Peak Objects

William R. Cook

Department of Computer Sciences
University of Texas at Austin
wcook@cs.utexas.edu

I was aware of a need for object-oriented programming long before I learned that it existed. I felt the need because I was using C and Lisp to build medium-sized systems, including a widely-used text editor, CASE and VLSI tools. Stated simply, I wanted flexible connections between providers and consumers of behavior in my systems. For example, in the text editor anything could produce text (files, in-memory buffers, selections, output of formatters, etc) and be connected to any consumer of text. Object-oriented programming solved this problem, and many others; it also provided a clearer way to *think about* the problems. For me, this thinking was very pragmatic: object solved practical programming problems cleanly.

The philosophical viewpoint that "objects model the real world" has never appealed to me. There are many computational models, including functions, objects, algebras, processes, constraints, rules, automata – and each has a particular ability to model interesting phenomena. While some objects model some aspects of the real world, I do not believe they are inherently better suited to this task than other approaches. Considered another way, what percentage of classes in the implementation of a program have any analog in the real world?

In the mid-'80s, when I was learning about objects, it was frequently said that objects could not be explained, they must be experienced. Sufficient experience would lead to an "Ah ha!" insight after which you could smile knowingly and say "it can't be explained... it must be experienced." This is, unfortunately, still true to a degree. Many students (and programmers) do not feel comfortable with dynamic dispatch, the higher-order nature of objects and factories, or complex subtype/inheritance hierarchies. Advanced programmers also struggle to design effective architectures using advanced techniques – where the best approach is not obvious.

In what follows I describe some long-standing myths about object-oriented programming and then suggest future directions.

Classes are Abstract Data Types.

The assumption that classes are *abstract data types* (ADTs) is one of the more persistent myths. It is not clear exactly how it started, but the early lack of a solid theoretical foundation of objects may have contributed.

ADTs consist of a *type* and *operations* on the type – where the type is *abstract*, meaning it name/identity is visible but is concrete representation is hidden. Hiding the representation type generally requires a *static type system*.

Objects, on the other hand, are *collections of operations*. The types of the operations (the object's interface) are completely public (no hidden types), whereas

the internal representation is completely invisible from the outside. The idea of *type abstraction*, or partially hiding a type, is not essential to objects, although it does show up when classes are used as types (see below). Since they don't require type abstraction, object work equally well in dynamically and statically typed languages (e.g. Smalltalk and C++).

The relationship between ADTs and objects is well-known [7, 17], yet even experts often treat "data abstraction" and "abstract data type" as synonyms, for example, in the history of CLU [11], and many textbooks. I suspect that the identification of "data abstraction" and "abstract data type" arose because ADTs seem natural and fundamental: they have the familiar structure of abstract algebra, support effective reasoning and verification [9], and have an elegant explanation in type theory [14]. In the late '70s ADTs appeared in practical programming languages, including Ada, Modula-2, and ML. However, ADTs in their pure form have never become as popular as object-oriented programming. It is interesting to consider what would have happened if Stroustrup had added ML-style ADTs, modules, and functors to "C" instead of adding objects [12].

Most modern languages combine both ADTs and objects: Smalltalk has built-in ADTs for integers, which are used to implement the object-oriented numbers. But Smalltalk does not support user-defined ADTs. OCAML allows user-defined ADTs and also objects. Java, C# and C++ have built-in ADTs for primitive types. They also support pure objects via interfaces and a form of user-defined ADTs: when a class is used as a type it acts as a bounded existential, in that it specifies a particular implementation/representation, not just public interface.

Integrating the complementary strengths of ADTs and objects is an active research topic, which now focuses on extensibility, often using syntactic expressions as a canonical example [10, 15, 19, 21]. However, the complete integration of these approaches has not yet been achieved.

Objects Encapsulate Mutable State.

Encapsulation is a useful tool for hiding implementation details. Although encapsulation is often cited as one of the strong points of object-oriented programming, complex objects with imperative update can easily break encapsulation.

For imperative classes with simple interfaces, like stacks or queues, the natural object implementation is effective at encapsulation. But if objects that belong to the private representation of an updatable object leak outside the object, then encapsulation is lost. For example, an object representing a graph may use other objects to represent nodes and edges – but public access to these objects can break encapsulation. This problem is the subject of ongoing research; several approaches have been developed to enforce encapsulation with extended type systems [6, 4, 22, 2]. Another approach uses multiple interfaces to prevent updates to objects that pass outside an encapsulation boundary [18].

With Orthogonal Persistence, We Don't Need Databases.

Orthogonal persistence is a natural extension of the traditional concept of variable *lifetime* to allow objects or values to persist beyond a single program

execution [1]. Orthogonal persistence is a very clean model of persistent data – in effect it provides garbage collection with persistent roots. Unfortunately, orthogonal persistence by itself does not eliminate the need for databases.

There are three main problems with orthogonal persistence: performance, concurrency, and absolutism – yet they are, I believe, all solvable [8]. The first problem is that orthogonal persistence does not easily support the powerful optimizations available in relational databases [13]. Until orthogonal persistence supports similar optimizations – without introducing non-uniform query languages – I believe it will not be successful outside small research systems. The second problem is the need for control of concurrent access to persistent data. Databases again have well-developed solutions to this problem, but they must be adapted to work with orthogonal persistence [3]. The final problem with orthogonal persistence is its absolutism: taken to the limit, anything can be persistent, including threads, user interface objects, and operating system objects. It also requires that object behavior (class definitions and methods) be stored along with an object's data. While this may be appropriate for some applications, a more pragmatic approach will likely be more successful in a wider range of applications, which just need to store ordinary data effectively.

We must also address the cultural problem: object-oriented programmers rarely have a deep understanding of database performance or transaction models. Database researchers don't seem very interested in how databases are actually incorporated into large systems.

Objects are the Best Model for Distributed Programming.

One of the grand projects at the end of the last millennium was the development of distributed object models. There were many contributors and results to this effort, but some of the most visible were CORBA, DCOM, and Java RMI [20]. I believe that this project met its goals, but was a mostly a failure because the goals were not the right goals. The problem is that *distance does matter* [16] and communication partners don't want to share stubs. I suspect the world wide web would have failed if the HTTP protocol had been designed and implemented using CORBA. Web services are a step in the right direction, but the document-oriented communication style must be better integrated with the application programming model.

Classes Are Types.

Object-oriented programming emphasizes the use of interfaces to separate clients and services, yet when classes are used as types they specify the implementation, not just the interface, of objects. It is even possible to inspect the runtime implementation of objects, thus breaking encapsulation, by indiscriminate use of `instanceof`. Programs that include many tests of the form `if (x instanceof C) ...` are quite common but undermine many of the benefits of using objects.

It is possible to define a language in which classes *are not* types. Such a language would be a more pure object-oriented language. Classes would only be used to construct objects. Only interfaces could be used as types – for arguments,

return values, and declarations of variables. Following tradition, part of the work could be titled "`instanceof` Considered Harmful".

Simula was the first Object-Oriented Language.

Although Simula was the first imperative object-oriented language, I believe that Church's untyped lambda-calculus was the first object-oriented language [5]. It is also the only language in which everything is an object – since it has no primitive data types. As a starting point, compare the Church booleans to the `True` and `False` classes in Smalltalk (note that ^e means return e):

Class	*Smalltalk method*	*Church Boolean*
`True`	`ifFalse: a ifTrue: b ^a value`	$\lambda a.\lambda b.a$
`False`	`ifFalse: a ifTrue: b ^b value`	$\lambda a.\lambda b.b$

Future

What does the future hold? In the late '90s I started working on enterprise software and found that object-oriented programming in its pure form didn't provide answers to the kinds of problems I was encountering.

It is still too difficult to build ordinary applications – ones with a user interface, a few algorithms or other kinds of program logic, various kinds of data (transactional, cached, session state, configuration), some concurrency, workflow, a security model, running on a desktop, mobile device, and/or server.

I find myself yearning for a new paradigm, just as I yearned for objects in the '80s. New paradigms do not appear suddenly, but emerge from long threads of development that often take decades to mature. Both pure functional programming (exemplified by Haskell) and object-oriented programming (Smalltalk & Java) are examples.

Thus it should be possible to see traces of future paradigms in ideas that exist today. There are many promising ideas, including generative programming, reflection, partial evaluation, process algebra, constraint/logic programming, model-driven development, query optimization, XML, and web services. It is unlikely that focused research in any of these areas will lead to a breakthrough that triggers a paradigm shift. What is needed instead is a wholistic approach to the problem of building better software more easily, while harnessing specific technologies together to create a coherent paradigm.

I want a more declarative description of systems. I find myself using domain-specific languages: for semantic data models, security rules, user interfaces, grammars, patterns, queries, consistency constraints, upgrade transformations, workflow processes. Little bits of procedural code may be embedded in the declarative framework, acting as procedural plugins.

Current forms of abstraction were designed to express isolated data abstractions, rather than families of interrelated abstractions. Today object models, e.g. a business application or the HTML document object model, have hundreds or thousands of interrelated abstractions. A the same time, it is very desirable to place each *feature* of a program into a separate module, even though the implementation of the features may be fairly interconnected.

Designs typically include aspects like security or persistence that are conceptually global, yet must be configured and specialized to each individual part of the system. The concept of an *aspect* is a powerful one – yet current aspect-oriented programming languages are only an initial step toward fulfilling the promise of this concept.

The underlying infrastructure for these higher levels will most likely be built using object-oriented programming. But at the higher levels of application programming, the system may not follow any recognizable object-oriented style.

For years there have been suggestions that object-oriented programming has reached its peak, that nothing new is to be discovered. I believe that objects will continue to drive innovation and will ultimately play a key role in the future of software development. However, it is still to be seen whether objects can maintain their position as a fundamental unifying concept for software designs, or if a new paradigm will emerge.

References

1. M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.
2. A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM*, 52(6):894–960, 2005.
3. S. Blackburn and J. N. Zigman. Concurrency — the fly in the ointment? In *POS/PJW*, pages 250–258, 1998.
4. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–223. ACM Press, 2003.
5. A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1951.
6. D. G. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 53–76. Springer-Verlag, 2001.
7. W. Cook. Object-oriented programming versus abstract data types. In *Proc. of the REX Workshop/School on the Foundations of Object-Oriented Languages*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
8. W. R. Cook and A. Ibrahim. Integrating programming languages & databases: What's the problem? Available from `http://www.cs.utexas.edu/users/wcook/projects/dbpl/`, 2005.
9. C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
10. S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 91–113. Springer-Verlag, 1998.
11. B. Liskov. A history of clu. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 133–147, New York, NY, USA, 1993. ACM Press.
12. D. B. MacQueen. Modules for standard ml. In *Proc. of the ACM Conf. on Lisp and Functional Programming*, 1984.

13. D. Maier. Representing database programs as objects. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages, Papers from DBPL-1*, pages 377–386. ACM Press / Addison-Wesley, 1987.
14. J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 37–51. ACM, 1985.
15. M. Odersky and M. Zenger. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, Jan. 2005.
16. D. A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004.
17. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
18. N. Scharli, A. P. Black, and S. Ducasse. Object-oriented encapsulation for dynamically typed languages. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–149. ACM Press, 2004.
19. M. Torgersen. The expression problem revisited - four new solutions using generics. In *Proceedings of ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 123–146. Springer, 2004.
20. M. Völter, M. Kircher, and U. Zdun. *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. Wiley, 2005.
21. P. Wadler. The Expression Problem. `http://www.cse.ohio-state.edu/~gb/cis888.07g/java-genericity/20`, November 1998.
22. T. Zhao, J. Palsber, and J. Vite. Lightweight confinement for featherweight java. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 135–148. ACM Press, 2003.