Object-based vs. Class-based Languages

Luca Cardelli

Digital Equipment Corporation
Systems Research Center

PLDI'96 Tutorial

Abstract

Class-based object-oriented programming languages take object generators as central, while object-based languages emphasize the objects themselves. This dichotomy, or spectrum, has been explicitly recognized for at least 15 years.

During this time, class-based languages have become relatively well-understood (or well-misunderstood), widely debated, and hugely popular. In contrast, the area of object-based languages has evolved within a smaller and more varied community, and is still underdeveloped. Much of its visibility is due today to the Self language which, in many ways, is the Smalltalk of object-based programming. But where are the Simula and the C++ of object-based programming? It seems that they have not been invented yet, or certainly they are not as popular.

History of this Material

- Designing a class-based language (Modula-3).
- Designing an object-based language (Obliq).
- Learning about other object-based languages.
  - ~ Organizing what I learned.
- Working on object calculi.
  - ~ Filling the gap between object calculi and object-oriented languages.

CLASS-BASED LANGUAGES

- The mainstream.
- We discuss only common, kernel properties.

Classes and Objects

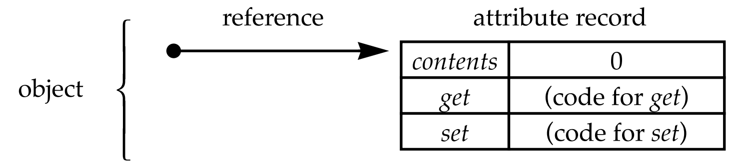
- Classes are descriptions of objects.
- Example: storage cells.

```
class cell is
    var contents: Integer := 0;
    method get(): Integer is
        return self.contents;
    end;
    method set(n: Integer) is
        self.contents := n;
    end;
end;
```

- Self.

Naive Storage Model

- Object = reference to a record of attributes.



Naive storage model

Object Creation

- Objects are created from classes via **new**.
- (*InstanceTypeOf(c)* indicates the type of an object of class *c*.)

```
var myCell: InstanceTypeOf(cell) := new cell;
```

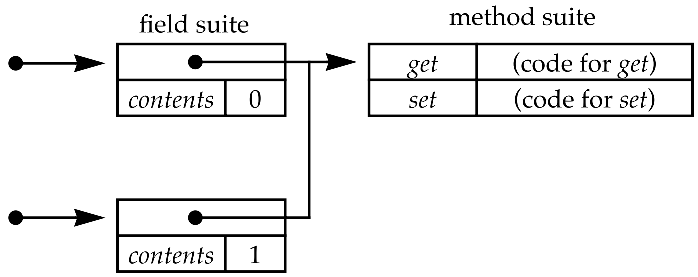

Object Operations

- Field selection.
- Field update.
- Method invocation.

```
procedure double(aCell: InstanceTypeOf(cell)) is
    aCell.set(2 * aCell.get());
end;
```

The Method-Suites Storage Model

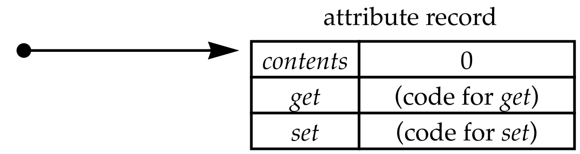
- A more refined storage model for class-based languages.



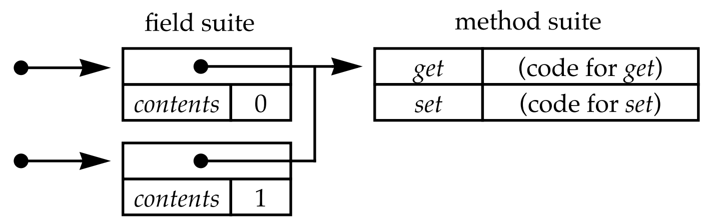
Method suites

Embedding vs. Delegation

- In the naive storage model, methods are *embedded* in objects.



- In the methods-suites storage model, methods are *delegated* to the method suites.



- Naive and method-suites models are semantically equivalent for class-based languages.
- The contrast between embedding and delegation will be our main theme.
- It will becomes particularly important in the context of object-based languages.

Method Lookup

- Method lookup is the process of finding the code to run on a method invocation *o.m*(…).
- The details depend on the language and the storage model.

The Great Class-Based Illusion

- In class-based languages, method lookup gives the *illusion* that methods are embedded in objects (cf. *o.x*, *o.m*(...)).
- Self is always the *receiver*: the object that *appears* to contain the method.
- Features that would distinguish embedding from delegation implementations (e.g. method update) are usually avoided.
- This illusion hides the details of the storage model and of method lookup.

Subclasses and Inheritance

- A *subclass* is a differential definition of a class.
- The *subclass relation* is the partial order induced by the subclass declarations.
- Example: restorable cells.

```
subclass reCell of cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        super.set(n);
    end;
    method restore() is
        self.contents := self.backup;
    end;
end;
```


Subclasses and Self

- Because of subclasses, the meaning of **self** becomes dynamic.

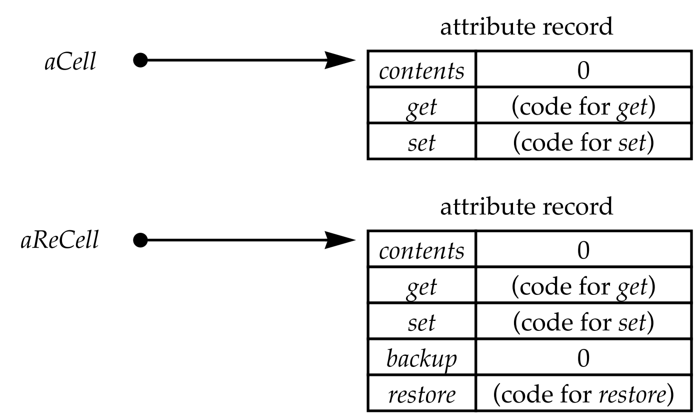
  self.*m*(...)

- Because of subclasses, the concept of **super** becomes useful.

  super.*m*(...)

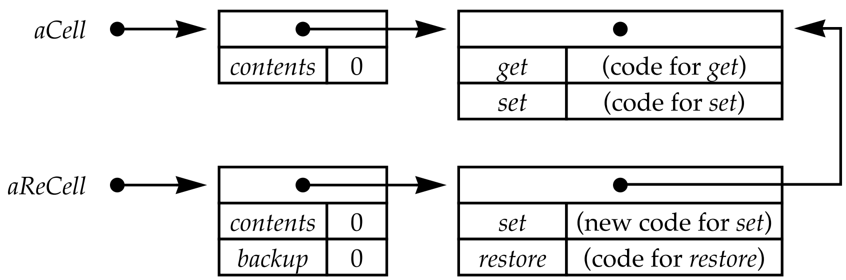
Subclasses and Naive Storage

- In the naive implementation, the existence of subclasses does not cause any change in the storage model.



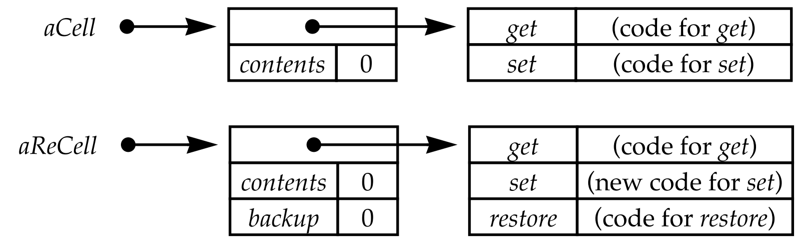
Subclasses and Method Suites

- Because of subclasses, the method-suites model has to be reconsidered. In dynamically-typed class-based languages, it is modified:



Hierarchical method suites

- In statically-typed class-based languages, however, it is maintained in its original form.



Collapsed method suites

Embedding/Delegation View of Class Hierarchies

- Hierarchical method suites: *delegation* (of objects to suites) combined with *delegation* (of sub-suites to super-suites).
- Collapsed method suites: *delegation* (of objects to suites) combined with *embedding* (of super-suites in sub-suites).

Class-Based Summary

- In analyzing the meaning and implementation of class-based languages we end up inventing and analyzing sub-structures of objects and classes.
- These substructures are independently interesting: they have their own semantics, and can be combined in useful ways.
- What if these substructures were directly available to programmers?

OBJECT-BASED LANGUAGES

- Slow to emerge.
- Simple and flexible.
- Usually untyped.

Objects without Classes

- Just objects and dynamic dispatch.
- When typed, just object types and subtyping.
- Direct object-to-object inheritance.

An Object, All by Itself

- Classes are replaced by object constructors.
- Object types are immediately useful.

```
ObjectType Cell is
    var contents: Integer;
    method get(): Integer;
    method set(n: Integer);
end;

object cell: Cell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;
```


An Object Generator

- Procedures as object generators.

```
procedure newCell(m: Integer): Cell is
    object cell: Cell is
        var contents: Integer := m;
        method get(): Integer is return self.contents end;
        method set(n: Integer) is self.contents := n end;
    end;
    return cell;
end;

var cellInstance: Cell := newCell(0);
```

- Quite similar to classes!

Decomposing Class-based Features

- General idea: decompose class-based notions. (And orthogonally recombine them.)
- We have seen how to decompose simple classes into objects and procedures.
- We will now investigate how to decompose inheritance.

Prototypes and Clones

- Object generation by parameterization.
- Vs. object generation by cloning and mutation.

Prototypes

- Classes describe objects.
- Prototypes describe objects and *are* objects.

Cloning of Prototypes

- Regular objects are clones of prototypes.

  var *cellClone*: *Cell* := **clone** *cellInstance;*

- **clone** is a bit like **new**, but operates on objects instead of classes.

Mutation of Clones

- Clones are customized by mutation (e.g., update).
- Field update.

cellClone.*contents* := 3;

- Method update.

cellClone.*get* :=
    method (): *Integer* **is**
        if self.*contents* < 0 **then return** 0 **else return self**.*contents* **end**;
    end;

Self-Mutation

- Restorable cells with no *backup* field.

ObjectType *ReCell* **is**
    var *contents*: *Integer*;
    method *get*(): *Integer*;
    method *set*(*n*: *Integer*);
    method *restore*();
end;

- The *set* method updates the *restore* method!

object *reCell*: *ReCell* **is**
    var *contents*: *Integer* := 0;
    method *get*(): *Integer* **is return self**.*contents* **end**;
    method *set*(*n*: *Integer*) **is**
        let *x* = **self**.*get*();
        self.*restore* := **method** () **is self**.*contents* := *x* **end**;
        self.*contents* := *n*;
    end;
    method *restore*() **is self**.*contents* := 0 **end**;
end;

Forms of Mutation

- Method update is an example of a mutation operation. It is simple and statically typable.
- Forms of mutation include:
  - ~ Direct method update (Beta, NewtonScript, Obliq, Kevo, Garnet).
  - ~ Dynamically removing and adding attributes (Self, Act1).
  - ~ Swapping groups of methods (Self, Ellie).

Object-Based Inheritance

- Object generation can be written as procedures, but with no real notion of inheritance.
- Object inheritance can be achieved by cloning (reuse) and update (override), but with no shape change.
- How can we inherit while changing shape?

An Option: Object Extension

- Object extension achieves inheritance and shape change. However:
  - ~ Not easy to typecheck.
  - ~ Not easy to implement efficiently.
  - ~ Provided rarely or restrictively.

Donors and Hosts

- General object-based inheritance: building new objects by "reusing" attributes of existing objects.
- Two orthogonal aspects:
  - ~ obtaining the attributes of a ***donor*** object, and
  - ~ incorporating those attributes into a new ***host*** object.
- Four categories of object-based inheritance:
  - ~ The attributes of a donor may be obtained ***implicitly*** or ***explicitly***. Orthogonally:
  - ~ those attributes may be either ***embedded*** into a host, or ***delegated*** to a donor.

Embedding vs. Delegation Inheritance

- A difference in execution.
- **Embedding inheritance**: the attributes inherited from a donor become part of the host (in principle, at least).
- **Delegation inheritance**: the inherited attributes remain part of the donor, and are accessed via an indirection from the host.
- Either way, self is the receiver.
- In embedding, host objects are independent of their donors. In delegation, complex webs of dependencies may be created.

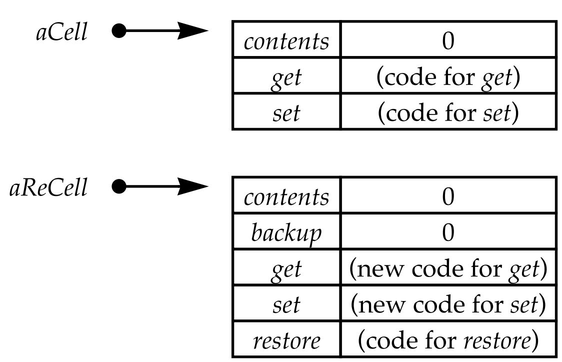
Implicit vs. Explicit Inheritance

- A difference in declaration.
- **Implicit inheritance**: one or more objects are designated as the donors (explicitly!), and their attributes are <u>implicitly</u> inherited.
- **Explicit inheritance**, individual attributes of one or more donors are <u>explicitly</u> designated and inherited.
- **Super** and **override** make sense for implicit inheritance, not for explicit inheritance.

- Intermediate possibility: explicitly designate a named collection of attributes that, however, does not form a whole object. E.g. *mixin* inheritance.
- (We can see implicit and explicit inheritance, as the extreme points of a spectrum.)

Embedding

- Host objects contain <u>copies</u> of the attributes of donor objects.



Embedding

Embedding and Self

- Embedding provides the simplest explanation of the standard semantics of **self** as the receiver.
- The invocation of an inherited method works exactly like the invocation of an original method.

Embedding-Based Languages

- Embedding was described by Borning as part of one of the first proposals for prototype-based languages.
- Recently, it has been adopted by languages like Kevo and Obliq. We call these languages *embedding-based* (*concatenation-based*, in Kevo terminology).
- Embedding inheritance can be specified explicitly or implicitly.
  - ~ Explicit forms of embedding inheritance can be understood as *reassembling* parts of old objects into new objects.
  - ~ Implicit forms of embedding inheritance can be understood as ways of *concatenating* or extending copies of existing objects with new attributes.

Explicit Embedding Inheritance

- Individual methods and fields of specific objects (donors) are copied into new objects (hosts).
- We write

  embed *o.m*(…)

  to embed the method *m* of object *o* into the current object.
- The meaning of **embed** *cell.set*(*n*) is to execute the *set* method of *cell* with **self** bound to the current self, and not with **self** bound to *cell* as in a normal invocation *cell.set*(*n*).
- Moreover, the code of *set* is embedded in *reCellExp*.

reCellExp

```
object cell: Cell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;
object reCellExp: ReCell is
    var contents: Integer := cell.contents;
    var backup: Integer := 0;
    method get(): Integer is
        return embed cell.get();
    end;
    method set(n: Integer) is
        self.backup := self.contents;
        embed cell.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```


- The code for get could be abbreviated to:

  method *get* **copied from** *cell*;

Implicit Embedding Inheritance

- Whole objects (donors) are copied to form new objects (hosts).
- We write

  object *o*: *T* **extends** *o′*

  to designate a donor object *o′* for *o*.
- As a consequence of this declaration, *o* is an object containing a copy of the attributes of *o′*, with independent state.

reCellImp

```
object cell: Cell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;

object reCellImp: ReCell extends cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        embed super.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```


Alternate *reCellImp* via method update

- We could define an equivalent object by a pure extension of *cell* followed by a method update.

```
object reCellImp1: ReCell extends cell is
    var backup: Integer := 0;
    method restore() is self.contents := self.backup end;
end;

reCellImp1.set :=
    method (n: Integer) is
        self.backup := self.contents;
        self.contents := n;
    end;
```

This code works because, with embedding, method update affects only the object to which it is applied. (This is not true for delegation.)

Stand-alone *reCell*

- The definitions of both *reCellImp* and *reCellExp* can be seen as convenient abbreviations:

```
object reCell: ReCell is
    var contents: Integer := 0;
    var backup: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is
        self.backup := self.contents;
        self.contents := n;
    end;
    method restore() is self.contents := self.backup end;
end;
```

Delegation

- Host objects contain links to the attributes of donor objects.
- Prototype-based languages that permit the sharing of attributes across objects are called *delegation-based*.
- Operationally, delegation is the redirection of field access and method invocation from an object or prototype to another, in such a way that an object can be seen as an extension of another.
- Note: similar to hierarchical method suites.

Delegation and Self

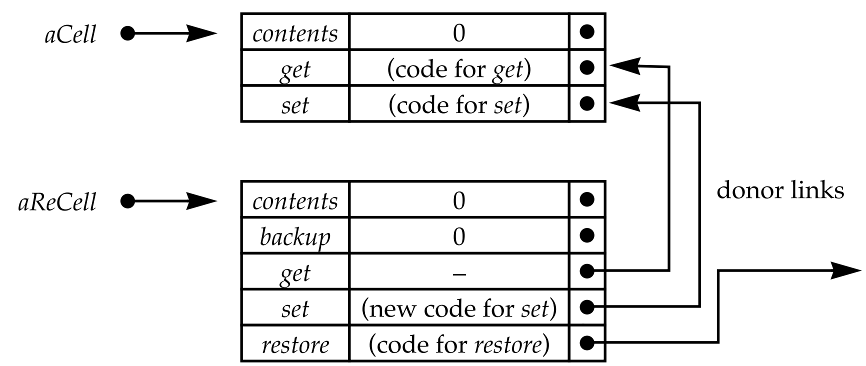
- A crucial aspect of delegation inheritance is the interaction of donor links with the binding of **self**.
- On an invocation of a method called *m*, the code for *m* may be found only in the donor cell. But the occurrences of **self** within the code of *m* refer to the original receiver, not to the donor.
- Therefore, delegation is not redirected invocation.

Explicit Delegation Inheritance

- Individual methods and fields of specific objects (donors) are linked into new objects (hosts).
- We write

  delegate *o.m*(...)

  to execute the *m* method of *o* with **self** bound to the current self (not to *o*).
- The difference between **delegate** and **embed** is that the former obtains the method from the donor at the time of method invocation, while the latter obtains it earlier, at the time of object creation.



(An example of) Delegation

reCellExp

object *reCellExp*: *ReCell* **is**
  var *contents*: *Integer* := *cell*.*contents*;
  var *backup*: *Integer* := 0;
  method *get*(): *Integer* **is return delegate** *cell*.*get*() **end**;
  method *set*(*n*: *Integer*) **is**
    self.*backup* := **self**.*contents*;
    delegate *cell*.*set*(*n*);
  end;
  method *restore*() **is self**.*contents* := **self**.*backup* **end**;
end;

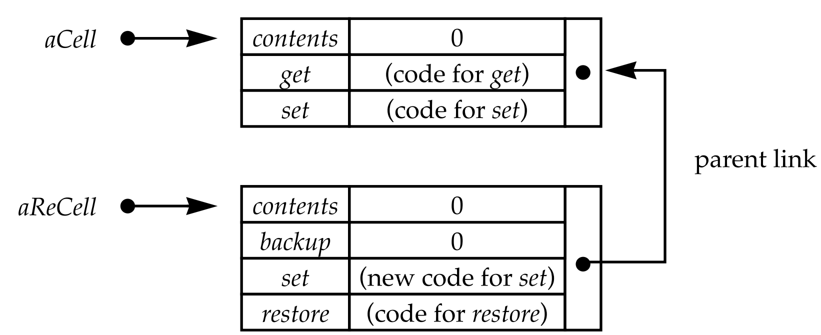
- Explicit delegation provides a clean way of delegating operations to multiple objects. It provides a clean semantics for multiple donors.

Implicit Delegation Inheritance (Traditional Delegation)

- Whole objects (donors/parents) are shared to from new objects (hosts/children).
- We write

  object *o*: *T* **child of** *o′*

  to designate a parent object *o′* for *o*.
- As a consequence of this declaration, *o* is an object containing a single *parent link* to *o′*, with parent state shared among children. Parent links are followed in the search for attributes.



(Single-parent) Delegation

reCellImp

- A first attempt.

```
object cell: Cell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;

object reCellImp': ReCell child of cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        delegate super.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```


- This is almost identical to the code of *reCellImp* for embedding.
- But for delegation, this definition is wrong: the *contents* field is shared by all the children.

- A proper definition must include a local copy of the *contents* field, overriding the *contents* field of the parent.

```
object reCellImp: ReCell child of cell is
    override contents: Integer := cell.contents;
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        delegate super.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```


- On an invocation of *reCellImp*.*get*(), the *get* method is found only in the parent cell, but the occurrences of **self** within the code of *get* refer to the original receiver, *reCellImp*, and not to the parent, *cell*.
- Hence the result of *get*() is, as desired, the integer stored in the *contents* field of *reCellImp*, not the one in the parent cell.

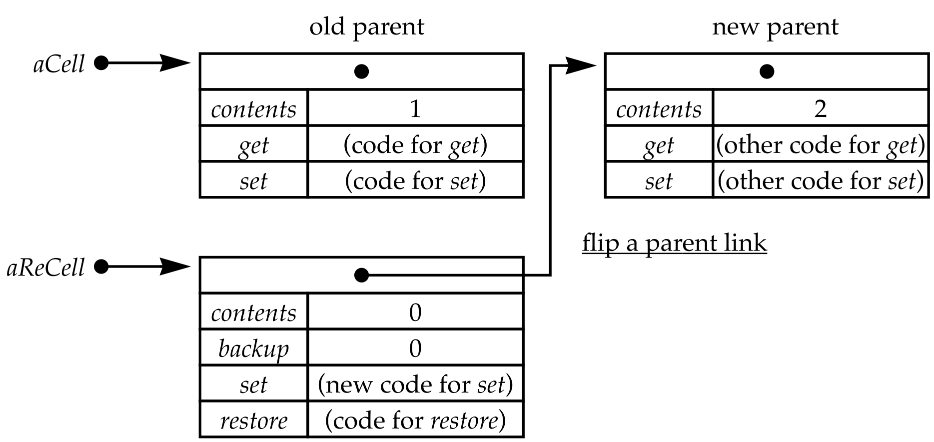
Dynamic Inheritance

- Inheritance is called *static* when inherited attributes are fixed for all time.
- It is *dynamic* when the collection of inherited attributes can be updated dynamically (replaced, increased, decreased).

Mode Switching

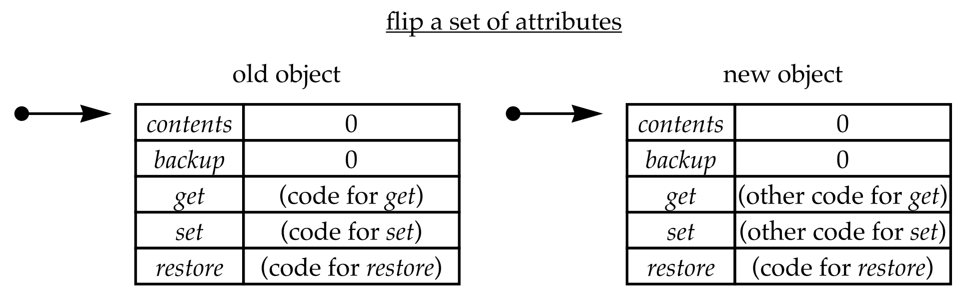
- Although dynamic inheritance is in general a dangerous feature, it enables rather elegant and disciplined programming techniques.
- In particular, *mode-switching* is the special case of dynamic inheritance where a collection of (inherited) attributes is *swapped* with a similar collection of attributes. (This is even typable.)

Delegation-Style Mode Switching



Reparenting

Embedding-Style Mode Switching



Method Update

Embedding vs. Delegation Summary

- In embedding inheritance, a freshly created host object contains copies of donor attributes.
- Access to the inherited donor attributes is no different than access to original attributes, and is quick.
- Storage use may be comparatively large, unless optimizations are used.

- In delegation inheritance, a host object contains links to external donor objects.
- During method invocation, the attribute-lookup procedure must preserve the binding of **self** to the original receiver, even while following the donor links.
  - ~ This results in more complicated implementation and formal modeling of method lookup.
  - ~ It creates couplings between objects that may not be desirable in certain (e.g. distributed) situations.

- In class-based languages the embedding and delegation models are normally (mostly) equivalent.
- In object-based languages they are distinguishable.
  - ~ In delegation, donors may contain fields, which may be updated; the changes are seen by the inheriting hosts.
  - ~ Similarly, the methods of a donor may be updated, and again the changes are seen by the inheriting hosts.

- ~ It is often permitted to replace a donor link with another one in an object; then all the inheritors of that object may change behavior.
- ~ Cloning is still taken to perform shallow copies of objects, without copying the corresponding donors. Thus, all clones of an object come to share its donors and therefore the mutable fields and methods of the donors.

- Thus, embedding and delegation are two fundamentally distinct ways of achieving inheritance with prototypes.
- Interesting languages exist that explore both possibilities.

Advantages of Delegation

- Space efficiency by sharing.
- Convenience in performing dynamic, pervasive changes to all inheritors of an object.
- Well suited for integrated languages / environments.

Advantages of Embedding

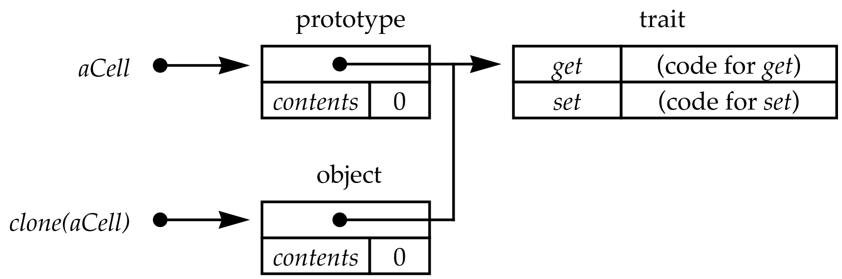
- Delegation can be criticized because it creates dynamic webs of dependencies that lead to fragile systems. Embedding is not affected by this problem since objects remain autonomous.
- In embedding-based languages such as Kevo and Omega, pervasive changes are achieved even without donor hierarchies.
- Space efficiency, while essential, is best achieved behind the scenes of the implementation.
  - ~ Even delegation-based languages optimize cloning operations by transparently sharing structures; the same techniques can be used to optimize space in embedding-based languages.

Traits: from Prototypes back to Classes?

- Prototypes were initially intended to replace classes.
- Several prototype-based languages, however, seem to be moving towards a more traditional approach based on class-like structures.
- Prototypes-based languages like Omega, Self, and Cecil have evolved usage-based distinctions between objects.

Different Kinds of Objects

- Trait objects.
- Prototype objects.
- Normal objects.



Traits

Not all These Object are “True” Objects

- In the spirit of classless languages, traits and prototypes are still ordinary objects. However, there is a separation of roles.
- Traits are intended only as the shared parents of normal objects: they should not be used directly or cloned.
- Prototypes are intended only as object (and prototype) generators via cloning: they should not be used directly or modified.

- Normal objects are intended only to be used and to carry local state: they should rely on traits for their methods.
- These distinctions may be seen as purely methodological, or may be enforced: for example, some operations on traits and prototypes may be forbidden to protect these objects from accidental damage.

Trait Treason

- This separation of roles violates the original spirit of prototype-based languages. Traits objects cannot function on their own.
- With the separation between traits and other objects, we seem to have come full circle back to class-based languages and to the separation between classes and instances.

Object Constructions vs. Class Implementations

- The traits-prototypes partition in delegation-based languages looks exactly like an implementation technique for classes.
- A similar traits-prototypes partition in embedding-based languages corresponds to a different implementation technique for classes that trades space for access speed.
- Class-based notions and techniques are not totally banned in object-based languages. Rather, they resurface naturally.

The Contribution of the Object-Based Approach

- The achievement of object-based languages is to make clear that classes are just one of the possible ways of generating objects with common properties.
- Objects are more primitive than classes, and they should be understood and explained before classes.
- Different class-like constructions can be used for different purposes. Hopefully, more flexibly than in strict class-based languages.

CONCLUSIONS

- Class-based: various implementation techniques based on embedding and/or delegation. Self is the receiver.
- Object-based: various language mechanisms based on embedding and/or delegation. Self is the receiver.
- Object-based can emulate class-based. (By traits, or by otherwise reproducing the implementations techniques of class-based languages.)

Foundations

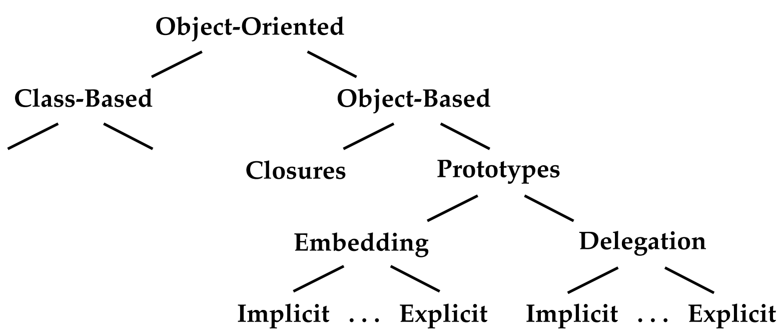
- Objects can emulate classes (by traits) and procedures (by “stack frame objects”).
- *Everything* can indeed be an object.

Future Directions

- I look forward to the continued development of typed object-based languages.
  - ~ The notion of object type arise more naturally in object-based languages.
  - ~ Traits, method update, and mode switching are typable (general reparenting is not easily typable).
- No need for dichotomy: object-based and class-based features can be merged within a single language, based on the common object-based semantics (Beta, O–1, O–2, O–3).

- Embedding-based languages seem to be a natural fit for distributed-objects situations. E.g. COM vs. CORBA.
  - ~ Objects are self-contained and are therefore *localized*.
  - ~ For this reason, Obliq was designed as an embedding-based language.

A New Hierarchy



BIBLIOGRAPHY

[1] Abadi, M. and L. Cardelli, **A theory of objects**. Springer. 1996 (to appear).

[2] Abadi, M. and L. Cardelli, **A theory of primitive objects: untyped and first-order systems**. *Proc. Theoretical Aspects of Computer Software*. Lecture Notes in Computer Science 789, 296-320. Springer-Verlag. 1994.

[3] Adams, N. and J. Rees, **Object-oriented programming in Scheme**. *Proc. 1988 ACM Conference on Lisp and Functional Programming*, 277-288. 1988.

[4] Agesen, O., L. Bak, C. Chambers, B.-W. Chang, U. Hölzle, J. Maloney, R.B. Smith, D. Ungar, and M. Wolczko, **The Self 3.0 programmer's reference manual**. Sun Microsystems. 1993.

[5] Alagic, S., R. Sunderraman, and R. Bagai, **Declarative object-oriented programming: inheritance, subtyping, and prototyping**. *Proc. ECOOP'94*. Lecture Notes in Computer Science 821, 236-259. Springer-Verlag. 1994.

[6] Andersen, B., **Ellie: a general, fine-grained, first-class, object-based language**. *Journal of Object Oriented Programming* **5**(2), 35-42. 1992.

[7] Apple, **The NewtonScript programming language**. Apple Computer, Inc. 1993.

[8] Blaschek, G., **Type-safe OOP with prototypes: the concepts of Omega.** *Structured Programming* **12**(12), 1-9. 1991.

[9] Blaschek, G., **Object-oriented programming with prototypes**. Springer-Verlag. 1994.

[10] Borning, A.H., **The programming language aspects of ThingLab, a constraint-oriented simulation laboratory**. *ACM Transactions on Programming Languages and Systems* **3**(4), 353-387. 1981.

[11] Borning, A.H., **Classes versus prototypes in object-oriented languages**. *Proc. ACM/IEEE Fall Joint Computer Conference*, 36-40. 1986.

[12] Cardelli, L., **A language with distributed scope**. *Computing Systems*, **8**(1), 27-59. MIT Press. 1995.

[13] Chambers, C., **The Cecil language specification and rationale**. Technical Report 93-03-05. University of Washington, Dept. of Computer Science and Engineering. 1993.

[14] Chambers, C., D. Ungar, and E. Lee, **An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes**. *Proc. OOPSLA'89*, 49-70. ACM Sigplan Notices 24(10). 1989.

[15] Dony, C., J. Malenfant, and P. Cointe, **Prototype-based languages: from a new taxonomy to constructive proposals and their validation.** *Proc. OOPSLA'92*, 201-217. 1992.

[16] Lieberman, H., **A preview of Act1**. AI Memo No 625. MIT. 1981.

[17] Lieberman, H., **Using prototypical objects to implement shared behavior in object oriented systems**. *Proc. OOPSLA'86*, 214-223. ACM Press. 1986.

[18] Lieberman, H., **Concurrent object-oriented programming in Act 1**. In *Object-oriented concurrent programming*, A. Yonezawa and M. Tokoro, ed., MIT Press. 9-36. 1987.

[19] Madsen, O.L., B. Møller-Pedersen, and K. Nygaard, **Object-oriented programming in the Beta programming language**. Addison-Wesley. 1993.

[20] Paepcke, A., ed., **Object-oriented programming: the CLOS perspective**. MIT Press, 1993.

[21] Rajendra, K.R., E. Tempero, H.M. Levy, A.P. Black, N.C. Hutchinson, and E. Jul, **Emerald: a general-purpose programming language**. *Software Practice and Experience* **21**(1), 91-118. 1991.

[22] Stein, L.A., H. Lieberman, and D. Ungar, **A shared view of sharing: the treaty of Orlando**. In *Object-oriented concepts, applications, and databases,* W. Kim and F. Lochowsky, ed., Addison-Wesley. 31-48. 1988.

[23] Taivalsaari, A., **Kevo, a prototype-based object-oriented language based on concatenation and module operations**. Report LACIR 92-02. University of Victoria. 1992.

[24] Taivalsaari, A., **A critical view of inheritance and reusability in object-oriented programming**. Jyväskylä Studies in computer science, economics and statistics No.23, A. Salminen, ed., University of Jyväskylä. 1993.

[25] Taivalsaari, A., **Object-oriented programming with modes**. *Journal of Object Oriented Programming* **6**(3), 25-32. 1993.

[26] Ungar, D., C. Chambers, B.-W. Chang, and U. Hölzle, **Organizing programs without classes**. *Lisp and Symbolic Computation* **4**(3), 223-242. 1991.

[27] Ungar, D. and R.B. Smith, **Self: the power of simplicity**. *Lisp and Symbolic Computation* **4**(3), 187-205. 1991.