Bad Engineering Properties of Object-Oriented Languages

Luca Cardelli

Digital Equipment Corporation, Systems Research Center
<http://www.research.digital.com/SRC/personal/Luca_Cardelli/home.html>

Abstract
Object-oriented languages dominate procedural languages in certain software-engineering categories, but not in others. Further progress may involve adapting and reintroducing principles that are already well understood and widely exploited in procedural languages.

The object-oriented paradigm emerged in the 60's, roughly during the time that important notions such as data abstraction, polymorphism, and modularization were applied to the procedural paradigm. Eventually, object-oriented languages also acquired notions of data abstraction, polymorphism, and modularization, but not quite in the same way, and not quite as effectively.

In the last decade, object-oriented languages have been widely adopted as engineering tools because of their superiority with respect to software extensibility, which is a critical engineering property [4]. A large and growing fraction of software engineering is now carried out in object-oriented languages, taking over the domain traditionally covered by procedural languages. However, object-oriented languages have not incorporated all the engineering insights that have been successfully deployed in procedural languages. For example, they have emphasized code reuse at the expense of modularization, and dynamic inspection at the expense of static detection. Therefore, opportunities remain for applying important ideas that hopefully will result in even stronger engineering advantages for object-oriented languages.

I will begin by considering some obvious engineering metrics. For comparison purposes, I will first recall how advances in language design have resulted in the past in engineering improvements, according to these metrics. Next, I will review how object-oriented languages have yet to show similar improvements in some of these areas. Finally, I will discuss what remains to be done to integrate certain good engineering principles into object-oriented languages.

Consider the following (informal) metrics:

- *Economy of execution*.
  How fast does a program run?
- *Economy of compilation*.
  How long does it take to go from sources to executables?
- *Economy of small-scale development*.
  How hard must an individual programmer work?

- *Economy of large-scale development*.
  How hard must a team of programmers work?
- *Economy of language features*.
  How hard is it to learn or use a programming language?

Let us consider first procedural languages. Over the course of many years, these languages acquired features that (more often than not) resulted in considerable improvements in engineering. This is of course not an accident: most language features were directly inspired or corroborated by engineering considerations. These engineering-friendly features include the notions of static typing, data abstraction, modularization, and parameterization, which resulted in the following improvements:

- *Economy of execution*. Type information was first introduced in programming to improve code generation and run-time efficiency for numerical computations, for example in FORTRAN. In ML, accurate type information eliminates the need for *nil*-checking on pointer dereferencing. In general, accurate type information at compile time leads to the application of the appropriate operations at run-time without the need of expensive tests.
- *Economy of compilation*. Type information can be organized into *interfaces* for program modules, for example as in Modula-2 and Ada. Modules can then be compiled independently of each other, with each module depending only on the interfaces of the others. Compilation of large systems is made more efficient because, at least when interfaces are stable, changes to a module do not cause other modules to be recompiled. The messy aspects of system integration are thus eliminated.
- *Economy of small-scale development*. When a type system is well designed, typechecking can capture a large fraction of routine programming errors, eliminating lengthy testing and debugging sessions. The errors that do occur are easier to debug, simply because large classes of other errors have been ruled out. Moreover, experienced programmers adopt a coding style that causes some logical errors to show up as typechecking errors: they use the typechecker as a development tool. (For example, by changing the name of a type when its invariants change even though the type structure remains the same, so as to get error reports on all its old uses.)
- *Economy of large-scale development*. Data abstraction and modularization have methodological advantages for code development. Large teams of programmers can negotiate the interfaces to be implemented, and then proceed separately to implement the corresponding pieces of code. Dependencies between pieces of code are minimized, and code can be locally rearranged without fear of global effects. Polymorphism is important for reusing code modularly.
- *Economy of language features*. Some well-designed constructions can be naturally composed in orthogonal ways. For example, in Pascal an array of arrays models two-dimensional arrays; in ML, a procedure with a single argument that is a tu-

ple of n parameters models a procedure of n arguments. Orthogonality of language features reduces the complexity of programming languages. The learning curve for programmers is thus reduced, and the re-learning effort that is constantly necessary in using complex languages is minimized as well.

Let us now consider object-oriented languages, and see how they compare on these metrics.

- *Economy of execution*. Object-oriented style is intrinsically less efficient that procedural style. In pure object-oriented style, every routine is supposed to be a (virtual) method. This introduces additional indirections through method tables and prevents optimizations such as inlining. The traditional solution to this problem (analyzing and compiling whole programs) violates modularity and is not applicable to libraries.
- *Economy of compilation*. Often there is no distinction between the code of a class and the interface of a class. Some object-oriented languages are not sufficiently modular and require recompilation of superclasses when compiling subclasses. Therefore, the time spent in compilation may grow disproportionally with the size of the system.
- *Economy of small-scale development*. This is a big win of object-orientation: individual programmers can take good advantage of class libraries and frameworks, drastically reducing their work load. When the level of ambition grows, however, programmers must be able to understand the details of those class libraries, and this turns out to be more difficult than understanding module libraries (see also the next point). The type systems of most object-oriented languages are not expressive enough; programmers must often resort to dynamic checking or to unsafe features, damaging the robustness of their programs.
- *Economy of large-scale development*. Teams of programmers are often involved in developing class libraries and specializing existing class libraries. Although reuse is a big win of object-oriented languages, it is also the case that these languages have extremely poor modularity properties with respect to class extension and modification. For example, it is easy to override a method that should not be overridden, or to reimplement a class in a way that causes problems in subclasses. Other large-scale development problems include the confusion between classes and object types, which limits the construction of abstractions, and the fact that subtype polymorphism is not good enough for expressing container classes.
- *Economy of language features*. Smalltalk was originally intended as a language that would be easy to learn [2]. C++ is based on a fairly simple model, inherited from Simula, but is otherwise daunting in the complexity of its many features [6]. Somewhere along the line something went wrong; what started as economical and uniform ("everything is an object") ended up as a baroque collection of

class varieties. Java represents a healthy reaction to the complexity trend, but is more complex than many people realize [3].

These problems form obstacles to the further development of object-oriented software engineering, and in some situations are beginning to cause its outright rejection. Such problems can be solved either by a variety of ad hoc tools and methodologies, or by progress in language technology (both design and implementation). Here are some things that could or should be done in the various areas.

- *Economy of execution*. Much can be done to improve the efficiency of method invocation by clever program analysis, as well as by language features (e.g. by "final" methods and classes); this is the topic of a large and promising body of current work. We also need to design type systems that can statically check many of the conditions that now require dynamic subclass checks.
- *Economy of compilation*. We need to adopt languages and type systems that allow the separate compilation of (sub)classes, without resorting to recompilation of superclasses and without relying on "private" information in interfaces.
- *Economy of small-scale development*. Improvements in type systems for object-oriented languages will improve error detection and the expressiveness of interfaces. Much promising work has been done already and needs to be applied or further deployed [1, 5].
- *Economy of large-scale development*. Major progress should be achieved by formulating and enforcing inheritance interfaces: the contract between a class and its subclasses (as opposed to the instantiation interface which is essentially an object type). This recommendation requires the development of adequate language support. Parametric polymorphism is beginning to appear in many object-oriented languages, and its interactions with object-oriented features need to be better understood. Subtyping and subclassing must be separated. Similarly, classes and interfaces must be separated.
- *Economy of language features*. Prototype-based languages have already tried to reduce the complexity of class-based languages by providing simpler, more composable features. Even within class-based languages, we now have a better understanding of how to achieve simplicity and orthogonality, but much remains to be done. How can we design an object-oriented language that is powerful and simple; one that allows powerful engineering but also simple and reliable engineering?

In conclusion, object-oriented languages still have to learn some engineering lessons from procedural languages. In fairness, designers of object-oriented languages did not simply "forget" to include properties such as good type systems and good modularity: the issues are intrinsically more complex than in procedural languages. Therefore, we have to work harder to produce object-oriented language designs that entail good engineering properties. We have to work even harder to produce *good* language designs.

References

[1] Dami, L., **OO Type Theory** web page, <http://cuiwww.unige.ch/OSG/Hop/types. html>, 1996.

[2] Goldberg, A. and D. Robson, **Smalltalk-80. The language and its implementation**. Addison-Wesley. 1983.

[3] Gosling, J., B. Joy, and G. Steele, **The Java language specification**. Addison-Wesley. 1996.

[4] Meyer, B., **Object-oriented software construction**. Prentice Hall. 1988. (Second edition to appear in 1997.)

[5] Odersky, M. and P. Wadler, **Pizza into Java: Translating theory in to practice**. *Proc. 24nd Annual ACM Symposium on Principles of Programming Languages*. 1997.

[6] Stroustrup, B., **The C++ Programming Language**. Second Edition. Addison-Wesley. 1991.