with providing precise definitions for the object-oriented paradigm, it was difficult to come up with a single generally accepted definition.

Rentsch [10] defines object-oriented programming in terms of inheritance, encapsulation, methods, and messages, as found in Smalltalk. Objects are uniform in that all items are objects and no object properties are visible to an outside observer. All objects communicate using the same mechanism of message passing, and processing activity takes place inside objects. Inheritance allows classification, sub-classification and super-classification of objects, which permits their properties to be shared.

Pascoe [11] also presents object-oriented terminology from the Smalltalk perspective. Pascoe defines an object-oriented approach in terms of encapsulation, data abstraction, methods, messages, inheritance, and dynamic binding for object-oriented languages. Pascoe also affirms that some languages that have one or two of these features have been improperly called object-oriented languages. For instance, Ada could not be considered an object-oriented language because it does not provide inheritance.

Other authors, such as Robson [12] and Thomas [13], emphasize the idea of message passing between objects and dynamic binding as fundamental to object-oriented programming. There is no doubt those authors have also been influenced by the Smalltalk language, wherein the message passing mechanism plays a fundamental role as the way of communication among objects. On the other hand, Stroustrup [14] claims that object-oriented programming can be seen as programming using inheritance, and that message-passing is just an implementation technique, not at all an inherent part of the paradigm.

Nygaard [15] discusses object-oriented programming in terms of the concept of objects in Simula. In that language an execution of a computer program is organized as the joint execution of a collection of objects. The collection as a whole simulates a real-world application, and objects sharing common properties are said to constitute a class. Madsen and Moller-Pedersen [16], like Nygaard [15], sees object-oriented programming as a model that simulates the behavior of either a real or imaginary part of the world. The

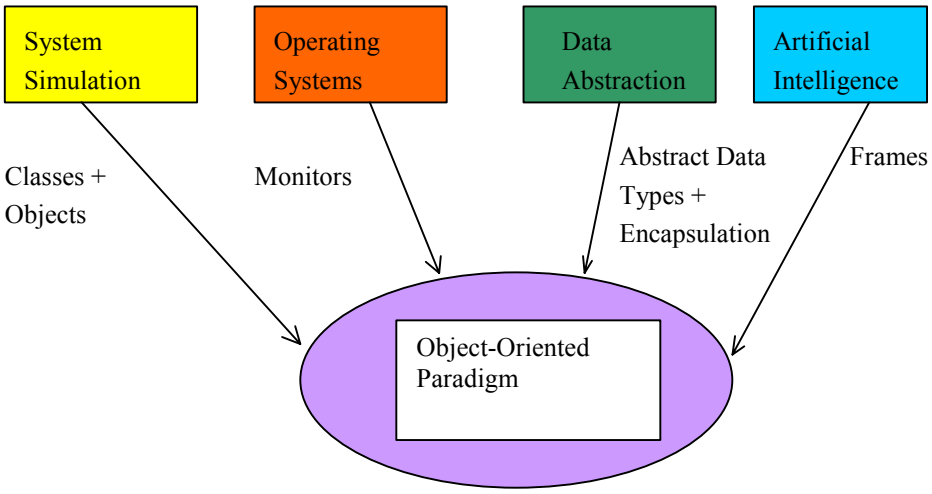

Figure 1: The Background of the Object-Oriented Paradigm.

model consists of objects defined by attributes and actions, and the objects simulate phenomena. Any transformation of a phenomenon is reflected by actions on the attributes. The state of an object is expressed by its attributes and the state of the whole model is the state of the objects in that model.

The object-oriented paradigm was still lacking a well-known and profound theoretical understanding, then some research come out in this area. Cardelli and Wegner [17], for example, with formal methods that used denotational semantics, described the essential features of the object-oriented paradigm, such as abstract data types, objects, classes and inheritance.

Lastly, Wegner [18] characterized an object-oriented approach in terms of objects, classes and inheritance. Objects are autonomous entities that have a state and respond to messages; classes arrange objects by their common attributes and operations; inheritance serves to classify classes by their shared commonality. Thus: object-orientation = objects + classes + inheritance. The characterization of an object-oriented approach proposed by Wegner has been the most accepted one.

As it has been described, there are many different interpretations of the object-oriented paradigm. Nevertheless, one thing that all definitions have in common, not surprisingly, is the recognition that an object is the primitive concept of the object-oriented paradigm. The object is an encapsulation of protected data along with all the legal operations that act on that hidden information.

4. COMPARISON BETWEEN "OBJECT-ORIENTED" LANGUAGES

At the beginning of programming language development, assembly languages only enabled programmers to write code based on machine instructions (operators) that manipulated the contents of memory locations (operands). Therefore the level of control and data abstraction achieved was very low. A great leap forward occurred when the first higher-level languages, e.g. Fortran and Algol, appeared. The operators turned into statements and operands into variables and data structures. The traditional view of programs in these languages is that they were composed of a collection of variables that represented some data and a set of procedures that manipulated those variables. The majority of traditional programming languages supported this data-procedure paradigm. That is, active procedures operate upon passive data that is passed to them. Things happen in a program by invoking a procedure and giving to it some data to manipulate. Through a sequence of statements and procedures, early higher-level languages had reasonable support to implement actions; however, they had shortcomings to represent abstract data types.

Abstract data types are abstractions that may exist at a higher level than operands and operators, or variables and procedures separately. Some languages provided a construct that allowed both variables and procedures to be defined within a single unit; for instance the cluster construct in CLU, which satisfies the definition of abstract data types. The same idea can also be found in Ada through the package construct. Nevertheless, if two abstract data types are similar but not identical, there is no means of expressing their commonality conveniently in a programming language that supports only abstract data types.

The object-oriented paradigm goes a step further than abstract data types; that is, object-oriented languages allow similarities and differences between abstract data types to be expressed through inheritance, which is a key defining feature of the object-oriented paradigm. Therefore it would be better to characterize the evolution of object-oriented languages based on the support for both abstract data types and inheritance; in this case the immediate ancestor of object-oriented languages was Simula, which was an Algol-based language. Simula was the first language to introduce the concept of class and to allow inheritance to be expressed, and it should be recognized as the "mother" of a few object-oriented programming languages. Besides, because object-oriented concepts have also arisen from the artificial intelligence community, it is not surprising that Lisp has influenced a number of object-oriented languages. For instance, Flavors [19], Loops [20] and CLOS [21], have all borrowed ideas from Lisp and Smalltalk.

The prominence of the object-oriented paradigm has influenced the design of other programming languages. There are languages that incorporate object-oriented constructs into the popular C, Pascal and Modula-2, resulting in the hybrid languages Objective-C [22], C++ [23], ObjectPascal [24] and Modula-3 [25]. The inclusion of object-oriented concepts into traditional languages sophisticated them, in that programmers had the flexibility to use or not to use the object-oriented extensions and benefits. Although these hybrid languages became more complex, those extensions enabled programmers who had considerable experience with those traditional procedure languages to explore incrementally the different concepts provided by the object-oriented paradigm. Nevertheless, when using a hybrid language, programmers had to exercise more discipline than when using a pure object-oriented language because it was too easy to deviate from sound object-oriented principles. For instance, C++ allows global variables, which violates the fundamental principle encapsulation.

As far as concurrency is concerned, objects can also be viewed as concurrent agents that interact by message passing, thus emphasizing the role of entities such as actors and servers in the structure of a real-world application. The main idea behind object-oriented languages that support concurrency is to provide programmers with powerful constructs that allow objects to run concurrently. Concurrency adds the idea of simultaneously executing objects and exploiting parallelism. Languages to which this applies include: Actor [26], ABCL [27], POOL-T [28], Orient84 [29] and ConcurrentSmalltalk [30].

Other languages influenced basically by Simula and CLU, such as Beta [31] and Eiffel [32] have also appeared and are believed to give good support for the object-oriented paradigm. Although Eiffel and Smalltalk seem to be coherent object-oriented languages with integrated programming environments, C++ has become the most used object-oriented programming language, due to the influence of UNIX and the popularity of the C language from which C++ derived. Finally, Java [33] should look familiar to C and C++ programmers because Java was designed with similar but cleaner constructs; it also provides a more robust library of classes. Java is rapidly gaining territory among programmers, and it is expected to become the most popular object-oriented language. Analyzing the evolution of all those languages over time leads to the dependency graph shown in Figure 2.

A programming language is called object-based if it permits the definition of objects as abstract data types only, whereas, a language is called object-oriented if it allows the definition of objects and supports the inheritance mechanism. According to this classification, the set of object-based languages includes Ada and CLU. This is so because objects in Ada are defined as packages and objects in CLU are instances of clusters. The set of object-oriented languages is smaller than the set of object-based languages, and excludes Ada and CLU but includes Smalltalk and C++ because the latter two support inheritance. Table 1 shows a comparison between some of the programming languages mentioned above.

When serious programming is mentioned, it is not just about the language, it is also about library support that has been built around a language, forming a platform that helps to develop software systems.

It can be concluded that, despite the possibility of following an object-oriented fashion using languages (e. g. Ada and CLU) with less or more difficulty, direct language support is beneficial in facilitating as well as encouraging the use of the object-oriented tenets such as in Eiffel or Java. Not only do these languages support the object-oriented paradigm, but also they enforce it because the main language constructs dealt with are related to objects, classes and inheritance. The danger in trying to force object-oriented concepts into a language that does not provide inheritance is that weird constructions may be produced, impairing software development and jeopardizing the quality of the resulting software.

Table 1: Comparing Languages

Features X Languages	Abstract Data Types	Inheritance Support	Dynamic Binding	Extensive Library
Simula	yes	yes	yes	no
CLU	yes	no	yes	no
Ada	yes	no	no	yes
Smalltalk	yes	yes	yes	yes
ObjectiveC	yes	yes	yes	yes
C++	yes	yes	yes	yes
CLOS	yes	yes	yes	no
Obj.Pascal	yes	yes	yes	no
Beta	yes	yes	yes	no
Eiffel	yes	yes	yes	yes
Actor	yes	yes	yes	no
Java	yes	yes	yes	yes

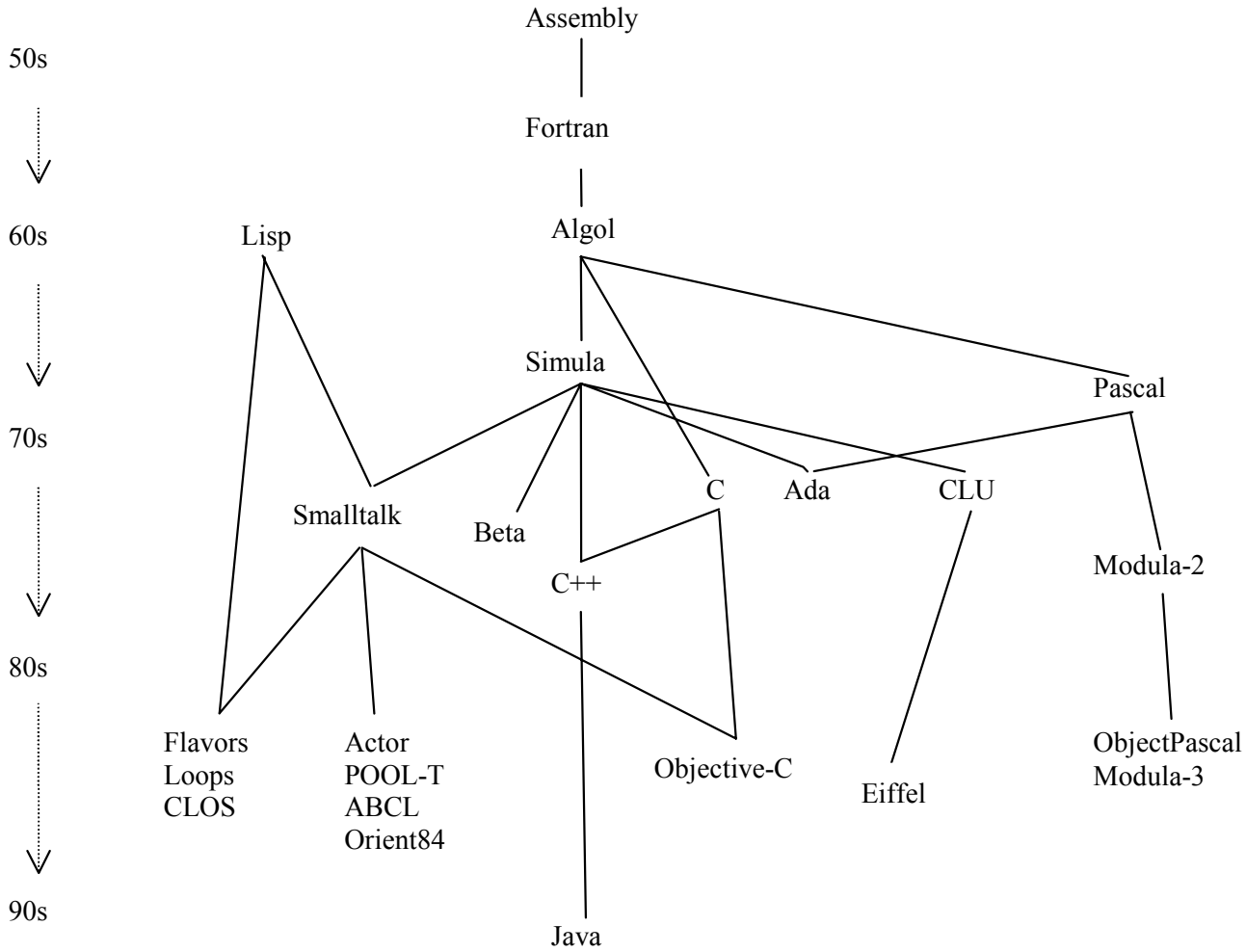

Figure 2: Language Evolution.

5. CLASSIFICATION OF OBJECT-ORIENTED METHODOLOGIES

An important idea brought forward by software engineering is the concept of software life cycle models. Several models have been proposed in order to systematize the several stages that a software system goes through [34, 35, 36]. In parallel, many software development methodologies have also been proposed over the last few decades. Such methodologies provided some discipline in handling the inherent software complexity because they usually offered a set of rules and guidelines that helped software engineers understand, organize, decompose and represent software systems.

Those methodologies may be classified into three approaches. First, some methodologies dealt with functions; they emphasized refinement through functional decomposition as, for example, Structured Design [37], HIPO [38] and Stepwise Refinement [39]. Typically, software development has to follow a top-down fashion by successively refining functions.

In a second line of thought, there were methodologies that recommended that software systems should be developed with emphasis on data rather than on functions. That is, the system architecture was based on the structure of the data to be processed by the system. The software system should be structured mainly through the identification of data components and their meaning. This technique could be noted in the early Jackson Structured Programming methodology [40] and the Entity-Relationship Model (ERM) [41]. The Entity-Relationship Model was the most common approach to data modeling in the 1970s and 1980s. ERM is a graphical technique easy to understand yet powerful enough to model real-world applications, then entity-relationship diagrams are readily translated into a database implementation.

A third style consisted of methodologies that aimed at developing software systems from both functional and data points of view, but separately. Examples of such methodologies are SADT [42], Structured Analysis [43] and Structured System Analysis [44]. SADT provides different kinds of diagrams to represent functions, control, mechanisms and data. As far as Structured Analysis and Structured System Analysis are concerned, designers can represent and refine functions through data flow diagrams, (which also show functions) and use a data dictionary to describe data. So that engineering applications could be better modeled, Ward and Mellor [45] introduced real-time extensions into structured analysis. Finally, Structured System Analysis and Design Methodology (SSADM) [46] is another renowned structured analysis approach.

These methodologies, known as structured, organize a system specification and design around hierarchies of functions. Structured analysis begins by identifying one or more high level functions that describe the overall purpose of a software system. Then, each high level function is broken down into smaller less complex functions, followed by structured design and structured programming. Needless to say, these methodologies have been supported by a myriad of CASE tools. The main purposes of the tools were to increase productivity, help with system documentation and enhance the quality of the developed software.

A combination of approaches that followed structured analysis, structured design, and structured programming was collectively known as structured development. This approach iteratively divides complex functions into sub-functions. When the resulting sub-functions are simple enough, decomposition stops. This process of refinement was known as the functional decomposition approach. Structured development also included a variety of notations for representing software systems. During the requirements and analysis phases, data flow diagrams, entity-relationship diagrams and a data dictionary are used to logically specify 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 appeared to be an attractive starting point to be followed by object-oriented design primarily because it was well known, many software professionals were trained in its techniques, and several tools supported its notations. However, structured analysis was not the ideal front-end to object-oriented design, mainly because it perpetuated a functional decomposition view of the system. Applying a functional decomposition approach first and an object-oriented approach later on the same software system led to trouble because functions could not be properly mapped into objects.

Ideally, object-oriented design and implementation should be part of a software development process in which an object-oriented philosophy is used throughout software development, as shown in Figure 3. In that figure, the dashed arrows represent an unnatural mapping between concepts of different approaches, as opposed to the bold arrows, which indicate a smooth transition from one phase to the next. Consequently, attempting to combine an object-oriented approach with a structured development approach gave rise to some problems.

Because, in early phases, a software system was described in terms of functions and later on the description was changed to object-oriented terms (see Figure 3), it jeopardized traceability from requirements to implementation. Structured development methodologies did not place data within objects but on the data flow between functions, and a software system was described with data flows and functions. In contrast, the object-oriented paradigm organizes a software system around classes and objects that exist in the designer's view of the real-world application.

On the other side, there has also been a profusion of so-called “object-oriented” methodologies for analysis and design influenced by different backgrounds, and found in a variety of software life cycle models. Nevertheless, two major trends can be noticed:

1) Adaptation: it has been concerned with mixing an object-oriented approach with well-known structured development methodologies.
2) Assimilation: it has emphasized the use of an object-oriented methodology for developing software systems, but has followed the traditional waterfall software life cycle model.

5.1 Adaptation

Adaptation proposes a framework to mix an object-oriented approach with existing structured methodologies. Henderson-Sellers and Constantine [47] suggested that a combination of structured development with an object-oriented approach could smooth software development. Based on a functional decomposition designers could 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. The adaptation of structured development to an object-oriented approach preserves the specification and analysis phases using data flow diagrams, and it proposes heuristics to convert these diagrams into an object model in such way that subsequent phases can then follow an object-oriented approach. Some advantages of this adaptive approach are:

- A complementary coupling between structured development and the object-oriented approach.
- A smoother migration from well-practiced and well-known approaches to a new one that included classes, objects and inheritance.
- Gradual change from old tools and environments to a new paradigm.

The most widely used software engineering methodologies have been those for structured development. Such methodologies have been popular because they were applicable to many types of application domains. Because of this popularity, structured development has been combined with an object-oriented approach. Software engineers, who had used functional decomposition and data modeling techniques, have probably found the methodologies of Shlaer and Mellor [48] as well as Coad and Yourdon [49] familiar because these methodologies are clearly adaptations of traditional structured development methodologies and data modeling techniques.

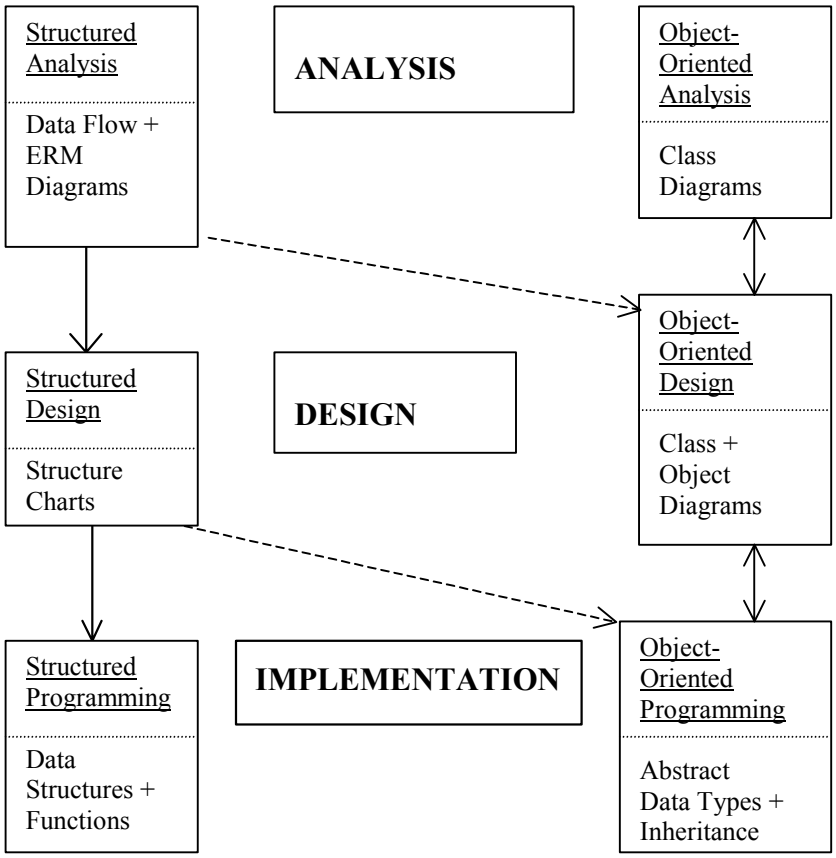

Figure 3: Some Combinations of Approaches

Those methodologies oversimplified the object-oriented paradigm by misusing the concepts of classes and objects during the analysis phase. Basically, they concentrated on modeling real-world entities as objects, and they can be considered as extensions of the Entity-Relationship Model [41], suggesting that they are incremental improvements of existing approaches to data modeling. Moreover they have not discussed the impact of their methodology on other phases of the software life cycle. These methodologies were used during a period of transition from structured development to object-oriented development as a compromise. However, they did not permit the full advantages of an object-oriented approach.

Jackson [50] has proposed a methodology called the Jackson System Development (JSD). JSD has some features that 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 little to support the object-oriented paradigm, and inheritance is completely ignored. Other less known proposals in which object-oriented concepts are derived from structured development can also be mentioned. Some of these methods were merely extensions of structured development techniques. Masiero and Germano [51] and Hull *et al*. [52] put together object-oriented design with JSD, and the by-product of the design is implemented in Ada. Bailin [53] and Bulman [54] combined object-oriented development with Structured System Analysis [44] and the Entity-Relationship Model [41] in an object-oriented requirements specification model. Lastly, Alabiso [55] and Ward [56] combined object-oriented development with Structured Analysis [43], Structured Design [37] and the Entity-Relationship Model [41].

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 [57] and Booch [58]. Booch rationalized Abbott's method, and referred to it as Object-Oriented Design [59]. Both Abbott and Booch have recommended that a design should start with an informal description of the real-world application and from that narrative description designers could identify classes and objects from nouns, and operations from verbs. Booch's work was significant because it was one of the earliest object-oriented design methodologies to be described. He was also one of the most influential advocates of object-oriented design within the Ada community.

Realizing the drawbacks of the technique based on identification of classes and objects from informal descriptions, later, Booch no longer used a narrative description. Instead, Booch [60] combined object-oriented design with existing methodologies and called it Object-Oriented Development. He suggested that existing methodologies such as SREM [61] or Structured System Analysis [44] or JSD [50] could be used during the system analysis phase as a step before object-oriented design. Subsequently, Booch [62] proposed a truly object-oriented design methodology.

As far as Booch's influences are concerned, they can be summarized as follows: what has come to be known as object-oriented design in the context of Ada was first proposed by Booch [58],

later extended and generalized by Booch [60], then refined by Seidewitz [63], Heitz [64] and Jalote [65]. Berard [66] and Sincovec and Wiener [67] also presented principles and methods biased by Booch [58] with implementation driven towards Ada. These design methodologies concentrated on identifying objects and operations, and were object-oriented in the sense that they viewed a software system as a collection of objects. Wasserman *et al*. [68] have proposed OOSD, a graphical representation for Object-Oriented Structured Design. OOSD provided a standard design notation by supporting concepts of both structured and object-oriented design. The main ideas behind OOSD came from Structured Design [37] and Booch [60] notation for Ada packages. Most of these methodologies were based on an informal description or representation of the software requirements, from which objects, attributes and operations were identified. Moreover, all of these methodologies applied hierarchical decomposition, a trend to decompose a software system by breaking it into smaller components through a series of top-down refinements towards an implementation in Ada.

5.2 Assimilation

In the 1980s and 1990s several object-oriented methodologies appeared but they covered only partially the software life cycle model. Assimilation was a trend that put the object-oriented paradigm within the traditional waterfall software life cycle model. Several authors tried to fit the object-oriented paradigm into this framework: Lorensen [69], Jacobson [70], Wirfs-Brock *et al*. [71], Rumbaugh *et al*. [72] and Booch [62] can be considered good examples.

Lorensen [69] described the rudiments of object-oriented software development by explaining that it was fundamentally different from traditional structured development methods, such as those based on data flow diagrams and a functional decomposition approach.

Jacobson [70] claimed to have a full object-oriented development methodology named the ObjectOry, which combined a technique to develop large software systems termed block design [73] with conceptual modeling [74] and object-oriented concepts. Jacobson stated that it was quite natural to unite these three approaches since they rely on similar ideas aiming at, among other things, the production of reusable software components.

Wirfs-Brock *et al*. [71] focused 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 that comprise a real-world application. A contract is a set of related responsibilities defined by a class, and describes the ways by which client objects can interact with server objects. Introduced by Beck and Cunningham [75], was a technique that recorded design on cards, and which proposed the Class, Responsibility, and Collaboration (CRC) cards. It has been suggested that using CRC cards is a simple technique for teaching object-oriented thinking to newcomers.

Rumbaugh *et al*. [72] developed the Object Modeling Technique (OMT), which focused on object modeling as a software development technique. OMT is a comprehensive methodology that incorporates structured development based on a functional decomposition approach following the traditional waterfall software life cycle model.

Booch [62] introduced a comprehensive object-oriented methodology for software development with a graphical notation to express a design, one that could form the basis for automated tools. He also included a variety of models that addressed the functional and dynamic aspects of software systems.

6. FINAL REMARKS

This paper has expanded upon the background of the object-oriented paradigm. This paradigm has provided a powerful set of concepts completely absorbed into the software development culture of the 1990s, just as, in the same way, structured development methodologies (and, to some extent abstract data types concepts) had been in the 1970s and 1980s. This is evident in the abundance of tools supporting all aspects of software development following this paradigm. Consequently the last decade has been a period of gradual acceptance of the object-oriented paradigm, which has become the main approach to developing software systems since the early 1990s.

One great advantage of using the object-oriented paradigm is the conceptual continuity across all phases of the software development life cycle; that is, the conceptual structure of the software system remains the same, from system analysis down through implementation. 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. Capretz and Capretz [76] describe a methodology for object-oriented design and maintenance, which takes domain analysis and software reusability into account as important aspects of an alternative software life cycle model. However, object-orientation has needed an organized and manageable view of software development permeating all phases of the software life cycle model. That demand has been met by the Unified Modeling Language (UML) [77] and by CASE tools such as Rational Rose.

Because there are unique object-oriented concepts involved in the whole software development process, there should have been specific methodologies suitable to the development of that object-oriented software. However, history shows that the object-oriented software development has been combined with other approaches; it was influenced by, and has been influencing, other ideas. After more than thirty years since the first object-oriented programming language was introduced, the debate over the claimed benefits of the object-oriented paradigm still goes on. But there is no doubt that most new software systems will be object-oriented; that, nobody disputes.

REFERENCES

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

[2] Micallef J. Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages, *Journal of Ob-*

ject-Oriented Programming, 1(1), pp. 12-36, April 1988.

[3] Gossain S. and Anderson B. An Iterative-Design Model for Reusable Object-Oriented Software, *ACM SIGPLAN Notices*, 25(10), pp. 12-27, October 1990.

[4] 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,* Englewood Cliffs, New Jersey: Prentice Hall, pp. 139-150, August 1992.

[5] Dahl O.-J., Myhrhaug B. and Nygaard K. *SIMULA67 Common Base Language*, Publication No. S-22, Oslo: Norwegian Computing Centre, 1970.

[6] Hoare C. A. R. Monitors: an Operating Systems Structuring Concept, *Communications of the ACM*, 17(10), pp. 549-577, October 1974.

[7] Liskov B., Snyder A., Atkinson R. and Schaffert, C. Abstraction Mechanisms in CLU, *Communications of the ACM*, 20(8), pp. 564-576, August 1977.

[8] Minsky M. A Framework for Representing Knowledge, Wiston P. (ed.) *The Psychology of Computer Vision*, New York: McGraw-Hill, 1975.

[9] Goldberg A. and Robson D. *Smalltalk-80: The Language and its Implementation*, Reading, Massachusetts: Addison-Wesley, 1983.

[10] Rentsch T. Object Oriented Programming, *ACM SIGPLAN Notices*, 17(9), pp. 51-57, September 1982.

[11] Pascoe G. A. Elements of Object-Oriented Programming, *Byte* 11(8), pp. 139-144, August 1986.

[12] Robson D. Object-Oriented Software Systems, *Byte*, 6(8), pp. 74-86, August 1981.

[13] Thomas D. What's in an Object, *Byte*, 14(3), pp. 231-240, March 1989.

[14] Stroustrup B. What is Object-Oriented Programming?, *Lecture Notes in Computer Science*, No. 276, pp. 51-70, Berlin: Springer-Verlag, 1987.

[15] Nygaard K. Basic Concepts in Object Oriented Programming, *ACM SIGPLAN Notices*, 21(10), pp. 128-132, October 1986.

[16] Madsen O. L. and Moller-Pedersen B. What Object-Oriented Programming May Be and What It Does Not Have to Be, *Lecture Notes in Computer Science*, No. 322, pp. 1-20, Berlin: Springer-Verlag, 1988.

[17] Cardelli L. and Wegner P. On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys*, 17(4), pp. 471-522, December 1985.

[18] Wegner P. Dimensions of Object-Based Language Design, *ACM SIGPLAN Notices*, 22(12), pp. 168-182, December 1987.

[19] Moon D. A. Object-Oriented Programming with Flavors, *ACM SIGPLAN Notices*, 21(11), pp. 1-8, November 1986.

[20] Stefik M. and Bobrow D. G. Object-Oriented Programming: Themes and Variations, *The AI Magazine*, 6(4), pp. 40-62, April 1986.

[21] DeMichiel L. G. and Gabriel R. P. The Common Lisp Object System: An Overview, *Lecture Notes in Computer Science*, No. 276, pp. 151-170, Berlin: Springer-Verlag, 1987.

[22] Cox B. J. *Object-Oriented Programming - An Evolutionary Approach*, Readings, Massachusetts: Addison-Wesley, 1986.

[23] Stroustrup B. *The C++ Programming Language*, Reading, Massachusetts: Addison-Wesley, 1986.

[24] Tesler L. *Object Pascal Report*, Santa Clara, California: Apple Computer, 1985.

[25] Cardelli L. *Modula-3 Report*, Palo Alto, California: Digital Equipment Corporation, 1989.

[26] Agha G. An Overview of Actor Languages, ACM *SIGPLAN Notices,* 21(10), pp. 58-67, October 1986.

[27] Yonezawa A., Shibayama E., Takada T. and Honda Y. Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, Yonezawa A. and Tokoro M. (eds.) *Object-Oriented Concurrent Programming*, pp. 55-90, Cambridge, Massachusetts: MIT Press, 1987.

[28] America P. POOL-T: A Parallel Object-Oriented Language, Yonezawa A. and Tokoro M. (eds.) *Object-Oriented Concurrent Programming*, pp. 199-220, Cambridge, Massachusetts: MIT Press, 1987.

[29] Yutaka I. and Tokoro M. A Concurrent Object Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation, *ACM SIGPLAN Notices*, 21(11), pp. 232-241, November 1986.

[30] Yokote A. and Tokoro M. Concurrent Programming in ConcurrentSmalltalk, Yonezawa A. and Tokoro M. (eds.) *Object-Oriented Concurrent Programming*, pp. 129-158, Cambridge, Massachusetts: MIT Press, 1987.

[31] Kristensen B. B., Madsen O. L., Moller-Pedersen B. and Nygaard K. Multi-Sequential Execution in the Beta Programming Language, *ACM SIGPLAN Notices*, 20(4), pp. 57-70, April 1985.

[32] Meyer B. *Object-Oriented Software Construction*, Englewood Cliffs, New Jersey: Prentice-Hall, 1988.

[33] Arnold K. and Gosling J. *The Java Programming Language*. Reading, Massachusetts: Addison-Wesley, 1996.

[34] Royce W. W. Managing the Development of Large Software Systems, *Proceedings of the 9th International Conference on Software Engineering*, pp. 328-338, IEEE Press, 1987.

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

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

[37] Yourdon E. and Constantine L. L. *Structured Design*, Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

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

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

[40] Jackson M. A. *Principles of Program Design*, New York, New York: Academic Press, 1975.

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

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

[43] DeMarco T. *Structured Analysis and System Specification,* Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

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

[45] Ward P. and Mellor S. *Structured Development for Real-Time Systems*, Englewood Cliffs, New Jersey: Prentice-Hall, 1985.

[46] Eva M. *SSADM Version 4: A User's Guide*, London: McGraw-Hill, 1994.

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

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

[49] Coad P. and Yourdon E. *Object-Oriented Analysis*, Englewood Cliffs, New Jersey: Prentice-Hall, 1990.

[50] Jackson M. A. *System Development*, London: Prentice-Hall, 1983.

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

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

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

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

[55] Alabiso B. Transformation of Data Flow Analysis Model to Object-Oriented Design, *ACM SIGPLAN Notices*, 23(11), pp. 335-353, November 1988.

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

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

[58] Booch G. *Software Engineering with Ada*, Menlo Park, California: Benjamin/Cummings, 1983.

[59] Booch G. Object-Oriented Design, Freeman P. and Wasserman A. I. (eds.) *Tutorial on Software Design Techniques*, 4th Edition, pp. 420-436, IEEE Press, 1983.

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

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

[62] Booch G. *Object-Oriented Design with Applications*, Redwood City, California: Benjamin/Cummings, 1991.

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

[64] Heitz M. *HOOD Reference Manual, Issue 3.0,* Noordwijk, The Netherlands: European Space Agency, 1989.

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

[66] Berard E. *An Object-Oriented Design Handbook*, Rockville, Maryland: EVB Software Engineering Inc., 1986.

[67] Sincovec R. F. and Wiener R. S. Modular Software Construction and Object-Oriented Design Using Ada, Peterson G. E. (ed.) *Tutorial: Object-Oriented Computing*, pp. 30-36, IEEE Press, 1987.

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

[69] Lorensen W. *Object-Oriented Design, CRD Software Engineering Guidelines*, General Electric Co., 1986.

[70] Jacobson I. Object Oriented Development in an Industrial Environment, *ACM SIGPLAN Notices*, 22(12), pp. 183-191, December 1987.

[71] Wirfs-Brock R., Wilkerson B. and Wiener L. *Designing Object-Oriented Software*, Englewood Cliffs, New Jersey: Prentice Hall, 1990.

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

[73] Jacobson I. Language Support for Changeable Large Real Time System, *ACM SIGPLAN Notices*, 21(11), pp. 377-384, November 1986.

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

[75] Beck K. and Cunningham W. A *Laboratory for Teaching Object-Oriented Thinking, ACM* SIGPLAN Notices, 24(10), pp. 1-6, October 1989.

[76] Capretz L. F. and Capretz M. A. M. *Object-Oriented Software: Design and Maintenance*. Singapore: World Scientific, 1996.

[77] Booch G., Rumbaugh J. and Jacobson I. *The Unified Modeling Language User Guide*. Reading, Massachusetts: Addison-Wesley, 1999.

Author's bio-sketch: Dr. L. F. Capretz has extensive experience in software engineering. He has worked (both at technical and managerial levels), taught and done research on the engineering of software in Brazil, Argentina, England and Japan since 1981. In the Faculty of Engineering at the University of Western Ontario (Canada), he teaches software design in an accredited program that offers a degree in software engineering. Currently, he is focusing his research in component-based software engineering and software product lines.