Is Object-Oriented Programming Structured Programming ?

Bernd Müller
Fachbereich Informatik
Universität Oldenburg
D-2900 Oldenburg
Bernd.Mueller@Informatik.Uni-Oldenburg.DE

Abstract

Object-oriented programming is one of today's buzzwords. On the one hand it is a programming paradigm of its own right. On the other hand it is a set of software engineering tools to build more reliable and reusable systems. One other kind of programming style, which has already shown its power on this field is structured programming.

In this paper we look at the relationship between structured programming and object-oriented programming. We give a definition of structured programming and check the object-oriented features encapsulation, inheritance and messages for compatibility. Two problems arise. The first addresses interfaces and inheritance, the second polymorphism and dynamic binding.

We do not end with an answer like *yes* or *no*, but try to give a basis for further discussions. This may help to classify object-oriented programming into a wide range of tools for computer programming and computer science. It may also help to abandon the hypothesis that object-oriented programming is the solution to all the problems.

1 Introduction

Much work has been done in the field of comparing Structured Programming (SP) with Object-Oriented Programming (OOP), or working out the relationship between both. For example, Grogono [13] promises that a carefully designed Object-Oriented Programming Language (OOPL) corresponds to the requirements of SP. Blaschek [4] shows how objects can be implemented in Modula-2, the programming language showpiece of SP. Edelson [12] compares Modula-2, C++, and Smalltalk-80 in terms of engineering large software systems effectively, which is the target of both, SP and OOP. Meyer [18] tries to motivate that Object-Oriented Design (OOD) and OOP, especially the language Eiffel [19] meets the demands of SP and, moreover, goes beyond it.

In this paper we also want to focus on the relationship between SP and OOP, but more detailed and comprehensive than done by that previous work. The above papers did not do that, because their primary purpose was completely different. We think the relationship between SP and OOP is not obvious enough and it is worth to look about it.

Both, SP and OOP are popular today although SP has a longer history. The current popularity of OOP and the connection to SP was pointed out by Rentsch:

> "What is object oriented programming? My guess is that object oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is." [22]

Beside the similarity in popularity SP and OOP are different paradigms[1], but, with respect to some metric or (partial) ordering, is it possible to order them or are they incomparable? We think that SP is a more general term than OOP. Thus, we have to show that every object-oriented program is a structured program. Or, to avoid some paranoiac programs, that every concept of OOP meets the demands of SP.

The notion of OOP is used by everyone, as mentioned in the above quotation, but used differently. The notion of SP is also widely used but has a somewhat more exact definition. Nevertheless, we have to define the notions of SP and OOP. In OOP we concentrate on inheritance, encapsulation and the message passing mechanism. We exemplify the relationship of these central object-oriented features to SP.

The rest of this paper is organized as follows. Section 2 introduces the notion of SP, section 3 the notion of OOP. Section 4 examines more closely the relationship between both paradigms. The paper concludes with section 5 where we summarize our results.

2 Structured Programming

SP is a very old[2] method to construct computer programs. Founded by [10] many people [29, 30, 15] have acknowledge this technique and SP has become the state of the art in programming. Nevertheless, we were not able to find[3] a complete but short enumeration of SP feature (a definition). We try to do this ourselves, without claiming to be complete. But, to apologize for incompleteness, we think it is enough for our comparison.

The method or paradigm of SP consists (at least) of the following (taken from [11, 10, 30, 15]):

SP1 The problem is decomposed into independent subproblems (see SP2). This decomposition recurses until the subproblems are atomic and can be solved.

SP2 The subproblems are connected by interfaces, and can therefore be solved independently.

SP3 The structure of (sub)problem(s) and (sub)solution(s) are (almost) identical.

SP4 There is a strong correspondence between the static description of the algorithm and its dynamic behavior (run-time).

SP5 It is 'easy' (the structures make it easy) to convince one about the correctness of the solution.

SP6 The only allowed control structures are selection, sequence, and repetition.

SP7 There is some block structure (visibility, scope of names)

This definition should be used as is, i. e. some interpretation should be possible. If we do not allow this we can stop after SP1. SP1 is a point of design and mainly constitutes top-down design. On the other hand OOP favors bottom-up design. Sometimes this is not stated explicitly, but it is agreed upon that OOD and OOP is done by composing objects or classes together, which is certainly a bottom-up technique ([31] and [20] give an overview of OOD techniques, [17] and [5] are two well established methods). We want to compare SP and OOP on the programming language level, not on the design level. Therefore, SP1 is not so much important. But the problem remains. For example to satisfy SP5, i. e. real correctness, formal verification techniques are necessary, which do not exist for SP and OOP, at least for complex enough (real life) applications. Therefore keep in mind that the list should act as a direction, not as a fixed mark.

[1] We think this is obvious and do not proof it. The above mentioned authors [13, 4, 12, 18] have the same opinion. Nevertheless, they will may be not agree with the following statements.

[2] In terms of computer science.

[3] Maybe due to our own inability.

3 Object-Oriented Programming

OOP is not a new programming paradigm. Simula in the late 60th and Smalltalk in the 70th started the development of a new programming paradigm, but did not succeed (in terms of wide spread use). The software crisis and a new interpretation of OOP, namely not as a programming paradigm, but as a set of software engineering concepts to produce better programs, has pushed OOP to the top. Cox [9], Meyer [17] and many others argue that object-oriented concepts will produce programs which are superior than conventional programs in terms of correctness, robustness, extendibility, reusability, and compatibility.

It is out of the scope of this paper to give a comprehensive introduction into OOP. A very extensive discussion of central object-oriented features can be found in [28]. Almost every textbook on OOP can be used for introductory purposes. Especially to mention is [6], which exemplifies the concepts into four different OOPLs, making clear the distinction between concepts and different implementation of this concepts.

For the discussion in this paper we will concentrate on encapsulation and inheritance[4], which Cox identifies as the central points, and in addition the message passing concept.

3.1 Encapsulation

In OOP each data-item is joined exclusively with the procedures manipulating this data-item. The attached procedures of a data-item are called *methods*, all methods together are called *interface*, the data-item together with its methods is called *object*. Because it is not allowed for unattached procedures to manipulate an object the data is encapsulated. Therefore the effects of changing data and/or procedures is always restricted and easy to localize.

Snyder requires for real encapsulation that

> "...one characteristic of an object-oriented language is whether it permits a designer to define a class such that its instance variables can be renamed without affecting clients." [23]

3.2 Inheritance

Inheritance is a technique which enables the reuse of behavior[5] of already defined classes (a class is a template from which objects can be created) in the definition of a new class. Inheritance plays such an important role in OOP because

> "Inheritance can express relations among behaviors such as classification, specialization, generalization, approximation, and evolution." [28]

Beside this conceptual benefits inheritance helps to avoid the duplication of code and the need for coding from scratch. It is therefore a tool to engineer software in a better way, too.

3.3 Message passing

A message is sent to an object and represents a request to perform some action. It is in the responsibility of the receiver how to react. Thus, it is possible that different objects react differently after receiving the same message. While this is a high-level concept which influences design (called responsibility-driven-design [31]) on the lower level of language implementation there is one more important difference to conventional procedure calls. Procedure calls are bound early (at compile or link time) to some code fragments, while messages trigger the execution of some code lately (which code to execute is determined at run-time) with respect to the above scheme of responsibility.

[4] While inheritance is really new, encapsulation is also possible in many other languages but mostly not enforced.

[5] And therefore of code.