The Power of Interoperability: Why Objects Are Inevitable

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, PA, USA
aldrich@cs.cmu.edu

Abstract

Three years ago in this venue, Cook argued that in their essence, objects are what Reynolds called *procedural data structures*. His observation raises a natural question: if procedural data structures are the essence of objects, has this contributed to the empirical success of objects, and if so, how?

This essay attempts to answer that question. After reviewing Cook's definition, I propose the term *service abstractions* to capture the essential nature of objects. This terminology emphasizes, following Kay, that objects are not primarily about representing and manipulating data, but are more about providing services in support of higher-level goals. Using examples taken from object-oriented frameworks, I illustrate the unique design leverage that service abstractions provide: the ability to define abstractions that can be extended, and whose extensions are interoperable in a first-class way. The essay argues that the form of interoperable extension supported by service abstractions is essential to modern software: many modern frameworks and ecosystems could not have been built without service abstractions. In this sense, the success of objects was not a coincidence: it was an inevitable consequence of their service abstraction nature.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-Oriented Programming

Keywords Object-oriented programming; frameworks; interoperability; service abstractions

1. Introduction

Object-oriented programming has been highly successful in practice, and has arguably become the dominant programming paradigm for writing applications software in industry. This success can be documented in many ways. For example, of the top ten programming languages at the LangPop.com index, six are primarily object-oriented, and an additional two (PHP and Perl) have object-oriented features.[1] The equivalent numbers for the top ten languages in the TIOBE index are six and three.[2] SourceForge's most popular languages are Java and C++;[3] GitHub's are JavaScript and Ruby.[4] Furthermore, objects' influence is not limited to object-oriented languages; Cook [8] argues that Microsoft's Component Object Model (COM), which has a C language interface, is "one of the most pure object-oriented programming models yet defined." Academically, object-oriented programming is a primary focus of major conferences such as ECOOP and OOPSLA, and its pioneers Dahl, Nygaard, and Kay were honored with two Turing Awards.

This success raises a natural question:

> *Why has object-oriented programming been successful in practice?*

To many, the reason for objects' success is not obvious. Indeed, objects have been strongly criticized; for example, in an interview Stepanov explains that his STL C++ library is not object-oriented, because he finds OO to be technically and philosophically unsound, and methodologically wrong [29]. In addition, popular object-oriented languages are often criticized for their flaws; for example, Hoare states that the null references most OO languages provide were his "billion dollar

[1] `http://www.langpop.com/`

[2] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

[3] `http://sourceforge.net/directory/`, click Advanced and mouse over Programming Language, accessed 4/7/2013

[4] `https://github.com/languages`, accessed 4/7/2013

mistake"[13]. These and other criticisms have led some to argue that object-orientation became popular essentially for marketing reasons: because "it was hyped [and] it created a new software industry."[5]

While there has unquestionably been some hype about objects over the years, I have too much respect for the many brilliant developers I have met in industry to believe they have been hoodwinked, for decades now, by a fad. The question therefore arises: might there be *genuine* advantages of object-oriented programming that could explain its success?

Some of the advantages of object-oriented programming may be psychological in nature. For example, Schwill argues that "the object-oriented paradigm...is consistent with the natural way of human thinking" [28]. Such explanations may be important, but they are out of scope in this inquiry; I am instead interested in whether there might be significant *technical* advantages of object-oriented programming.

The most natural place to look for such technical advantages is the essential characteristics that define the object-oriented paradigm. On this question there is some controversy, but I follow Cook's definition: "an object is a value exporting a procedural interface to data or behavior" [8]. In other words, objects are in their essence *procedural data structures* in the sense of Reynolds [27]. Cook's definition essentially identifies dynamic dispatch as the most important characteristic of objects. Each object is self-knowing (*autognostic* in Cook's terms), carrying its own behavior with it, but assuming nothing about other objects beyond their own procedural interfaces. This autognosis property, in turn, enables different implementations of an object-oriented interface to interoperate in ways that are difficult when using alternative constructs such as abstract data types.

Could data structure interoperability really be so important? It is an excellent question, but too narrow. As we will see, to understand the value of objects we must consider *service abstractions* that go beyond data structures, because it is for abstractions of components and services that interoperability becomes critical. This leads to the following thesis:

> *Object-oriented programming is successful in part because its key technical characteristic—dynamic dispatch—is essential to supporting independent, interoperating extensions; and because interoperable extension is in turn essential to the reuse of architectural code (as in frameworks), and more broadly to the design of modern software ecosystems.*

[5] Joe Armstrong, quoted at `http://harmful.cat-v.org/software/OO_programming/why_oo_sucks`

To support this thesis, we need a careful definition of interoperable extension and an analysis of how objects provide it—by contrast, we'll see that obvious alternatives such as abstract data types don't (Section 3). Alternative mechanisms can provide interoperable extension only by using service abstraction themselves—and thus are equivalent to what we consider the essence of objects.[6] Interoperable extension is essential to many modern software systems and frameworks, in that the core architectural requirements of these systems cannot be fulfilled without it (Section 4). The rise of objects, therefore, is not a coincidence: it is an inevitable response to the needs of modern software.

Pieces of the story told here are known, more or less formally, within the object-oriented and programming language communities, but the larger story has not been well told. With that story in place we can begin to explain the success of objects on a purely technical basis.

Along the way, we'll connect Cook's definition of objects with Kay's focus on messaging as a key to object-orientation; investigate a mystery in the design of object-oriented vs. functional module systems; and use the theory sketched here to make predictions both about the technical feasibility of statically typed, fully-parametric modules in object-oriented systems, and about the effect of adding first-class modules to languages that do not support objects.

I hope that the arguments in this essay will inspire researchers to gather data and build systems that can properly validate the propositions described here.

2. The Nature of Objects

A clear understanding of the nature of objects is essential to my argument. Cook's definition was intended to distinguish objects from ADTs, with which objects are often confused. However, his definition extends beyond data abstraction, following Alan Kay's emphasis on objects providing services that support high-level behavioral goals. As we will see, it is in its ability to abstract behavior that object-oriented programming provides its greatest benefits.

2.1 Objects and ADTs

Once again, Cook's definition states that "an object is a value exporting a procedural interface to data or behavior" [8]. His essay uses a simple set example to illustrate the distinction between objects and abstract

[6] As we will see, "simulated objects" do appear in practice in systems that require interoperable extension, but are not written in object-oriented languages. These examples supports my thesis, which is about object-oriented programming in the most general sense. Of course, when programming with objects in practice, programming language support is convenient.

data types (ADTs). For example, an object-oriented set type may be abstractly defined as follows:

```
type IntSet = {
   bool contains(int element);
   bool isSubsetOf(IntSet otherSet);
}
```

Note that different `IntSet` implementations can interoperate. For example, as shown in the code below, we can have instances of two different implementations of type `IntSet` and test if one contains all elements of the other. An object-oriented set type can be described using an interface in Java; an example is `java.util.Set`, or simply the interface above replacing the keyword **`type`** with **`interface`**. Different classes can then implement that interface. Each set object carries its hidden implementation type with it. In type theory, we say that objects have ***existential types***; the existential is opened on every access to each object.

```
class IntSet1 implements IntSet { ... }
class IntSet2 implements IntSet { ... }

// in main()
IntSet s1 = new IntSet1(...);
IntSet s2 = new IntSet2(...);
boolean x = s1.isSubsetOf(s2);
```

In contrast a set ADT might be abstractly defined as follows:

```
module SetModule1 {
     // implementation...
} with signature {
   type IntSet;

   bool contains(IntSet set, int element);
   bool isSubsetOf(IntSet s1, IntSet s2);
}
```

Set ADTs matching this identical signature can be implemented by multiple modules (`SetModule1`, `SetModule2`, etc.), but each module `SetModule`N defines a separate type `SetModule`N`.IntSet`. The ADT type `SetModule1.IntSet` denotes the fixed but hidden representation defined in module `SetModule1`. All instances of `SetModule1.IntSet` have the same representation, and both Cook and Reynolds point out that this has some advantages: the `isSubsetOf` operation can be defined in terms of the hidden representation, which can be asymptotically more efficient than implementing `isSubsetOf` in terms of `contains`, as must be done in a pure object-oriented implementation of the `IntSet` interface. This efficiency difference can be critical in practice.[7]

ADTs can also be defined in Java using classes as follows:

```
final class IntSetA {
   bool contains(int element) { ... }
   bool isSubsetOf(IntSetA other) { ... }
}
```

As with the abstract `SetModule1.IntSet` described earlier, all instances of `IntSetA` are instances of the same class and have the same internal representation.[8]

The disadvantage of ADTs, relative to objects, is the lack of interoperability. Consider what happens if we define a second Java class, called `IntSetB`, analogous to the first one above:

```
// different code but the same interface
final class IntSetB {
   bool contains(int element) { ... }
   bool isSubsetOf(IntSetB other) { ... }
}
// in main()
IntSetA sA = new IntSetA(...);
IntSetB sB = new IntSetB(...);
boolean x = sA.isSubsetOf(sB); // ERROR!
```

Now if we create an instance of `IntSetA` and an instance of `IntSetB`, we cannot invoke the `isSubsetOf` operation to compare their contents. The reason is that `IntSetA` and `IntSetB` are different abstract types; neither is a subtype of the other. This is, in fact, an inevitable consequence of the fact that binary operations on abstract data types can depend on the internal representation of the second object passed in. This dependence has performance advantages, but makes interoperation impossible, since `IntSetB` may

[7] In practical object-oriented settings, the efficiency of binary operations can often be regained, aside from any cost associated with dispatching itself. This is done most elegantly through the use of ***multiple dispatch***, in which the implementation of an operation is chosen dynamically according to the class of all arguments to a *generic function* [3]. Other solutions include using double-dispatching idioms or more ad-hoc **`instanceof`**-style class tests. A full discussion of these solutions, including the significant modularity issues involved, is out of scope here.

[8] Note that the Java implementation of `IntSetA` uses a **`final`** class in order to sidestep the use of inheritance. Cook argues that inheritance, while valuable, is a secondary and optional feature of objects. The design of the Self language illustrates this point, providing code reuse via delegation instead of inheritance [32]. Others agree that inheritance is secondary to dynamic dispatch. For example, when Kay discusses the six main ideas of Smalltalk, inheritance is not mentioned, but dynamic dispatch is (in the form of messaging) [18]. Likewise, Gamma *et al.* argue that a central principle of object-oriented design is to "favor object composition over class inheritance,"[11]—and note that composition typically leverages dynamic dispatch in the setting of patterns. When discussing framework design, Johnson and Foote distinguish *white-box* frameworks based on inheritance from *black-box* frameworks based only on dynamic dispatch, and argue that "black-box relationships are an ideal towards which a system should evolve" [16]. While acknowledging that inheritance is a feature of many OO languages and may have significant value in many designs, I will not consider it further, so as to focus on the most important, and most unique, feature of objects.

very well have an internal representation that is incompatible with that of `IntSetA`.

Because each module hides the representation of the ADT it defines, ADTs are also existential types. However, these existential types are opened once, when the defining module is imported into the program, rather than on every invocation of an operation. Thus the distinction between objects and ADTs can be thought of in type theory as the difference between ***closed*** and ***open*** existentials.

Discussion. Cook's essay focuses on the technical and theoretical differences between objects and ADTs. He highlights the tradeoff between the interoperability provided by objects and the shared representation provided by ADTs. But due to the focus of his essay, the larger consequences of the interoperability provided by objects are only briefly discussed. At this point a reader may be forgiven for asking, what is the big deal about interoperation? After all, Cook quotes Liskov making the following cogent argument:

> Although a program development support system must store many implementations of a type..., allowing multiple implementations within a single program seems less important. [22]

Perhaps Liskov is right. Does it really matter whether we have two different implementations of Set in a program? Why don't we just pick one that is efficient and has the features needed for the program?

As far as data abstraction goes, this defense of ADTs may be correct. To investigate why having multiple implementations of an abstraction might be important indeed, we must broaden our view and look beyond data abstraction.

2.2 Beyond Data Abstraction: Behavior, Messages, and Services

Cook's essay focuses primarily on data abstraction, because he is comparing objects to ADTs and data abstraction is what abstract data types are intended to do. However, his definition of objects is more general: "...a value exporting a procedural interface to data **or behavior**." This reflects a broader view of object-orientation, one that can be seen even more clearly in OO's origins in Simula and Smalltalk. Dahl and Nygaard, for example, related object-oriented programming to simulation, saying that "a program execution is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world" [21]. While this quote focuses more on the purpose of OO rather than its mechanisms, a simulation nevertheless focuses on behavior more than data. Alan Kay underscores this point:

> What I got from Simula was that you could now replace bindings and assignment with *goals*. The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. Instead, the objects should be presented as *sites of higher level behaviors more appropriate for use as dynamic components.* [18]

Thus, while comparing objects to ADTs may be useful for making intellectual distinctions, Kay suggests that the power of objects is not in representing data structures, but in representing higher-level ***goals***. The idea of goals suggests an analogy to planning in artificial intelligence: rather than express an algorithm with even high-level state and assignment, it is better to express declarative goals and declarative rules for achieving them, and rely on a search engine to apply the rules in a way that accomplishes the goal. Object-oriented programming is not fully declarative, but Kay's point is that the abstraction provided by a method-based interface enables a lot of client code to approach the declarative ideal.

When discussing his view of objects in Smalltalk, Kay writes of the "objects as server metaphor" in which every "object would be a server offering services" that are accessed via messages[9] to the object [18]. In fact, Kay considers objects the "lesser idea" and states that "the big idea is messaging."[10] On a technical level, in Smalltalk messages are "a procedural interface to data or behavior," which is consistent with Cook's definition, but again Kay de-emphasizes data abstraction in favor of behavior and high-level goals.

This focus on goals also suggests that whereas ADTs are focused on lower-level data representation and manipulation concerns, objects are focused more on abstractions that are useful in high-level program organization. Thus, in looking for the advantages of objects, perhaps we ought not to focus on data abstractions for our examples. Instead of Reynolds's procedural data structures, in the remainder of this essay I will generally use the term *service abstractions*, reflecting Kay's view of objects as servers that provide services to their clients. A ***service abstraction*** is, on a technical level, the same form of abstraction as a procedural data structure, but it may be used to abstract any set of services, not just data structure manipulations. We could also simply use the term object, following Cook's definition, but the term service abstraction will

[9] Note that messages in Smalltalk are synchronous method calls; they are not asynchronous or distributed in the sense of network messages, although Kay derives substantial inspiration from network-based metaphors.

[10] Alan Kay, email sent October 10, 1998, to squeak@cs.uiuc.edu

remind us that we are focused specifically on the dynamic dispatch feature of objects.[11]

Taking Smalltalk's pioneering work in GUI libraries as an inspiration, we will explore the generalization to service abstractions through the example of a widget:

```
interface Widget {
   Dimension getSize();
   Dimension getPreferredSize();
   void setSize(Dimension size);
   void paint(Display display);
}
```

While a widget certainly has data—its size, and perhaps widget-specific data related to what the widget is displaying—abstracting that data is not the primary purpose of a widget. Rather, a widget's purpose is to abstract the *behavior* of a user interface element. In the simple interface above, inspired by the `ConstrainedVisual` interface from Apache's Pivot UI framework,[12] the service abstraction captures the negotiation between a UI element and its container concerning how large the UI element should be, as well as the goal of painting itself on a display device.

More importantly, the interoperability advantages of object-oriented service abstractions over ADTs suddenly become more obvious with this example. While a framework such as Apache Pivot provides many widgets, the space of possible widgets is much larger, and so programmers using the framework will likely want to define their own, or even use widgets developed by strangers.[13] GUI frameworks provide facilities to recursively and dynamically compose atomic widgets into composite widgets, and ultimately into a complete user interface. For example, a `CompositeWidget` might be defined as:

```
interface CompositeWidget extends Widget {
  void addWidget(Widget chld, Position p);
}
```

Here `addWidget` is a binary operation: CompositeWidget is a kind of Widget, and it must interoperate with other kinds of widgets that are added to it. It is essential, in particular, that custom widgets written by programmers can take their place in a user interface together with widgets from the base framework, as well as widgets written by strangers.

3. The Design Leverage of Objects

Based on Cook's insight that different object-oriented implementations of a set can interoperate, and the intuition from the widget example that this might take on increased importance in service abstractions, I now propose a candidate for the leverage provided by object-oriented service abstractions in design:

> *The key design leverage provided by objects is the ability to define nontrivial abstractions that are modularly extensible, where instances of those extensions can interoperate in a first-class way.*

Let me define the these terms with more care:

- **Nontrivial abstraction.** An interface that provides at least two essential services.
- **Modular Extensibility.** New implementations not anticipated when the abstraction was designed can be provided without changing the original abstraction.
- **First-class Interoperability.** Clients can instantiate a number of different implementations of an abstraction and manipulate those instances together in a first-class way. First-class interoperability has three facets:
  - **Direct interoperation.** If the abstraction defines a binary operation, the arguments to that operation need not be instances of the same implementation. This form of interoperation is analogous to gears that mesh; mathematically, it corresponds to a fold operation.
  - **Uniform treatment.** Clients can uniformly invoke operations on instances of different implementations, without distinguishing the particular implementations involved (e.g. by using a static type or an explicit typecase operation). This form of interoperation is analogous to balls within a ball bearing, which may not touch each other directly but nevertheless must match each other closely; mathematically, it corresponds to a map operation.
  - **First-class manipulation.** Instances of different implementations can be manipulated together as first-class values. For example, it should be possible to store a collection of instances of different implementations in a single data structure instance, then dynamically select element(s) from the collection and invoke operations on them.

Discussion. The reason to consider only nontrivial abstractions is that if an abstraction has only one service, one can use a function to abstract it. Functions are completely ideal in such cases—but some abstractions that at first appear to be simple turn out to be richer than expected in practice. Consider the `Widget` example: the most important function in the interface is probably

[11] e.g. as stated before, I am not considering inheritance—surely the question of whether inheritance contributes to the success of objects is interesting, but it is out of scope for our current purposes.

[12] http://pivot.apache.org/

[13] we will see an example of this later when looking at the Microsoft Office plugin ecosystem.

`paint`, and a trivial academic example might include only that, leading to the mistaken impression that just passing a first-class function around suffices to encapsulate the widget abstraction. In reality, widgets must manage other concerns, such as their size, which requires communication with both their container and their children. I have in fact already oversimplified: the true interface on which the `Widget` example is based has nine methods, and adds the additional concern of revealing the widget's baseline.[14]

The definition of modular extension captures Meyer's *open-closed principle* [24], which states that abstractions "should be open for extension, but closed for modification." It is particularly important when an abstraction is designed by one organization, but extended in an unanticipated[15] way by another; in this case, the second organization may not be able to modify the original abstraction.[16]

Reynolds originally suggested interoperability as a benefit of procedural data structures [27], and Cook points out that as a consequence, when using a pure object model, binary operations should work on different implementations. This direct interoperability property is critical in the `Widget` example, where the `CompositeWidget` must be able to integrate widgets defined by strangers.

It turns out that simple forms of interoperability can be supported through mechanisms other than objects, however. A notable example is generic programming [25] as represented in the C++ Standard Template Library (STL). For example, iterators in the STL can be used to copy elements from one collection to another. The copy operation is binary, accepting a source and a destination iterator. Although iterators are C++ objects, dynamic dispatch is not used to enable interoperation of different iterator implementations. Instead, they are defined using C++ templates, which essentially generate a fresh version of the copy operation for every pair of source and destination iterator types.

A key limitation of the interoperability provided by generic programming mechanisms is that it is second-class, precluding the manipulation of iterators as truly first-class values. For instance, in the iterator example above, the C++ compiler must know the precise implementation class of each iterator so that it can support template code generation. This is a reasonable assumption in the typical modes of use of generic programming, but it is nevertheless a serious limitation in many other cases. For example, if we store different kinds of iterators in a list, typical type systems lose track of the particular implementation class of each iterator, and so we cannot use pure generic programming to invoke copy operations on them; we must fall back on dynamic dispatch to make copies.

Objects, on the other hand, are first-class values; we can see this in Cook's definition, which starts "objects are *values*...." We therefore characterize object-oriented service abstractions as providing a first-class notion of interoperability. With service abstractions, one can use multiple implementations of an abstraction together, even when instances of the abstractions are handled in a first-class way.

To illustrate first-class interoperation, consider a scenario in which we build a composite widget that allows the user to select one of several visualizations, which are swapped when the user clicks a button. Here the choice of a visualization widget to go inside the `CompositeWidget` is made dynamically, yet the composite widget interoperates with the visualization widget in the sense that the latter is passed to the `addChild` method of the former.

I use the term first-class interoperation broadly, to include not just invoking binary methods, but also the general ability to operate uniformly on instances of multiple different implementations. For example, the `CompositeWidget` must store its child widgets together in some kind of data structure. When the `paint` method of `CompositeWidget` is invoked, it must iterate over the child widgets and invoke `paint` in turn on each one of them. Note that the implementation of `CompositeWidget.paint()` must not rely on an explicit typecase to determine the implementation of each child widget before invoking `paint` on the child; a typecase would enable the container to work with a fixed set of widget implementations, but in that case the unmodified container code could not work with new widget implementations.

Alternatives to objects. Are there technical approaches, besides the service abstraction technique used in objects, that can provide the design benefits described above?

I have already established that abstract data types do not provide the required interoperability for binary methods. In fact, even in the absence of binary methods, they fail the first-class interoperability test: a data structure may be a list of `IntSet1` or a list of `IntSet2`, but we cannot define a list that mixes in-

[14] see `http://pivot.apache.org/2.0.3/docs/api/org/apache/pivot/wtk/ConstrainedVisual.html`

[15] By ***unanticipated extension*** I mean an extension for which neither the extension's code, nor the particular feature(s) being added by the extension, were explicitly anticipated by the designer of the abstraction being extended. Of course, in order to design an extension point at all, the designer must have some idea of the class of features the extension point is intended to support, but it is typically impractical to enumerate all features that fall into that class.

[16] Such modifications may be literally impossible if the abstraction is distributed in binary form; or they may merely be infeasible in practice because the second organization wishes to avoid modifying (and then maintaining) a component it did not develop

stances of the two types.[17] ADTs are therefore not a solution.[18]

A module is a similar kind of abstraction to an object: both typically present a procedural interface to the outside world, both encapsulate state, and both are typically given types (or module signatures) that permit subtyping. To attain the expressiveness of objects, however, modules must be made first class, a property rarely found in widely-used languages, but which is an ongoing topic of research [5]. I would consider a first-class module system, such as that provided in recent versions of OCaml, to provide service abstractions.[19]

Haskell's type classes [34] cannot directly express service abstractions, but rather are similar to generic programming in their support for interoperation. The natural signature of a binary operation in a Haskell type class requires both arguments to have the same type; thus, binary operations typed in this way cannot operate on two different implementations of a type class. More fundamentally, different implementations of a type class cannot be manipulated together in a first-class way, for example by putting them in the same list. Achieving direct interoperability with type classes is possible, but requires more complexity in the definition of the type class, so that binary operations take two type arguments as well as two value arguments. In order to regain first-class manipulation—for example to put different instances of different implementations of a type class into the same list—each instance must be wrapped in a datatype, creating an existential that stores the type class of each instance. The use of such an encoding is inconvenient, but I would consider it a use of service abstractions.

Object encodings. Various techniques have been proposed for encoding objects using functions [7, 19]. Not just topics of theoretical study, these techniques have been used to construct frameworks in an object-oriented style within non-OO languages. For example, the GTK+ framework is an object-oriented toolkit for creating graphical user interfaces that is implemented in C. I consider a use of these techniques to be uses of service abstractions.

[17] A language with union types could support heterogeneous lists, but then we cannot use the sets when we get them out of the list without using a typecase operation

[18] Please note that I am not arguing against ADTs. There are good reasons to support ADTs well in programming languages—for example, the increased efficiency that ADTs provide when implementing binary methods. I am only arguing that ADTs do not solve the particular interoperability problem that service abstractions solve. We will see, from a discussion of frameworks and ecosystems, that this problem is important in practice.

[19] Some module import constructs could even be viewed as a simple kind of inheritance—but as usual a detailed examination of this is out of scope.

As an example of such a technique, consider the following Standard ML implementation of a Set service abstraction:[20]

```
datatype IntSet = IntSet of {
  contains : int -> bool,
  isSubsetOf : IntSet -> bool
}

fun makeSingletonSet(x:int) = IntSet {
  contains   = fn(y:int) => (y = x),
  isSubsetOf =
     fn(s:IntSet) =>
        let
          val (IntSet srec) = s
          val scontains = #contains(srec)
        in
          scontains(x)
        end
}
```

Here the `IntSet` object interface is defined as an ML **`datatype`** consisting of a record with two fields. Each field stores a function that implements a method: the `contains` field, for example, holds a function that takes an `int` parameter and returns a `bool` result. The `makeSingletonSet` function creates a singleton set holding the single value `x`, passed as an argument to this "constructor." The constructor returns an instance of the `IntSet` datatype, with the "methods" implemented appropriately to match the behavior of a singleton set. `contains` simply holds a function (introduced with the **`fn`** keyword) that compares the method argument `y` to the single set element `x`. `isSubsetOf` is slightly more complex. We must unpack the `IntSet` datatype `s` to get at the record `srec` inside. Then we can select the `contains` method using ML's record selection operator, `#`. Finally, we can implement `isSubsetOf` by checking if the other set contains the singleton element `x`.

The code above is more verbose than necessary in order to make it more accessible to those not familiar with Standard ML. Using the pattern matching, type inference, and currying facilities of Standard ML, one can define `makeSingletonSet` much more succinctly as follows:

```
fun makeSingletonSet x = IntSet {
  contains   = fn y => y = x,
  isSubsetOf = fn IntSet S =>
                  #contains S x
}
```

Is this solution—or similar encodings [19] in other languages—an adequate replacement for objects? Clearly it is a service abstraction, and therefore realizes all the benefits that objects obtain from their

[20] Thanks to Adam Chlipala for suggesting this encoding style.

service abstraction nature. On the other hand, ML does not provide direct language support for service abstractions, e.g. via class and method constructs; this is instead an encoding, and that involves tradeoffs. In the example above, the drawbacks include some verbosity and awkwardness, caused by the need to wrap and unwrap the `IntSet` datatype to get at the record inside, and the encoding of methods by binding first-class functions to fields. There are potential advantages too—for example, we can avoid some type annotations due to the type inference provided by Standard ML.

For programs that have only incidental need of service abstractions, such encodings may be adequate or even desirable. On the other hand, it is my anecdotal observation that few object-oriented programmers find such encodings acceptable. While the GObject encoding used by GTK does have a significant user base, it is painfully verbose—to the extent that the Vala[21] language and the GObject Builder preprocessor[22] were developed to allow GTK/GObject developers to write higher-level object-oriented code, which is then translated into C code compatible with GObject. This suggests that direct language support for service abstractions is almost a requirement for programs that are expected to benefit significantly from their use. Let us now therefore look at what kind of software requires a significant use of service abstractions, and whether that class of software is important.

4. The Importance of Interoperability

So far I have argued that objects provide a form of extension in which extensions are interoperable, and that this form of extension can be achieved in other technologies only with service abstractions that closely simulate the essence of objects. Does this matter?

To answer this question, let us return to the idea that objects are focused on abstractions for high-level program organization. This suggests that we study how extensibility and interoperability are used in high-level software design. We will first review the theory explaining how modular extensibility can facilitate software change. Second, we will look at how interoperable extensions are leveraged by software frameworks in order to provide a higher-level form of reuse compared to libraries. Third, we will look at how frameworks facilitate software ecosystems, which constitute some of the most high-value software we see today.

Let us first consider the theory of how modular extension facilitates software evolution. The need for a software system to support new, unanticipated implementations of an abstraction was discussed in Parnas's seminal paper on the criteria to be used in decomposing systems into modules [26]. Parnas's argument has become a pillar of software design: nearly all software must change over time, so a software system should be decomposed in a way that hides (i.e. isolates) decisions that are likely to change. The implication is that when change comes, it can be accommodated by providing a new implementation of the abstraction captured by the module's interface.

So extension is important for facilitating software evolution. However, is the interoperability of extensions necessary in practice, beyond examples such as widgets?

4.1 Software Frameworks

To consider whether *interoperation* between extensions is commonly needed, let us turn our attention next to software frameworks. A software framework is "the skeleton of an application that can be customized by an application developer" [15]. This customization occurs when the developer "supplies it [the framework] with a set of components that provide the application specific behavior" [16]. These components are implementations of "abstract designs" defined by the framework; thus frameworks inherently require extensible abstractions.

How do frameworks differ from more well-known approaches to reuse, such as libraries? While libraries typically provide reusable primatives such as functions and data structures, frameworks provide ***architectural reuse*** [10, 14, 15]: reuse of the overall design of an application, along with code that realizes that design. This reused code may include architecturally important abstractions (typically interfaces), default implementations of those abstractions, and glue code that ties the abstractions together and allows them to communicate. The architectural reuse provided by frameworks is inherently higher-order in nature, because the framework code invokes extensions provided by the application.[23]

The importance of frameworks. Why are frameworks interesting to study? First, because the architectural reuse they provide is unique and important, both in theory and in practice, and second, because as we will see, many frameworks would not be possible without the technical benefits provided by service abstractions.

Consider first the theoretical benefits of frameworks in design. Frameworks support reuse at an architectural granularity, which is larger than classes:

> [Frameworks] provide a way of reusing code that is resistant to more conventional reuse at-

[21] `https://wiki.gnome.org/Vala`

[22] `http://www.jirka.org/gob.html` – developed because "writing a fully featured GObject is a hassle"

[23] This is sometimes called the Hollywood principle: "don't call us, we'll call you" [33].

tempts. Application independent components can be reused rather easily, but reusing the edifice that ties the components together is usually possible only by copying and editing it. [16]

Johnson and Foote are arguing that the reuse provided by frameworks is different in scale from the reuse provided by libraries. The higher-order nature of frameworks allows the reuse not just of a data structure or a set of routines, but of infrastructure that ties together many large-scale plugin components. Developers use frameworks because the larger-scale reuse they provide can reduce development times dramatically; Johnson suggests frameworks may reduce development effort by an order of magnitude [15].

Because of the economic benefits of larger-scale, architectural reuse, frameworks have become a critical part of the modern application landscape. Web-based software is almost invariably written on top of a web or application framework such as Rails, Django, Spring, .NET, Servlets, or EJB. The popularity of the Ruby language, in fact, appears to derive mostly from the popularity of Rails. In the mobile space, every Android application must build on the Android framework. The dominant platform for Java tools is the Eclipse framework. Virtually all graphical user interfaces are developed using GUI frameworks such as AWT, SWT, Swing, Qt, and GTK+. Frameworks such as Google's MapReduce and Apache Hadoop are ubiquitous in big data applications.

Do frameworks need objects? Let us now consider whether frameworks can be defined without the service abstractions (e.g. as provided by objects). I first examine the definition of unique design leverage provided by service abstractions described earlier: the ability to define nontrivial abstractions that are modularly extensible, where instances of those extensions can interoperate in a first-class way. I evaluate whether frameworks require each part of this definition, both in the abstract and with respect to the example widget framework.

Abstraction. As described by Johnson and Foote, frameworks unquestionably define abstractions [16]: it is these abstractions that are extended by applications in order to provide application-specific behavior. While some of the abstractions defined by frameworks may be simple observers that could be implemented with a first-class function, many frameworks define nontrivial abstractions. In Apache's Pivot UI framework, the equivalent of the widget interface has nine methods dealing with three separate concerns; it is not trivial. Even the Mapper and Reducer interfaces in Hadoop have three methods each—in addition to the functions the respective interfaces are named after, they provide methods for configuration and teardown.[24] Every major framework of which I am aware similarly defines abstractions that are similarly rich, if not much richer.

Extensibility. The abstractions a framework defines are designed to be extended in order to customize the framework. Meyer's open-closed principle [24] is a cardinal rule of framework extension: the application should extend the framework without modifying its code. For example, when Apache's Pivot UI framework is used, a binary JAR library stores the framework code. Although the code is available in this open-source project, application developers would be ill-advised to modify it; if they did, they would find it more difficult to incorporate framework enhancements and bug fixes into their applications in the future. Thus, extension of framework abstractions must be modular and should not change the framework source code.

Furthermore, frameworks typically must support *unanticipated* extension as well. For example, when one organization defines a framework, it is unrealistic to assume that organization can anticipate all the extensions that strangers will make, because it does not know the particular problems that the extensions are intended to solve. The designers know the framework will be extended *somehow*, and the abstractions the framework provides define the kinds of extensions that are possible, but the designers cannot anticipate the details of specific extension implementations.

Interoperability and uniform treatment. It is possible to design a framework, achieving all the important reuse benefits that frameworks typically bring, without requiring interoperability or uniform treatment of different implementations of the framework abstractions. Such a framework might be instantiated with exactly one implementation for each customizable abstraction it defines. In this case, the framework could be implemented using modules or abstract data types.

The FoxNet system [2], an implementation of a network protocol stack in Standard ML, is a concrete example of this scenario. FoxNet defines a family of abstractions using an ML signature called `PROTOCOL`, with subsignatures for transport protocols such as `TCP` and `UDP` as well as protocols from other layers of the network stack, including `IP`, `DNS`, `ETHERNET`, etc. Different modules implement these abstractions, and an application can be formed by composing the abstractions together. In the FoxNet design, implementations

[24] see `http://developer.yahoo.com/hadoop/tutorial/module5.html` for an example of how this may be useful

of TCP and UDP can be substituted for one another at compile time, but it is not possible to instantiate one of each and choose between them at run time: the abstract data types they define are different. Thus the uniform treatment criterion is not met by this framework—and indeed, FoxNet did not require anything like objects for its implementation.

Most frameworks, however, do rely critically on uniform treatment and/or interoperability. We have already seen that this is important for a UI framework such as Apache's Pivot: the user must be able to hierarchically compose widget instances at run time, which requires a composite widget to store different kinds of child widgets in a data structure, treating them uniformly. Furthermore, the operation to add a child to a composite widget is a binary method, requiring direct interoperability.

Typically, frameworks are not like FoxNet: rather than being limited to a single implementation for each abstraction, they accommodate many implementation plugins for each extension point. As the framework typically owns the application's thread of control, it must invoke operations of plugins at appropriate times. To do this, the framework must keep the various plugins in some kind of data structure and, when some event happens, determine which of the plugins to invoke. Any framework that does this is relying on uniform treatment of implementations, and must be implemented using service abstractions.

Similarly, many frameworks exist specifically in order to enable plugins to interact and build on one another. This is true of UI frameworks, in which programmer-defined widgets can be dynamically selected to be composed (using the Composite pattern) or wrapped (using the Decorator pattern [11]) by framework-defined widgets. This dynamic selection of a programmer-defined plugin to interoperate with framework-defined implementations also requires interoperability of implementations, and must be implemented using service abstractions.

4.2 Interoperability in Particular Frameworks

Let us now consider additional examples of frameworks, illustrating the degree to which they require the unique interoperability of extensions provided by service abstractions.

Servlets. The Java Servlets framework supports the implementation of dynamic web pages. A simplified `Servlet` interface is shown in the next column. The interface encapsulates several concerns. The most important concern is responding to web requests, in the `service` method. Other concerns include lifecycle methods `init` and `destroy` which provide mechanisms for the servlet to initialize itself and clean up resources, respectively; and `getServletInfo`, which is useful for administration of the web server on which the servlets are installed. Although the heart of the abstraction is the `service` method, the other concerns are quite important in practice; the abstraction is not trivially reducible to a single function.[25]

```
interface Servlet {
  void service(Request req, Response res);
  void init(ServletConfig config);
  void destroy();
  String getServletInfo();
  ServletConfig getServletConfig();
}
```

The point of `Servlet` is to support unanticipated extensions of a web server without changing the web server's code. Furthermore, the web server may have several servlets installed, which respond to requests at different URLs. These servlets must necessarily be stored in a data structure mapping each URL to the appropriate servlet, thus requiring uniform treatment of servlet extensions. Therefore, we observe that the full functionality of the servlet framework cannot be realized in a system without using service abstractions.

LLVM. The Low-Level Virtual Machine (LLVM) is "a compiler framework designed to support transparent, lifelong program analysis and transformation for arbitrary programs" [20]. As a framework, it is designed to be extended with new analysis and transformation passes, and we are interested in whether this extensibility support requires service abstractions. LLVM includes a `FunctionPass` abstraction, defined using a C++ class whose simplified interface is shown below. Clearly, `FunctionPass` is a rich abstraction, supporting methods not just for running an analysis or transformation pass (`runOnFunction`), but also a pair of methods for initialization and cleanup, a method for printing analysis results in a human readable form, and a way of specifying usage constraints such as which other analyses this analysis depends on.

```
class FunctionPass {
  virtual bool runOnFunction(Function &F);
  virtual bool doInitialization(Module &M);
  virtual bool doFinalization(Module &M);
  virtual void print(raw_ostream &O...);
  virtual void getAnalysisUsage(Usage &I);
}
```

[25] Interestingly, this is done with a single function in the Rails framework. However, Rails is well-known for its "convention over configuration" philosophy, which makes it very easy to use if your application can fit into Rails's design—and almost impossibly awkward if your application is a poor fit. This suggests that Rails is optimizing for the most common case, while the servlet framework is trying to achieve broad coverage across many kinds of applications.

While the LLVM provides many analyses, it is intended to be extended with analyses for language-specific virtual machines built on the LLVM framework. These extensions should not require modification of the underlying LLVM, and must interoperate with other custom and built-in extensions. For example, a `PassManager` class stores a list of passes and schedules them to run in an efficient order. Because it stores passes in a list, `PassManager` can only be implemented if `FunctionPass` is a service abstraction.

Linux and other C frameworks. GTK+ has already been mentioned as an example of an GUI framework that needs service abstractions for extensibility—but it turns out the Linux operating system can also be viewed as a framework that uses service abstractions. Robert Love states:

> People often miss this, or even deny it, but there are many examples of object-oriented programming in the kernel. Although the kernel developers may shun C++ and other explicitly object-oriented languages, thinking in terms of objects is often useful. The VFS [Virtual File System] is a good example of how to do clean and efficient OOP in C, which is a language that lacks any OOP constructs. [23]

Linux uses service abstractions in order to support multiple file systems. There are vtable-like structures such as `file_operations` that are used to dispatch operations such as `read` to the code that implements file reading in a particular driver. Files in Linux are a rich abstraction, supporting 13 different operations in Linux version 2.4.2, and Linux is intended to support extension with new file systems. These extensions must interoperate: directories from one file system may have symbolic links to files in another file system, for example. Linux file systems also define service abstractions—with dispatch tables—for the mounted file system itself (a superblock), as well as inodes and directory entries.

The use of objects in Linux is not limited to files—there are on the order of 100 different object structures similar to `file_operations` [6]. More broadly, service abstractions are used not just in Linux and GTK, but also in several other C-based framework settings. For example, the GStreamer multimedia framework uses the same object system as GTK+ to support plugins for different container formats, streaming protocols, and codecs, among others. GStreamer supports plugins written by strangers, and plugins can be loaded and selected dynamically based on the media type being played.

Discussion. These frameworks and framework-like systems are not anomalies. Every framework of which I are aware, with the exception of FoxNet, relies on abstractions like these. It is fair to say that without a service abstraction mechanism, software frameworks as we know them, with all of their economic impact, would not exist.

4.3 Interoperability in Software Ecosystems

Interoperability is important, not only because of the reuse provided by frameworks, but also because it supports the development of software ecosystems. A ***software ecosystem*** is a "set of software solutions that enable, support, and automate the activities...[of] actors in the associated social or business ecosystem..." [4]. Examples of software ecosystems include mobile platforms such as iOS and Android, operating systems such as Windows or Linux, application suites such as Microsoft Office, tool platforms such as Eclipse, and retail marketplaces such as Amazon's.

Software ecosystems have enormous economic importance because of network effects [17]: a platform such as the iPhone becomes more valuable the more people use it, and (critically for our purposes) the more apps that are available on the platform. To consider the value of networks, consider that in 2012, the largest company in the world by market capitalization was Apple, whose success is largely attributable to its control of the iOS ecosystem; and the four next technology companies were Microsoft, IBM, Samsung, and Google, each of which is notable for its central involvement in one or more software ecosystems.[26]

Ecosystems may or may not be based on frameworks in the sense that Johnson defines them, but they have similar requirements for extensibility and interoperability. Bosch argues that extensibility is a critical success factor of both operating system-centric and application-centric software ecosystems [4]. However, it is also critical that independently developed extensions interoperate. Interoperation of independent extensions is enabled in software ecosystems because "the architecture provides a formalization of the rules of interoperability and hence teams can, to a large extent, operate independently" [4].

Android. The Android platform illustrates the critical need for interoperability in software ecosystems. One of the benefits of Android is that user-defined apps can substitute for system apps. For example, the Smartr Contacts app[27] provides the same functionality as Android's built in Contacts application, but adds features

[26] Source: Financial Times Global 500 2012, available at `http://www.ft.com/intl/companies/ft500`

[27] `https://www.xobni.com/download/android`

such as contact synchronization, integration with Twitter and Facebook, and an integrated history of interactions. When another app, such as the phone dialer, needs contact information, the user is asked which contact manager to use, and the Smartr Contacts app can be dynamically chosen. The ability to replace default apps with customized ones is notably used to develop skins such as the recently announced Facebook Home,[28] which overrides many default elements of the Android platform with Facebook-enhanced functionality.

Android is an interesting case because on the Android platform different apps execute in different virtual machine processes, so they are not directly using object-oriented method calls to communicate. However, the processes nevertheless have an service-based interface, in the sense that they communicate only via messaging, and in the sense that different implementations of functionality (e.g. different app that can be used to select a contact) are dynamically substitutable.

In addition to the abstract use of service abstractions in the inter-app Android framework, there is Android framework code within each app that critically depends on the service abstraction capabilities of objects. For example, the Android contacts application makes its data available using a `ContentProvider` object, a portion of whose interface is shown below.[29] The `ContentProvider` abstraction is rich, consisting of query, insert, update, and delete operations (among others). Different kinds of data need different `ContentProvider`s in order to support queries and other operations in a way that is appropriate to that kind of data. An app that manages several kinds of data can therefore define several `ContentProvider`s, and the Android framework must manage them all in a data structure in order to determine which to use when a request for content comes in. Thus, even within the application, content providers (and many other abstractions) require the features provided by service abstractions.

```
class ContentProvider {
  abstract Cursor query(Uri uri, ...);
  abstract int insert(Uri uri,
                      ContentValues vals);
  abstract Uri update(Uri uri,
                      ContentValues vals,
                      ...);
  abstract int delete(Uri uri, ...);
  ... // other methods not shown
}
```

[28] `https://www.facebook.com/home`

[29] `ContentProvider` is actually a class that provides some reusable method implementations (not shown) to subclasses, but the abstract methods shown here form a subinterface that all content providers must implement.

This is neither an isolated nor an unimportant example. The content provider functionality is the most important means by which applications in Android share data; it is critical to the economic network effect where the availability of one app on Android makes another app more valuable. Nor is the use of service abstractions incidental. I am not merely observing that Android uses objects for this abstraction; I am arguing that *any Android-like framework that provides support for content providers must do so via service abstractions*. That is because the framework must accept multiple implementations of the content provider from the application, and it must store these implementations uniformly in a data structure; this is possible only if content providers are service abstractions.

Now, a non-OO app platform might force each app to provide a single content provider implementation that serves all the different kinds of content from that app, but this solution sacrifices the (substantial) reused code in the framework that dispatches incoming requests to different content providers according to the type of content requested. So without the use of service abstractions, each app would have to reimplement this code—ultimately making app interoperability more difficult, and thereby lowering the value of the entire ecosystem. This illustrates concretely how objects facilitate reuse: the service abstraction provided by objects is what enables the content provider framework code to be reused.

Microsoft COM applications and Office plugins. Microsoft COM is an interface standard that enables applications such as Microsoft Word to communicate with each other and with plugins written by strangers. An ecosystem has grown up in particular around Microsoft's Office suite, with many third-party developers writing Office plugins, often using Visual Basic to conveniently provide a GUI for the plugin. These plugins can be dynamically added to Office, where they coexist with the native Win32 Office widgets. Many Pffice plugins either build on other plugins, or incorporate GUI widgets provided by third parties. The result is one of the most robust component marketplaces in the software industry. Cross-plugin, cross-widget, and cross-language interoperation could not be achieved without the service abstractions provided by Microsoft's Component Object Model (COM).

Coppit and Sullivan also studied the reuse achievable based on the COM model and its extensions, OLE and Active Document [9]. Observing that component-based software development has largely been elusive, they explore a model in which the components are not simply libraries, but are full-fledged applications themselves, developed by different companies.[30] The model was successful, partly due to business factors (as with other ecosystems), but also due to technical factors such as the interoperability and shared architecture provided by COM and related standards.

4.4 The Architectural Flexibility of Objects

Recently, Bracha *et al.* proposed the Newspeak module system, in which every module is parametric in its dependencies [5]. This design is intended to reduce coupling between modules, and at the same time subsume ideas like dependency injection.

The Newspeak design stands in contrast to the discussion of *fully functorized* (i.e. fully parameterized) modules in Harper and Pierce's chapter on module systems [12]. Harper and Pierce state that "Experience has shown this to be a bad idea: all this parameterization—most of it unnecessary—gives rise to spurious coherence issues, which must be dealt with by explicitly (and tediously) decorating module code with numerous sharing declarations, resulting in a net decrease in clarity and readability for most programs." Why is this not a problem for Newspeak?

The obvious answer is that Newspeak is not statically typed, and sharing declarations are about making sure static types that have to match up, do. But a more subtle examination shows that the problem of proliferating sharing declarations exists only because of abstract types. The problem occurs when we have a module M that defines an abstract type t, and two different libraries L1 and L2 depend on module M. The abstract type t may then show up in the signature of L1 and L2. Now imagine that we have a program P that is intended to import L1 and L2, but we wish to make it parametric in the actual implementation of L1 and L2. We need the type t that appears in L1 and L2 to be the same, so that we can pass values of the abstract type back and forth between L1, L2, and P; in ML, this constraint is enforced by a sharing declaration. These declarations proliferate, because if we have n abstract types, each of which is shared between m imported modules, we will have $O(n \times m)$ sharing declarations.

What if all abstractions used object types, rather than abstract data types? If we take Cook's definition of objects to be definitive, then an object type does not denote a particular, but hidden, representation; instead, an object type primarily characterizes the set of methods to which an object responds (i.e. a service abstraction). This notion of object type is also reflected in typed object calculi, many of which use sets of methods as types [1].[31] The definition of the type in the signature of each module says what the methods are, so there is never any question about whether types from two imported modules match. Thus sharing declarations are unnecessary. Hence, the use of abstract data types rather than object types increases coupling between modules and impedes the flexibility of code.

The problem described above is roughly analogous to "DLL Hell," in which a program needs to use two libraries, but those libraries depend on different versions of a third library [30]. Object designs can still suffer from these problems in cases where incompatible changes are made to an object's interface between versions. But where the core interface is compatible, objects sidestep the problem entirely, because different implementations of the same (pure) object interface will always be interoperable at the type level. To the extent that Java programs suffer from an analogous "JAR Hell," it may be attributable in part to the fact that Java's object model is not pure, but rather, as discussed by Cook, it contains some support for abstract data types as well.

5. Conclusion

In summary, I have argued that objects provide a unique form of service abstraction that supports interoperable extensions. This interoperability cannot be duplicated in other programming paradigms without likewise creating service abstractions, thus simulating the essence of objects. Furthermore, service abstractions are not just of academic interest—they are the foundation of software frameworks and ecosystems, which arguably constitute the richest form of software reuse currently known, and the most significant source of value in the software industry, respectively. Assuming that the market's adoption of programming language technology is even somewhat rational over the long term, these forces ought to be sufficient to make direct language support for service abstractions inevitable. This, in turn, explains the rise of object-oriented programming languages: they represent the only class of programming languages to date that provides good support for service abstractions.

Implications. An implication for future language designers is that if a language is to be a suitable foundation for a software ecosystem, or indeed to support architectural reuse in any application domain, it should have good support for interoperable extensions—and therefore for service abstractions. Left open, for now,

[30] The case study was done with Microsoft Word and Shapeware's Visio. Ironically, Microsoft purchased Visio shortly after the case study was completed.

[31] While my argument about modularity here focuses on static types, note that the run-time type of an object in a dynamic language can also often be thought of as a set of methods: you get a method-not-understood error if you call a method not in the set.

is what form that support should take: row polymorphism with subtyping as in OCaml, Go's interfaces, and delegation as in Self appear to support interoperable extension just as well as the inheritance constructs common to today's popular industrial languages. Furthermore, an argument for object-orientation need not be an argument against other paradigms. Good support for service abstractions is compatible with good support for ADTs, for functional programming, and for many other desirable language features.

Predictions. A scientific hypothesis should not only explain, but also make testable predictions. In that spirit, I offer two predictions suggested by the hypothesis described above concerning the reasons behind the success of objects. The following hypotheses can be tested either by experiment, or in the natural course of the future evolution of software, languages, and tools:

- Lightweight, first-class modules are service abstractions in that they provide unanticipated extension of rich abstractions, and interoperability of the extensions via module signature subtyping. If such a module system is added to a language, such as Standard ML, that does not currently have good support for objects, framework-like designs will begin to show up in the enhanced language.
- A practical, statically typed object-oriented language can be designed to support Newspeak-style modules that are parametric in their dependencies, provided that all types are given in an object-oriented, rather than an ADT, style.

Future work. There is much more to be done in exploring the potential benefits (or lack thereof) of objects. I have focused on technical benefits, but non-technical benefits (e.g. of a psychological nature) may be important too. Furthermore, I have focused on service abstractions as representing the core of objects, but in doing so I have neglected other interesting features of typical object-oriented programming languages. Tempero *et al.* have shown that a broad range of object-oriented programs use inheritance frequently and in non-trivial ways [31]; more research is needed to better understand the benefits (and possible drawbacks) of inheritance, and to explain why it is so widely used.

It is my hope that the argument, hypotheses, and predictions above will be further tested and refined in future research, ultimately contributing to a broader and deeper understanding of the role service abstractions play in modern software systems.

Acknowledgments

I thank William Cook for inspiring this essay with his earlier one, and the many people with whom I have informally discussed the arguments in this essay. I also thank Marwan Abi-Antoun, Luís Caires, Adam Chlipala, Karl Crary, Ray Dillinger, Richard Gabriel, David Garlan, Alec Heller, Stefan Holdermans, Ralph Johnson, James Koppel, Darya Kurilova, Du Li, Michael Maass, Gregory Malecha, Sean McDirmid, Jeff Meister, Matthew Might, Scott McMurray, Brad Myers, Robb Nebbe, Cyrus Omar, Simon Peyton-Jones, Alex Potanin, Anton van Straaten, Éric Tanter, Philip Wadler, Todd Wilson, John Zabroski, and the anonymous reviewers for comments and suggestions on earlier drafts of this essay.

The writing of this essay was supported by the U.S. National Science Foundation under grant #CCF-1116907, "Foundations of Permission-Based Object-Oriented Languages," the U.S. Department of Defense, and the Air Force Research Laboratory.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] E. Biagioni, R. Harper, and P. Lee. A network protocol stack in Standard ML. *Higher Order Symbolic Comput.*, 14 (4):309–356, Dec. 2001.

[3] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In *Object-Oriented Programming Systems, Languages, and Applications*, 1986.

[4] J. Bosch. From software product lines to software ecosystems. In *Software Product Line Conference*, 2009.

[5] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in Newspeak. In *European Conference on Object-Oriented Programming*, 2010.

[6] N. Brown. Object-oriented design patterns in the kernel, part 1. *Linux Weekly News*, 2011. URL `http://lwn.net/Articles/444910/`.

[7] K. Bruce, L. Cardelli, and B. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software*. 1997.

[8] W. R. Cook. On understanding data abstraction, revisited. In *Onward! Essays*, 2009.

[9] D. Coppit and K. J. Sullivan. Multiple mass-market applications as components. In *International Conference on Software Engineering*, 2000.

[10] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, Oct. 1997.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] R. Harper and B. C. Pierce. Design considerations for ML-style module systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2004.

[13] T. Hoare. Null references: The billion dollar mistake. In *QCon London*, 2009.

[14] C. N. C. Jaspan. Proper plugin protocols. Carnegie Mellon University Ph.D. Dissertation, available as technical report CMU-ISR-11-116, 2011.

[15] R. E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, Oct. 1997.

[16] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[17] M. L. Katz and C. Shapiro. Network externalities, competition, and compatibility. *The American Economic Review*, 75(3):424–440, 1985.

[18] A. C. Kay. The early history of Smalltalk. In *History of programming languages—II*, 1993.

[19] O. Kiselyov and R. Lämmel. Haskell's overlooked object system. Manuscript available at `http://arxiv.org/abs/cs/0509027`, 2005.

[20] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization*, 2004.

[21] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley, 1993.

[22] B. Liskov. A history of CLU. In *History of programming languages—II*, 1993.

[23] R. Love. *Linux Kernel Development (2nd Edition)*. Novell, 2nd edition edition, 2005.

[24] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[25] D. R. Musser and A. A. Stepanov. Generic programming. In *International Symposium on Symbolic and Algebraic Computation*, 1988.

[26] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15 (12):1053–1058, 1972.

[27] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Advances in Algorithmic Languages*, 1975.

[28] A. Schwill. Cognitive aspects of object-oriented programming. In *IFIP WG 3.1 Working Conference "Integrating Information Technology into Education"*, 1994.

[29] A. Stepanov. STLport: An interview with A. Stepanov. Available at `http://www.stlport.org/resources/StepanovUSA.html`.

[30] C. Szyperski. Greetings from DLL hell. *Dr. Dobbs Journal*, Oct. 1999.

[31] E. Tempero, H. Yang, and J. Noble. What programmers do with inheritance in Java. In *European Conference on Object-Oriented Programming*, 2013.

[32] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Object-oriented programming systems, languages, and applications*, 1987.

[33] J. Vlissides. Protection, part 1: The Hollywood principle. *C++ Report*, Feb. 1996.

[34] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages*, 1989.

A. Supporting Data

As of 4/7/2013, LangPop.com's top languages were, in order: C, Java, C++, PHP, JavaScript, Python, C#, Perl, SQL, and Ruby. All of these languages except C and SQL have object-oriented features. The Wikipedia page on Object-Oriented Programming[32] describes Java, C++, JavaScript, Python, C#, and Ruby as being primarily object-oriented languages; PHP and Perl have object-oriented features but were designed (and are still probably used) primarily for procedural programming.

At the same date, the corresponding top languages at the TIOBE index were, in order: C, Java, C++, Objective-C, C#, PHP, (Visual) Basic, Python, and Perl. All but C have object-oriented features, and I consider all but C, PHP, Perl, and (Visual) Basic to be primarily object-oriented.

[32] `http://en.wikipedia.org/wiki/Object-oriented_programming`