Object-oriented programming: Some history, and challenges for the next fifty years

Andrew P. Black

Portland State University, Portland, Oregon, USA

Abstract

Object-oriented programming is inextricably bound up with the pioneering work of Ole-Johan Dahl and Kristen Nygaard on the design of the SIMULA language, which started at the Norwegian Computing Center in the Spring of 1961. However, object orientation, as we think of it today — fifty years later — is the result of a complex interplay of ideas, constraints and people. Dahl and Nygaard would certainly recognize it as their progeny, but might also be amazed at how much it has grown up.

This article is based on a lecture given on 22nd August 2011, on the occasion of the scientific opening of the Ole-Johan Dahl Hus at the University of Oslo. It looks at the foundational ideas from SIMULA that stand behind object-orientation, how those ideas have evolved to become the dominant programming paradigm, and what they have to offer as we approach the challenges of the next fifty years of informatics.

Keywords:

1. Introduction

On 22nd August 2011, a public event was scheduled to open both the 18th International Symposium on Fundamentals of Computation Theory and the Ole-Johan Dahl hus, the new building that is home to the University of Oslo's Department of Informatics, and which is shown in Figure 1. The morning

Email address: `black@cs.pdx.edu` (Andrew P. Black)
URL: `http://www.cs.pdx.edu/~black` (Andrew P. Black)

session opened with an Introduction by Morten Dæhlen, which was followed by two invited talks, one by Andrew Black and one by Jose Meseguer, and a discussion panel on the future of object-orientation and programming languages, which was chaired by Arne Maus, and comprised Andrew P. Black, Yuri Gurevich, Eric Jul, Stein Krogdahl, Jose Meseguer, and Olaf Owe.

As it happened, none of these events took place in the Dahl hus, because the beautiful lecture room that had been scheduled for the conference was put out of commission, less than half an hour before the start of the session, by an electrical fault: the scientific opening of the Dahl hus was actually conducted in the neighbouring Kristen Nygaard building. Thus, nine years after their deaths, Dahl and Nygaard were able to form a symbolic partnership to solve a pressing problem.

Figure 1: The Ole Johan Dahl Hus. (Photograph © the author)

This article is based on the invited talk given by the author. It is not a transcript, and I have taken the opportunity to elaborate on some themes and to précises others, to add references, and to tidy up some arguments that seemed, in hindsight, a bit too ragged to set down in print.

2. The Birth of Simula

In American usage, the word “drafted” has many related meanings. It can mean that you have been conscripted into military service, and it can mean that you have been given a job that is necessary, but that no one else wants to take on.

In 1948, Kristen Nygaard was drafted, in both of these senses. He started his conscript service at the Norwegian Defence Research Establishment, where his assignment was to carry out calculations related to the construction of Norway’s first nuclear reactor [1]. Years later, Nygaard recalled that he had no wish to be responsible for the first nuclear accident on the continent of Europe[1].

After extensive work on a traditional numerical approach, Nygaard turned to Monte Carol simulation methods, and first headed the "computing office" at the Defence Establishment, later (in 1952) turning full-time to operational research. He earned a Master of Science degree from the University of Oslo in 1956, with a thesis on probability theory entitled "Theoretical Aspects of Monte Carlo Methods" [2]. In 1960, Nygaard moved to the Norwegian Computing Center (NCC), a semi-governmental research institute that had been established in 1958. His brief was to expand the NCC's research capabilities in computer science and operational research. He wrote "Many of the civilian tasks turned out to present the same kind of methodological problems [as his earlier military work]: the necessity of using simulation, the need of concepts and a language for system description, lack of tools for generating simulation programs" [1]. In 1961, he started designing a simulation language as a way of attacking those problems.

In January 1962, Nygaard wrote what has become a famous letter. It was addressed to the French operational research specialist Charles Salzmann. Nygaard wrote: "The status of the Simulation Language (Monte Carlo Compiler) is that I have rather clear ideas on how to describe queueing systems, and have developed concepts which I feel allow a reasonably easy description of large classes of situations. I believe that these results have some interest even isolated from the compiler, since the presently used ways of describing such systems are not very satisfactory. ...The work on the compiler could not start before the language was fairly well developed, but this stage seems now to have been reached. The expert programmer who is interested in this part of the job will meet me tomorrow. He has been rather optimistic during our previous meetings."

The "expert programmer" was of course Ole-Johan Dahl, shown in Figure 2, and widely recognized as Norway's foremost computer scientist. Along with Nygaard, Dahl produced the initial ideas for object-oriented programming, which is now the dominant style of programming for commercial and industrial applications. Dahl was made Commander of The Order of Saint Olav by the King of Norway in 2000; along with Nygaard he received the ACM Turing Award in 2001 for ideas fundamental to the emergence of object

[1]David Ungar, private communication.

oriented programming, through their design of the programming languages SIMULA I and SIMULA 67, and in 2002 he was awarded the IEEE's von Neumann medal, once again with Nygaard. Dahl died on 29th June 2002.

In this article, I will try to identify the core concepts embodied in Dahl and Nygaard's early languages, and see how these concepts have evolved in the fifty years that have passed since their invention. I will also hazard some guesses as to how they will adapt to the future.

3. History in Context

In seeking guidance for the hazardous process of predicting the future, I looked at another event that also took place in 1961: the Centennial Celebration of the Massachusetts Institute of Technology. Richard Feynman joined Sir John Cockcroft (known for splitting the atom), Rudolf Peierls (who first conceptualized a bomb based on U-235) and Chen Ning Yang (who received the 1957 Nobel Prize for his work on parity laws for elementary particles), to speak on "The Future in the Physical Sciences." While the other speakers contented themselves with predicting 10, or perhaps 25, years ahead, Feynman, who was not to receive his own Nobel Prize for another four years, decided to be really safe by predicting 1000 years ahead. He said "I do not think that you can read history without wondering what is the future of your own field, in a wider sense. I do not think that you can predict the future of physics alone [without] the context of the political and social world in which it lies." Feynman argued that in 1000 years the discovery of fundamental physical laws would have ended, but his argument was based on the social and political context in which physics must operate, rather than on any intrinsic property of physics itself [3].

Figure 2: Ole-Johan Dahl (Photograph courtesy of Stein Krogdahl)

If social and political context is ultimately what determines the future of science, we must start our exploration of the SIMULA languages by looking at the context that gave them birth. Nygaard's concern with modelling the phenomena associated with nuclear fission meant that SIMULA was designed as a process *description* language as well as a programming language. "When SIMULA I was put to practical work it turned out that to a large extent it was used as a system description language. A common attitude among its simulation users seemed to be: sometimes actual simulation runs on the computer provided useful information. The writing of the SIMULA program was almost always useful, since … it resulted in a better understanding of the system" [1]. Notice that modelling means that the actions and interactions *of the objects created by the program* model the actions and interactions of the real-world objects that they are designed to simulate. It is not the SIMULA code that models the real-world system, but the objects created by that code. The ability to see "through" the code to the objects that it creates seems to have been key to the success of SIMULA's designers.

Because of Nygaard's concern with modeling nuclear phenomena, SIMULA followed Algol 60 in providing what was then called "security": the language was designed to reduce the possibility of programming errors, and so that those errors that remained could be cheaply detected at run time [4]. A SIMULA program could never give rise to machine- or implementation-dependent effects, so the behavior of a program could be explained entirely in terms of the semantics of the programming language in which it was written [1].

Dahl took a more technical view, as he explained in his role as discussant at the first HOPL conference [5]. For Dahl, SIMULA's contribution was the generalization and liberalization of the Algol 60 block, which gave the new language

1. record structures (blocks with variable declarations but no statements);
2. procedural data abstraction (blocks with variable and procedure declarations, but liberated from the stack discipline so that they could outlast their callers);
3. processes (blocks that continue to execute after they have been detached from their callers);
4. prefixing of one "detached" block with the name of another; and
5. prefixing of an ordinary Algol-60–style in-line block with the name of another, which let the prefix block play the rôle of what Dahl called a context object.

This generalized block was the Simula "class"; nowadays item 4 would be called inheritance, and item 5 would be called packaging, or modularity.

This list of features might lead one to think that the real contribution of Simula was the class construct, and that object-oriented programming is nothing more than class-oriented programming. However, I don't believe that this is historically correct. Dahl himself wrote "I know that Simula has been criticized for perhaps having put too many things into that single basket of class. Maybe that is correct; I'm not sure myself. ... It was great fun to see how easily the block concept could be remodeled and used for all these purposes. It is quite possible, however, that it would have been wiser to introduce a few more specialized concepts, for instance, a "context" concept for the contextlike classes." [5]. I interpret these remarks as Dahl saying that the important contributions of Simula were the five individual features, and not the fact that they were all realized by a single piece of syntax. While semantic unification can lead to a true simplification in the design of a language, syntactic unification may be a mistake.

4. The Origin of Simula's Core Ideas

As the above quotations reveal, Dahl was inspired to create the Simula class by visualizing the *runtime representation* of an Algol 60 program. To those with sufficient vision, objects were already in existence inside every executing Algol program — they just needed to be freed from the "stack discipline". In 1972 Dahl wrote:

> In Algol 60, the rules of the language have been carefully designed to ensure that the lifetimes of block instances are nested, in the sense that those instances that are latest activated are the first to go out of existence. It is this feature that permits an Algol 60 implementation to take advantage of a stack as a method of dynamic storage allocation and relinquishment. But it has the disadvantage that a program which creates a new block instance can never interact with it as an object which exists and has attributes, since it has disappeared by the time the calling program regains control. Thus the calling program can observe only the results of the actions of the procedures it calls. Consequently, the operational aspects of a block are overemphasised; and algorithms (for example, matrix multiplication) are the only concepts that can be modelled. [6]

In Simula, Dahl made two changes to the Algol 60 block: small changes, but changes with far-reaching consequences. First, "a block instance is permitted to outlive its calling statement, and to remain in existence for as long as the program needs to refer to it" [6]. Second, object references are treated as data, which gives the program a way to refer to those block instances. As a consequence of these changes, a more general storage allocation mechanism than the stack is needed: a garbage collector is required to reclaim those areas of storage occupied by objects that can no longer be referenced by the running program. Dahl and Nygaard may not have been the first to see the possibilities of generalizing the Algol block, but they were the first to realize that the extra complexity of a garbage collector was a small price to pay for the wide range of concepts that could be expressed using blocks that outlive their calling creators. It was Dahl's creation of a storage management package "based on a two-dimensional free area list" that made possible the Simula class [1]. This package seems to have been completed by May 1963, the date of a preliminary presentation of the Simula Language. The Simula I compiler was completed in January 1965, and accepted by Univac, who had partially financed its development, a year later — see Figure 3.

The idea of the *Class Prefixing* — now called subclassing, or inheritance — came to Simula from the work of Tony Hoare on record handling [7]. Hoare had introduced the notion of a class of record instances; a class could be specialized into several subclasses, but the class and its subclasses had to be declared together, with the subclass definitions nested inside the class definition. Dahl and Nygaard realized that it would be useful if a class could be declared separately from its subclasses, so that it could be specialized separately for diverse purposes. However, nesting precluded such a sepa-

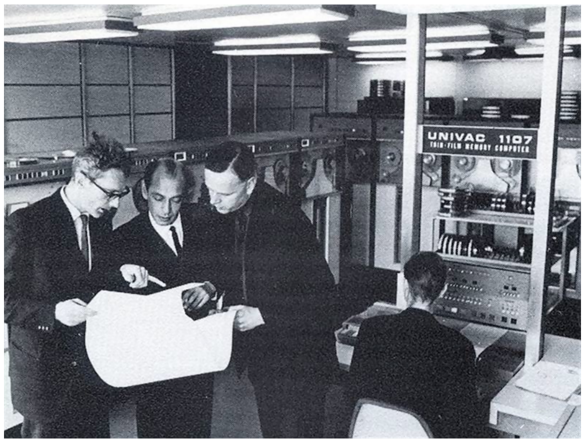

Figure 3: Dahl and Nygaard during the development of Simula, around 1962. The Univac 1107 was obtained by Nygaard for the NCC at a favorable price in exchange for a contract to implement Simula I. (Photograph from the Virtual Exhibition *People Behind Informatics*, http://cs-exhibitions.uni-klu.ac.at/)

ration. “The solution came with the idea of class prefixing: using C as a prefix to another class, the latter would be taken to be a subclass of C inheriting all properties of C.” [8]. The semantics of prefixing was defined as concatenation, but the syntactic separation of class and subclass turned the class of SIMULA 67 into a unit of reuse. This was so successful that the special-purpose simulation facilities of SIMULA were replaced in SIMULA 67 by a SIMULATION class, which was defined in SIMULA 67 itself.

Inheritance has become the *sine qua non* of object-orientation, indeed, an influential 1987 paper of Wegner’s defined “object-oriented” as “objects + classes + inheritance” [9]. Some of us objected to this definition at the time — it seemed more reasonable for the term “object-oriented” to imply only the use of objects — particularly since Wegner’s definition excluded any delegation-based language from being object-oriented. Wegner did admit that delegation, which he defined as “a mechanism that allows objects to delegate responsibility for performing an operation or finding a value to one or more designated ‘ancestors’ ”, was actually the purer feature. In any case, inheritance has stood the test of time as a useful feature. Since 1989, thanks to Cook, we have known that inheritance can be explained using fixpoints of generators of higher-order functions [10]. What this means is that, in theory, functions parameterized by functions are “as good as” inheritance, in the sense that they can enable equivalent reuse. However, they are not as good in practice, because, to enable reuse, the programmer has to plan ahead and make every part that could possibly change into a parameter.

Functional programmers call this process of parameterization “abstraction”. It has two problems: life is uncertain, and most people think better about the concrete than the abstract. Let’s examine those problems briefly. We have all experienced trying to plan ahead for change. Inevitably, when change comes, it is not the change we anticipated. We find that we have paid the price of generality and forward planning, but still have to modify the software to adapt to the change in requirements. Agile methodologies use this: they tell us to eschew generality, and instead make the software ready to embrace change by being changeable. Inheritance aids in this process by allowing us to write software in terms of the concrete, specific actions that the application needs now, and to abstract over them only when we know — because the change request is in our hands — that the abstraction is needed.

The second problem derives from the way that we think. There are people who are able to grasp abstractions directly, and then apply them: we call such people mathematicians. However, most of us can more easily grasp new ideas if we are first introduced to one or two concrete instances, and are then shown the generalization. Putting it another way: people learn from examples. That is, we might first solve a problem for $n = 4$, and *then* make the changes necessary for 4 to approach infinity. Inheritance uses this: it first provides a concrete example, and then generalizes from it.

To make this hand-waving argument a bit more concrete, let's look at an an example from the functional programming literature. In *Programming Erlang* [11, Chapter 16], Armstrong introduces the OTP (Open Telecom Platform) generic server. He comments: "This is the most important section of the entire book, so read it once, read it twice, read it 100 times — just make sure the message sinks in". To make sure that the message about the way that the OTP server works does sink in, Armstrong writes "four little servers ... each slightly different from the last. server1 runs some supplied code in a server, where it responds to remote requests; server2 makes each remote request an atomic transaction; server3 adds hot code swapping, and server4 provides both transactions and hot code swapping." Each of these fragments of sever code is self-contained: server4, for example, makes no reference to any of the preceding three servers.

Why does Armstrong describe the OTP sever in this way, rather than just presenting server4, which is his destination? Because he views server4 as too complicated for the reader to understand in one go (it is, after all, 31 lines long!) Something as complex as server4 needs to be introduced step-by-step. However, his language, lacking inheritance (and higher-order functions) does not provide a way of capturing this stepwise development.

In an effort to understand the OTP server, I coded it up in Smalltalk. As George Forsythe is reputed to have said: "People have said you don't understand something until you've taught it in a class. The truth is you don't understand something until you've taught it to a computer — until you've been able to program it." [12]. First I translated server1 into Smalltalk; I called it BasicServer, and it had three methods and 21 lines of code. Then I needed to test my code, so I wrote a name server "plug-in" for BasicServer, and set up unit tests, and made them run green. In the process, as Forsyth had predicted, I gained a *much* clearer understanding of how Armstrong's server1 worked. Thus equipped, I was able to implement TransactionServer by subclassing BasicServer, and HotSwapServer by subclassing TransactionServer, bringing me to something that was equivalent to Armstrong's server4 in two steps, each of which added just one new concern. Then I refactored the code to increase the commonality between TransactionServer and BasicServer, a commonality that had been obscured by following Armstrong in simply rewriting the whole of the main server loop to implement transactions. This refactoring added one method and one line of code.

Once I was done, I discussed what I had learned with Phil Wadler. Walder has been writing, and thinking deeply, about functional programming since the 1980s; amongst other influential articles he has authored "The Essence of Functional Programming" [13] and "Comprehending Monads" [14]. His first reaction was that the Erlang version was simpler because it could be described in straight-line code with no need for inheritance. I pointed out that I could refactor the Smalltalk version to remove the inheritance, simply by copying down all of the methods from the superclasses into HotSwapServer, but that doing so would be a *bad idea*. Why? Because the series of three classes, each building on its superclass, *explained* how HotSwapServer worked in much the same way that Armstrong explained it in Chapter 16 of *Programming Erlang*. I believe that this was an "ah-ha moment" for Phil.

To summarize: most people understand complex ideas incrementally. Properly used, code that uses inheritance can explain complex code incrementally, that is, in a way that is well-adapted to the way that people think.

5. Object-Oriented Frameworks

In my view, one of the most significant contributions of SIMULA 67 was the idea of the Object-Oriented Framework. By the 1960s, subroutine libraries were well-known. A subroutine, as its name suggests, is always subordinate to the main program. The relationship is always "don't call us, we'll call you". The only exception to this is when a user-written subroutine p is passed as an argument to a library subroutine q; in this case, q can indeed call p. However, in the languages of the time, any such "callback" to p could occur only during the lifetime of the original call from the main program to q. An Object-Oriented framework allows subroutines, but also enables the reverse relationship: the "main program" is now a component that can be invoked by the framework.

This reversal of roles enabled the special-purpose simulation constructs of SIMULA I to be expressed as a framework within SIMULA 67. The programmer of a simulation wrote code that described the behavior of the individual objects in the system: the nuclear fuel rods, or the air masses, or the products and customers, depending on the domain being simulated. The simulation framework then *called* those user-defined objects. The framework was in control, but users were able to populate it with objects that achieved their diverse goals.

Of course, for this scheme to work, the user-provided objects called by the framework must have methods for all of the requests made of them. Providing these methods might be a lot of work, but the work is generally made manageable by having the user-provided objects *inherit* from a framework object that already understands the majority of these requests; all the user need do is override the inherited behaviour selectively.

One of the most common instantiations of this idea is the user-interface framework. In most object-oriented languages, objects that are to appear on the computer's screen are sent requests by the display framework. They must respond to enquires such as those that ask for their bounds on the screen, and to draw themselves within these bounds. This is normally achieved by having displayable objects subclass a framework-provided class, such as Squeak's Morph or java.awt.Component.

Dahl and Nygaard realized that these ideas could be used to provide the simulation features of SIMULA I. SIMULA 67 is a general-purpose language; its simulation features are provided by an object-oriented framework called SIMULATION. SIMULATION is itself implemented using a more primitive framework called SIMSET.

6. From Simula to Smalltalk

Smalltalk-72 was an early version of Smalltalk, used only within Xerox PARC. It was clearly inspired by SIMULA, and took from SIMULA the ideas of classes, objects, object references, and inheritance. It is described by Alan Kay in his paper on the Early History of Smalltalk [15].

Smalltalk-72 refined and explored the idea of objects as little computers, or, as Kay puts it: "a recursion on the notion of computer itself". Objects combined data with the operations on that data, in the same way that a computer combines a memory to store data with an arithmetic and logic unit to operate on the data.

However, Smalltalk-72 dropped some of SIMULA's key ideas, notably the idea that objects were also processes, and that classes, used as prefixes to inline blocks, were also modules. Smalltalk 80 made the same omissions; as a result, what we may call the "North-American School" of object-orientation views these features as relatively unimportant, and perhaps not really part of what it means to support objects.

By the late 1980s, objects had attracted significant industrial interest. May different object-oriented languages had been developed, differing in the weight that they gave to classes, objects, inheritance, and other features. In 1991, Alan Snyder of Hewlett Packard wrote an influential survey paper "The Essence of Objects" [16], which was later published in *IEEE Software* [17].

Unlike SIMULA and Smalltalk, Snyder's is a descriptive work, not a prescriptive one. He describes and classifies the features of contemporary object-oriented languages, including Smalltalk, C++, the Common Lisp Object System, and the Xerox Star, as well as a handful of other systems. In Snyder's view, the essential concepts were as follows:

- An object embodies an abstraction.
- Objects provide services.
- Clients issue requests for those services.
- Objects are encapsulated.
- Requests identify operations.
- Requests can identify objects.
- New Objects can be created.
- The same operation on distinct objects can have different implementations and observably different behaviour.
- Objects can be classified in terms of their services (interface hierarchy).
- Objects can share implementations.
  - Objects can share a common implementation (multiple instances).
  - Objects can share partial implementations (implementation inheritance or delegation).

We see that objects as processes and classes as modules are not on this list. Snyder does mention "Active Objects" as an "associated concept", that is, an idea "associated with the notion of objects, but not essential to it". The idea that classes can serve as modules does not appear at all. Table 1 compares the features of objects as viewed by SIMULA, Smalltalk and Snyder.

7. Objects as Abstractions

The word "Abstraction" is used in Informatics in many different senses. As I mentioned previously, it is used by functional programmers to mean

Table 1: Evolution of the features of an "object". Simula's idea of procedural encapsulation became more complete, both over time and as objects migrated across the Atlantic. However, other features of the Simula Class, such as Active Objects and Classes as Modules, have not persisted.

Feature	**Simula 67**	**Smalltalk 80**	**Snyder (1981)**
Procedural Abstraction	Attributes exposed	Attributes encapsulated	Objects characterized by services
Active Objects	Yes	No	"Associated Concept"
Dynamic object creation	Yes	Yes	Yes
Classes	Yes	Yes	Shared implementations
Inheritance	class prefixing	subclassing	"Shared partial implementations"
Overriding	Under control of superclass	Under control of subclass	Optional; delegation as alternative
Classes as Modules	Yes	No	No

"parameterisation". In the context of objects, I'm going to use the word abstraction to capture the idea that what matters about an object is its protocol: the set of messages that it understands, and the way that it behaves in response to those messages. Nowadays, this is sometimes also referred to as the object's interface. The key idea is that when we use an object, we "abstract away from" its internal structure; more simply, that the internal structure of an object is hidden from all other objects.

Abstraction, in this sense, is one of the key ideas behind objects, but it does not appear explicitly until Snyder's 1991 survey. Simula doesn't mention abstraction specifically; it speaks of instead of modelling, which captures the importance of interacting with the object, but not the information hiding aspect. Dahl remarks [8] "According to a comment in [the Simula user's manual, 1965] it was a pity that the variable attributes of a *Car* process could not be hidden away in a subblock", but this was not possible in Simula without also hiding the procedures that defined the Car's behaviour — and exposing those procedures was the whole reason for defining the Car.

Smalltalk 80 fixed this problem by making the *instance variables* that contain the representation of an object accessible only to the methods of that object. However, Dan Ingalls' sweeping 1981 *Byte* article "Design Principles behind Smalltalk" refers to abstraction only indirectly. Ingalls singles out *Classification* as the key idea embodied in the Smalltalk class. He writes:

> Classification is the objectification of *ness*ness. In other words, when a human sees a chair, the experience is taken both literally as "that very thing" and abstractly as "that chair-like thing". Such abstraction results from the marvelous ability of the mind to merge "similar" experience, and this abstraction manifests itself as another object in the mind, the Platonic chair or chair*ness*.

However, Smalltalk classes do not classify objects on the basis of their behavior, but by representation. It seems that the the idea of separating the internal (concrete) and external (abstract) view of an object was yet to mature. It can be glimpsed elsewhere in Ingall's article. After Classification, his next principle is "*Polymorphism*: A program should specify only the behavior of objects [that it uses], not their representation", and in the "Future Work" section, where he remarks that

> message protocols have not been formalized. The organization provides for protocols, but it is currently only a matter of style for protocols to be consistent from one class to another. This can be remedied easily by providing proper protocol objects that can be consistently shared. This will then allow formal typing of variables by protocol without losing the advantages of polymorphism.

Borning and Ingalls did indeed attempt just that [18], but it proved not to be as easy to get the details right as Ingalls had predicted. Still, Smalltalk as defined in 1980 did succeed in abstracting away from representation details: it never assumes that two objects that exhibit the same protocol must also have the same representation.

The related concept of *data abstraction* can be traced back to the 1970s. Landmarks in its evolution are of course Hoare's "Proof of Correctness of Data Representations" [19], and Parnas' "On the Criteria to be used in Decomposing Systems into Modules" [20], both published in 1972, and the CLU programming language, which provided explicit conversion between the concrete data representation seen by the implementation and the abstract view seen by the client [21]. However, none of these papers pursues the idea that multiple representations of a single abstraction could co-exist. They assume that abstraction is either a personal discipline backed by training the programmer, or a linguistic discipline backed by types. The latter view — that abstraction comes from types — has been heavily marketed by the proponents of ML and Haskell.

Objects follow a different route to abstraction, which can be traced back to Alonzo Church's work on representing numbers in the lambda-calculus [22]. In 1975 John Reynolds named it "procedural abstraction" and showed how it could be used to support data abstraction in a programming language [23], as well as contrasting it with the approach based on types. Procedural abstraction is characterized by using a *computational* mechanism — a representation of a function as executable code — rather than type discipline or self-discipline — to enforce abstraction boundaries. Reynolds writes:

> Procedural data structures provide a decentralized form of data abstraction. Each part of the program which creates procedural data will specify its own form of representation, independently of the representations used elsewhere for the same kind of data, and will provide versions of the primitive operations (the components of the procedural data item) suitable for this representation. There need be no part of the program, corresponding to a type definition, in which all forms of representation for the same kind of data are known.

Unfortunately, neither this paper nor a later version [24] are widely known, and many practitioners do not understand the fundamental distinction between type abstraction and procedural abstraction. Why is this distinction important? Because there is a trade-off to be made when choosing between the two approaches. Procedural abstraction provides secure data abstraction without types, and allows multiple implementations of the same abstraction to co-exist; this supports change, and helps to keep software soft. The cost is that it is harder to program "binary operations" — those that work on two or more objects — with maximal efficiency.

William Cook made another attempt to explain the difference in a 2009 OOPSLA Essay [25]. Cook coined the term "Autognostic", meaning "self-knowing", for what Reynolds called "decentralization": an object can have detailed knowledge only of itself. All other objects are abstract. Cook remarks: "The converse is quite useful: any programming model that allows inspection of the representation of more than one abstraction at a time is not object-oriented."

The idea of procedural data abstraction was certainly in SIMULA I — indeed, I will argue that it was the genesis of SIMULA I. As Dahl and Hoare observe in "Hierarchical Program Structures" [6], the key concept is already present in Algol 60: an Algol 60 block is a single mechanism that can contain both procedures and data. The limitation of Algol is that blocks are constrained not to outlive their callers. Dahl saw this, and "set blocks free" by devising a dynamic storage allocation scheme to replace Algol 60's stack [1].

SIMULA's class construct can be used to generate both records (unprotected, or protected by type abstraction) and objects (protected by procedural abstraction). The same is true for some later languages, for example, C++.

The encapsulation of objects in SIMULA derives directly from the encapsulation offered by the Algol 60 block. Just as code in one block cannot access variables declared in another block (unless it is nested inside), the code inside one object cannot access the variables declared inside another. While SIMULA introduced the **inspect** statement specifically to expose those variables, Smalltalk made them inaccessible to all other objects. Indeed, the modern view is that it is only the *behaviour* of an object that matters: whether or not that behaviour is ultimately supported by methods or variables is one of the hidden implementation details of the object. Thus, while the approach to data abstraction from types makes a fundamental distinction between a representation — a value protected by an existential type — and the functions that operate on it, the procedural approach requires *only* methods: there may be no representation variables at all.

Perhaps the clearest example of this is the implementation of the Booleans in Smalltalk, which is outlined in Figure 4. The objects true and false implement methods &, |, not, ifTrue:ifFalse:, *etc.* (The colon introduces an argument, in the same way that parentheses are used to indicate arguments in other languages, and ↑ means "return from this method with the following answer".) For example, the & method of true answers its argument, while the | method always answers self, *i.e.*, true.

The ifTrue:ifFalse: methods are a bit more complicated, because the arguments must be evaluated conditionally. The ifTrue:ifFalse: method of false ignores its first argument, evaluates its second, and returns the result, while

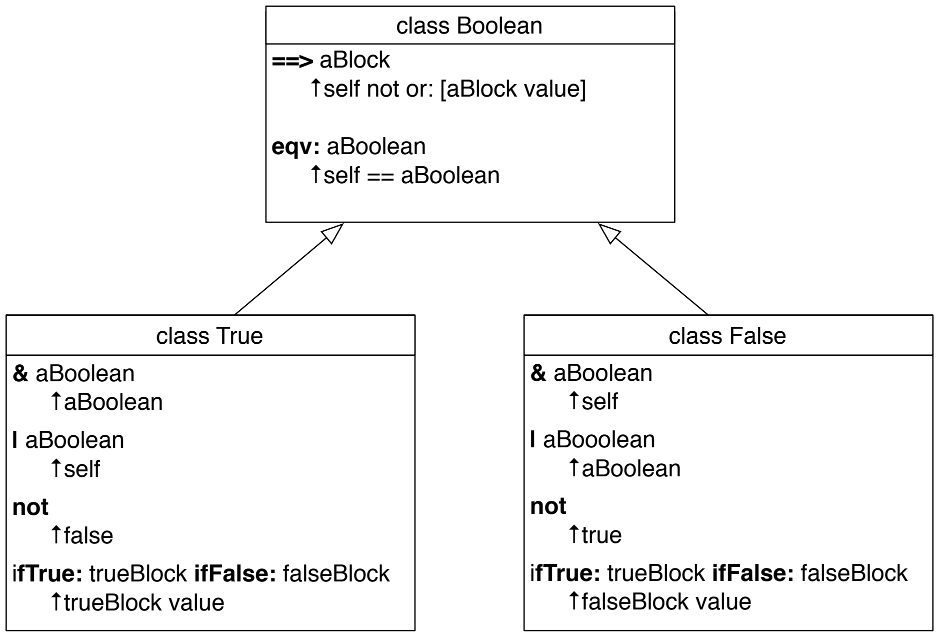

Figure 4: An extract from the Boolean classes in Smalltalk. Boolean is an abstract class with no instances; true and false are the only instances of the classes True and False, which inherit from Boolean. There is no "hidden state"; the methods themselves are sufficient to define the expected behavior of the Booleans.

the ifTrue:ifFalse: method of true does the reverse.

A more complicated example is a selection of objects implementing numbers. The objects 1 and 2 might include a representation as a machine integer, and methods $+$, $-$, *etc.*, that eventually use hardware arithmetic. However, the object representing 2^{67} could use a three-digit representation in radix 2^{32}, and the methods for $+$ and $-$ would then perform multi-digit arithmetic. This scheme, augmented with operations to convert machine integers to the multi-digit representation, is essentially how Smalltalk implements numbers. Notice that new number objects can be added at any time, without having to change existing code, so long as they provide the necessary conversion methods. Because the representation of an object can never be observed by any other object, there is no need to protect it with an existential type.

8. Active Objects

Active objects seem to be one idea from SIMULA that has become lost to the object-oriented community. Activity was an important part of SIMULA; after all, the original purpose of the language was to simulate activities that from the real world. SIMULA's "quasi-parallelism" was a sweet-spot in 1961: it allowed programmers to think about concurrency while ignoring synchronization. Because another task could execute only when explicitly resumed, programmers could be confident that their data would not change "out from under them" at unexpected times.

Hewitt's Actor model [26] built on this idea, as did Emerald [27], in which every object could potentially contain a process. However, activity has gone from "mainstream" object-oriented languages. Since Smalltalk, processes and objects have been independent concepts.

Why are Smalltalk objects passive? I don't know. Perhaps Kay and Ingalls had a philosophical objection to combining what they saw as separate ideas. Perhaps the realities of programming on the Alto set limits as to what was possible at that time. Perhaps they wanted real processes, not co-routines; this requires explicit synchronization, and thus a significantly different language design. Smalltalk does indeed have processes, rather than coroutines. However, they are realized as separate objects, rather than being a part of every object.

9. Classes and Objects

It's interesting to revisit Wegner's idea, discussed in Section 4, that object-orientation is characterized by the existence of classes. As we have seen, historically, most of the concepts that we recognize as making up object-orientation came from the class concept of SIMULA. However, as both Dahl and Nygaard would be quick to point out, it's the dynamic objects, not the classes, that form the system model, and modeling was the *raison d'être* of SIMULA: classes were interesting only as a way of creating the dynamic system model, which comprised interacting objects. They are indeed a most convenient tool if you want hundreds of similar objects. But what if you want just one object? In that case they impose a conceptual overhead. The fact is that classes are meta, and meta isn't always better!

Classes are meta, relative to objects, because a class is a factory that describes how to make an object, and describes the implementation and behavior of those embryonic objects. If you need just one or two objects, it's simpler and more direct to describe those objects in the program directly, rather than to describe a factory (that is, a class) to make them, and then use it once or twice. This is the idea behind Self, Emerald, NewtonScript and JavaScript, which take the object, rather than the class, as their fundamental concept. Which is not to say that classes are not useful: just that, at least in their role of object factories, they can be represented as objects.

In class-based languages, classes play many roles. Borning, in an early paper contrasting class-based and object-based languages [28], lists eight. In addition to creating objects, Smalltalk classes describe the representation of those objects and define their message protocol, serve as repositories for the methods that implement that protocol, provide a taxonomy for classifying objects, and, of course, serve as the unit of inheritance. However, as we saw when discussing Ingalls' view of classification (Section 7 above), this taxonomy, being based on implementation rather than behaviour, is fundamentally at odds with the autognosis that other researchers, such as Cook, believe to be fundamental to object-orientation.

If the only thing that we can know about an object is its behaviour, the most useful way of classifying objects must be according to their behaviour. This brings us back to types. Recall that types are not needed to provide abstraction in object-oriented systems. The true role of types is to provide a *behavioural* taxonomy: types provide a way of classifying objects so that programmers can better understand what roles a parameter is expected to play, or what can be requested of an answer returned from some "foreign" code.

Why is such a taxonomy useful? To add redundancy. A type annotation is an assertion about the value of a variable, no different in concept from asserting that a collection is empty or that an object is not `nil`. Redundancy is in general a good thing: it provides additional information for readers, and it means that more errors — at least those errors that manifest themselves as an inconsistency between the type annotation and the code — can be detected sooner. Nevertheless, I'm about to argue that types can be harmful.

On the surface, this is a ridiculous argument. If types add redundancy, and redundancy is good, how can types be harmful? The problem is not that adding type annotations will mess up your program, but that adding types to a language can, unless one is very careful, mess up the *language design*.

I've been an advocate of types for many years. The Emerald type system [29, 30] was one of the ways in which the Emerald project attempted to improve on Smalltalk. I spent many years working on type-checking and type systems, but have only recently really understood the sage advice that I was given by Butler Lampson in the late 1980s: stay away from types — they are just too difficult.

The difficulty springs from two sources. One is Gödel's incompleteness theorem, which tells us that there *will be* true facts about any (sufficiently rich) formal system that cannot be proved, no matter what system of proof we adopt. In this case, the formal system is our programming language, and the system of proof is that language's type system. The other source is our very natural desire to *know* that our program won't go wrong, or at least that it won't go wrong in certain circumscribed ways. This has led most statically-typed languages to adopt an interpretation of type-checking that I am going to refer to as "The Wilson interpretation". The name honors Harold Wilson, prime minister of the UK 1964–70 and 1974–76. Wilson fostered what has been called "the Nanny State", a social and political apparatus that said the government will look after you: if it even remotely possible that something will go wrong, we won't even let you try. The Wilson interpretation of type-checking embraces two tenets: first, that the type system must be complete, that is, every type assertion that can be made in it must be provably true or false, and second, that every part of the program must be type-checked before it can be executed. The consequence is that if it is even remotely possible that something may go wrong when executing your program, then the language implementation won't even let you try to run it — assuming that the "something" that might go wrong is a thing over which the type system believes it has control.

The Wilson interpretation is not the only interpretation of type-checking. At the other extreme is what I will call the "Bush interpretation", honoring George W. Bush, president of the USA 2001 — 2009, who abolished many government regulations and weakened the enforcement of those that remained. The Bush interpretation of type checking is that type information is advisory. The program should be allowed to do what you, the programmer, said that it should do. The type-checker won't stop it; if you mess up, the PDIC — the Program Debugger and Interactive Checker — will bail you out. The Bush interpretation amounts to not having static type-checking at all: the programmer is allowed to write type annotations in the program, but they won't be checked until runtime.

There is a third interpretation of type-checking that lies between these extremes. This might be called the "Proceed with caution" approach. If the type-checker has been unable to prove that there are no type errors in your program, you are given a warning. “It may work; it may give you a run-time error. Good night, and good luck.” I’m going to name this interpretation in honour of Edward R. Murrow, 1908–1965, an American broadcaster, whose signature line that was.

I’m referencing to Wilson, Murrow, and Bush as interpretations of type-checking, because I’m taking a decidedly “Churchian” view of types: the semantics of the program, in the absence of errors, is independent of the type system. The rôle of the type system does is to control how and when errors are reported. There is an alternative view in which programs that don’t type-check *have no semantics*; with this view, Wilson, Murrow, and Bush define different languages, connected by a superset relation. Since, as we have seen, types are not necessary to define object-based abstraction, I feel that the Churchian view is simpler and more appropriate, but the comparison of these views is a topic worthy of its own essay, and I’m not going to discuss it further here.

So, under all three interpretations, an error-free program has the same meaning. Under Wilson, we have conventional static typing. An erroneous program will result in a static error, and won’t be permitted to run. Because of Gödel, this means that some error-free programs won’t be permitted to run either, no matter how complex the type system becomes.

Under Bush, we have conventional dynamic typing. All checks will be performed at runtime — even those that are guaranteed to fail. Enthusiasts for dynamic typing point out that a counter-example is often more useful than a type-error message. The disadvantage is that the programmer will not receive any static warnings. Under the Murrow interpretation, the programmer will get a mix of compile-time warnings and run-time checks. A Murrow warning may say that a certain construct is provably a type error, or it may say that the type-checker has been unable to prove that it is type safe. The difference between these two warnings is dramatic, but Wilson treats them as if they were the same, confounding a programming error with a deficiency in the type system.

Let me say here that I’m for Murrow! I believe that the Murrow interpretation of types is the most useful, not only for programmers, but also for language designers. Wilson’s “Nanny Statism” is an invitation to mess up your language design. The temptation for the language designer is to exclude any construct that can’t be statically checked, regardless of how useful it may be.

I believe that SIMULA was for Murrow too! Recall that, according to Nygaard, the core Ideas of SIMULA were first modelling, and second, security. Modelling meant that the actions and interactions of the objects created by the program model the actions and interactions of the real-world objects that they were designed to simulate. Security meant that the behavior of a program could be understood and explained entirely in terms of the semantics of the programming language in which it is written. Modelling came first! SIMULA did not compromise its modelling ability to achieve security; it compromised its run-time performance, by incorporating explicit checks when a construct necessary for modelling was not statically provable to be safe.

I feel that many current language designers are suffering from a Wilson obsession. This obsession has resulted in type systems of overwhelming complexity, and languages that are larger, less regular, and less expressive than they need to be. The fallacy is the assumption that, because type checking is good, *more* type-checking is necessarily better.

Let's consider an example of how the insistence on static type-checking can mess a language design. I'm currently engaged, with Kim Bruce and James Noble, in the design of a language for student programmers. The language is called Grace, both to honour Rear Admiral Grace Hopper, and because we hope that it will enable the *Graceful* expression of algorithms. Grace is an object-oriented language, and contains an inheritance facility, based on early proposals by Antero Taivalsaari [31]. Here is an example, using a syntax invented for this article that is intended to need little explanation.

```
class Dictionary .new {  initialSize  →
    extends Hashtable.new( initialSize )
    method findIndex (predicate) override { ...}
    method at (key) put (value)  { ...} ...
}
```

The idea is that the object created by the execution of Dictionary.new(10) will have (copies of) all of the methods of the object Hashtable.new(10), except that the given method for findIndex() will override that inherited from Hashtable, and the given method for at()put() will be added.

Static checks that one might expect to take place on such a definition would include that Hashtable actually has a method new, and that the object answered by that method have a findIndex() method and *not* have an at()put() method. There is no problem with that expectation so long as Hashtable is a globally known class. However, suppose that I want to let the client choose the actual object that is extended, in other words, suppose that I make the super-object a parameter.

```
class Dictionary.new { superObj ,  initialSize  →
  extends superObj
  method findIndex (predicate) override { ...}
  method at (key) put (value)  { ...} ...
}
```

Although the same implementation mechanisms will still work, static checking is no longer simple. What arguments may be substituted for superObj? The requirement that superObj have a (possibly private) method findIndex and that it *not* have a method at()put() cannot be captured by the usual notion of type, which holds only the information necessary to *use* an object, not the information necessary to inherit from it. Type-checking this definition in a modular way seems to require a new notion of "heir types", that is, types that capture the information needed to check inheritance. Another possible solution is to make classes a new sort of entity, different from objects, and to give them their own, non-Turning complete sublanguage, including class-specific function, parameter and choice mechanisms. A third possibility is to ban parametric super-objects. How then does one use inheritance, that is, how does the programmer gain access to a superclass to extend? The classical solution is to refer to classes by global variables, and to require that the superclass be globally known. This is a premature commitment that runs in opposition to the late-binding that otherwise characterizes object-orientation; its effect is to reduce reusability. Some languages alleviate this problem by introducing yet another feature, known as open classes, which allows the programmer to add methods to, or override methods in, an existing globally-known class.

Regardless of the path chosen, the resulting language is both larger and less-expressive than the simple parametric scheme sketched above. Indeed, in the scheme suggested by the Dictionary example, a new object could extend *any* existing object, and classes did not need to be "special" in any way; they were ordinary objects that did not require any dedicated mechanisms for their creation, categorization or use.

Virtual classes, as found in BETA, are another approach to this problem [32]. Virtual classes feature co-variant methods — methods whose arguments are specialized along with their results. This means that they cannot be guaranteed to be safe by a modular static analysis. Nevertheless, they are useful for modelling real systems.

Another example of this sort of problem is collections that are parameterized by types. If types are desirable, it certainly seems reasonable to give programmers the opportunity to parameterize their collection objects, so that a library implementor can offer Bags of Employees as well as Bags of Numbers. The obvious solution is to represent types as objects, and use the normal method and parameter mechanisms to parameterize collections — which is exactly what we did in Emerald [33]. Unfortunately, the consequence of this is that type checking is no longer decidable [34]. The reaction of the Wilsonian faction to this is to recoil in shock and horror. After all, if the mission of types is to prevent the execution of any program that might possibly go wrong, and the very act of deciding whether the program will go wrong might not terminate, what is the language implementor to do?

However, a language that takes the Murrow interpretation of type can react to the possibility that type-checking is statically undecidable more pragmatically. The language implementation will in any case need perform run-time type checks, to deal with situations in which the program cannot be statically guaranteed to be free of type errors. So, if a type assertion cannot be proved true or false after a reasonable amount of effort, it suffices to insert a run-time check.

The alternative way of parameterising objects by types is to invent a new parameter passing mechanism for types, with new syntax and semantics, but with restricted expressivity to ensure decidability. Nevertheless, because of Gödel's first incompleteness theorem, some programs will still be un-typeable. The consequence is that the language becomes larger, while at the same time less expressive.

10. The Future of Objects

Now I'm going to turn to some speculations about the programming languages of the future. I could follow Feynman, and predict, with a certain confidence, that in 1000 years object-oriented programming will no longer exist as we know it. This is either because our civilization will have collapsed, or because it has not; in the latter case humans will no longer be engaged in any activity resembling programming, which will instead be something that computers do for themselves.

However, I think that it is probably more useful, and certainly more within my capabilities, to look 10 or 20 years ahead. The changes in computing that will challenge our ability to write programs over that time span are already clear: the trend from multicore towards manycore, the need for energy-efficiency, the growth of mobility and "computing in the cloud", the demand for increased reliability and failure resilience, and the emergence of distributed software development teams.

10.1. Multicore and Manycore

Although the exact number of "cores", that is, processing units, that will be present on the computer chips of 2021–31 is unknown, we can predict that with current technology trends we will be able to provide at least thousands, and perhaps hundreds of thousands. It also seems clear that manycore chips will be much more heterogeneous than the two- and four-processor multicore chips of today, exactly because not all algorithms can take advantage of thousands of small, slow but power-efficient cores, and will need larger, more power-hungry cores to perform acceptably. The degree of parallelism that will be available on commodity chips will thus depend both on the imperatives of electronic design and on whether we can solve the problem of writing programs that use many small cores effectively.

What do objects have to offer us in the era of manycore? The answer seems obvious. Processes that interact through messages, which are the essence of SIMULA's view of computation, are exactly what populate a manycore chip. The similarity becomes even clearer if we recall Kay's vision of objects as little computers. Objects are also offer us heterogeneity. There are many different classes of objects interacting in a typical program, and there is no reason that these different classes of objects need be coded in the same programming language, or compiled to the same instruction set. This is because the "encapsulation boundary" around each object means that no other object need know about its implementation. For example, objects that perform matrix arithmetic to calculate an image might be compiled to run on a general-purpose graphical processing unit, while other parts of the same application might be compiled to more conventional architectures.

Objects can also provide us with a "Cost Model" for manycore, that is, a way of thinking about computation that is closely-enough aligned with the real costs of execution that it can help us reason about performance. Most current computing models date from the 1950s and treat computation as the expensive resource — which was the case when those models were developed, and computation made use of expensive valve or discrete transistor logic. Data movement, in contrast, is regarded as free, since it was implemented by copper wires. As a consequence, these models lead us to think, for example, of moving an operation out of a loop and caching the result in memory as an "optimization". The reality today is very different: computation is essentially free, because it happens "in the cracks" between data fetch and data store. In contrast, data movement is expensive, in that it uses both time and energy. Today's programming languages are unable to even express the thing that needs to be carefully controlled on a manycore chip: data movement.

I would like to propose an object-oriented cost model for manycore computing. To the traditional object model we need add only two things: each object has a size and a spatial location. The size of an object is important because it needs to fit into the cache memory available at its location. "Size" includes not just the data inside the object, but also the code that makes up its method suite. Local operations on an object can be regarded as free, since they require only local computations on local data. Optimizing an object means reducing its size until it fits into the available cache, or, if this is not possible, partitioning the object into two or more smaller objects. In contrast, requesting that a method be evaluated in another object may be costly, since this requires sending a message to another location. The cost of a message is proportional to the product of the distance to the receiver and the amount of data; we can imagine measuring it in byte nanometers. The cost of a whole computation can be estimated as proportional to the number of messages sent and the cost of each message. We can reduce this cost by relocating objects so that frequent communication partners are close together, and by recomputing answers locally rather than requesting them from a remote location. Such a model does not, of course, capture all of the costs of a real computation, but it seems to me that it will provide a better guide for optimization than models from the 1950s.

10.2. Mobility and the Cloud

The world of mobile computing is a world in which communication is transient, failure is common, and replication and caching are the main techniques used to improve performance and availability. The best models for accessing objects in such an environment seem to me to those used for distributed version control, as realized in systems like subversion and git [35, 36]. In such systems all objects are immutable; updating an object creates a new version. Objects can be referred to in two ways: by version identifier, and by name. A version identifier is typically a large numeric string and refers to (a copy of) a particular immutable object. In contrast, a name is designed for human consumption, but the binding of a name to a version object changes over time: when a name is resolved, it is by default bound to a recent version of the object that it describes.

Erlang uses a similar naming mechanism to realise its fail-over facility. Erlang messages can be sent either to a process identifier, or to a process name. A particular process identifier will always refer to the same process; sending a message to a process identifier will fail if the process is no longer running. In contrast, a process name indirects through the name server, so messages sent to a process name will be delivered to the most recent process to register that name. When a process crashes and its monitor creates a replacement process, the replacement will usually register itself with the name sever under the same name. This ensures that messages sent to the process name will still be delivered, although successive messages may be delivered to different processes [11]. Perhaps a similar mechanism should be part of every object-oriented system? This would mean that we would be able to reference an object either by a descriptor, such as “Most recent version of the Oslo talk”, or by a unique identifier, such as Object16x45d023f. The former would resolve to one replica of a recent version; the latter would resolve to a specific object.

10.3. Reliability

Total failures are easy to handle, but will become increasingly rare as distributed and manycore systems become the norm. It is often possible to mask a failure using replication in time or space, but this is not always wise. Resending lost messages is not a good idea if they contain out-of date information; it may be better to send a new message containing fresh information. Similarly, it may be more cost-effective to recompute lost data than to retrieve it from a remote replica. Thus it seems to me that facilities for dealing with failures must be visible in the programming model, so that programmers can choose whether or not to use them. What, then, should the unit of failure be in an object-oriented model of computation? Is it the object, or is there some other unit? Whatever unit we choose, it must “leak failure”, in the sense that its clients must be able to see it fail and take action appropriate to the context.

10.4. Distributed Development Teams

With the ubiquity of computing equipment and the globalization of industry, distributed software development teams have become common. You may wonder what this trend has to do with objects. The answer is packaging: collaborating in loosely-knit teams demands better tools for packaging code and sharing it between the parts of a distributed team. Modules, which I regard as the unit of code sharing, are typically prameterised by other modules. The point is that module parameters should be bound at the time that a module is used, not when it is written. In other words, module parameters should be "late bound". This means that there is no need for a global namespace in the programming language itself; URLs or versioned objects provide a perfectly adequate mechanism for referring to the arguments of modules.

11. The Value of Dynamism

Before I conclude, I'm going to offer some speculations on the value of dynamism. These speculations are motivated by the fact that in designing SIMULA, Dahl recognized that the *runtime structures* of Algol 60 already contained the mechanisms that were necessary for simulation, and the recognition of Nygaard and Dahl that it is the *runtime behavior* of a simulation program that models the real world, not the program's text. I see a similar recognition of the value of dynamism in "Agile" software development, a methodology in which a program is developed in small increments, in close consultation with the customer. The idea is that (a primitive version of) the code runs at the end of the first week, and new functionality is added every week, under the guidance of the customer, who determines which functions have the greatest value. Extensive test suites make sure that the old functionality continues to work while new functionality is added.

If you have never tried such a methodology, you may wonder how it could possibly work! After all, isn't it important to *design* the program? The answer is yes: design is important. In fact, it is *so* important, that agile practitioners don't do it only at the start of the software creation process, when they know nothing about the program. Instead, they design every day: the program is continuously re-designed as the programmers learn from the code, and from the behavior of the running system.

The observation that I wish to make about the Agile design process is that the program's run-time behavior is a powerful teaching tool. This is obvious if you view programs as existing to control computers, but perhaps less obvious if you view programs as static descriptions of a system, like a set of mathematical equations. However, as Dahl and Nygaard discovered, it is a program's behavior, not the program itself, that models the real-world system. The program's text is a meta-description of the program behavior, and it is not always easy to infer the behavior from the meta-description.

In my own practice as a teacher of object-oriented programming, I know that I have succeeded when students anthropomorphize their objects, that is, when they turn to their partners and start to speak of one object asking another object to do something. I have found that his happens more often and more quickly when I teach with Smalltalk than when I teach with Java: Smalltalk programmers tend to talk about objects, while Java programmers tend to talk about classes. I suspect that this is because Smalltalk is the more dynamic language: many features of the language and the programing environment help the programmer to interact with objects, rather than with code. Indeed, I am tempted to define a "Dynamic Programming Language" as one designed to help the programmer learn from the run-time behavior of the program.

12. Summary

Fifty years ago, Dahl and Nygaard had some profound insights about the nature of both computation and human understanding. "Modelling the world" is not only a powerful technique for designing and re-designing a computer program: it is also one of the most effective ways yet found of communicating that design to other humans.

What are the major concepts of object-orientation? The answer depends on the social and political context. Dahl listed five, including the use of objects as record structures and as procedurally-encapsulated data abstractions, and the use of inheritance for incremental definition. Some of what he though were key ideas, such as active objects and objects as modules, have been neglected over the last 30 years, but may yet be revived as the context in which we program becomes increasingly distributed and heterogeneous, in terms of both execution platform and programming team. It certainly seems to me that, after 50 years, there are still ideas in SIMULA that can be mined to solve twenty-first century problems. There may not be any programming a thousand years from now, but I'm willing to wager that some form of Dahl's ideas will still be familiar to programmers in fifty years, when SIMULA celebrates its centenary.

Acknowledgements

I thank Stein Krogdahl, Olaf Owe and the committee of FCT11 for honouring me with the invitation to speak at the scientific opening of the Ole-Johan Dahl hus, as well as supplying me with information about Dahl and Simula. I also thank Olaf for convincing me to turn my lecture into this article. William Cook was an encouraging and through critic; Ivan Sutherland helped me understand where the costs lie in modern processors, and provided useful feedback on a draft of my manuscript. David Ungar shared many insights with me and helped to improve both my lecture and this article.

References

[1] K. Nygaard, O.-J. Dahl, The development of the SIMULA languages, in: R. L. Wexelblat (Ed.), History of programming languages I, ACM, New York, NY, USA, 1981, pp. 439–480.

[2] University of Oslo, Department of Informatics, Kristen Nygaard — Education and Career, 2011. Web page last visited 25 December 2011. http://www.mn.uio.no/ifi/english/about/kristen-nygaard/career/.

[3] R. Feynman, Speech at centenial celebration of Massachusetts Institute of Technology (version C), 1961. The Feynman Archives at California Institute of Technology.

[4] C. Hoare, Hints on Programming Language Design, Memo AIM-224, Stanford Artificial Intelligence Laboratory, 1973. Invited address, 1st POPL conference.

[5] O.-J. Dahl, Transcript of discussant's remarks, in: R. L. Wexelblat (Ed.), History of programming languages I, ACM, New York, NY, USA, 1981, pp. 488–490.

[6] O.-J. Dahl, C. Hoare, Hierarchical program structures, in: Structured Programming, Academic Press, 1972, pp. 175–220.

[7] C. Hoare, Record handling, Lectures at the NATO summer school, 1966.

[8] O.-J. Dahl, The roots of object-oriented programming: the Simula language, in: M. Broy, E. Denert (Eds.), Software Pioneers: Contributions to Software Engineering, Springer-Verlag, Berlin, Heidelberg, 2002, pp. 79–90.

[9] P. Wegner, Dimensions of object-based language design, in: N. Meyrowitz (Ed.), Proceedings Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications, ACM Press, Orlando, Florida, 1987, pp. 168–182.

[10] W. Cook, J. Palsberg, A denotational semantics of inheritance and its correctness, in: Conference on Object-oriented programming systems, languages and applications, ACM Press, New Orleans, LA USA, 1989, pp. 433–443.

[11] J. Armstrong, Programming Erlang: Software for a concurrent world, Pragmatic Bookshelf, 2007.

[12] B. Archer, Programming quotations, 2011. Web page last visited 22 January 2012. http://www.bobarcher.org/software/programming_quotes.html.

[13] P. Wadler, The essence of functional programming, in: Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages, ACM Press, Albuquerque, NM, 1992, pp. 1–14.

[14] P. Wadler, Comprehending monads, Mathematical Structures in Computer Science 2 (1992) 461–493. Originally published in ACM Conference on Lisp and Functional Programming, June 1990.

[15] A. C. Kay, The early history of Smalltalk, in: The second ACM SIGPLAN conference on History of programming languages, HOPL-II, ACM, New York, NY, USA, 1993, pp. 511–598.

[16] A. Snyder, The Essence of Objects: Common Concepts and Terminology, Technical Report HPL-91-50, Hewlett Packard Laboratories, 1991.

[17] A. Snyder, The essence of objects: Concepts and terms, IEEE Softw. 10 (1993) 31–42.

[18] A. H. Borning, D. H. H. Ingalls, A type declaration and inference system for smalltalk, in: Conference Record of the Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, NM, USA, pp. 133–141.

[19] C. Hoare, Proof of correctness of data representations, Acta Informatica 1 (1972) 271–281.

[20] D. L. Parnas, On the criteria to be used in decomposing systems into modules, CACM 15 (1972) 1053–1058.

[21] B. Liskov, A. Snyder, R. Atkinson, C. Schaffert, Abstraction mechanisms in CLU, Comm. ACM 20 (1977) 564–576.

[22] A. Church, The Calculi of Lambda-Conversion, volume 6 of *Annals of Mathematical Studies*, Princeton University Press, 1941.

[23] J. C. Reynolds, User-defined types and procedural data structures as complementary approaches to data abstraction, in: Conference on New Directions in Algorithmic Languages, IFIP Working Group 2.1, Munich, Germany, p. 12.

[24] J. C. Reynolds, User defined types and procedural data structures as complementary approaches to data abstraction, in: D. Gries (Ed.), Programming Methodology, A Collection of Articles by IFIP WG2.3, Springer Verlag, 1978, pp. 309–317. Reprinted from S. A. Schuman (ed.), New Advances in Algorithmic Languages 1975 Inst. de Recherche d'Informatique et d'Automatique, Rocquencourt, 1975, pages 157-168.

[25] W. R. Cook, On understanding data abstraction, revisited, in: S. Arora, G. T. Leavens (Eds.), OOPSLA, ACM, 2009, pp. 557–572.

[26] C. Hewitt, P. Bishop, R. Steiger, A universal modular actor formalism for artificial intelligence, in: IJCAI, Stanford University, Stanford, California, pp. 235–245.

[27] A. Black, N. Hutchinson, E. Jul, H. Levy, Object structure in the Emerald system, in: Proceedings of the First ACM Conf. on Object-Oriented Programming Systems, Languages and Applications, volume 21, ACM, Portland, Oregon, pp. 78–86. Published as SIGPLAN Notices **21**(11), November 1986.

[28] A. Borning, Classes versus prototypes in object-oriented languages, in: FJCC, IEEE Computer Society, 1986, pp. 36–40.

[29] A. P. Black, N. Hutchinson, E. Jul, H. M. Levy, L. Carter, Distribution and abstract types in emerald, IEEE Tran Software Eng. SE-13 (1987) 65–76.

[30] A. P. Black, N. Hutchinson, Typechecking Polymorphism in Emerald, Technical Report Technical Report CRL 91/1 (Revised), Digital Equipment Corporation Cambridge Research Laboratory, 1991.

[31] A. Taivalsaari, Delegation versus concatenation, or cloning is inheritance too, SIGPLAN OOPS Mess. 6 (1995) 20–49.

[32] O. L. Madsen, An overview of beta, in: J. Knudsen, O. Madsen, B. Magnusson (Eds.), Object-Oriented Environments, Prentice Hall, Englewood Cliffs, NJ, 1993, pp. 99–118.

[33] A. P. Black, N. C. Hutchinson, E. Jul, H. M. Levy, The development of the Emerald programming language, in: B. G. Ryder, B. Hailpern (Eds.), HOPL III: Proceedings of the third ACM SIGPLAN conference on History of Programming Languages, ACM, San Diego, CA, 2007, pp. 11–1–11–51.

[34] A. R. Meyer, M. B. Reinhold, 'Type' is not a Type: Preliminary report, in: Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages, St Petersburg Beach, FL, USA, pp. 287–295.

[35] M. Mason, Pragmatic Version Control Using Subversion, Pragmatic Bookshelf, Pragmatic Programmers, 2005.

[36] C. Duan, Understanding Git: Repositories, 2010. Accessed on 13 Feb 2012.