Classes Considered Harmful

Andrew P. Black
Portland State University
black@cs.pdx.edu

Abstract

Classes are as essential to object-oriented programming as the **goto** statement: while there are probably circumstances where both are useful, on the whole they do more harm than good. Programmers should avoid them, and language designers should provide simpler, comprehensible alternatives with tractable semantics.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Classes and Objects

Keywords class, object

1. Introduction

Donald Knuth famously wrote "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil" [6]. I take the position that the introduction of classes into object-oriented programming languages is just such a premature optimization. After all, as languages implementors we *know* that there must be just one copy of the code for all the objects that share similar behaviour, so before we have written a single program in our language, before we know whether shared behaviour will be important in the applications that will be written in it, and before we have considered the alternatives, we decide that our languages must have classes.

2. Why Classes?

The necessity, and even the advisability, of classes has long been questioned. In 1986, Alan Borning pointed out the conceptual cost of classes: "the emphasis on classes . . . is at odds with the goal of interacting with the computer in a concrete way. When designing a new object, one must first move to the abstract level of the class, write a class definition, then instantiate it and test it, rather than . . . incrementally building an object" [5]. In a Gedanken Experiment he proposed instead a language in which "objects are completely self-contained", and the only way to create an object is to copy an existing object — an idea that he traced back to *Thinglab* [4].

As coincidence would have it, at the same time that Borning was writing, Emerald [2] had shown that it was indeed possible to base a language on self-contained objects. In Emerald, there are neither classes nor prototypes: instead there is a language primitive, the *object constructor*, that builds a new object out of whole cloth.

Let us also be clear that classes do not "model the real world". *Objects* may or may not model the real world, but classes certainly don't. Although there are many chairs in the real world, there is no "chair class" in the real world.

Borning identified 8 functions for classes in Smalltalk, of which two (updating objects when a method is changed and locating all of the instances) relate to Smalltalk as a dynamic programming *system* rather than as a language. Of the remaining 6,

1. generators of new objects
2. descriptions of the representation of their instances
3. descriptions of the message protocol of their instances
4. elements in the object taxonomy
5. a means for implementing differential programming (like this object, but with the following differences), and
6. repositories for methods,

functions 2, 3 and 4 are meta: they have to do with introspecting on objects rather than creating or using them, and rightfully belong to object mirrors (2 and 3) and types (4), not classes. The first function, generating new objects, is not accomplished by the class alone: it requires collusion with some "system magic", for example, the **new** keyword in Java, or the basicNew primitive in Pharo. Once we conclude that there has to be some such magic, it seems advisable to choose magic that is as self-contained as possible: I claim that an object constructor, as found in Emerald and Grace [3], is better magic than **new**.

Function 6 I have already dismissed as a premature optimization: let us design our language and its semantics first, and *then* decide how to implement method suites efficiently. This leaves us with 5: how should we support differential programming, or, in other words, how should we support the reuse of code between objects?

3. Alternatives

The known alternatives to Classes are delegation, trait objects, and copying prototypes. In the *Treaty of Orlando*, Lieberman, Stein and Ungar seem to agree to a truce just in time to avoid splitting the then-young OO community over the question of how to best support differential programming [7]. This was probably wise: the task then before the OO community was to establish a united front, not to squabble amongst ourselves. We have lived with that truce now for 18 years, and I believe that it is time to re-raise the question: can't we do better than classes?

So far I've argued that classes may not be necessary, but not that they are bad, and indeed this is what I have believed for the last 20 years. But recently, because of my involvement in the design of Grace, I've spend many hours looking at the options for class-like inheritance. I don't like any of them. A long discussion of the issues can be found in my submission to MASPEGHI 2013 [1]; there is no space to repeat it here. In brief, inheritance from classes seems to

1. require either metaclasses, meta-metaclasses, and so on in infinite regress (as in Smalltalk), or the postulation of a new kind of entity that is not an object and therefore has no class (as in Java);
2. demand a complex semantics with two levels of fixpoints, one to create the classes, and another to create the objects;
3. does not accommodate immutable objects, that is, objects that are *created* with instance-specific data that never changes. Moreover,
4. attempts to simulate the semantics of class-inheritance without classes, using copying or delegation, create their own problems which are arguably worse than those caused by classes.

4. Conclusion

I've reluctantly come to the conclusion that the only viable solution for behaviour-sharing in an egalitarian (i.e., classless) language is delegation. Delegation is a flexible primitive that can allow for the re-use of code from multiple sources, the overriding of features of the delegate, and a compositional semantics. Protection concerns may convince us that only certain objects be allowed as delegates, for example, objects that explicitly declare their willingness to serve in this capacity.

It is time to re-open the discussions of the mid-1980s on the advantages and disadvantages of classes and of other approaches to code-reuse. The NOOL workshop seems like an excellent venue to do so.

References

[1] A. P. Black. What shall we tell the children (about inheritance)? In *Workshop on MechAnisms for SPEcialization, Generalization and inHerItance (MASPEGHI)*, 2013. URL `http://www.cs.jyu.fi/maspeghi2013/papers.html`.

[2] A. P. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings First ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, Portland, Oregon, October 1986. ACM Press.

[3] A. P. Black, K. B. Bruce, M. Homer, and J. Noble. Grace: the absence of (inessential) difficulty. In *Onward! '12: Proceedings 12th SIGPLAN Symp. on New Ideas in Programming and Reflections on Software*, pages 85–98, New York, NY, 2012. ACM. URL `http://doi.acm.org/10.1145/2384592.2384601`.

[4] A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *TOPLAS*, 3(4):353–387, 1981.

[5] A. Borning. Classes versus prototypes in object-oriented languages. In *FJCC*, pages 36–40. IEEE Computer Society, 1986. ISBN 0-8186-0743-2.

[6] D. Knuth. Structured programming with go-to statements. *ACM Computing Surveys*, 6(4):261–302, 1974.

[7] H. Lieberman, L. Stein, and D. Ungar. Treaty of orlando. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, pages 43–44, New York, NY, USA, 1987. ACM. ISBN 0-89791-266-7. . URL `http://doi.acm.org/10.1145/62138.62144`.