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

D. Thomas (Ed.): ECOOP 2006, LNCS 4067, pp. 180–185, 2006.

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