

# Simple, eh?

Erik Ernst

Dept. of Computer Science, University of Aarhus, Denmark  
ernst@daimi.au.dk

**Abstract.** The aspect-oriented software development community has always shown great interest in language design, and simplicity is a very common criterion for the quality of such language design. However, we claim that simplicity is not so simple as it looks, and this paper proposes three different variants of simplicity in language design. Setting out from the recommended topic structure of papers for this workshop, we try to illustrate how the choice of variant heavily influences the effect of simplicity on the comprehensibility, predictability, and evolvability of programs.

## 1 Introduction

This paper intends to shed some light on the nature of simplicity in connection with programming language design. The reason why this is relevant is that simplicity is a very often mentioned criterion for good language design, and the aspect-oriented software development community has always shown great interest in programming language design. The reason why it is worthwhile to dissect simplicity as such is that it is deceptively complex. This paper hopes to bring a little bit more structure to discussions about simplicity in language design by promoting three different variants, each with different properties when it comes to the software engineering consequences.

The first variant is *hardware-simplicity*, by which we refer to the kind of language design that pushes the language closer to the hardware, such that compiler implementation and related tasks become simple(r). Our prime example of this is C++ [5] inheritance, and in particular the notion of ‘virtual base class’. The semantics of C++ is such that a superclass which is reachable via multiple paths in the inheritance hierarchy by default gives rise to multiple subobjects, i.e., data members from that class (or those classes) will exist in multiple, distinct copies. If, however, the class that would be duplicated is inherited virtually, then this duplication is avoided. People have different opinions on multiple inheritance, e.g., we think virtual inheritance is the well-defined mechanism and the default duplication mechanism in multiple inheritance is useful only for wrong modeling.

This is an example of hardware-simplicity because the presence or absence of the keyword `virtual` in a superclass clause directly determines whether the inherited fields are accessed by a fixed offset or via an object-internal pointer (a ‘vbptr’). This is not a required implementation strategy, but it is common and the language seems to have been designed to allow compiler writers to avoid the ‘vbptr’ except when programmers explicitly ask for it. This is also the logical choice in C++, where by design any run-time cost beyond that of a C program is avoided except where the programmer explicitly asks for it—for instance, a function is not virtual unless marked as such, and an object has no run-time type information (not even a ‘vtable’ pointer) unless it contains or inherits a virtual method.

The second variant is *syntax-simplicity*, by which we refer to the kind of language design that makes the language syntax simpler. This is a complex concept, because there are so many ways to do it, so we mention a few of them and end up with a prime example.

Problem being  
addressed by simplicity  
(aka motivation)

Smalltalk [2] is rightfully famous for having a simple grammar. This means that it is very easy to learn how to write Smalltalk programs at the syntactic level, but it also means that programmers must use many parentheses, and the numerical expression syntax is confusing because there are no precedence rules. This was, after all, not what we were looking for.

Ruby [1] is an interesting little scripting language whose syntax is designed in such a way that it is very convenient to use regular expressions and certain other entities, even though this convenient syntax is in fact interpreted in a quite consistent object-oriented manner. In other words, we get real objects everywhere, but it looks like raw regular expressions etc. In this case, the syntactic design is such that it becomes easier to write typical constructs, while less common needs must be satisfied using the more complex and tedious standard syntax. This is what we usually call ‘syntactic sugar’, an elegant exemplar even, but this is not the core of syntax-simplicity, either.

Self [6] is a very innovative language which has been promoted exactly for its radical simplicity, and in this language we find a very clearcut trade-off between syntax-simplicity and the last kind of simplicity, to be discussed below. The language Self is prototype based, i.e., all entities are objects (really) and there are no classes. New objects are created by cloning existing objects, and abstraction is achieved by means of delegation between objects. This is sufficient to emulate inheritance (single, multiple, even dynamic), sharing (like instance variables vs. class variables in Smalltalk, or `static` attributes in the Java [3] programming language), name space management (like packages or modules), etc.etc. Self has succeeded in providing a very simple set of concepts (object, slot, message send, delegation) that is powerful enough to support more mechanisms from other languages than most programmers may ever experience.

However, in Self, there are actually multiple kinds of objects! In particular, there are method objects and (ordinary) data objects. A data object evaluates to itself, but when a method object is evaluated, it is cloned, equipped with actual arguments, placed in context of the message receiver, and its code is executed. Because these two kinds of objects behave so differently, evaluating a data slot and calling a zero-argument method looks exactly the same, and writing to a data slot looks exactly the same as calling a method that takes one argument. It is even possible to override an inherited data slot with a method slot, or vice versa. In other words, in such a simplicity fanatic language design, the unlikely deviation of having multiple kinds of objects has been made, and the outcome is that it becomes invisible whether a given feature is stored or computed, to an extent which is not common in other languages. This representation transparency is an impressive piece of language design, and it shows the trade-off between syntax and semantics very clearly, exactly because it is such a deep and subtle difference. Here, the trade-off is to get more consistent (and simple) syntax, in return for a more complex set of entities.

The last kind of simplicity is concept-simplicity, by which we refer to the kind of language design where the semantics may be described by means of few concepts. Note that Self exhibits this kind of simplicity to an unusual degree. However, there is a language which is even more stubborn on reducing the number of language concepts, namely BETA [4]. In this language there is only one kind of objects, and both “ordinary objects” and method invocations are represented this way. The consequence is that it is easier than in Self to get hold of a “method object”, and execution of an object (think: running a special ‘default’ method) makes sense and is used for user-defined evaluation and assignment semantics, but on the other hand the representation transparency is less complete in some ways—in particular, it is not possible to override a method by a field or vice versa.

To summarize, we propose to focus on *hardware-simplicity*, i.e., pushing languages closer to the hardware such that compilers etc. become easier to implement;

*syntax-simplicity*, i.e., making deep choices in language design with the effect that the simplicity of the syntax that programmers must write is given priority over the simplicity of the semantic entities in terms of which the program is executed; and *concept-simplicity*, i.e., simplicity measured in the number of concepts needed in order to describe the semantics of the language, and to some extent the complexity of each of those concepts.

## 2 Does simplicity support comprehensibility?

Comprehensibility is concerned with the ability, subsequently ease, with which a subject can be understood by people. Complexity is a related concept, in that most people would probably accept the correlation: More complex = less comprehensible. However, the crucial point is that there are many different presentations of any given subject, and they differ radically in complexity. Based on this, it is tempting to assume that there is an *inherent* complexity for any given subject, and the best possible presentation would have that complexity, with all other presentations being (needlessly) more complex. Even though this is of course a very naïve description, it is usable enough to motivate the use of such an ‘inherent complexity’ concept in connection with software engineering projects, and the question is then how the three kinds of simplicity in language design affect the needless complexity overhead.

We believe that concept-simplicity is crucial for good comprehensibility, in the sense that it is easier for people to understand a complex subject when it is formulated in few, well-defined basic terms rather than in a plethora of more or less overlapping concepts and mechanisms. However, these few concepts must be powerful enough such that the system complexity can be organized well, i.e., there must be strong support for abstraction. Abstraction is *the* tool making it possible to build complex systems in such a way that the complexity that programmers need to manage is kept under control even when the raw amount of details gets much larger than what a human being can juggle at the same time. The syntax-simplicity seems to be a more local issue at first sight, in the sense that a somewhat more tedious or less consistent syntax makes the concrete program elements look different, but the large picture would not be affected. However, the simplicity of program *changes* may be highly influenced by the difference between just changing a few declarations, as opposed to having to change every usage site for those declarations, too, and this difference is exactly the consequence of a kind of syntactic simplicity, namely representation transparency. Finally, hardware-simplicity is all about letting human beings pay for the convenience of the hardware, so the effect of hardware-simplicity on comprehensibility would generally be detrimental.

All in all, we believe that simplicity in language design is absolutely crucial for the comprehensibility of software artefacts, but concept-simplicity and syntax-simplicity correlate positively and hardware-simplicity correlates negatively. Moreover, simplicity is not enough because the language must have powerful (and simple!) abstraction mechanisms, such that the complexity can be kept under control when the system scales up.

## 3 Does simplicity support predictability?

Predictability is an aspect of comprehensibility, because it is concerned with understanding the future of the behavior of some system. Since source code studies are all about predicting the behavior of the hardware when it runs the software, program understanding is inherently prediction oriented. However, we may seek out an aspect of predictability which is not already trivially covered by the previous case, namely predicting which kind of entity is denoted by a given language expression.

General properties, assumptions, hypotheses about comprehensibility.

How simplicity supports comprehensibility.

How simplicity reduces comprehensibility.

Conclusion about comprehensibility.

General properties, assumptions, hypotheses about predictability.

How simplicity supports predictability.

How simplicity reduces predictability.

In a language like Smalltalk or Self, where ‘everything is an object’ is taken seriously, language expressions denote objects. However, as we have seen, there are different kinds of objects in Self, and we may want to know whether a computation will take place at a given point or not. In this case the syntax-simplicity of Self, and in particular its representation transparency, makes it impossible to predict whether an expression is an object or a method. In BETA a similar problem may arise, because there is no difference between objects and method invocations. In this case it is not even possible at run-time and with a debugger to determine whether a “thing” is one or the other of the two. When the concepts are unified such that there *is* no difference between a method invocation and an object, then it stops being a question about predictions and starts being a non-question. Finally, most widely known languages including Java and C++ make this distinction explicit directly in the syntax (a method invocation is marked by `()`, as in `anObject.aMethod()`, and a field evaluation is not, as in `x.y`), and since this has been motivated with performance considerations (I’ve seen that kind of argument many times when I followed discussions on `comp.lang.c++.c++` during the standardization process), it might as well be described as an instance of hardware-simplicity.

Conclusion about predictability.

We have chosen to focus on predictability in a tiny subject area, namely whether it is possible to predict which kind of entity a given expression denotes, and it turns out that both syntax-simplicity and concept-simplicity may make it hard/impossible, whereas hardware-simplicity would probably lead to a design where it is trivial to detect the difference. This is interesting because it has the opposite direction of the other two iltities. However, we consider this as a case of healthy information hiding: Representational transparency hides which kind of entity is being used, but the design flexibility associated with the ability to change our mind on this question is much more valuable than the performance gains that could have been obtained by having programmers exploit this information in myriads of little ways all over the program. In fact, the system rigidity that such an optimization culture would cause is a very serious problem—which brings us to the next topic, evolvability.

## 4 Does simplicity support evolvability?

General properties, assumptions, hypotheses about evolvability.

Software system evolution is an activity that inevitably occurs with successful systems (other systems are discarded too soon for that...), and the historical developments all point in the direction of software evolution as a more and more crucial activity. In other words, simplicity had better be good for evolvability.

How simplicity supports evolvability.

How simplicity reduces evolvability.

The ability to change a software system in a way that affects the semantics in a desired manner is highly affected by the complexity of the changes themselves. In other words, if the desired semantic changes are *small* in some sense then it would be highly desirable if the syntactic changes needed (i.e., the amount of programming) is *also small*. This principle of proportionality is very hard to make precise, but we believe that the most important tool here is again abstraction. If the software contains much of its inherent complexity in terms of applications of abstractions, then modification of those abstractions is a powerful tool on the way to the desired system. We believe that concept-simplicity is crucial in order to achieve the most powerful abstraction mechanisms. Moreover, representation transparency may be achieved by syntax-simplicity, and this is also extremely critical during a software evolution process. As usual, hardware-simplicity is probably rather undividedly damaging.

Conclusion about evolvability.

To support evolvability is hard, but necessary. The syntax-simplicity of Self is an example of a design that enables a large category of changes in implementation of features without changing client code. In the more general case, we believe that concept-simplicity is required for good abstraction mechanisms, and abstraction is crucial in order to obtain the flexibility in a design that allows relevant and powerful

semantic changes to be made without having to perform a large number of complex syntactic changes.

## 5 Does simplicity cause semantic interactions?

Semantic interactions are inevitable and essential in software, starting with such a simple thing as an assignment statement which may affect the future behavior of code that has no connection with the assignment statement, but only shares access to the assigned variable's declaration. However, this is generally considered acceptable because of the natural expressive power of variables, but semantic interactions become dangerous when they are unexpected and complex, including the cases where they are associated with widely scattered locations in the source code and with seemingly unrelated entities. In such cases they radically ramp up the system complexity, because they make it incorrect for programmers to try to understand the system piece by piece, they have to try to grasp the whole thing in one go. We consider it an important quality criterion for abstraction mechanisms that they allow software systems to be constructed in such a way that dangerous semantic interactions are avoided or rare.

We believe that concept-simplicity is the primary source of help to avoid bad semantic interactions, especially if the semantics of a complex system is compositional, i.e., the semantics of the entire system can be described in terms of the semantics of its components. This would again be made possible by means of high-quality abstraction mechanisms. Once again abstraction plays the crucial role as *simplicity-builder*, the tool that enables system developers to construct complex systems that do not break down having all kinds of bad properties—in this case dangerous semantic interactions—as they scale up.

In summary, semantic interactions are a problem if they are unexpected and/or unwanted, especially if they are complex, and if they are associated with software artifacts or runtime entities that seem to have little to do with each other. We believe that the right kind of simplicity in the language design is the most important tool of them all in the effort to reduce such unwanted semantic interactions, and as usual good abstraction support enters the picture as the tool that lets us scale up systems without (too many) bad effects.

## 6 Conclusion

We have presented a division of the concept of language design simplicity into three different kinds, and discussed the impact of those variants of simplicity on the selected main 'ilities' of the workshop, namely comprehensibility, predictability, evolvability, and semantic interactions. The three kinds of simplicity in language design are hardware-simplicity, which is about making the relation between programs and hardware simpler and more direct; syntax-simplicity, which is about obtaining simpler and more consistent syntactic expressions of programs at the expense of a more involved set of semantic entities, and concept-simplicity, which is about having as few and simple concepts behind the language as possible. This discussion illustrates why it is important to distinguish between multiple different notions of simplicity when discussing whether or not a language design should have been simple(r) or not. Finally, high-quality support for abstraction kept turning up in all contexts, playing the role of the magic wand that keeps complexity under control as the system scales up—assuming, of course, that it is waved by a real wizard. :-)

General properties, assumptions, hypotheses about semantic interactions.

How simplicity causes semantic interactions.

How simplicity reduces semantic interactions.

Conclusion about semantic interactions.

## References

1. Hal Fulton. *The Ruby Way*. SAMS, 2001. ISBN 0672320835.
2. Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA, USA, 1989.
3. Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *Java(TM) Language Specification (2nd Edition)*. Addison-Wesley Publishing Company, 2000.
4. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, USA, 1993.
5. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.
6. David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87*, pages 227–242, Orlando, FL, October 1987.