The Power of Interoperability: Why Objects Are Inevitable

Onward! Essay, 2013

http://www.cs.cmu.edu/~aldrich/papers/objects-essay.pdf

Comments on this work are welcome. Please send them to aldrich at cmu dot edu

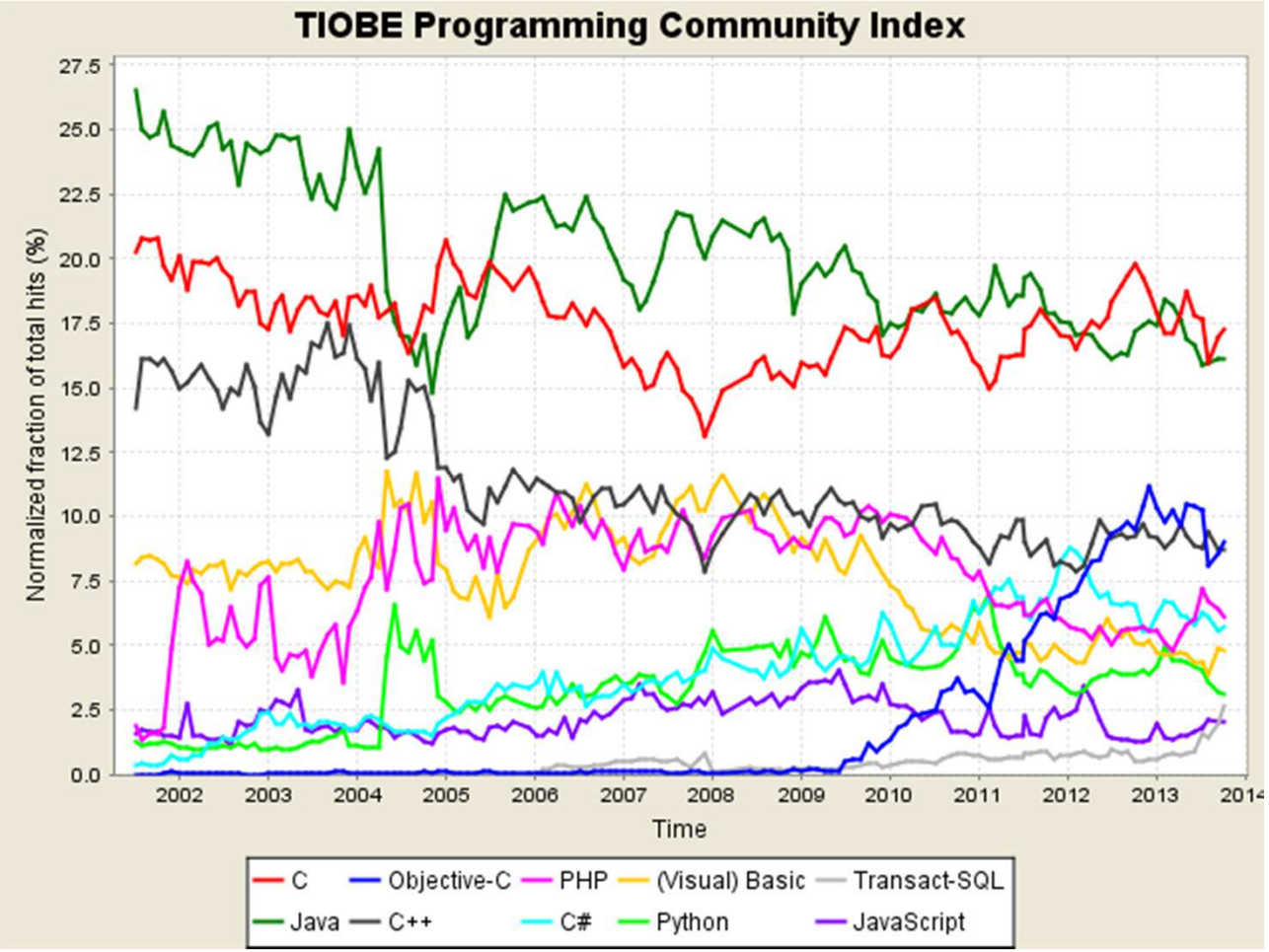
Jonathan Aldrich

Institute for Software Research

School of Computer Science

Carnegie Mellon University

Object-Oriented Programming is Widespread



6-8 of top 10 PLs are OO – TIOBE

Object-Oriented Programming is Influential

- Major conferences: OOPSLA, ECOOP
- Turing awards for Dahl and Nygaard, and Kay
- Other measures of popularity
  - Langpop.com: 6-8 of most popular languages
  - SourceForge: Java, C++ most popular
  - GitHub: JavaScript, Ruby most popular
  - Significant use of OO design even in procedural languages
    - Examples: GTK+, Linux kernel, etc.

- Why this success?

OOP Has Been Criticized

"I find OOP **technically unsound… philosophically unsound… [and] methodologically wrong.**"

- Alexander Stepanov, developer of the C++ STL

Why has OOP been successful?

Why has OOP been successful?

"…it was hyped [and] it created a new software industry."

- Joe Armstrong, designer of Erlang

Marketing/adoption played a role in the ascent of OOP.

But were there also genuine advantages of OOP?

Why has OOP been successful?

"the object-oriented paradigm...is consistent with the natural way of human thinking"

- [Schwill, 1994]

OOP may have psychological benefits.

But is there a technical characteristic of OOP that is critical for modern software?

What kind of technical characteristic?

Talk Outline

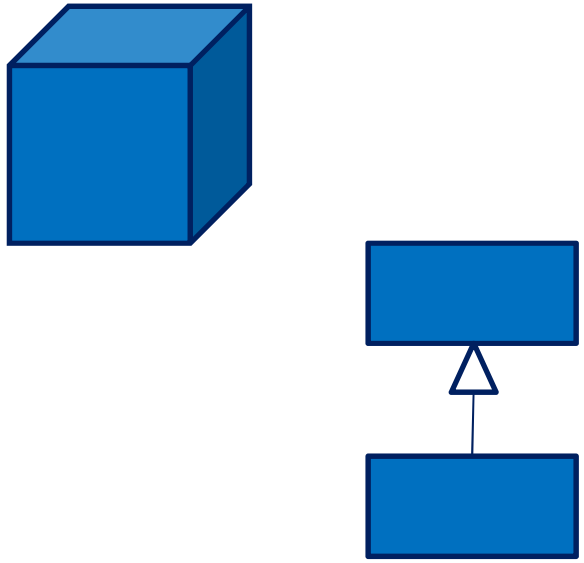
1. A technical characteristic **unique** to objects
   - Addressed in Cook's 2009 Onward! Essay

2. That has a **big impact**
   - Our focus: why that characteristic matters
   - I.e. how it affects **in-the-large software development**

What Makes OOP Unique?

Candidates: key features of OOP

- Encapsulation?
  - Abstract data types (ADTs) also provide encapsulation
- Inheritance?

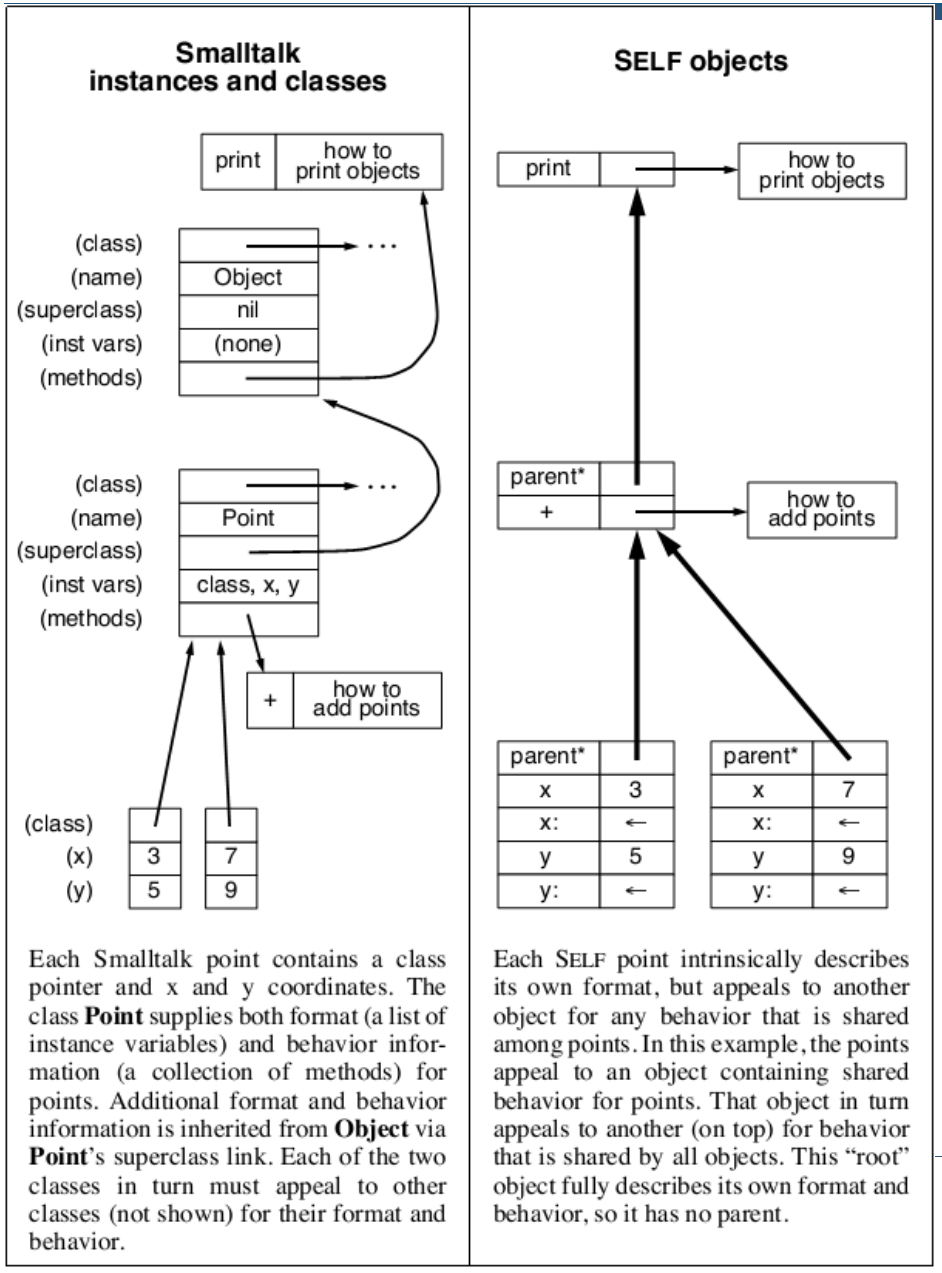


Not all OO Languages Have Inheritance

- A Modern Example: Go
  - Provides encapsulation, interfaces, dynamic dispatch
  - But no inheritance of code from a superclass

- An Alternative: Delegation
  - Supported in Self, JavaScript, others
  - There are no classes, only objects. To get a new object:
    - Create an empty object
      - Add things to it
      - Optionally, delegate to an existing object via a *parent* field
        - If you call method *m* on an object, and *m* is not defined, the system will look for it in the parent object
    - Clone an existing object

Inheritance vs. Delegation, Graphically



Each Smalltalk point contains a class pointer and x and y coordinates. The class **Point** supplies both format (a list of instance variables) and behavior information (a collection of methods) for points. Additional format and behavior information is inherited from **Object** via **Point**'s superclass link. Each of the two classes in turn must appeal to other classes (not shown) for their format and behavior.

Each SELF point intrinsically describes its own format, but appeals to another object for any behavior that is shared among points. In this example, the points appeal to an object containing shared behavior for points. That object in turn appeals to another (on top) for behavior that is shared by all objects. This "root" object fully describes its own format and behavior, so it has no parent.

The Self project also had a big impact on optimization of dynamic compilers

- E.g. [Chambers & Ungar, 1989]
- Used in Java, JavaScript, etc.

Source: Ungar and Smith. Self: The Power of Simplicity. *Lisp and Symbolic Computation*, 1991.

Inheritance has Benefits, Drawbacks

- Benefits
  - No easier way to reuse a partial implementation of an abstraction
  - Alternative requires forwarding each method individually
  - Especially useful when subclass and superclass call each other
    - E.g. a class with both super calls and a template method
    - Implementing Template Method, Factory is awkward in Go
      [Schmager, Cameron, and Noble 2010]

- Drawbacks
  - Tight coupling between subclass and superclass
    - E.g. fragile base class problem

Fragile Base Class Problem

```
class List {
   private Link head;
   public void add(int i) {…}
   public void addAll(List l) {…}
   public int size() {
      … // traverses the list
   }
}

class CachedSizeList extends List {
   private int cachedSize;
   public int size() { return cachedSize; }
   public void add(int i) {
      cachedSize++;
      super.add(i);
   }
   // do we need to override addAll?
}
```

- Correct impl of subclass depends on the base class implementation
  - Couples classes, breaks modularity
- Worse: if the base class changes, the subclass will be broken
- What causes this coupling *is also what makes the template method pattern work!*
- Some solutions
  - Document internal method calls that can be intercepted
    - Document whether addAll() calls add()
  - Only make self-calls to **`abstract`** or **`final`** methods
  - Selective open recursion – language feature describes which methods are used for downcalls [Aldrich and Donnelly, 2004]

Inheritance has Benefits, Drawbacks

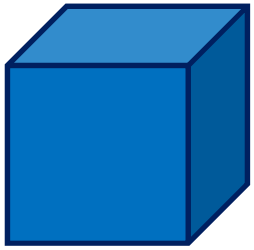
- Benefits
  - No easier way to reuse a partial implementation of an abstraction
  - Alternative requires forwarding each method individually
  - Especially useful when subclass and superclass call each other
    - E.g. a class with both super calls and a template method
    - Implementing Template Method, Factory is awkward in Go
      [Schmager, Cameron, and Noble 2010]

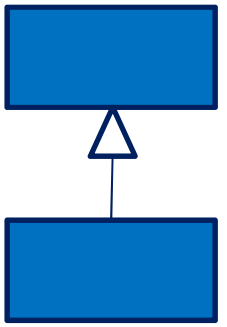
- Drawbacks
  - Tight coupling between subclass and superclass
    - E.g. fragile base class problem
  - Drawbacks mitigated by careful methodology

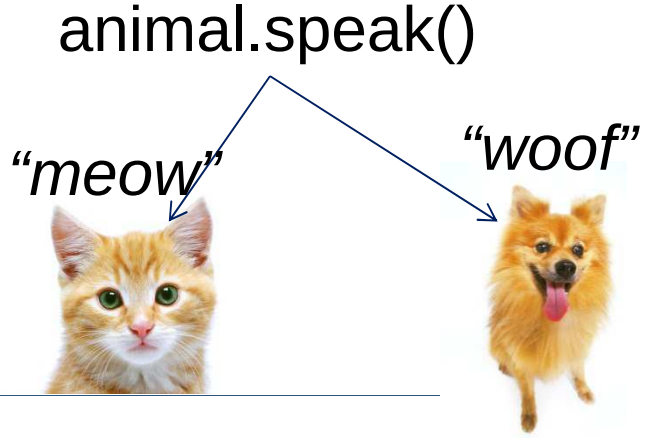
What Makes OOP Unique?

Candidates: key features of OOP

- Encapsulation?
  - Abstract data types (ADTs) also provide encapsulation
- Inheritance?
  - Neither universal nor unique in OOPLs
  - Worth studying, but not our focus
- Polymorphism/Dynamic dispatch?
  - Every OOPL has dynamic dispatch
  - Distinguishes objects from ADTs







Dynamic Dispatch as Central to OOP

Significant grounding in the OO literature

- Cook's 2009 Onward! essay
  - Object: "value exporting a procedural interface to data or behavior"
  - Objects are self-knowing (*autognostic*), carrying their own behavior
  - Equivalent to Reynolds' [1975] *procedural data structures*

- Historical language designs
  - "the big idea [of Smalltalk] is messaging" [Kay, 1998 email]

- Design guidance
  - "favor object composition over class inheritance" [Gamma *et al.* '94]
  - "black-box relationships *[based on dispatch, not inheritance]* are an ideal towards which a system should evolve" [Johnson & Foote, 1988]

Objects vs. ADTs

Two Object-Oriented Sets

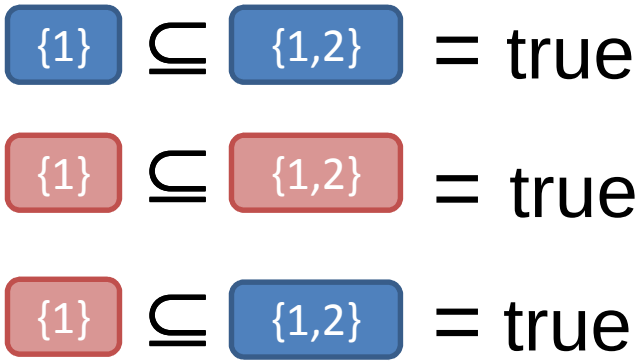
```
interface IntSet {
    bool contains(int element)
    bool isSubsetOf(IntSet otherSet)
}
```

Interface is a set of **messages**

All communication is message-based; isSubsetOf() implemented by calling contains() on otherSet

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

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



Set Objects

Different implementations interoperate freely

Objects vs. **ADTs**

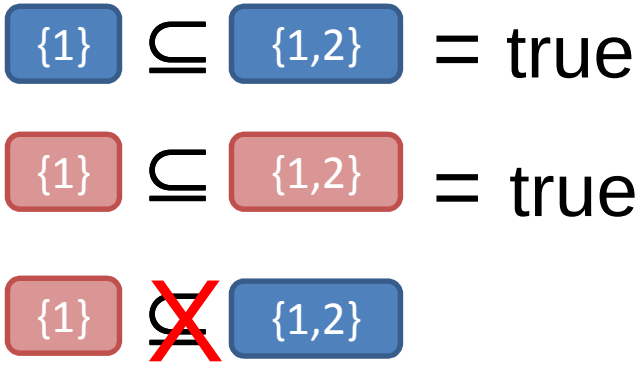
Two Set ADTs

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

// in main()
IntSet sA = new IntSetA(...);
IntSet sB = new IntSetB(...);
bool x = sA.isSubsetOf(sB); // ERROR!
```

Interface is a set of **operations** over a **fixed** but hidden type (IntSetA)

isSubsetOf() is a binary method that only works with other instances of IntSetA. Good for performance.



Set ADTs

Different ADT implementations cannot interoperate

ADT Performance Up Close

- Assume Set ADT uses sorted array
- isSubsetOf can be O(n)
  - Single linear scan through both arrays
  - Find each element of first array in second
  - Because both are sorted, never backtrack in second array
- OO implementation cannot use this trick
  - The other object may not be implemented based on an array
    - Unless we do an **instanceof** test – but that is at least ugly and at worst unmodular
  - The only possible implementation makes O(n) calls to contains()
  - If a contains() call is worse than O(1), we lose performance
- Java, C++ support both ADT-style and OO-style abstractions
  - Ironically, OO languages helped make ADTs mainstream, and vice versa
  - Neither objects nor ADTs is "better" than the other
    - They are design styles (or patterns) that are useful in different situations

Does Interoperability Matter?

- For data structures such as Set, maybe not
  - Maybe optimization benefits of ADTs dominate interoperability

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

  - A History of CLU [Liskov, 1993]

- But are data structures what OOP is really about?

Are Objects "Procedural Data Structures?"

An object is "...a value exporting a procedural interface to data ***or behavior***." [Cook, 2009]

"a program execution is regarded as a physical model, ***simulating the behavior*** of either a real or imaginary part of the world"

[Madsen, Møller-Pedersen, Nygaard (and implicitly Dahl), 1993]

"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." [Kay, 1993]

Service Abstraction

- Objects can implement data structures
  - Useful, but not their **primary purpose**
  - Not a **unique** benefit of objects

- Kay [1993] writes of the "objects as server metaphor" in which every "object would be a server offering services" that are accessed via messages to the object.

- A better term is ***service abstraction***
  - Definition: a value exporting a procedural interface **to behavior**
  - Identical to procedural data abstraction, but focused on behavior
  - Captures the characteristic of objects in which we are interested

Side Note: Distributed Objects

- Kay's work on objects was inspired by network servers
  - Xerox PARC was a leader in networking research at the time
  - Other OO languages took this quite literally
- Emerald provided a single distributed virtual machine
  - One program running on many server nodes
  - Every object has a globally unique address
  - Objects can refer to, or invoke methods of, remote objects
  - Objects can move themselves to other nodes
    - Generally for efficiency purposes
  - Fun problems in optimization, distributed GC
- Emerald, like Smalltalk, was beautiful but never caught on
  - The most attractive distributed programming model of which I know
  - Practical challenges
    - Interoperation with systems not written in Emerald
    - Abstractions like global address space, GC are nice, but also expensive
    - Often distributed system programmers want lower-level control

Service Abstraction provides Interoperability

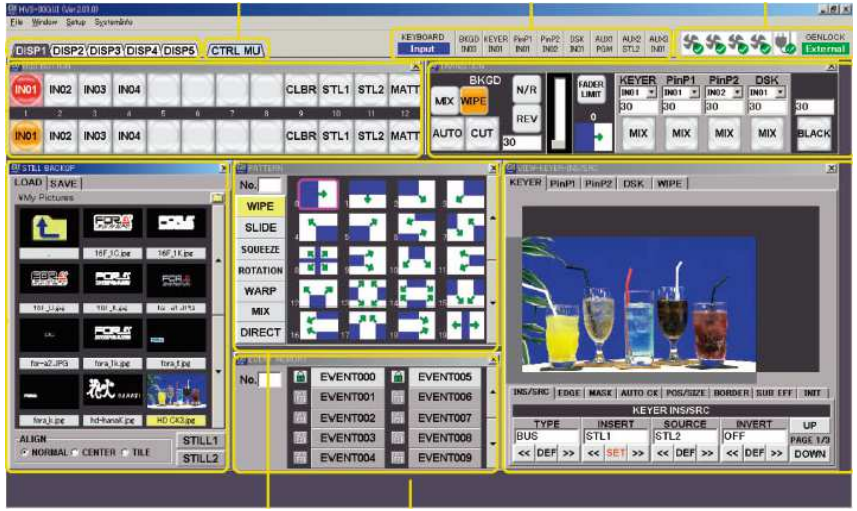
- Let's assume service abstraction is the core of OO

- What are the benefits of service abstraction?
  - Reynolds/Cook: procedural data abstraction provides ***interoperability***
  - But so do functions, type classes, generic programming, etc.

- **What makes service abstraction unique?**

Interoperability of Widgets

- Consider a Widget-based GUI
  - Concept notably developed in Smalltalk



Source: http://www.for-a.com/products/hvs300hs/hvs300hs.html

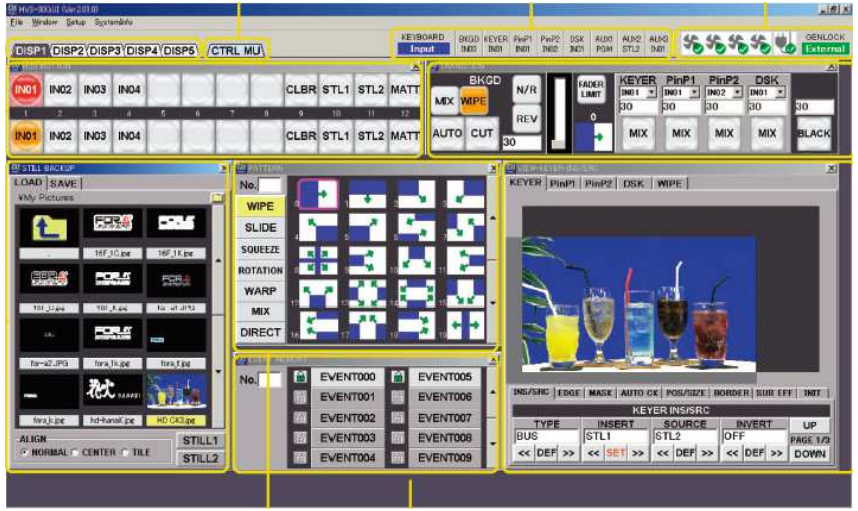
```
interface Widget {
    Dimension getSize();
    Dimension getPreferredSize();
    void setSize(Dimension size);
    void paint(Display display);

    … /* more here */ }          // based on ConstrainedVisual from Apache Pivot UI framework
```

- Nontrivial abstraction – not just paint()
  - A single first-class function is not enough

Interoperability of Composite Widgets

- Consider a **Composite GUI**
  - Concept notably developed in Smalltalk



Source: http://www.for-a.com/products/hvs300hs/hvs300hs.html

```
class CompositeWidget implements Widget {
    Dimension getSize();
    Dimension getPreferredSize();
    void setSize(Dimension size);
    void paint(Display display);
    void add(Widget widget)
    … /* more here */ }          // based on Container from Apache Pivot UI framework
```

- Nontrivial abstraction – not just paint()
  - A single first-class function is not enough
- Composite needs to store diverse subcomponents in a list
  - Can't do this with type classes, generic programming
- Composite needs to invoke paint() uniformly on all subcomponents
  - Also breaks type classes, generic programming

Design Leverage of Service Abstractions

*The ability to define **nontrivial abstractions** that are **modularly extensible**, where instances of those extensions can **interoperate in a first-class way**.*

- **Nontrivial abstractions**
  - 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**
  - Interoperability of **binary methods**
    - Such as adding a subcomponent to a composite

  - **First-class manipulation** of different implementations
    - Such as putting subcomponents in a list
  - **Uniform treatment** of different implementations
    - Such as invoking paint() on all subcomponents

Talk Outline

1. A technical characteristic **unique** to objects
   - Objects, for our purposes, are **service abstractions** that provide **dispatch**
   - Service abstractions uniquely provide **first-class interoperability**

2. That has a big **impact**
   - Well, first-class interoperability is nice for GUIs
   - Does this affect **in-the-large software development** more broadly?

Large-Scale Development Impact

- How might service abstractions impact in-the-large software development?

- Some hints
  - We are likely looking for an approach to **design**
  - We already know service abstractions are useful for **GUIs**
  - Anecdotally, one can argue that GUIs drove OO
    - Smalltalk, MacApp, Microsoft Foundation Classes, Java Applets, …
  - What are these GUI designs an instance of?

- A likely candidate: **software frameworks** [Johnson, 1997]

Software Frameworks

- A framework is "the skeleton of an application that can be customized by an application developer" [Johnson, 1997]

- Frameworks uniquely provide **architectural reuse**
  - Reuse of "the edifice that ties components together" [Johnson and Foote, 1988]
  - Johnson [1997] argues can reduce development effort by 10x

- As a result, frameworks are ubiquitous
  - GUIs: Swing, SWT, .NET, GTK+
  - Web: Rails, Django, .NET, Servlets, EJB
  - Mobile: Android, Cocoa
  - Big data: MapReduce, Hadoop

Frameworks need Service Abstraction

- Frameworks define **abstractions** that extensions implement
  - The developer "supplies [the framework] with a set of components that provide the application specific behavior" [Johnson and Foote, 1988]
  - Sometimes the application-specific behavior is just a function
  - More often, as we will see, these abstractions are **nontrivial**
- Frameworks require **modular extensibility**
  - Applications extend the framework without modifying its code
    - Frameworks are typically distributed as binaries or bytecode
    - *cf.* Meyer's [1988] open-closed principle
  - Framework developers cannot anticipate the details of extensions
    - Though they do plan for certain kinds of extensions
- Frameworks require **first-class interoperability**
  - Plugins often must **interoperate** with each other
  - Frameworks must **dynamically** and **uniformly** manage diverse plugins
  - We have already seen this for GUI widgets – let's look at other examples

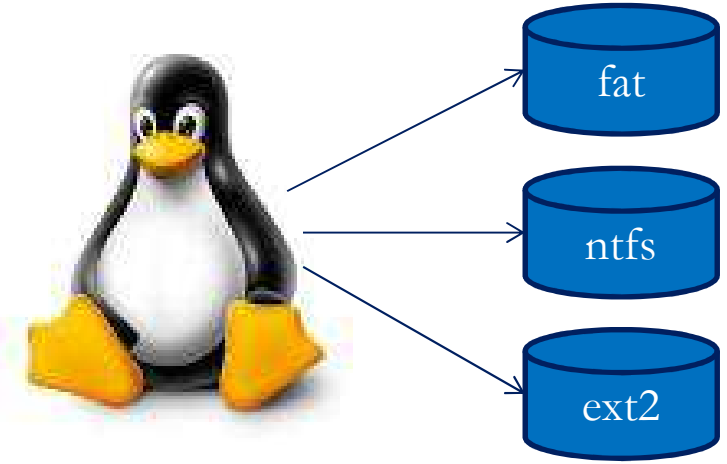
Web Frameworks: Java Servlets

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

- Nontrivial abstraction
  - Lifecycle methods for resource management
  - Configuration controls
- Modular extensibility
  - Intent is to add new Servlets
- First-class interoperability required
  - Web server has a list of diverse Servlet implementations
  - Dispatch is required to allow different Servlets to provide their own behavior

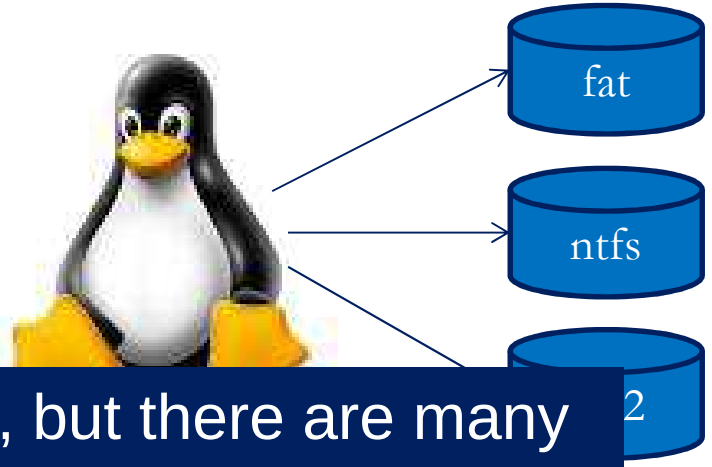
Operating Systems: Linux

- Linux is an OO framework!
  - In terms of design—not implemented in an OO language
- File systems as service abstractions
  - Interface is a struct of function pointers
  - Allows file systems to interoperate
    - E.g. symbolic links between file systems
- Not just file systems
  - Many core OS abstractions are extensible
  - ~100 Service abstractions in the kernel



Operating Systems: Linux

- Linux is an OO framework!
  - In terms of design—not implemented in an OO language
- File systems as service abstractions
  - Interface is a struct of function



People often miss this, or even deny it, but there are many examples of object-oriented programming in the kernel. Although the kernel devel-opers 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.

- Robert Love, *Linux Kernel Development (2nd Edition)*

Objection: If I want objects, I can build them!

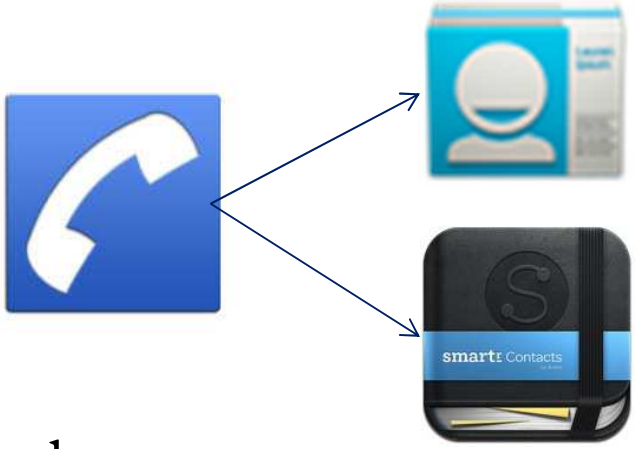
- Works nicely in a dynamically-typed setting with macros
  - Exhibit A: PLT Scheme / Racket
- Works poorly in a statically typed language
  - Certainly possible [Kiselyov and Lämmel, 2005]
  - Painful in C, Standard ML, Haskell, etc.
    - No built-in type gives you exactly what you want
    - Annoying object packing/unpacking is necessary
    - Feels like an encoding, rather than a natural expression of ideas
  - Typed Racket works because of special OO types
- Programmers do it when really necessary
  - *cf.* GTK+ GUI framework, Microsoft COM, Linux drivers, etc.
  - My take: people only do this if OO languages are excluded *a priori*

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" [Bosch, 2009]
  - Examples: iOS, Android, Windows, Microsoft Office, Eclipse, Amazon Marketplace, …
- Ecosystems have enormous **economic impact**
  - Driven by network effects [Katz and Shapiro, 1985]
  - Top 5 tech firms control or dominate an ecosystem
    - Apple, Microsft, IBM, Samsung, Google
- Ecosystems require **first-class interoperability**
  - Critical to achieving benefit from network effects
  - "the architecture provides a formalization of the rules of interoperability and hence teams can, to a large extent, operate independently" [Bosch, 2009]

Mobile Devices: Android

```
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
}
```



- Network effects (apps) give Android value
- Apps build on each other
  - Example: contact managers
    - Smartr Contacts is a drop-in replacement for the default contact manager
    - Phone, email apps can use Smartr Contacts *without preplanning*
  - Enabled by service abstraction interfaces
    - Android keeps a list of heterogeneous ContentProvider implementations

ADTs vs. Objects, Revisited

- What have we learned about objects?
- Especially compared to alternatives such as abstract types (ADTs)
  - *cf.* 15-150: functional programming languages focus on good support for ADTs, but not (typically) objects

- ADTs are more useful in performance-critical algorithmic code
- Objects are more useful for large-scale program organization
  - Particularly when solving framework-like design problems

- **We need both ADTs and objects!**
  - Languages like Java provide adequate—though imperfect—support for both

Conclusions: Why Objects Were Successful

- The essence of objects is **dispatch**, or **service abstraction**
- Dispatch uniquely provides **first-class interoperability**
- First-class interoperability is critical to **frameworks** and **ecosystems**
- Frameworks and ecosystems are **economically critical** to the software industry

- Likely a significant factor in objects' success
  - Future study is warranted to validate the story above
  - Other factors (psychology, benefits of inheritance) are worth exploring too