Programs as objects

M. O'Brien, S. Maglaras
Computer Science Department, The University of Nottingham, University Park - NG7 2RD Nottingham UK
E-Mail: szm@cs.nott.ac.uk

Abstract

In the field of software engineering, the last decade has seen the emergence of object-oriented data models. Aspects of the object paradigm have been applied throughout computer science. This model has emerged as a possible solution to the problem of designing and implementing large software systems and increasingly complex applications. A major step towards this object-oriented implementation approach has been made with the advent of the so called object-oriented languages, either as extensions of existing languages or as new pure object-oriented programming environments. There has been substantial theoretical work on the features these languages should offer to the programmer in order to conform to the object paradigm. However, the actual treatment of the programs themselves as objects has been somewhat overlooked. This paper offers a perspective of programs from that point of view; it is an attempt to redefine a computer program's attributes in terms of object-oriented theory, assigning each of the main concepts of a computer program its place in the object model.

Introduction

The incorporation of the object-oriented approach in software engineering has already been suggested as the main mechanism for integrating the various stages of software production under the same data model. Major research attempted to use object-oriented ideas in software engineering (Coad & Yourdon[9]). Yet these efforts have been predicated on taking new tools and applying them to an old problem; the emphasis has been on application to the detriment of theory. This rush to apply new tools has led to a certain lack of reflection which this paper seeks to redress. This paper considers a particular application of the object model in software engineering. More specifically, not only the logical but the physical state (textual features, implementation files) of computer programs is treated from the object-oriented perspective. Additionally, the effects of that approach on present software engineering tools is investigated.

One of the main concerns of software engineering has been 'programming in the large'. The advent of the Modula-2 programming language (Wirth[27]) addressed the challenge of building efficient large scale systems.

Modula-2 introduced the concept of modularity in which a software system is treated as a collection of autonomous components, each one handling a conceptual aspect of the system. Other major research considerations have included the need for robustness and effortless maintenance of software.

This paper shifts the emphasis away from such all-embracing concerns and looks in detail at the small scale manifestations of textual source code and associated structures. Moreover, it attempts to comment on the nature of these manifestations, as redefined and re-labelled within the concepts and terminology of the object model.

The Program Inconsistency - The New Approach

The contribution of object-oriented languages in the propagation of the object model as a powerful data modelling tool among computer scientists has been substantial. Either as pure object languages like Smalltalk (Goldberg & Robson[14]), Eiffel (Meyer[21]) and Trellis/Owl (Schaffert *et. al.*[22]) or as hybrid object languages like C++ (Stroustrup[26]), Objective-C (Cox[12]) and Object Pascal (Schmucker[23]) and functional languages like the Common Lisp Object System-CLOS (Bobrow *et. al.*[5]), all object languages have served as testbeds for thorough testing and continuous improvement of the theory underlining the use of objects. Many of these object programming systems were focused at serving particular types of applications such as graphical interfaces (Eiffel, Object Pascal) and database applications (ODE (Agrawal & Gehani[2]), O_2 (Bancilhon *et. al.*[3])).

However, research on the fact that programs are also physical entities has been somewhat confined to the quantitative and qualitative analysis of the structural properties of source code such as complexity and computational completeness. All the syntactic and semantic information that a program carries is stored and manipulated through a series of actions which take place on that underlying physical level. Therefore, applying the object model only to the syntactic and semantic level of a program creates an obvious inconsistency. As a result, programs that contain object-oriented code as an improvement on their design and implementation, are still not being treated, manipulated and operated upon, as objects themselves.

This paper addresses this inconsistency by introducing an object-oriented critique of the major physical aspects and operations of a program. The analysis of the traditional form of programs that follows examines the current features of a program under the scope of object orientation.

Useful Assumptions and Clarifications

It is helpful to clarify the concept of a program as used within this paper. The attention of the following analysis is directed towards procedural programs. The term 'procedural' denotes source code written in some procedural language (C, Pascal, etc.) or in any of the aforementioned object languages whose code encompasses traditional textual and physical attributes (code blocks, linked

libraries, source/object files). In addition, programs developed in some database related procedural Data Manipulation Language (DML) such as Access Basic, are included in the model as well.

Some common physical characteristics are assumed to be present in all these programs mentioned above. They all follow more or less the standard 'Edit - Compile - Link - Run' model of software production and usage. Conformance to that traditional model of compilation implies the existence of a program in various forms (source code, object file, executable program). All the forms of a program will be referred to throughout the text under the following filename extensions:

a) ***.SRC*** -> Uncompiled source code
b) ***.OBJ*** -> Compiled, Non-linked Object Files
c) ***.EXE*** -> Executable programs

Other points relative to this model are assumptions on the physical storage location and execution of the programs under study. An attitude has been adopted under which source code is thought of as always stored externally of its development environment. Additionally, programs are considered to be directly executable (command line execution) or indirectly executable (called by other programs). In either case, they are supposed to be running within the functional environment of the operating system. Programs not conforming to this set of assumptions (for example, SQL programs) are beyond the scope of this study.

An Object Perspective of Programs

In the beginning, the epicentre of attention has been the object analysis of software systems. Several methodologies emerged in that field such as (Booch[6]), (Coad & Yourdon[9]), and (Seidewitz & Stark[24]). All these are requirement specification methods focused on producing a set of object-oriented design guidelines. Unfortunately, none of them advanced further than their design scope in order to include programs as physical entities in their object models. However, there have been some efforts pointing towards that direction, such as Brown[7], where it has been suggested that programming languages can be treated as base classes in their own right.

It is a significant argument on its own what is and what is not considered a basic element of the object model. This problem arises due to the continuing lack of standardization of the field (Maier[19]), as no equivalent of Codd's paper for the relational model (Codd[10]) exists. For that reason, the selection of the object features used in this analysis has been made purely by observing the current debate of this matter in the literature.

Transactions on Information and Communications Technologies vol 14, © 1997 WIT Press, www.witpress.com, ISSN 1743-3517

1) Encapsulation

The inherent encapsulation that programs display in their physical form emerges if the old concept of treating programs as black boxes is considered. According to that early software engineering theory a program is an opaque black box with an input, a processing mechanism invisible to the outside world and an output. The resemblance of that definition to the main principle of information hiding implemented in encapsulation is nonetheless striking. Extending the 'black-box' definition with the proper object-oriented distinction between interface and implementation the following perspective of programs is suggested:

i) The Interface. The interface of a program consists of both the parameters of its execution and the resulting output which the user gets as feedback (including error messages). The same applies to calling programs, since they are never aware of the internal operation of the programs included in their code.

ii) The Implementation. It is the whole internal structure of the program which performs the actual tasks of the program. Routines, variables, constructs and library calls form the non-transparent mechanism which in the operational state of the program (executable) are completely isolated from the external environment.

This is a generalized view of a program used to introduce the object-oriented behaviour of programs. To construct a more detailed perspective of the encapsulation that programs contain, some issues have to be addressed. The first would be defining the actual limits of the object 'program'. This could be quite a trivial task considering the various states that a program simultaneously exists in. The existence of the various physical files which compose the logical entity 'program' (***.SRC, .OBJ, .EXE***), as stated on the assumptions part, poses a problem in to what might be included in the object 'program'. This consideration leads into two different forms of encapsulation. Their distinct approaches on the previous problem carry along many implications to other aspects of the object analysis, so it would be useful to be thoroughly presented.

a) The "narrow" form. According to the narrow model of encapsulation the limits of the object 'program' are constrained around the source file (***.SRC***) where the actual code is stored. Subsequently, all the rest states of the program are considered to be mere by-products of operations (compilation, linking) taking place on the source file. The immediate effect of that point of view is that ***.OBJ*** and ***.EXE*** files are considered to be output produced by initiating those operations. In this model, a shift of importance is emphasized towards the source file in relation to the other files of program which as it will be displayed further on affects the size and complexity of the way the actual object is constructed and managed.

b) The "broad" form. The term 'broad encapsulation' implies the inclusion of all different states of a program into the environment of the object. The emphasis here is on the fact that by definition these states are tied up together in an inseparable way; if any are missing the object 'program' is

incomplete. Broad encapsulation dictates that no object 'program' actually exists unless the succession of these stages has been completed. The result in this case is that a more complex mechanism has to be devised in order to create and manipulate the object. However, there could be a serious argument in favour of the broad form if looked upon from a pure object-oriented point of view. The argument is related to the conformance of this form to the principles of object orientation. All possible manifestations of the scope of the object 'program' have to be contained within the object; therefore that should be the case with ***.OBJ*** and ***.EXE*** files as well. That is particularly the case when examining the case of projects where a complete functional program is a collection of many programs.

The selection of either form of encapsulation in constructing a system which treats programs as objects must be based upon the technical parameters of the development process. Such parameters can be the functional scope of the system or the practical limitations (performance, complexity etc., physical space) that apply.

2) Object Attributes

The encapsulation that has just being ascribed to programs can be additionally justified by identifying the object attributes through which encapsulation is expressed according to the object theory. Therefore, a private memory that holds its state is needed as well as the methods that operate on that state. Additionally, messages which constitute the communication link of the object with the outer world need to be detected.

a) Data Members

The storage area of the object 'programs' can be attributed to the physical files stored in the physical media. The aforementioned distinction between the narrow and broad form encapsulation finds its practical confirmation when considering the files belonging to the development environment of a program. If the narrow approach is followed, the only data member of an object 'program' should be its source code file. Therefore, the only operations taking place under this development environment are the ones concerning the source file. However, including the entirety of files connected to the program at any stage of its development (***.OBJ, .EXE***) as the broad form suggests, means that any operation involved in the construction of the functional program becomes part of its object behaviour.

b) Constructor, Destructor and Member functions

The clear distinction between the forms of encapsulation has a more immediate effect on the definition of basic object attributes such as the constructor, destructor and member functions of the object 'program'. For both cases of encapsulation though a minimal set of operations performed during the creation of the source code file have to be included in the constructor. As a result, the combination of New/Edit/Save operations (the Edit operation

denoting all Cut, Copy, Paste and similar edit actions) performed within an editor can be thought of as the simplest constructor of the object 'program'. However, following the approach of the broad model, the construction of the object cannot be considered complete unless the compilation and linking are completed first.

That last assumption carries significant weight when deciding which are going to be treated as member functions of the object. If compilation, and linking are to be included in the constructor, then the most obvious method of a program is its execution. Otherwise, all three Compile, Link, and Run stages have to be treated as methods of the object. A direct correspondence between the states of a program and how they could be interpreted under the object model is presented with the use of C++-like pseudo-code in Figure 1.

```
Command                 Object Data Member
-----------------------------------
Compile prog.c          prog.Compile()
Run prog.c              prog.Run()
Edit (new) prog.c       prog() Constructor
Delete prog.c           ~prog()Destructor
```

Figure 1 Object Representation of Program Manipulation Commands

The same principles apply to the destructor of the object 'program' as well. Whilst the deletion of the ***.SRC*** file is enough to constitute a destructor for under the narrow model, a broad approach would require the deletion of all related files (***.OBJ, .EXE***). An interesting case emerges if the deletion of the .SRC file only is examined. Although it may seem to create an inconsistency to the object model at first, this case can be thought of as the modification of the data member .SRC, into an empty file.

c) Messages

In software engineering there has been a significant research on topics covering the whole software life cycle. Many methodologies have been produced in order to improve the performance and correctness of programs. The messages that can be sent to an object 'program' could therefore express all these major software engineering aspects. A good example to demonstrate this concept is the CASE tools available for the design and testing of software. These tools provide the developer all types of information about the program under development. In a proper object environment the operations which produce this information would be attached to the object as methods. All messages to the object would cause these methods to produce the appropriate response.

The feedback a user could get from these methods is related to the physical and textual properties of a program and covers a whole range of software engineering techniques for testing and evaluation. Thus, it would be

more useful to group that information into types of responses acquired from the object when messages are sent.

- Statistical: Messages can be conveying information about the natural size of textual constructs within the source code such as lines, blocks of code, and functions. Additional statistical information like the number of lines, or the number of functions contained in a code module can be provided.

- Defect testing: A whole range of methods for performing functional and structural testing can be included herein. These methods can be implementing testing techniques such as equivalence partitioning (Somerville[25]) and cyclomatic complexity (McCabe[20]).

- Quality Metrics: The structural quality of a program could be easily assessed by the user if quality metrics are include in the messaging system of the programs. These could include fan-in/fan-out metrics both design (Constantine & Yourdon[11]) and informational (Henry & Kafura[17]). Other existing techniques such as Halstead's metrics (Halstead[16]) can be included as well.

The embedding of major software techniques as functional parts of the object environment of computer programs can lead to maximum usability for both users and software developers. The theoretical substrate required is existent within the object model. A study on the pure operational properties of such a system need only be proposed.

3) Inheritance

The concept of inheritance initially emerged as a feature of Simula-67 (Dahl & Nygaard[13]) which is widely considered to be the first language incorporating object features in its programming scheme. The form of inheritance supported therein was a single IS-A relation. That is, a derived class could only have one parent. This modelling of hereditary behaviour proved to be less flexible and certainly less representing of real-world conditions where objects can be usually traced as heirs of a lot more than one classes. Therefore, multiple inheritance was introduced (Cardelli[8]). As a result, many object-oriented systems such as Eiffel (Meyer[21]) have acknowledged its advantages and have incorporated it as an essential part of their object model.

Inheritance is of particular interest to this model. Although not apparent at first sight, the notion of inheritance appears to "natively" exist in programs. However, its manifestations in the type of programs this work focuses on, can only be highlighted if these programs are investigated as objects. For that reason the relation of the object "program" to the other objects of its kind has to be taken into account. One of the primary messages or member functions of that object is the linking of libraries and external programs in order for the program to be able to use their functions. If inheritance is thought of as the incorporation into an object of behaviour developed elsewhere, we come to the conclusion that all library declarations (such as *#include* in C) and their subsequent linking

process act as the equivalents of an IS-A declaration of pure object semantics; this constitutes a clear manifestation of inheritance. Furthermore, it is a multiple inheritance which is attributed to programs since they usually include calls to more than one library and external modules.

The discussion so far has been on the linking of code of the same programming language (base class). More advanced examples of hereditary behaviour can be traced in existing programs that allow mixed code, that is code from various programming languages (different base classes) to coexist in the same source file (e.g. .ASM declarations or "exec query" declarations in C).

A simpler form of inheritance can be also traced in programs. The object theory suggests that an integral part of inheritance is the derivation of objects which carry along some (if partial/selective inheritance occurs) or all of the data members of its parent. In that case, if the aforementioned discussion on data members is taken into account, the simple Copy/Paste operation where pieces of code are duplicated in more than one programs (objects) constitutes an act of inheritance. Should one want to be purist, claiming that inheritance should be full and not partial, he should consider program versioning where a newer version of a program mainly constitutes of an older version plus some new code added.

4) Object Composition / Decomposition of Programs

Another indispensable concept of the object model that many like Hughes[18] list as an equally important tool of object analysis is called the object composition/ decomposition. The term refers to what is known as an IS-PART-OF relation under which an object can be depicted as a substructure consisting of a lower level objects, them having their own individual data members, member functions, etc. An object that can be represented using such an IS-PART-OF hierarchy of other objects is referred to as a composite or a complex object (Goodman[15] and Banerjee *et. al.*[4]). That complex object structure can be identified on the textual features of the object 'program'. Objects like 'token' (lexical unit) and 'grammar rule' (syntactical unit) compose (are-parts-of) a program in a similar way that object 'wheel' and 'engine' are parts of the object 'car'.

A revised analysis on the structured piece of text called source code can produce interesting results if the last observation is taken into account. Starting from the very fundamental units that a program consists of (lexical units) and building the program's object hierarchy all the way up the scale (syntactical, semantic level), a whole structure of complex objects appears to be arising, from tokens to grammar rules to blocks of code, to functions and finally to a complete program. A schematic representation of that hierarchical structure can be seen in figure 2.

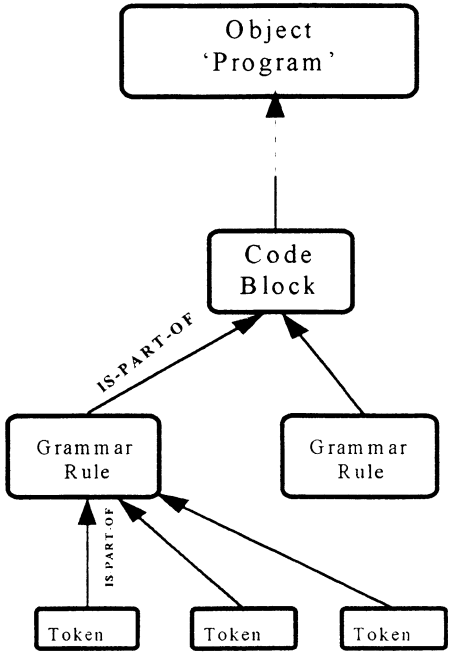


Figure 2 Object Structure of Programs

The result of such an object composition/decomposition process, is that every textual attribute included in a program as dictated by the principles of modular and structured programming, it is allocated on that well structured object hierarchy; the description of a program as an object becomes considerably clearer. Moreover, if that line of thinking is advanced, every analysis or processing of the source code at any level of that hierarchy (as described in the messages section) becomes a mere sending of messages to the object constructs of the appropriate level (e.g. semantic analysis, parsing, statistical, qualitative information etc.).

Conclusion

This paper has focused on presenting the current involvement of the object-oriented model into the theoretical substructure and research of software engineering. More particularly, this work has examined a particular aspect of that involvement by applying the object model to programs. The theoretical approach presented herein was directed towards adding more data abstraction to the object-oriented design theory by applying the principles of the object model to the physical attributes of programs. Moreover, it hopes to have helped object orientation get integrated further into software engineering by initiating a discussion within the research community over the matter of treating programs as objects physically as well as logically. The fact that this theoretical discussion is using a mainstream object-oriented model as its base should not prevent

alternative studies on the nature and attributes of computer programs from emerging into the research literature. As an example, the adoption of classless methodologies such as these offered by the actors theory (Agha[1]), could offer interesting additional interpretations.

References

1. Agha G., (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, USA.

2. Agrawal R. and Gehani N., (1989). ODE: The Language and the Data Model, *Proc. ACM SIGMOD Conf. on the Management of Data*, Portland, OR, USA.

3. Bancilhon F., Delobel C. and Kanellakis P., (1992). *Building an Object-Oriented Database System: The Story of O_2*, , Morgan Kaufmann Publishers, San Mateo, CA, USA.

4. Banerjee J., Chou H.-T., Garza J., Kim W., Woelk D., Ballou N. and Kim H., (1987). Data Model Issues for Object-Oriented Applications, *ACM Transactions on Office Information Systems*, **5**, 1

5. Bobrow D.G., et. al., (1988). Common Lisp Object System Specification, *X3J13 Document 88-002R.*

6. Booch G., Object-Oriented Development, (1986). *IEEE Transactions on Software Engineering*, **SE-12**, 2.

7. Brown A., (1991). *Object-Oriented Databases, Applications in Software Engineering*. McGraw-Hill Book Company, Maidenhead, UK.

8. Cardelli A., (1984). A semantics of Multiple Inheritance, Semantics of Data Types, *Lecture Notes in Computer Science*

9. Coad P. and Yourdon E., (1990). *Object Oriented Analysis*, Prentice Hall International, London, UK.

10. Codd E.F., (1970). A Relational Model of Data for Large Shared Data Banks, *Communications of the ACM*, **13**, 377-387.

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

12. Cox B.J., (1986). *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley Publishing, Reading, MA, USA.

13. Dahl O. and Nygaard K., (1966). Simula, An Algol-based Simulation Language. *Communications of the ACM*, **9**, 671-678

14. Goldberg A. and Robson D., (1983). *Smalltalk-80, The Language and its Implementation*, Addison-Wesley, Reading, MA, USA.

15. Goodman N., (1991). The Object Data Model in Action, *Info DB*. Fall '91, **6** , 2

Transactions on Information and Communications Technologies vol 14, © 1997 WIT Press, www.witpress.com, ISSN 1743-3517

16. Halstead M.H., (1977). *Elements of Software Science,* North Holland, Amsterdam.

17. Henry S. and Kafura D., (1981). Software structured metrics based on information flow. *IEEE Trans. Software Engineering*, 7, 5, 510-518

18. Hughes J. G., (1991). *Object-Oriented Databases*, Prentice Hall International, UK.

19. Maier D., (1989). Why isn't there an Object-Oriented Data Model, *Proceedings of Information Processing (IFIP-89),* San Francisco, USA.

20. McCabe T.J., (1976). A complexity measure, *IEEE Trans. Software Engineering,* **2,** 4.

21. Meyer B., (1988). *Object-Oriented Software Construction*, Prentice Hall International, London, UK.

22. Schaffert C., Cooper T., Bullis B., Killian M. and Wilpolt C., (1986). An Introduction to Trellis/Owl., *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86),* Portland, OR, USA.

23. Schmucker K.J., (1986). *Object-Oriented Programming for the Macintosh*, Hayden Book Company

24. Seidewitz E. and Stark M., (1986). General Object-Oriented Software Development, *Report No. SEL-86-002*, Software Engineering Laboratory, NASA Goddard Space Flight Centre., USA

25. Somerville I., (1992). *Software Engineering*, Addison-Wesley.

26. Stroustrup B., (1986). *The C++ Programming Language*, Addison-Wesley, Reading, MA., USA.

27. Wirth N., (1977). Modula: A language for modular multiprogramming. *Software - Practice and Experience*, 7, 3-35.