

What Object-Oriented Programming Was Supposed to Be

Two Grumpy Old Guys' Take on Object-Oriented Programming

Ole Lehrmann Madsen
Department of Computer Science
Aarhus University
Aarhus Denmark
ole.l.madsen@cs.au.dk

Birger Møller-Pedersen
Department of Informatics
University of Oslo
Oslo Norway
birger@ifi.uio.no

Abstract

Object-oriented programming has been around for more than 50 years and is now the most dominant style of programming. In recent years there has been an increasing criticism of object-oriented programming. Some people argue that the mainstream object-oriented languages do not capture the intentions of object-orientation as intended by its founding fathers. There are indeed issues with object-orientation as practiced by mainstream. In this essay, we identify a number of issues that we think are problematic. We argue that the primary reason for these issues is that reuse is considered the main advantage of object-orientation at the expense of modeling. We argue that modeling should be the main focus, that programming is modeling, and we describe a number of principles to follow when practicing object-oriented modeling.

CCS Concepts: • **Software and its engineering** → Software notations and tools → General programming languages → Language types → Object oriented languages.

Keywords

Object-oriented programming, Modeling, Reuse, Programming

ACM Reference Format:

Ole Lehrmann Madsen and Birger Møller-Pedersen. 2022. What Object-Oriented Programming Was Supposed to Be: Two Grumpy Old Guy's Take on Object-Oriented Programming. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '22)*, December 8–10, 2022, Auckland, New Zealand. ACM, New York, NY, USA, 20 pages. <https://doi.org/10.1145/3563835.3568735>

“Why do we keep writing that programming is modeling”,
“Guess we’ll never know”

freely after the Muppet Show balcony grumpy guys

Waldorf: “Why do we always come here?”
Statler: “I guess we’ll never know”

1 Introduction

Object-oriented programming has been around for more than 50 years and is now the dominant style of programming. Most programming languages support object-orientation, and a substantial number of applications have been made in these languages through the years.

In recent years there has been an increasing criticism of object-oriented programming – some people advocate functional programming as an alternative and some people argue that the dominant object-oriented languages do not capture the intention of object-orientation as intended by its founding fathers, including Dahl & Nygaard [1] and Alan Kay [2] – the creators of SIMULA and Smalltalk, respectively.

Most of the criticism appears in blogs, and most of the posts do not seem to be based on solid knowledge, and they often contain a number of factual errors. In addition, object-orientation is also blamed for things, e.g. spaghetti code, that is obviously not grounded in object-orientation. The starting point for this essay was one of our highly esteemed colleagues who pointed to a blog by Ilya

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Onward! '22, December 8–10, 2022, Auckland, New Zealand
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9909-8/22/12
<https://doi.org/10.1145/3563835.3568735>

Suzdalnitski: “Why is OOP Such a Waste”¹ and he suggested that we commented on it.

We do find the posts by Suzdalnitski and similar ones to be flawed and full of mistakes and we do not find it useful to comment on these kinds of post. We do, however, hear critique of object-oriented programming from colleagues, programmers and others and there are of course valid points of criticism that gives rise to reflection. In this paper we discuss what we consider to be critical issues of object-oriented programming and argue that modeling should be the main focus of object-oriented programming.

It all started with SIMULA as a language, which was intended for modeling as well as programming. It introduced objects, classes, subclasses, and virtual methods. Classes were meant for the representations of domain concepts, and subclasses as representations of specialized concepts. A side effect was that subclasses inherited the code of the superclass.

The modeling approach of SIMULA was not dominant in most of the programming languages that followed. Instead reuse of classes was considered the main advantage of object-oriented programming. The mainstream interpretation, represented by the use of today’s dominant languages, backed by textbooks and various papers including research papers on object-oriented programming, has focus on the inheritance of code. Some languages consider classes as object factories, and subclass/inheritance as implementation inheritance: reuse of code by defining a subclass and inherit the code from the superclass and redefine what needs to be redefined. This may lead to class libraries that are difficult to understand since there is no conceptual relation between a class and its possible subclasses, in the sense that a subclass does not represent a specialization of the superclass.

Some of the blog posts claim that the current mainstream object-oriented languages do not correspond to the intentions of object-orientation as formulated by Alan Kay as part of the development of Smalltalk – according to Kay, the central idea of object-oriented programming is that a program execution is a set of objects sending messages to each other, i.e. a message passing interpretation.

To complete the picture, there is also an interpretation that does without classes and subclasses, i.e. only have objects (and delegation). This is a prototype-based interpretation, with Self [3] as the most striking language.

Object-oriented modeling methods, e.g. [4-6], that lead to UML [7] obviously followed the original modeling interpretation. However, as stated in [8, 9], the community on object-orientation is split into a programming community that ‘make programs’, but do not consider modeling, and a modeling community focusing on models more or less independent of programs. We learn from

programmers that they do not model as also pointed out by Bran Selic [10], and prominent researchers like W. Cook in Peak Objects, [11] do not seem to buy into the modeling interpretation.

In this essay we will argue that most of the criticisms of object-orientation may be due to the mainstream interpretation of object-orientation. In this interpretation, a class is just considered a factory for generating objects and inheritance is considered a mechanism for reuse – Section 3.

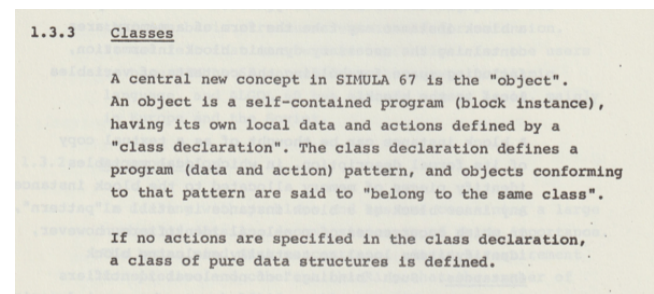
We will argue that a program always describes a model of the application domain even if the programmers claim they do not model. But being explicit about that programming is modeling will improve the readability of software, we will argue that most of the problems with the mainstream interpretation do not exist in the modeling interpretation of object-oriented programming.

We briefly summarize the principles behind a modeling interpretation of object-oriented programming and list a number of principles to follow when considering programming as modeling. We think programming is modeling and you do not need to use UML to carry out modeling – Section 4.

In Section 5 we will show that both the message-passing interpretation and elements of the prototype-based interpretation may gracefully be unified with the modeling interpretation. Instead of regarding functional programming as an alternative to object-oriented programming, we think that functions, values, and value types are important for objects: Objects have states in terms of values, and some properties of objects are readily represented by functions producing values. In addition, there is no reason that functions cannot be a mechanism along with classes.

2 Background

In the beginning there were objects and classes of objects.



SIMULA 67 Common Base Language, 1968 [1]

Despite its name, SIMULA was a general-purpose programming language, with a framework for simulation. Moreover, it was not just a programming language, but it was also intended for modeling (at that time called

¹ <https://suzdalnitski.medium.com/oop-design-patterns-bd2c4fb3014c>

‘description’), see the preface to the language definition for SIMULA I [12]:

The two main objects of the SIMULA language are:

- *To provide a language for a precise and standardised description of a wide class of phenomena, belonging to what we may call “discrete event systems”.*
- *To provide a programming language for an easy generation of simulation programs for “discrete event systems”.*

The modeling approach was strengthened by the notion of subclasses. These were inspired by discussions on record handling and data abstraction with Tony Hoare [13], and it was obvious that the main focus was on modeling and not on code reuse:

1. 1 Objects and Records.

... When we wish to solve a problem on a computer, we often need to construct within the computer - a model of that aspect of the real or conceptual world to which the solution of the problem will be applied. In such a model, each object of interest must be represented by some computer quantity which is accessible to the program, and which can be manipulated by it.

...

SIMULA took the notion of records a step further by objects having behavior, either in terms of procedures or behavior of the object itself.

SIMULA was developed by Kristen Nygaard and Ole-Johan Dahl at the Norwegian Computing Center in the sixties. The first version, SIMULA I, was a simulation language whereas its successor, SIMULA 67, was a general-purpose programming language.

The appearance of Smalltalk-80 [2] implied a major boost in the popularity of object-orientation. Smalltalk took the concepts of SIMULA one step further in the sense that all data-items are objects, and all methods are virtual. Alan Kay is credited for the term ‘object-oriented’, but according to Alan Kay the main interpretation of object-orientation represented by the early Smalltalk was message passing between objects and not classes and subclasses of objects; when introduced in later versions of Smalltalk the main rationale was code reuse, not modeling. We are not aware of versions of Smalltalk without classes, and classes were central already in Smalltalk-72 [14]. However, Smalltalk-72 did not have subclasses.

C++ in the eighties founded the basis for the popularity of object-oriented programming in the software industry. There were other early languages like Eiffel [15], and Objective-C [16] that contributed to the popularity and since the nineties a myriad of new languages like Java [17], C# [18], Python [19], and JavaScript [20] have been important contributions. All of this laid the ground for the mainstream interpretation, with focus on reuse and implementation inheritance. This was specifically the reason that some languages introduced multiple inheritance, it was not for multiple classifications of a single object. Multiple inheritance involved a number of

rules to resolve name conflicts and diamond inheritance, so other languages instead introduced that the class could have several interfaces with just method signatures. While interfaces have found the way into mainstream languages, other ideas like aspects, mixins, and traits have not.

As mentioned, SIMULA was intended for both modeling and programming, and this was also the case for the Beta language [21]. A major contribution from Beta was the formulation of a conceptual framework for object-oriented modeling – The conceptual framework of Beta is described in chapters 2 and 18 of [19] and is in the following referred to as *The Conceptual Framework*. The approach behind SIMULA and Beta was coined the Scandinavian approach to object-oriented programming.

Modeling was taken up in the eighties, by people like Booch [4], Coad & Yourdon [6], and Rumbaugh [5], who introduced the idea of Object-oriented Analysis, Design and Modeling as a replacement for Structured Analysis and Structured Design (SA/SD). This work eventually led to the design of UML.

The unfortunate implication of this was that the object-oriented programming and object-oriented modeling diverged into two more or less disjoint communities. Programmers don’t model and modelers are detached from programming. We have discussed this situation in [8, 9, 22], and [23, 24]. We have only mentioned some of the early developments of object-oriented programming, but most new languages follow the mainstream interpretation.

3 The Mainstream Interpretation

In this section, we discuss what we consider to be problematic issues in the mainstream interpretation of object-oriented programming. These issues are problems to a smaller or larger extent depending on the actual programming language being used. Although mainstream languages like C++, Java, and C# are all based on objects, classes, subclasses and virtual methods, there are important differences, and these will be noted. Each description of an issue is summarized in a short, numbered sentence. In Section 4 we discuss how the modeling interpretation may provide an alternative to most of the issues.

In the descriptions below, we list examples of languages and authors promoting code reuse and these examples are more or less randomly chosen among many examples and the purpose here is not to bash the authors. On the contrary, we highly respect these people and the contributions they have made to programming languages. And once again there are no right or wrong answers to the issues we discuss – it is to a large extent a matter of one’s perspective and choices with regard to what is considered important.

3.1 Reuse versus Modeling

In the mainstream interpretation of object-oriented programming, reuse of code is considered the main

advantage. When object-oriented programming started to become mainstream, most people advocating and practicing object-oriented programming claimed that the main benefit of object-orientation was the great possibilities of reusing code. Object-oriented languages indeed provided new language mechanisms for reuse by means of the class-, subclass- and virtual mechanisms. The downside of this is that reuse became *the* main benefit of object-oriented programming compared to modeling.

As mentioned, a major goal of SIMULA was to develop a language that could be used for modeling as well as programming. In the mainstream interpretation of object-oriented programming, modeling does not seem to play an equally important role.

We often hear that ‘Real programmers do not model’ because modeling is assumed to be the same as using UML. Many programmers do not like using tools to draw boxes and bobbles, just as UML diagrams are hard to keep up-to-date with the program text.

This ignorance of modeling is backed by most textbooks on object-oriented programming, research papers on object-oriented programming, etc. Their primary focus is on a technical description of object-oriented language mechanisms.

Smalltalk is built on a model of objects communicating with messages and a class describes the implementation of a set objects. The word model does not appear in the index, and the description of Smalltalk is purely technical. This is reflected in the Smalltalk library where in our opinion inheritance seems to have been used as implementation inheritance.

W. Cook emphasizes a technical view on object-oriented programming in his paper on ‘Peak Objects’ from the Special 20th Anniversary Session at ECOOP 2006 [11]. He states: *“The philosophical viewpoint that ‘objects model the real world’ has never appealed to me.”* And in his Onward 2009 essay: ‘On Understanding Data Abstraction, Revisited’ [25] he presents a strict technical and to some extent mathematical definition of object-oriented programming.

Andrew Black in ‘Object-oriented programming: Some history, and challenges for the next fifty years’ [26] states: *“Inheritance has stood the test of time as a useful feature, and I now believe that some code-reuse mechanism like inheritance or delegation is a very important part of any object-oriented programming system.”*

Bono et al in [27] is a recent example of a paper advocating reuse – they states that *“The ultimate goal of object-oriented programming should be code reuse.”*

Note that if you read the literature on SIMULA, modeling is not very explicit although it is underlying the goal of SIMULA. On the other hand, the SIMULA literature does not mention reuse of code, which does not have seem to have been an issue, but reuse is of course also possible with SIMULA.

As mentioned in the background section, SIMULA is to a large extent based on Hoare’s Record Handling and here modeling is an explicit motivation.

1. By focusing on reuse, the benefit of modeling is lost.

3.2 Implementation Inheritance

In the mainstream interpretation of object-oriented programming, a class is primarily considered a template/factory for creating objects and inheritance (subclassing) is considered *implementation inheritance* in the sense that the primary use of inheritance/subclassing is to *reuse* an existing class by making subclass and redefine the inherited methods so that they fit with this actual use. The reason for this interpretation is probably that reuse of code was considered the main benefit of object-oriented languages. There is of course nothing wrong with reuse and implementation inheritance, since a subclass does reuse the code of the superclass just as a class is a template for generating objects, but focusing on reuse and implementation inheritance may easily lead to large unstructured class libraries.

As an example of implementation inheritance mentioned by Barnes [28] is class `Stack` from `java.util`. Class `Stack` is defined as a subclass of class `Vector`, which means that `Stack` inherits all the methods of `Vector`. A `Stack` is obviously not a `Vector`.

Eiffel is an example of a language where *“inheritance is a key reusability technique”* seen from a module viewpoint, (10.2.1, page 228 in [29]). In Eiffel a class is considered a module. However, Eiffel also considers inheritance from a type perspective and mentions that inheritance represents the is-a relation (10.2.2). So here inheritance is used both for implementation inheritance and for defining type hierarchies.

The highly cited paper by Wegner and Zdonik – ‘Inheritance as an Incremental Modification Mechanism or What Like Is and Isn’t Like’ [30], is an influential representative of a primarily technical view on object-oriented programming. As the title states, they considered inheritance as an incremental modification mechanism that facilitates reuse. They define various relations between a class and a subclass: behavioral compatibility, signature compatibility, name compatibility, and cancellation.

Behavioral compatibility obviously belongs to a modeling interpretation of inheritance, whereas the other relations are purely technical and belong to the implementation interpretation of inheritance. We return to the paper by Wegner & Zdonik below.

Although they include behavioral compatibility, they do not consider modeling as the primary use of object-oriented programming (in this case inheritance), but they do touch upon modeling issues at some places.

If one insists on implementation inheritance then a mechanism like private inheritance in C++ should be used. With private inheritance, a derived class may not be assigned to a reference variable typed with the superclass and substitutability is not a problem. Public inheritance may then be used for representing classification hierarchies, but they are essentially two different mechanisms.

With implementation inheritance, there is no conceptual relation between a subclass and its superclass in the sense of the subclass being a specialization of the superclass.

Reuse of code is of course useful, but unrestricted use of implementation inheritance leads to the following issues:

2. Classes in a class hierarchy have *no conceptual* relations other than a reuse relation.
3. Class hierarchies may be *difficult to maintain* and navigate in.

3.3 Substitutability

In most object-oriented languages, a reference variable *R* of type *T* may be assigned a reference to an object of any subclass of *T*. This notion of substitutability was a central idea of the subclass mechanism of SIMULA, and it was assumed that a subclass should always be compatible with its superclass.

The problem with substitutability in SIMULA is that the language and compiler is not able to check if a subclass is compatible with its superclass.

This is evident if implementation inheritance is considered the main purpose of subclassing, and if a given subclass of *T* is not compatible with *T*, then this may lead to failures during execution.

4. Using implementation inheritance, a subclass may not be compatible with its superclass.

3.4 Multiple Inheritance

Some languages have multiple inheritance so that a class can inherit code from more than one superclass. Most proposals for multiple inheritance are motivated by code reuse and the semantics of multiple inheritance is often technically complicated. The order of the superclasses is often significant, the handling of name collision is complicated, and the so-called *diamond problem* may give rise to ambiguities.

In the eighties a number of object-oriented extensions of Lisp was made including LOOPS [31], Flavors [32] and CLOS [33]. They are examples of languages with complicated definitions of multiple inheritance. Eiffel is also an example of a language supporting multiple inheritance.

C++ has both overlapping (virtual) multiple inheritance and non-overlapping multiple inheritance. The ordering of

superclasses is relevant only to determine the order of default initialization by constructors and cleanup by destructors. Name collision has to be resolved by prefixing the name with the name of the superclass.

Python also has multiple inheritance and here the ordering of the superclasses is used to determine possible name collisions.

Eiffel has means to say that inheritance does not imply subtyping [34]. If an inheritance clause for a class starts with `inherit {NONE}`, the class does not define a subtype of the type(s) of the super classes. The rationale for this is said to be that 'Sometimes inheritance is for reuse only and is not intended for polymorphism.' Inheritance normally allows polymorphism (and hence subtyping): if *B* inherits from *A*, then $x := y$ is legal for *x* of type *A* and *y* of type *B*. This corresponds to private inheritance in C++

A representative example where multiple inheritance is used for a combination of classification and reuse is the example [35] where the class `Linked_Queue` is defined with two superclasses: `Queue` because a `Linked_Queue` is a `Queue` and `Linked_List` because `Linked_List` is used to implement the `Linked_Queue`.

Smalltalk, Java, and C# do not have multiple inheritance; neither do SIMULA and Beta. Java has interface types with multiple inheritance, which to a large extent give many of the benefits of multiple inheritance but with less complications.

Mixin classes and traits have been proposed as alternatives to multiple inheritance. Mixin classes support that classes can be defined by mixing a number of mixin classes, see e.g. [36, 37].

Traits [38] are defined simply as collections of methods that may be included in any class independently of its position in a class hierarchy.

We should say that these are the original definitions. As of today, traits of e.g. Scala [39] can have both data items and methods. In Scala it is possible to restrict in which (sub)classes it is possible to include a trait, just like some mixins. C# interfaces have been extended to have data items (members) and (default) implementations of methods, so these work as traits.

5. Multiple inheritance, mixins, and traits add complexity and ambiguity compared to single inheritance.

3.5 All Methods Are Virtual

SIMULA distinguished between virtual and non-virtual methods, and this is also the case for C++ and C#. In some languages, like Smalltalk, and Java all methods of a class are by default virtual. In Java a method can be declared as *final*, which means that it cannot be redefined. C# has a similar mechanism called *sealed*, but only redefinitions can be

sealed, the first definition of a method in a class cannot be sealed. Smalltalk cannot finalize redefinitions. Beta also distinguishes between virtual and non-virtual methods, and a binding of a virtual may be marked as final.

From a reuse point of view, it of course provides maximum flexibility if all methods are virtual, but the implication is that it is not possible to express that some part of class behavior is the same for any subclass. When defining a subclass, it is not obvious which methods of the superclass are supposed to be redefined, and which should not.

Final and to some extent sealed may be used to declare methods as non-virtual, but since it is not default, most methods in practice are virtual.

Virtual procedures in SIMULA, were introduced as a replacement for name parameters and formal procedures in ALGOL 60 [40]. If all methods are virtual, they are in fact all potential parameters of the class to be defined in subclasses. See also the discussion of 'ex post facto parameterization' as proposed by Black et al. below.

6. Methods being virtual by default makes it unclear which methods to redefine in a subclass, and which should not.

3.6 Methods and Functions

SIMULA had procedures both as part of objects/classes (today's methods) and as first-class citizens. With Smalltalk this was kind of forgotten in that everything should be objects. In the object-oriented languages to follow it is also the case that the emphasis is on classes with methods, with the implication that the representation of general behavior phenomena (read: pure methods and functions that really do not belong to an object/class) has to be represented by so-called static methods of a class. The distinction between static and non-static methods comes with a set of rules that are difficult for newcomers to object-oriented programming.

Note however, that this does not apply to all languages; notable exceptions are C++, Scala, and Python, and it is of course a minor issue.

7. It is inconvenient to have to define functions and procedures as static methods of a 'ghost' class.

3.7 Classes as Types, or not?

Wegner & Zdonik [30] also discuss the difference between type and class and provide an example where a subclass cannot be substituted for its superclass. This is an early example of a suggestion to add a separate type mechanism in addition to class.

The paper by W. Cook et al, 'Inheritance is not Subtyping' [41] is another example of a paper that influenced the view of subclassing primarily being used as

implementation inheritance and this thus calls for a separate type mechanism.

The consequence of this is that several people have proposed to add a separate type concept to object-oriented languages and Liskov came up with the concept of *subtype substitutability* [42], which is actually a reinvention of the SIMULA notion of substitutability.

However, a separate type mechanism has not made it into mainstream languages. The reason may be that this adds further complexity to the languages, but also that language designers and user do not see a need for it. There are language proposals like Grace that have a separate type mechanism.

8. Adding a type mechanism is an unnecessary complication.

3.8 Cancellation

Wegner & Zdonik has an interesting comment regarding cancellation of attributes. They have an example of a class representing birds with a property that a bird can fly. Cancellation can then be used to define subclasses of non-flying birds like penguins and ostriches.

This is an interesting example, but one may discuss if flying should be a property of all birds since there are birds that cannot fly. One must also be aware that object-oriented languages have limitations with respect to modeling. As mentioned in The Conceptual Framework, object-oriented languages are primarily well suited to model so-called Aristotelian concepts whereas they are not good at modeling so-called prototypical concepts (this should not be confused by prototype-based languages) – we return to this in Section 5.

Cancellation of attributes is obviously a problem with regard to substitutability, but as far as we know, no mainstream language supports cancellation.

9. Cancellation of attributes is a problem with regard to substitutability.

3.9 Inheritance for Development

In the paper by Wegner and Zdonik mentioned above [30] they consider inheritance as an incremental modification mechanism that facilitates reuse.

Black, Bruce and Noble in [43] take this understanding a step further. They consider inheritance as a mechanism for starting with a concrete program and then later on abstracting over concrete elements without having to edit the concrete program, but just adding new attributes or redefine the attributes of the superclass that needs to be modified. They call this *ex post facto parameterization*. Consequently, inheritance is not just to represent classification hierarchies – and as mentioned above, their example language Grace has a separate type system.

The viewpoint that this is the essence of inheritance is of course valid – we tried to argue differently in the ‘The Essence of Subclassing’ [44].

We agree that concrete programs are easier to understand than abstract programs and that teaching should start by concrete examples and only later introduce the corresponding abstractions.

However, we do not agree that inheritance is the best mechanism for going from the concrete to the abstract. We think that after experimenting with a number of examples, it is good practice to make the abstractions and apply these to the examples. This is in accordance with the process of knowledge – see also the Appendix.

Perhaps the use of inheritance as presented here may be useful for illustrating a series of examples leading to a general abstraction, but for developing software, maintenance may be a problem.

10. A subclass hierarchy used to express evolution of code conflicts with both modeling and implementation inheritance.

3.10 Extreme Definition of Object-Orientation

In his blog post: ‘A Proposal for Simplified, Modern Definitions of "Object" and "Object Oriented"’ [45], which is a companion to his essay ‘On understanding data abstraction, revisited’ [25], W. Cook further refines his interpretation of object-oriented programming. He has the following technical definition of object: “*An object is a first-class, dynamically dispatched behavior.*” Inheritance is not essential in his definition of object-oriented programming.

The proposal of Cook is interesting and novel but far from mainstream languages. It is an example of a reuse/technical interpretation taken to the extreme. However, we don’t think it brings object-oriented programming in the right direction.

11. An abstract technical definition of object-oriented programming does not make object-oriented programming easier to understand.

4 The Modeling Interpretation

The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly.

- Donald Knuth

To program is to understand

- Kristen Nygaard

Modeling isn't boring.

- Amber Valletta

As mentioned above, it all started with object-oriented programming being introduced both to describe complex systems and to program these. SIMULA I had this as an explicit goal, applied to processes, SIMULA 67 had it for objects in general, and Beta completed this interpretation by defining a conceptual framework for object-oriented programming – as mentioned called *The Conceptual Framework* in this essay. At the time of SIMULA I ‘description’ was the term use for ‘model’, the main application domain was processes with their own behavior, and interaction between processes, while SIMULA 67 was a general-purpose programming language with objects and classes and subclasses of potentially active objects; processes for the purpose of simulation was supported by an application framework.

In the following we will provide some principles for how to practice modeling. These principles are just a short summary of how to apply modeling in object-oriented programming since space do not allow for an in-depth presentation, but we think it may be useful anyway for programmers familiar with object-orientation. Most of the principles may be applied when programming in mainstream languages like C++, Java, C# and Python.

We recommend reading about object-oriented modeling as described in [4-6], and [7] including *The Conceptual Framework*. We also recommend that object-oriented modeling is part of any course teaching object-oriented programming [46], see also [47].

We present a number of principles that are the basis for the modeling interpretation of object-oriented programming. To ease reading, we also shortly informally define what we understand by object-oriented language mechanisms, and we illustrate these by simple examples. These definitions and examples should all be familiar to programmers using object-oriented languages.

The fundamental principle in the modeling interpretation is that

1	Programming is modeling
---	-------------------------

We consider *the program execution to be the model* being developed during programming. That is, the dynamic process of objects being generated during program execution is the model. Objects, properties of objects and processes generated during the program execution constitute the model and these elements represent phenomena from the application domain. We think it is important that the programmer thinks about what is being generated during program execution.

1.1	The program execution is the model
-----	------------------------------------

The *program* is thus a description of the model (program execution) mainly in the form of a text. The program

describes how to generate the model from objects. This is analogous to a box of LEGO bricks for building a model of a Porsche: the bricks are the objects, and the instruction manual is a program that describes how to build the model of the Porsche.

This view on an object-oriented model is in contrast to e.g. UML where the diagrams are considered the model. In our perspective, diagrams are also descriptions similar to a program text.

The above view of a model first appeared in [48] and has been further discussed in [8, 9, 21, 22].

1.2 The program is a description of the model

Although most textbooks and language definitions do not mention modeling as an important part of programming, many examples do actually show class hierarchies representing classification hierarchies. Our experience is also that most programmers to a large degree actually do model. They may not do it by means of using modeling tools, but rather by sketching for the purpose of understanding the domain and communication of the design.

As a subprinciple of 1.0, we emphasize that

1.3 You do not have to use UML for real for modeling

As mentioned, most of the communities concerned with modeling have separated from the programming community, and much of the object-oriented modeling efforts have been centered on UML. But you do not need to use UML in order to do modeling. Modeling is possible using any object-oriented language. It is, however, useful to make informal diagrams showing e.g. class hierarchies when developing the program, and you may of course use UML if you find that to be useful.

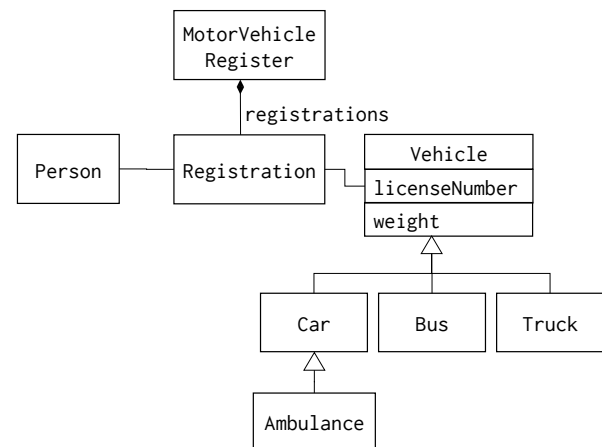
In the modeling interpretation, classes and subclasses represent concepts in the application domain and a class/subclass hierarchy represents a corresponding classification hierarchy in the application domain. Objects represent the corresponding phenomena.

We use two examples to illustrate this interpretation. The first example is a so-called ‘real-world’ example of designing a system for registration of vehicles. The other example is a technical domain just to emphasize that object-oriented modeling applies to all kinds of domains – not just so-called ‘real-world’ domains. For that we use the domain of producers and consumers communicating via buffers.

Consider the development of a motor vehicle registration system. Most programmers will make classes that represent e.g. vehicles (of different types), and registrations; the program executions will therefore have objects that represent the corresponding real phenomena

like vehicles, and registrations. There will also be a class representing the persons being owners of vehicles.

Below we show an informal diagram as well as a program text illustrating these classes. Note that both descriptions contain the same information, and although the diagram may resemble UML it is not a complete UML diagram. Diagrams may be useful for communicating designs in the initial phases, but in later phases textual descriptions may be just as efficient since they are more compact – it is, however, important that such textual descriptions can be shown at the right level of abstraction without details.



```

class MotorVehicleRegister:
    registrations:...
    ...

class Registration:
    ...

class Vehicle:
    licenseNumber:...
    weight:...
    ...

class Person:
    ...

class Car: Vehicle
    ...
class Bus: Vehicle
    ...
class Truck: Vehicle
    ...
class Ambulance: Car
    ...
  
```

A simple example like this illustrates several principles of the modeling interpretation. We use a syntax inspired by Python with respect to using indentation for blocks instead of curly brackets.

4.1 Object

An object is some computerized material characterized by attributes and a statement part, which both may be empty. The attributes of an object may be data-items, methods, and nested classes. An object may inherit from a superclass. We

return to methods, nested classes, and statement part later in this section.

An object represents a phenomenon from the application domain. For the vehicle registration system, we must have objects representing vehicles including cars, busses, and trucks. The attributes of an object represent properties of the phenomena being represented by the object.

2	An object represents a phenomenon from the application domain and the attributes of the object represent properties of the phenomenon
---	---

A singular object represents a phenomenon, which is one of its kind in the application domain.

2.1	If there is only one phenomenon of its kind in the domain, then do not define a class, but use a singular object instead
-----	--

In the 1964 film "Goldfinger" James Bond drove an Aston Martin DB5 replete with gadgets like machine guns, an ejector seat and an oil slick maker.

There is only one instance of this car, and it should therefore be described by a singular object as inheriting from `AstonMartinDB5`:

```
JamesBond_Goldfinger_car: obj AstonMartinDB5
    frontGun, backGun: obj Machinegun
    escape: obj EjectorSeat
    slicker: obj OilSlickMaker
...
```

`AstonMartinDB5` is in turn a subclass of `Car`.

The first keyword **obj** above describes an object that is not an instance of a class. The object `JamesBond_Goldfinger_car` is a singular object since it adds attributes in addition to those inherited from `AstonMartinDB5`. The objects (**obj**) `frontGun`, `backGun`, `escape`, and `slicker` are part objects embedded in `JamesBond_Goldfinger_car`. See also the section about composite objects below.

If your favorite language does not have singular objects (as Beta), use anonymous classes in Java, singletons in Scala or the singleton pattern in other languages.

4.2 Data-Items

Data-items may be constants or variables representing constant or variable values or constant or variable references to objects.

Data-items are relevant for modeling as well as implementation. From a modeling point of view a data-item may represent a property of a physical phenomenon, such as the color of a vehicle; or it may represent a reference to an object: a vehicle may have a reference to the owner of the vehicle.

However, in most cases data items are used for implementation and thus private to the object. There may be good reasons to declare data items to be private just as they in some languages are always private. In this case, if the value represented by a data-item is part of the model, it may of course be accessed via set- and get-methods or via general methods.

3	A public data item should represent a property of the phenomenon represented by the object
---	--

A vehicle will obviously have a license number and a weight, and these may be represented by simple data-items. Calculation of `licenseFee` and e.g. the change of owner of a vehicle may be represented by methods.

4.3 Class and Subclass

A class is a pattern for creating objects with the same properties in terms of attributes and actions. A subclass is a class that inherits attributes and actions from another class – its superclass, by extending the structure of its superclass.

The following three principles describe role of class and subclasses in the modeling interpretation.

4	A class represents a concept in the application domain
---	--

4.1	A subclass represents a specialization of the concept represented by the superclass, and it must be behavioral compatible with its superclass
-----	---

4.2	A class hierarchy represents a classification hierarchy in the application domain
-----	---

As mentioned in the Substitutability subsection in Section 3, a key property of class and subclass is that a variable typed with a class name may be assigned a reference to an object of a subclass. A variable `v`: **ref** `Vehicle` may thus refer to objects being instances of `Car`, `Truck`, and `Bus`. Code that involves such a variable `v` must work independently of which kind of object being referenced by `v`. As mentioned in Section 3, this was a fundamental idea in SIMULA, and most object-oriented languages have adopted this, i.e. classes define types of reference variables (and reference expressions) and subclass define subtypes.

The example with class `vehicle` and subclasses `car`, `bus`, `truck`, was actually the example that triggered the idea of subclass in SIMULA, although applied to passing a toll bridge and paying a fee according to the type of vehicle [40].

4.4 Method

A method is an attribute of an object that is a piece of code that performs a task when executed. The task may compute one or more values from the state of the object and its parameters and/or implement an algorithm that changes the state of the object.

Phenomena in a given domain may exercise behavior resulting in changes to the phenomena and other phenomena and/or compute a property of the phenomena – a *behavioral property*.

5	A public method is an attribute of an object that represents a behavioral property of the phenomenon represented by the object
---	--

A method property of `Vehicle` would be `licenseFee` that calculates the license fee based upon the properties of the vehicle, e.g. length and weight. Another would be e.g. `changeOwner`

4.5 Cancellation of Properties

Most programmers will agree that the properties defined in class `Vehicle` (e.g. `licenseNumber`) will be a property of all subclasses of `Vehicle`, i.e. there will be no cancellation of properties as supported by a number of languages. This is also one of the principles of the modeling interpretation:

6	No cancelation of properties in subclasses, including data items and methods
---	--

It is obviously important that these are not cancelled in subclasses, as the main part of the system will count on all types of `Vehicle` objects, i.e. including objects of subclasses, to have these methods.

4.6 Virtual and Non-Virtual Methods

As mentioned, in Section 3, SIMULA distinguished between virtual and non-virtual methods (called procedures in SIMULA). In the design of a class, it was (and is) important to consider which methods may be redefined in subclasses and which should not be redefined. In Smalltalk, Java (see the note on final below), and Python, all methods of a class are virtual, implying that they can be redefined in subclasses – this of course supports reuse and gives flexibility, but for a given class, it is no longer clear which methods are intended to be redefined. Languages like C++, C#, and Beta also distinguish virtual and non-virtual methods.

The `changeOwner` method of class `Vehicle` is obviously a method that should not be redefined, while `licenseFee` has to be redefined in order to reflect the difference in license fee of the different types of vehicles.

A non-virtual method is a method where the task to be computed is the same for all objects being instances of the class and its subclasses.

A virtual method is a method where the task to be computed depends on the actual subclass of the class.

7	In the design of a class, one must be explicit about which methods are virtual and which are non-virtual
---	--

Beta supports these principles by supporting a final binding of a virtual, which means that it no longer can be extended. The final mechanism has also been included in Java, and it may be used to declare a method as non-virtual.

8	For a virtual binding one must be explicit about whether or not the binding is final or is still virtual
---	--

Most object-oriented languages allow a virtual method in a superclass to be redefined completely in a subclass. Doing this makes it possible to change the effect of the method compared to the one in the superclass. The `licenseFee` method for the `Bus` object could e.g. be redefined to issue a 'Hello World' instead of calculating the `licenseFee`.

The fact that a variable typed with a class may be assigned a reference to an object of a subclass implies that one must be careful when redefining a virtual method.

Design by Contract as proposed by Bertrand Meyer [29] is a useful technique for building reliable software. The idea is that each class has an *invariant* which is an assertion describing a property which holds of all instances of a class in between (public) method invocations. Each method of the class has a pre-condition which is an assertion that must hold before the invocation of the method. And finally, each method has a post-condition which is an assertion that must hold after invocation of the method.

For a given method one must then prove/assert/validate that if the invariant and the pre-condition is true before the invocation, then the post-condition must be true after the invocation.

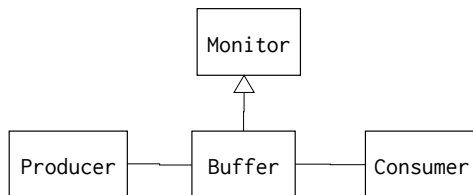
A subclass must inherit the invariant of its superclass and all new methods must have a pre-condition and a post-condition.

For a method being redefined, it may have the same pre-condition and post-condition as the method in the superclass. It is possible to weaken the pre-condition (but not strengthen it) and strengthen the post-condition.

9	A method being redefined must provide the same effect as the one in the superclass with respect to fulfilling a possible class invariant, and possible pre-, and post-conditions for the method
---	---

An often-encountered criticism of object-oriented programming is that programming is not just modeling the so-called ‘real world’! Still, motor vehicle registers would not be around had there not been real cars and real persons owning them. In addition to these, the class `Registration` defines objects that are the electronic registrations as part of the system.

The following *technical domain example* of buffers with producers and consumers will demonstrate that the modeling interpretation does not have to do with just ‘real world’ applications but is relevant for all domains including technical domains. The example also demonstrates how the principle 9 above may be partially supported.



```

class Monitor:
  ...

class Buffer: Monitor
  ...

class Producer:
  buff: ref Buffer
  ...

class Consumer:
  buff: ref Buffer
  ...
  
```

Buffer objects will have methods `put` and `get`; for the purpose of illustration, we just have buffers of Integers. A Producer object will call the `put` method on a Buffer object, and a Consumer object will call the `get` method. The methods `put` and `get` should have exclusive access to the representation of the Buffer, and below this is solved by defining Buffer as a subclass to Monitor, and the methods `put` and `get` methods as extension of the `entry` method of Monitor. The `entry` method ensures exclusive access, in addition to the inheritance of properties:

```

class Monitor:
  S: obj Semaphore
  entry:
    S.wait
    inner(entry)
    S.signal
  S.init
  inner(Monitor)
  
```

```

class Buffer: Monitor
  L: ref List
  put: entry
    in v: var integer
    ...
    L.put(v)

  get: entry
    out v: var integer
    ...
    v := L.get

  L := List(integer)
  
```

As can be seen, class `Buffer` is specified as a subclass of `Monitor` by having `Monitor` in front of the description of the class; similarly, the methods `put` and `get` inherit behavior from `entry`.

The keyword **ref** describes a variable that may refer instances of a given class; **var** describes a data-item representing a value; and **in** describes that the data-item is an input parameter.

In SIMULA and Beta, a description of an object may contain statements. These statements often play the role of a constructor in other languages but do also describe the actions of active objects (see below). Statements in a superclass and a subclass are combined by means of **inner**. When an object is generated, execution starts by the statements in the topmost superclass. Execution of an **inner** implies that the statements in a possible subclass are executed. For a more detailed explanation of **inner**, see [21].

In the example above, class `Monitor` contains the statements

```

S.init
inner(Monitor)
  
```

and class `Buffer` has a statement

```

L := List(integer)
  
```

The example below shows a class `Buffer` that inherits from `Monitor`. The Buffer has two virtual (`<`) methods, `put` and `get`. Class `BoundedBuffer` and class `VariableSizedBuffer` are both subclasses of `Buffer` and they have bindings of `put` and `get` that are extensions of `put` and `get` in class `Buffer`. The methods `put` and `get` inherit from method `entry` in class `Monitor` and since `put` and `get` are extended in `BoundedBuffer` and `VariableSizedBuffer`, the code in `entry` is executed as a wrapper around the code for `put` and `get` defined in `BoundedBuffer` and `VariableSizedBuffer`. For a detailed explanation of this, see [21, 49, 50].

```

class Buffer: Monitor
  put:< entry
  ...
  get:< entry
  ...
  ...

class BoundedBuffer: Buffer
  in size: var integer
  put:<:
  ...
  get:<:
  ...
  ...

class VariableSizedBuffer: Buffer
  put:<:
  ...
  get:<:
  ...
  ...

```

The example above illustrates the following principle:

- | | |
|----|--|
| 10 | Redefinition of virtual methods must be designed as extensions of the definition in the superclass and not a plain/complete redefinition |
|----|--|

In most object-oriented languages, a virtual method may be redefined, and the definition in the superclass may be invoked using a mechanism like `super`. But since invocation of `super` is optional, the code in `super` may never be executed. Beta supports this principle in that a method cannot be redefined, but only extended using the inner-mechanism. This guarantees that the code in the `super` method is always executed. It does, however, not ensure behavioral compatibility since the code in a submethod may have an arbitrary effect. The fact that a virtual method cannot be redefined but only extended in a subclass is called *structural compatibility* in Beta and may be considered a generalization of signature compatibility as defined by Wegner & Zdonik.

In some cases, a virtual method is defined to be a default method in case no binding occurs in a subclass. In this case it makes sense to redefine the default. The SIMULA class `HashTable` has a default virtual method for computing a hash key and this default method may be redefined in a subclass.

- | | |
|----|---|
| 11 | One must be explicit about whether or not a virtual method is considered a default method or can be replaced by a method with effect similar to the default |
|----|---|

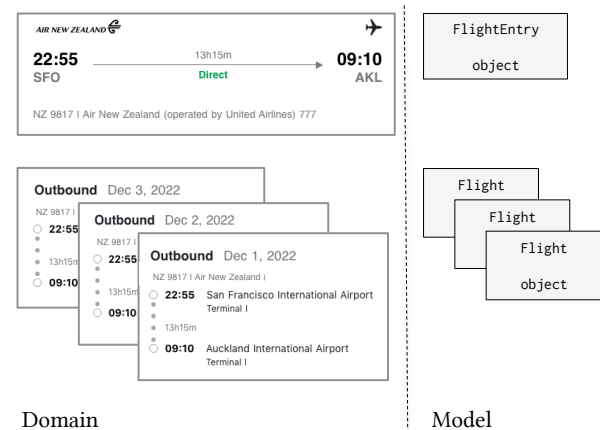
As mentioned above, inheritance may be viewed as a mechanism for reuse since the code in the superclass is reused in each subclass. However, reuse should not be the primary use of inheritance.

- | | |
|----|--|
| 12 | Implementation inheritance must be avoided |
|----|--|

4.7 Nested Class

A nested class represents a concept that belongs to the context of a phenomena or concept represented by an object or class. A non-virtual nested class represents the same concept for all instances of the enclosing class and instances of its subclasses.

As an example², consider the notion of flight entries defining routes (e.g. NZ9817 from San Francisco to Auckland) served by a specific airline company (in this case Air New Zealand) and the corresponding flights.



For each such route, there will just one flight entry represented by an object of class `FlightEntry`. Such an object has attributes that represent the name of the route, the source and destination cities and the scheduled departure time, arrival time, and flying time. For simplicity, we here assume that these flights take place every day during the week.

In addition, to a given flight entry, there is a need for an object representing each of the actual flights taking place. These flights are represented by objects of a nested class `Flight`. A given flight object has attributes representing the reservations of seats, the actual departure time, the actual arrival time and the actual flight time.

For a given object representing a flight entry (an object of class `FlightEntry`) there will a nested object per day representing the actual flight taking place on that day.

² This example first appeared in [39] and is also included in [19].

```

class FlightEntry:
  name: ref String -- e.g. NZ7
  source, destination: ref City Airport
  departureTime: var TimeOfDay
  arrivalTime: var TimeOfDay
  flyingTime: var Time

  class Flight:
    seats: ...
    departureDate: var Date
    actualDepartureTime: var TimeOfDay
    actualArrivalTime: var TimeOfDay
    actualFlyingTime: var Time
    delay:
      out V: var Time
      V := actualArrivalTime - arrivalTime

```

A Flight object has a method delay that computes the delay of the flight from arrivalTime in the enclosing FlightEntry and the local actualArrivalTime.

The role of nested class is reflected in the following principle:

13	If a concept is only meaningful in the context of a specific phenomenon or concept, then it should be represented by a nested class
----	---

A virtual class [51] is a nested class that may represent different specializations in subclasses of the enclosing class.

13.1	If a concept is only meaningful in the context of a specific phenomenon, but have different specializations in subclasses, then it should be represented by a virtual class
------	---

As an example of a virtual class consider the following change to MotorVehicleRegister:

```

class MotorVehicleRegister:
  registrations:...
  ...
  class Registration:<
    owner: ref Person
    theVehicle: ref Vehicle
    ...

```

```

Obj MI6MotorVehicleRegister:
  MotorVehicleRegister
  class Registration::
    licenseToKill:...
    ...

```

The class Registration is now defined as a class nested in MotorVehicleRegister (as it really only make sense there); in addition, it is defined as a virtual class (:<). In the MotorVehicleRegister for MI6, where the James Bond car is

registered, the Registration is extended with some additional attributes, among these a licenseToKill.

4.8 Active Object

Most mainstream languages have a task mechanism – SIMULA and Beta have active objects, and we use the term *active object* below to also cover tasks. In the following, we briefly describe active objects.

A SIMULA object may be active in the sense that a class may have a statement part just like a procedure. The statement part is executed as part of the generation of the object. A SIMULA object behaves like a coroutine in the sense that execution of the statement part may be suspended, and execution returns to the caller. Execution may later be resumed at the point of suspension.

The ability to describe active objects was due to the need for describing systems with concurrent activities to be simulated by SIMULA program. SIMULA did not have true concurrency but only what they called quasi-parallel systems implemented using coroutines. True concurrency was later added by the Lund SIMULA compiler.

Except for Beta, we do not know of any object-oriented language that supports active objects in the sense of SIMULA. Most mainstream object-oriented languages have a task mechanism in addition to passive objects, which adds to the complexity of the languages. Concurrent Pascal [52] was inspired by SIMULA classes and in addition to classes, it has processes and monitors that are a special form of classes. Concurrent Pascal, however, did not have inheritance or virtual methods.

In Beta the notion of SIMULA objects was further developed and Beta supports coroutines as well as concurrent objects. As pointed out in [49, 50, 53], concurrency in Beta (as in Java, and other mainstream languages) is not safe with respect to race conditions and proposals for making Beta objects safe are presented.

According to an interview [54] with Alan Kay he *thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages*. Smalltalk objects, however, are not active as in SIMULA.

14	An active object represents a process in the application domain
----	---

4.9 Composite Objects

Beta has a mechanism for defining composite objects by means of embedded objects – an embedded object is a fixed part of the composite object, and it has a name that is a constant reference to the object. In [55] we showed that the combination of embedded objects and nesting as in Beta may be used to inherit/include code from other classes in a form that avoids most of the problems with multiple

inheritance. The proposal is thus also an alternative to mixins and traits.

Consider a class Address:

```
class Address:
  street: ref String
  streetNo: var integer
  town: ref String
  country: ref String
  print:<
    inner(print)
    print{street,streetNo,town,Country}
```

An Address may be embedded in a Person object:

```
class Person:
  name: ref String
  addr: obj Address
  ...
```

The keyword **ref** declares an ordinary reference to an object of the class – in the above cases String objects. The keyword **obj** specifies an embedded object of the class – in this case `addr` is the name of an Address object being embedded in the holding Person object.

The code of Address is made available in Person using composition as an alternative to inheritance. However, the attributes of the Address object have to be qualified by `addr` both within Person and when accessed from outside as in `P.addr.print` where P is a reference to a Person object. This is thus different from multiple inheritance, mixins and traits, and of course more verbose, but ambiguities will never appear. From a modeling approach, a Person *is not* an Address, but Person *has* an Address property.

We are aware that Black et al. reject a solution where it is necessary to qualify the town attribute by `addr` as in `P.addr.town`. They use as example `self.asRectangle.area` and their point is that the programmer may forget to qualify with `asRectangle` just as there will be a run-time overhead. We do not think that qualifying by `addr` is inconvenient – it improves readability and for Beta, the compiler will catch if the programmer forgets about ‘`addr`’. This is perhaps a difference between a static type check and a run-time type check. In Beta such errors are caught by the compiler whereas in Squeak [56] it will only be caught at run-time. They also mention that `asRectangle` gives a run-time overhead, which is not the case with `addr`. In Beta there is no run-time overhead.

An Address has a virtual attribute `print`, which may be bound as shown below:

```
class Person:
  name: ref String
  addr: obj Address
  print::
    name.print
```

The part object `addr` is then a singular object inheriting from Address. As can be seen, the binding of `print` may access the `name` field in the enclosing Person object.

An Address may also be a part of a Company object:

```
class Company:
  name: obj BusinessName
  addr: obj Address
  print::
    name.print
```

The embedded object `addr` of Company is a singular object inheriting from Address. As can be seen, the binding of `print` may access the `name` field in a holding Company object in a similar way as `print` in `Person.addr` may access the `name` of a Person object.

The above example is possible in Beta, since singular objects, nesting and embedded objects are general language mechanisms without special rules.

Here we provide a definition of `Linked_Queue` that does not use multiple inheritance as mentioned in Section 3. Instead, we do this by a combination of classification and composition. In the following sketch the `Linked_List` embedded object is there just to provide for the reuse of `Linked_List` in the implementation of `Linked_Queue`:

```
class Queue:
  ...
class Linked_List:
  ...
class Linked_Queue: Queue
  ll: obj Linked_List
  ...
```

As in the Address-example, the `ll` may of course be defined as a singular object inheriting from `Linked_List` and redefine what is needed in order to use it for the implementation of a `Linked_Queue`.

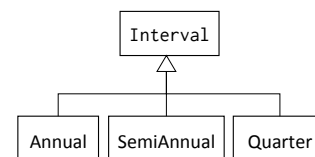
The idea of using nested singular embedded objects is of course not a general ‘solution’ to the multiple inheritance. In [21], Section. 6.2 and Chapter 18 multiple inheritance is discussed from a classification point of view.

15	Use a language with general mechanisms without special cases/rules
----	--

4.10 Covariance Versus Contravariance

For redefinition of virtual methods and classes, and for generics, there is an issue with respect to covariance versus contravariance of parameters [57].

Covariance and contravariance are tricky issues and we thus provide a short explanation here. Given the following class hierarchy on intervals of paying license fee,



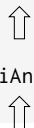
the following simple example explains the issues:

Covariance

```

class Vehicle:
  payLicenseFee(period: Interval):< {...}
  ...
class Truck: Vehicle
  payLicenseFee(period: SemiAnnual):< {...}
  ...

```



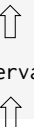
With covariance, the types of parameters of redefined methods are redefined to be subtypes of the parameter types of the virtual method. In the example above, the type of `period` is redefined to `SemiAnnual`, which is a subtype of `Interval`.

Contravariance

```

class Vehicle:
  payLicenseFee(period: SemiAnnual):< {...}
  ...
class Truck: Vehicle
  payLicenseFee(period: Interval):< {...}
  ...

```



With contravariance, the types of parameters of redefined methods are redefined to be supertypes of the parameter types of the virtual method. In the example above, the type of `period` is redefined to `Interval`, which is the supertype of `SemiAnnual`.

Contravariance does not make sense from a modeling point of view since specialization of a class with virtual methods or classes in most cases implies a corresponding specialization of the virtual methods or classes.

However, it is well known that contravariance will ensure full static type checking and substitutability. Covariance implies a run-time type check [57] which is the case for Beta.

In the following example, we illustrate the issue of covariance versus contravariance. We use the example of a classification hierarchy of `Vehicle` with subclasses `Car`, `Bus`, and `Truck`.

We have a general register defined by class `Register`, which has a virtual class `ElmType` being the type of elements that may be inserted in the register. The type of the parameter `Elm` of `insert` is `ElmType`. Since `ElmType` is qualified by `Object`, any object may be inserted in a `Register` object.

```

class Register:
  ElmType: < object
  insert:
    in Elm: ref ElmType
  ...

```

We then specify a register of vehicles as a subclass of `Register`. The virtual class `ElmType` is then further bound (specialized) to be a `Vehicle`, which means that only `Vehicle` objects may be inserted in a `VehicleRegister`.

```

class VehicleRegister: Register
  ElmType: < Vehicle

```

We define a register of busses as a subclass of `VehicleRegister`, binding `ElmType` to `Bus`, which means that only `Bus` objects may be inserted in a `BusRegister`.

```

class BusRegister: VehicleRegister
  ElmType: < Bus
  --- insert now requires a Bus

```

The code below declares a `BusRegister` and a `Bus` object.

```

BR: ref BusRegister
B: ref Bus
BR := new BusRegister
B := new Bus
BR.insert(B)

```

Insertion of a `Bus` object in `BR` is ok since the parameter of `insert` is typed with `Bus`. If we, however, attempt to insert, e.g. a `Truck`, it will not work.

```

T: ref Truck
T := new Truck
T := BR.insert(T)

```

The type checker for Beta will in general not catch this kind of error, and instead a runtime check is inserted here.

The type of the parameter `Elm` of `insert` is covariant which seems natural here. If your language only supports contravariance, one may only bind `ElmType` to a superclass of the type of `ElmType` in the superclass.

The situation is analogous the so-called reverse assignment in SIMULA:

```

V: ref Vehicle
B: ref Bus
T: ref Truck
B := new Bus
T := new Truck
V := B // ok
B := V // ok, but run-time check
V := T // ok
B := V // not ok, run-time check fails

```

As mentioned in, [57], final bindings and part objects (corresponding to `obj` in this paper) may often avoid a run-time type check since the binding of the virtual is known at compile-time.

16	Think covariance instead of contravariance and accept a limited amount of run-time checking
----	---

5 Miscellaneous

5.1 The Message-Passing Interpretation

As mentioned in the introduction, some of the blogposts claim that according to Kay [54], the central idea of object-oriented programming is that a program execution is a set of objects sending messages to each other, and he apparently also regretted that he named it object-oriented programming – since the important characteristic is messages between objects. In any case it does not conflict

with the modeling interpretation whether object or message is the central term.

For SIMULA, processes were just as important to be able to model as (passive) objects as in Smalltalk. In [58] Dahl and Nygaard states: “*The original purpose was to create classes and subclasses of data structures and processes.*” In [40] they stated: “*We needed subclasses of processes with own actions and local data stacks, not only of pure data records.*”

Smalltalk uses the term *message* where SIMULA used the term *remote procedure*. Technically there is no difference between message in Smalltalk-80 and remote procedure in SIMULA. However, for Smalltalk-72 it was actually possible to manipulate the message by looking at its name, save it for later execution, etc. In addition, messages are not asynchronous as in ‘real’ message passing systems.

A major difference between SIMULA and Smalltalk is that objects in SIMULA may be active and model parallel processes from the domain.

5.2 Prototype-Based Languages

In this essay we have mainly been concerned with class-based object-oriented languages. Prototype-based languages is another interpretation of object-oriented programming and has been advocated by several authors [59], [60], and languages, with Self [3] as the most prominent.

We agree that in the initial experimental activities of development of a system it may be useful to be able to experiment with prototypical objects for obtaining knowledge of your domain. However, as you obtain more and more knowledge of your domain, you have to organize your knowledge, and classification and composition are important means for doing that. You develop concepts covering your phenomena, you understand your phenomena in terms of compositions of parts and you organize your concepts in classification hierarchies. This is in accordance with how people in general obtain and organize knowledge – see also the Appendix and The Conceptual Framework.

There is nothing that prevents a class-based language to include a mechanism for describing objects that are not instances of a class. Beta has singular objects; Java has anonymous classes, and Scala has singletons. None of the languages support implicit delegation but there is nothing to prevent including that. For a more elaborate discussion of how to integrate prototype-based languages in class-based languages, see [24].

5.3 Functional Programming

Functional programming is often suggested as an alternative to object-oriented programming and some of the blog posts seem to consider functional programming as a silver bullet.

We agree that functional programming languages include many languages mechanism that are useful for

programming in general and this also includes object-oriented programming.

We have never advocated a pure object-oriented language like Smalltalk. The Beta pattern was designed to be a unification of class and method from object-oriented languages but also to cover value, type and function from functional programming. Beta does thus support higher order types and classes and functions, but it may lack some of the expressiveness of functional languages.

We do think that it is necessary to support state of systems in terms of states of objects. We think that the handling of state in functional languages is very indirect.

Objects are the natural unit for representing state whereas state changes and computation of properties by method and functions is best expressed in a functional style. See also [61], [21], [22], Scala, and Python.

6 Summary

The motivation for writing this essay was the apparently increasing critique of object-orientation as a disaster especially appearing in many blogs on the net.

Our observation was that most of the postulates in these blog posts are not grounded in solid knowledge say research just as they contain many errors.

When reflecting on the critique it became apparent that there are issues with object-orientation as practiced in the mainstream interpretation that need to be dealt with. The main cause of most of these issues seems to be the overwhelming focus on reuse at the expense of modeling.

The goal of this essay has been to identify critical issues in the mainstream interpretation of object-orientation and to argue for more focus on modeling.

We have identified 12 issues which we think are critical for object-oriented programming and which one should be aware of when practicing object-oriented programming.

We recommend a modeling approach to object-oriented programming, which we think will avoid many of the pitfalls of object-oriented programming as identified by the issues. For this we have identified 16 main principles (and a number of subprinciples).

The issues and principles may be part of a method for object-oriented programming; however, they can also just be used in programming even if no specific method is followed.

The issues and principles cannot stand alone – there are many other principles and techniques that must/can be applied for programming, but they may be a useful supplement.

Object-oriented programming has limitations for modeling as have all programming (and modeling) languages. The Conceptual Framework was an attempt to develop a conceptual framework that goes beyond the Beta language. As stated in [21], it is important to be aware of when the languages falls short of modeling a given aspect

of a domain and then how you get around it in your program/model. Programming languages evolve and limitations at one point of time may be overcome with new language mechanisms.

Multiple inheritance is an example of limitations of state of art of object-oriented languages. Multiple inheritance is technically complicated, and we are not aware of any object-oriented language that has a satisfactory mechanism for multiple inheritance from a modeling point of view just as most proposals are technically complicated.

Modeling was here. A common thread in this essay is that reuse has been dominating object-oriented programming at the expense of modeling. Although modeling does not dominate the mainstream interpretation, there are elements of modeling in many of the (early) papers on object-oriented programming. One example is John F Shoch in 'An Overview of the Programming Language Smalltalk-72' [14] that has a section about the *Origins of Classes and Instances* where he describes Plato's theory of Forms – form being the origin of class and thing being the origin of instance. By the way, the term object-oriented is not used in the paper by Shoch, and Plato did not seem to survive to Smalltalk-80.

Bertrand Meyer in the design of Eiffel has emphasized that object-orientation provides strong support for both reuse and modeling although reuse seems to have a more important role than modeling [62].

And as mentioned there was a great interest in modeling in the eighties, but unfortunately this implied that the modeling community somehow got isolated from the programming community.

Acknowledgement

We thank Kaj Grønbaek for inspiring us to write this essay and we thank the anonymous referees and the PC Chair Jeremy Singer for many useful comments.

Appendix – On Phenomena and Concepts

Modeling is about identifying relevant phenomena and concepts from the application domain and describing these in a program. Programmers practicing modeling must therefore be aware of the nature of phenomena and concepts and The Conceptual Framework was developed for this purpose. The conceptual framework provides conceptual means for understanding and organizing knowledge about a given application domain – being a real-world domain or a technical domain.

Functional programming is based on a foundation in mathematics – The Conceptual Framework is supposed to be a foundation for object-oriented programming. In the rest of this section, we provide a short summary of the elements of The Conceptual Framework.

According to The Conceptual Framework:

- “a phenomenon is a thing that has definite, individual existence in reality or in the mind; anything real in itself”, and
- “a concept is a generalized idea of a collection of phenomena, based on knowledge of common properties of instances in the collection.”

An application domain may contain different kinds of phenomena:

- A phenomenon may be a physical entity that has identity and takes up space. It may have properties/state that changes over time and it is unique and distinguishable from other phenomena. In the vehicle example, the actual vehicles being registered are examples of material phenomena. In the queue example the queues are examples of material phenomena.
- A phenomenon may be a property characterizing a physical entity. Such properties may be the length, weight, and color of a given phenomenon and vehicles have such properties. Vehicles may also have properties such as its owner and registration number. Queues may have properties such as the number of items in the queue, and the successor of a given element in the queue.
- A phenomenon may be a transformation in the form of events, sequences of events, and actions going on in sequence, alternation or in parallel. Such phenomena may change the state of substantial phenomena and thereby its associated property phenomena. The registration of a vehicle and the change of owner are examples of events/actions. The insertion and deletion of elements in buffers are also examples of events/actions.

A concept is an abstraction over a collection of phenomena with similar properties. Concepts and abstraction are essential for our understanding and organization of knowledge.

A concept is traditionally characterized by the following terms:

- The *extension* of a concept refers to the collection of phenomena that the concept somehow covers.
- The *intension* of a concept is a collection of properties that in some way characterize the phenomena in the extension of the concept.
- The *designation* of a concept is the collection of names (or pictures) by which the concept is known.

As mentioned, a class may represent a concept: the intension of the concept is represented by the attributes of the class; the extension is the set of objects generated from the class; and the designation of the concept is represented by the name of the class.

A subclass represents a specialization of the concept represented by the superclass:

1. The intension of the specialization is an extension of the intension of the superclass: more attributes are added.
2. The extension is a subset of the extension of the superclass: the objects do belong to the extension of the superclass.

In reality we have to deal with a huge number of concepts and phenomena, and it would not be possible to deal with that amount of complexity without tools for hierarchical organization of concepts and phenomena. *Classification* and *composition* are means to organize complexity in terms of hierarchies. A classification is represented by a class/subclass hierarchy. Composition is related to part-whole hierarchies, and decomposition of methods (procedures and functions) into sub-methods. We do not consider composition further here since it should be well-known from programming languages in general. It is, however, important to be aware that they are two different conceptual means – the one cannot replace the other and vice versa.

The vehicle hierarchy is an example of a classification hierarchy where car, bus, and truck are specializations of vehicle.

It is important to stress that modeling (and thus programming) is an iterative and evolutionary process. In The Conceptual Framework, the process of knowledge is described as follows:

1. Identification of phenomena in the domain and representation of these by objects, attributes, and behavior
2. Identification and development of abstractions / concepts and representation of these by classes.
3. Application of the concepts and classes to understand the relations between phenomena and objects.

Phase 1 typically corresponds to the exploratory phase where prototypical objects may be useful. In phase 2 objects may be organized in classes and in phase 3 the classes are used to organize our understanding about the objects and in the further identification of phenomena and objects.

It is important to stress that the above is not three sequential phases; iteration and evolution takes place within each phase, and several iterations of the three phases take place. A major message here is that objects and classes including class hierarchies will undergo several changes during the analysis of the domain and design of a system.

A note about terminology, as mentioned, a class may represent a concept and a subclass may represent a

specialization. The term subclass was used in SIMULA and refers to the fact that the extension of a subclass/-specialization is a subset of the extension of the superclass.

Java uses the term *extend* to define a subclass – *extend* here means that the intension of the superclass in the form of attributes is extended with more attributes in the subclass.

This leaves us with derived class and base class from C++. According to Stroustrup [63] p. 49 he invented those terms since he could never remember what was sub and what was super, and he was not the only one with that problem. He may be right about that! We think it may be useful to think about how a subclass extends the intension and defines a subset of the extension of the superclass. In [29] Section 10.2.3, Meyer recommends avoiding the term subclass – ‘*So inheritance is specialization from the type viewpoint and extension from the module viewpoint*’.

Adding new attributes in a subclass is one way of specializing a class into a subclass – extending the intension. Another way is to specialize the possible virtual methods and virtual classes of the superclass.

We conclude this appendix by re-emphasizing what we understand by an object-oriented model:

- A program execution is regarded as a physical model, representing the behavior of either an existing or planned system (in an application domain) from a particular viewpoint.

The model is thus considered the dynamic process of objects created during a program execution. The program is considered a description of the model.

This is in contrast to e.g. UML where the diagrams are considered the model. In our perspective, diagrams are also descriptions similar to a program text.

The above definition first appeared in [48] and has been further discussed in [8, 9, 22].

References

- [1] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. 1968. SIMULA 67 Common Base Language (Editions 1968, 1970, 1972, 1984). Norwegian Computing Center, Oslo
- [2] A. Goldberg and D. Robson. 1983. *Smalltalk-80: the language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
- [3] D. Ungar and R.B. Smith. 1987. *Self: The Power of Simplicity*. In: *OOPSLA'87 – Object-Oriented Programming Systems, Languages and Applications*. Orlando, Florida, USA. ACM Press.
- [4] G. Booch. 1991. *Object-Oriented Analysis and Design with Applications*. Redwood City: Benjamin/Cummings.

- [5] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. 1991. *Object-Oriented Modeling and Design*. Prentice Hall.
- [6] P. Coad and E. Yourdon. 1991. *Object-Oriented Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- [7] G. Booch, J. Rumbaugh, and I. Jacobson. 1998. *The Unified Modeling Language Reference Manual*. Addison Wesley.
- [8] O.L. Madsen and B. Møller-Pedersen. 2010. *A Unified Approach to Modeling and Programming*. In: *MoDELS 2010*. Oslo. Springer.
- [9] O.L. Madsen and B. Møller-Pedersen. 2018. *This is not a Model*. In: *8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Limassol. Springer LNCS 11244.
- [10] B. Selic. 2012. *Key Note: Model-Based Software Engineering in Industry: Revolution, Evolution, or Smoke?* In: *NOKOBIT 2012*.
- [11] W. Cook. 2006. *Peak Objects*. In: *ECOOP 2006 - European Conference on Object-Oriented Programming*. Springer-Verlag Berlin Heidelberg.
- [12] O.-J. Dahl and K. Nygaard. 1966. *SIMULA: an ALGOL-based Simulation Language*. Communications of the ACM, 9(9): p. 671–678.
- [13] C.A.R. Hoare. 1965. *Record Handling*. ALGOL Bulletin, (21).
- [14] J.F. Shoch. 1979. *An overview of the programming language Smalltalk-72*. ACM SIGPLAN Notices, 14(9): p. 64–73.
- [15] B. Meyer. 1992. *Eiffel: The Language*. Prentice Hall.
- [16] B. Cox. 1986. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley Longman, Incorporated.
- [17] J. Gosling, B. Joy, and G. Steele. 1999. *The Java (TM) Language Specification*. Addison-Wesley.
- [18] A. Hejlsberg, S. Wiltamuth, and P. Golde. 2003. *The C# Programming Language*. Addison-Wesley.
- [19] Python Software Foundation. 2019. *Python*. <https://www.python.org>, <https://docs.python.org/3/reference/index.html>.
- [20] Pluralsight. 2022. *JavaScript*. <https://www.javascript.com/>.
- [21] O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. 1993. *Object-Oriented Programming in the BETA Programming Language*. Addison Wesley.
- [22] B.B. Kristensen, O.L. Madsen, and B. Møller-Pedersen. 2007. *The When, Why and Why Not of the BETA Programming Language*. In: *HOPL III - The third ACM SIGPLAN Conference on History of Programming Languages*. San Diego.
- [23] O.L. Madsen. 2006. *From ECOOP'87 to ECOOP 2006 and beyond*. In: *ECOOP'06: Proceedings of the 20th European conference on Object-Oriented Programming*. Nantes, France. Springer-Verlag Berlin, Heidelberg.
- [24] O.L. Madsen. 1995. *Open Issues in Object-Oriented Programming*. Software Practice & Experience, 25(S4).
- [25] W.R. Cook. *On Understanding Data Abstraction, Revisited*, in *OOPSLA - 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 2009, ACM: Orlando, Florida, USA.
- [26] A.P. Black. 2013. *Object-oriented programming: Some history, and challenges for the next fifty years*. Information and Computation, 231: p. 3–20.
- [27] V. Bono, J. Kuśmierek, and M. Mulatiero. 2012. *Magda: A New Language for Modularity*. In: *26th European conference on Object-Oriented Programming*.
- [28] D.J. Barnes 2000. *Object-Oriented Programming with Java – An Introduction*,. Prentice Hall, Upper Saddle River, New Jersey.
- [29] B. Meyer. 1988. *Object-Oriented Software Construction*. Prentice-Hall.
- [30] P. Wegner and S. Zdonick. 1988. *Inheritance as an Incremental Modification Mechanism or What Like is and Isn't Like*. In: *ECOOP'88 – European Conference on Object-Oriented Programming*. Oslo, Norway. Springer Verlag.
- [31] D.G. Bobrow and M. Stefik. 1986. *The LOOPS Manual*.
- [32] H. Cannon. 1982. *Flavors: A Non-Hierarchical Approach to Object-Oriented Programming*.
- [33] S.E. Keene. 1989. *Object-Oriented Programming in COMMON Lisp—a Programmer's Guide to CLOS*. Reading MA: Addison Wesley
- [34] B. Meyer. 2006. *Basic Eiffel language mechanisms*. <http://se.ethz.ch/~meyer/publications/online/eiffel/basic.html>.
- [35] Eiffel. 2022. *Eiffel - ET: Inheritance*. https://www.eiffel.org/doc/eiffel/ET_Inheritance.
- [36] M. Flatt, S. Krishnamurthim, and M. Felleisen. 1998. *Classes and Mixins*. In: *POPL*. San Diego.
- [37] G. Bracha and W. Cook. 1990. *Mixin-based Inheritance*. In: *Joint OOPSLA/ECOOP'90 – Conference on Object-Oriented Programming: Systems, Languages, and Applications & European Conference on Object-Oriented Programming*. Ottawa, Canada. ACM Press
- [38] A. Black, N. Schärli, and S. Ducasse. 2003. *Applying traits to the Smalltalk collection hierarchy*. In: *OOPSLA'03 - 17th International Conference on Object-Oriented Programming Systems, Languages and Applications*.
- [39] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala, Third Edition. A comprehensive step-by-step guide*. 2016.
- [40] O.-J. Dahl and K. Nygaard. 1978. *The Development of the SIMULA Languages*. In: *ACM SIGPLAN History of Programming Languages Conference*.
- [41] W.R. Cook, W. Hill, and P.S. Canning. 1990. *Inheritance is not subtyping*. In: *POPL '90 - 17th ACM*

- SIGPLAN-SIGACT symposium on Principles of programming languages*. San Francisco, California, US. ACM New York, NY, USA
- [42] B.H. Liskov and J.M. Wing. 1994. *A behavioral notion of subtyping*. ACM Trans. Program. Lang. Syst., 16(6): p. 1811–1841.
 - [43] A.P. Black, K.B. Bruce, and J. Noble. 2016. *The Essence of Inheritance*. In: *The -1th Workshop on New Object-Oriented Languages (NOOL)*. Amsterdam.
 - [44] O.L. Madsen and B. Møller-Pedersen. 2016. *The Essence of Subclassing*. In: *The -1th Workshop on New Object-Oriented Languages (NOOL)*. Amsterdam.
 - [45] W. Cook. 2012. *A Proposal for Simplified, Modern Definitions of "Object" and "Object Oriented"*. <https://wcook.blogspot.com/2012/07/proposal-for-simplified-modern.html>.
 - [46] B. Meyer. 2003. *The Outside-In method of teaching introductory programming*. <http://se.ethz.ch/~meyer/publications/teaching/teaching-ispj.pdf>.
 - [47] J.L. Knudsen and O.L. Madsen. 1988. *Teaching Object-Oriented Programming is more than teaching Object-Oriented Programming Languages*. In: *ECOOP '88 European Conference on Object-Oriented Programming Springer, 1988. s. 21-40 (Lecture Notes in Computer Science, Bind 322)*. Oslo, Norway. Springer, LNC 322.
 - [48] O.L. Madsen and B. Møller-Pedersen. 1988. *What Object-Oriented Programming May Be—and What It Does Not Have to Be*. In: *ECOOP'88 – European Conference on Object-Oriented Programming*. Oslo, Norway. Springer Verlag.
 - [49] O.L. Madsen. 2014. *Building Safe Concurrency Abstractions*, in *Concurrent Objects and Beyond*, G. Agha, et al., Editors. Springer, Berlin, Heidelberg.
 - [50] O.L. Madsen. 2021. *Using Coroutines for Multi-core Preemptive Scheduling*, in *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*. ACM. p. 46–52.
 - [51] O.L. Madsen and B. Møller-Pedersen. 1989. *Virtual Classes—A Powerful Mechanism in Object-Oriented Programming*. In: *OOPSLA'89 – Object-Oriented Programming, Systems Languages and Applications*. New Orleans, Louisiana. ACM Press.
 - [52] P. Brinch-Hansen. 1975. *The Programming Language Concurrent Pascal*. IEEE Transactions on Software Engineering, SE-1(2).
 - [53] O.L. Madsen and B. Møller-Pedersen. 2022. *Using supplementary properties to reduce the need for documentation*. In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Rhodes. Springer.
 - [54] X. Lee: Alan Kay on Object Oriented Programming. Interview with A. Kay. http://xahlee.info/comp/Alan_Kay_on_object_oriented_programing.html 2015
 - [55] O.L. Madsen and B. Møller-Pedersen. 1992. *Part Objects and Their Location*. In: *TOOLS'92: Technology of Object-Oriented Languages and Systems*. Dortmund. Prentice Hall.
 - [56] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. 1997. *Back to the future: the story of Squeak, a practical Smalltalk written in itself*. ACM SIGPLAN Notices, 32(10): p. 318–326.
 - [57] O.L. Madsen, B. Magnusson, and B. Møller-Pedersen. 1990. *Strong Typing of Object-Oriented Languages Revisited*. In: *Joint OOPSLA/ECOOP'90 – Conference on Object-Oriented Programming, Systems, Languages, and Applications & European Conference on Object-Oriented Programming*. Ottawa, Canada. ACM Press
 - [58] O.-J. Dahl and K. Nygaard. 1967. *Class and Subclass Declarations*, in *Simulation Programming Languages*, J.N. Buxton, Editor. North Holland, Amsterdam.
 - [59] A. Borning. 1986. *Classes versus Prototypes in Object-Oriented Languages*. In: *ACM/IEEE Fall Joint Computer Conference*. Dallas, Texas.
 - [60] H. Liebermann. 1986. *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*. In: *OOPSLA*. Portland, Oregon.
 - [61] O.L. Madsen. 2000. *Towards a Unified Programming Language*. In: *ECOOP'2000 – European Conference on Object-Oriented Programming*. Cannes. Springer Verlag.
 - [62] informIT: Interview with Programming Expert Bertrand Meyer. Interview with B. Meyer. 2001
 - [63] B. Stroustrup. 1994. *Design and Evolution of C++*. Addison-Wesley Professional.