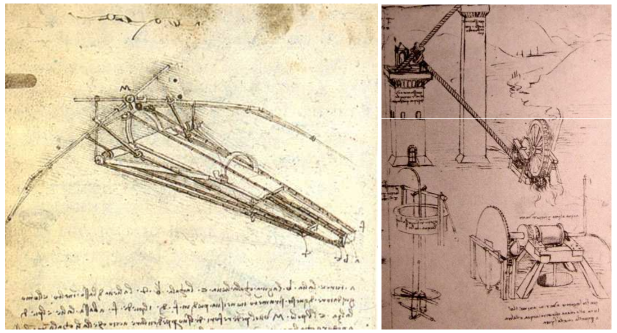
HY540 – Advanced Topics in Programming Language Development



Lecture 4

Untyped inheritance – introduction (1/2)

- Untyped inheritance is the analogy of class-based inheritance in the object-based world
  - how to produce objects compliant to a designed subclass
  - but without rewriting code that corresponds to the bases classes
  - while respecting late binding semantics
- Class-based notions and techniques are not totally abandoned in object-based inheritance
  - Rather they resurface naturally with more flexible implementation techniques

Untyped inheritance – introduction (2/2)

- Irrespective of the language features, there are two possible *design patterns* for untyped inheritance
  - perform inheritance at the levels of class-emulating objects (*prototypes* or *class objects*)
    - *class graph emulation* *- objects share their heritage*
  - perform inheritance at the level of individual objects
    - *object structure emulation* *- objects copy their heritage*
- We will study whether and how the existing mechanisms for untyped inheritance support both

Roadmap

- Patterns
  - Class graph emulation
  - Object structure emulation
- Methods
  - Embedding
  - Delegation
  - Subobject trees
- Issues
  - Uninheritance
  - Non-monotonic object evolution
  - Mode switching

Class graph emulation (1/5)

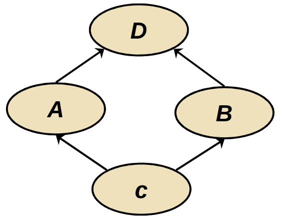
- In this method the language allows to convert the designed class graph to a corresponding runtime object graph
  - each node in the graph represents a distinct class
    - *implemented as a prototype for this class*
  - every class node delegates to the nodes corresponding to its base classes
    - *subclassing implies delegation among prototypes*
  - note that dynamic updates on a class node alter the objects of all subclasses

Class graph emulation (2/5)

Sample class structure (graph)

class D;
class A subclass of D;
class B subclass of D;
class C subclass of A,B;

Respective runtime object graph



*If we affect **D** then all subclasses **A**, **B** and **C** are directly affected*

******Class graph emulation results in runtime webs of prototypes isomorphic to the static class graphs that is produced during compilation.***

Class graph emulation (3/5)

- Every object will have to delegate to the node of its respective class
  - we did that before with normal prototypes
  - but also every subobject should delegate to the node of the of the bottommost class in the tree
    - *this is necessary to guarantee late binding*
- Linking of prototypes via delegation should be done carefully
  - in a way that delegation lookup is equivalent to MRO (like linearization)
- Rule is that *state is copied methods are shared*

Class graph emulation (4/5)

```
class Person {
  var name:string = "";
  virtual GetName():string  { return self.name;}
}
```

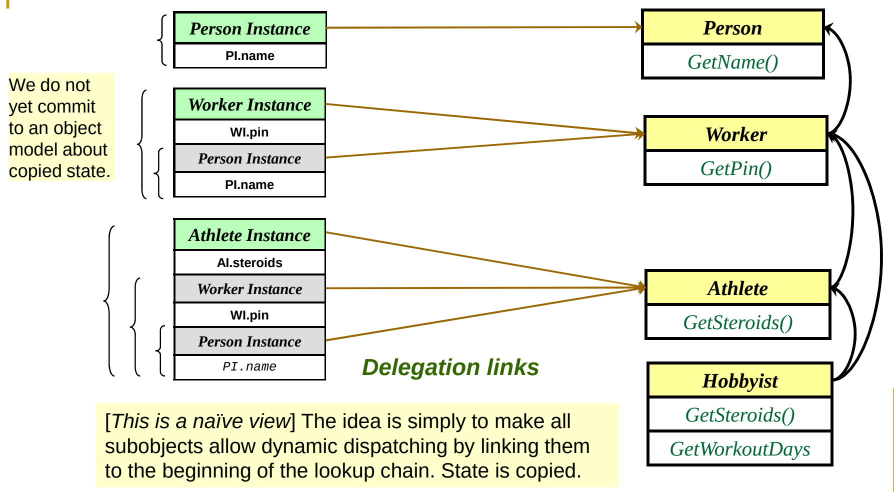
```
subclass Worker of Person {
  var pin:integer = 0;
  virtual GetPin(): integer  { return self.pin; }
}
```

```
subclass Athlete of Person {
  var steroids: list[string];
  virtual GetSteroids(): list[string] { return seroids; }
}
```

```
subclass Hobbyist of Athlete, Worker {
 var: workoutDays: integer[0…7] = 0;
 override GetSteroids(): list[string] { return nil; }
 method  GetWorkoutDays():integer[0…7]  { return workoutDays; }
}
```

Sample class design (exists in the design domain only)

Class graph emulation (5/5)



We do not yet commit to an object model about copied state.

[*This is a naïve view*] The idea is simply to make all subobjects allow dynamic dispatching by linking them to the beginning of the lookup chain. State is copied.

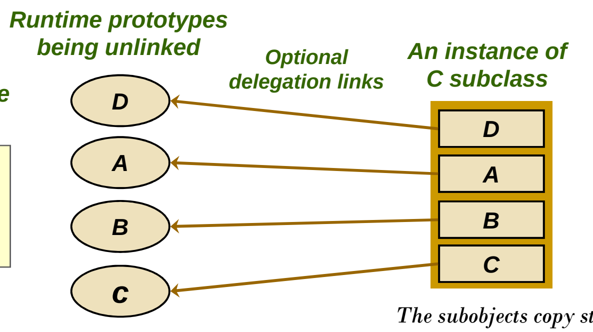
Object structure emulation (1/2)

- Every object is complete and contains all the data and methods of its respective class
  - normally we adopt a structure in which a main object (most derived) is composed of subobjects
    - *but in all cases we need guarantee late binding*
- Our definition of subobject: *an object being part of another object due to inheritance*
- Prototypes may optionally exist but they are never linked to each other
  - meaning they are not to linked according to subclassing
  - but since prototypes are not linked, the MRO should be implied by the way objects are structured
- Rule is that *state and methods are copied*

Object structure emulation (2/2)

Sample class structure (graph)

class D;
class A subclass of D;
class B subclass of D;
class C subclass of A,B;


Runtime prototypes being unlinked
Optional delegation links
An instance of C subclass
D
A
B
C
D
A
B
C

*If we affect **D** then all **D** objects are directly affected. The effect on subclass objects depends on their object structure.*

The subobjects copy state and behavior of their class and are somehow glued or linked together

Embedding (1/11)

- Embedding or concatenation is an inheritance method in which all *attributes inherited from a base become an integral part of the derived*
  - thus derived objects become independent and unrelated at runtime to their bases
- Since embedding packs all inherited attributes together it means
  - overridden method names should map always to their most recent version
  - if data fields are also late bound in the language then they should similarly map to most recent versions

Embedding (2/11)

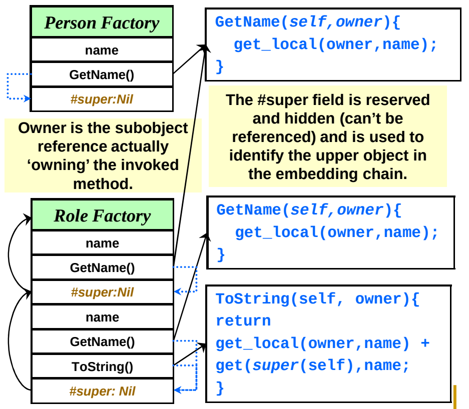
- When earlier versions (like ***super***) of overridden methods (or fields) can be invoked in the language
  - then their addresses (or storage) should be somehow retained within the packed structure
- When inherited data fields are late bound then similar qualified access (like ***local***) should be necessarily allowed
  - else the semantics of base methods involving late bound fields reappearing in derived objects may be broken

Embedding (3/11)

```
class Person {
  var name:string = “”;  ← person  name
  virtual GetName():string  { return self..name;}
}
***Assume self.id is late bound access and that
self..name implies local access
```

```
subclass Role of Person {
  var name:string = “”;  ← role name
  override GetName():string  { return self..name;}
  method ToString():string
    {  return self..name + super.name };
}
```

```
record Method {
    Address    addr;        // Code address
    Attributes begin;       // First attribute of  owner object.
};
a.f() is equivalent to a.f.addr(a.begin≡self, a.f.begin ≡owner)
self.x  →       get(self, x)
self..x →       get_local(owner, x)
super  →        self.#super
***Local means search attributes until a #super index is met
```



```
GetName(self,owner){
  get_local(owner,name);
}
```

Owner is the subobject reference actually ‘owning’ the invoked method.

The #super field is reserved and hidden (can’t be referenced) and is used to identify the upper object in the embedding chain.

```
GetName(self,owner){
  get_local(owner,name);
}
```

```
ToString(self, owner){
return
get_local(owner,name) +
get(super(self),name;
}
```

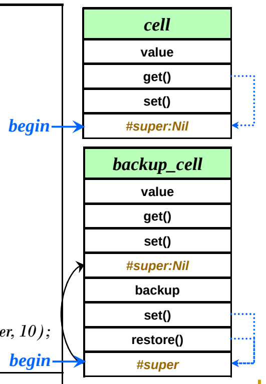
Embedding (4/11)

- No upcasting / down casting is necessary
  - We pass the object reference of an entire object in all cases
  - Being actually a reference to the first item of the attribute list
- Late binding is guaranteed by *lookup*
  - resolution of the most recent attribute version is mandated
  - since we search attributes from begin (bottom) to end (top)
  - inherently objects are attribute lists (not a field dictionary)
  - this is the only time we adopt such an object model
- In methods we set the ***begin*** field only once, upon construction of the object owning the method

Embedding (5/11)

```
cell = object {
        value:0,
        method get()  { return self..value; },
        method set(x) { self..value = x; },
};
backup_cell_sub = object {
        backup:0,
        method set(x)
          { self..backup = self.value; super.set(x); },
        method restore() { self.value = self..backup; }
};
backup_cell = embed(backup_cell_sub, cell);
backup_cell.set(10)→backup_cell.set(backup_cell.begin, backup_cell.set.owner, 10);
backup_cell.restore();
```

```
set(self, owner, x) {       of backup_cell
   set_local(owner, “backup”, get(self, “value”));
   get_local(super(self), “set”)(x);
}
```



Embedding (6/11)

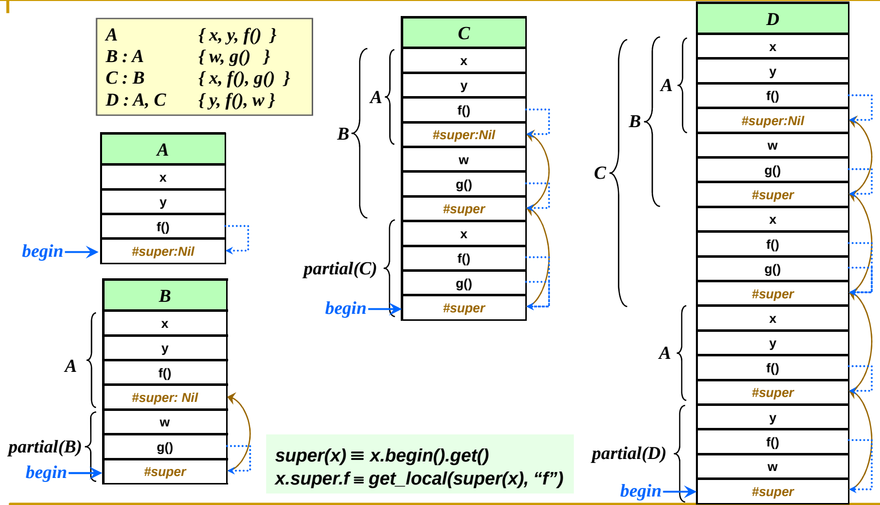
- An earlier version of *x* to the base object being ***N*** levels up the inheritance chain is visible through `(super.)`^{K}`x`
  - *K≤N: K* is the number of classes in the chain of *N* base classes which also have an attribute *x*
- Prototypes are used only as object generators
  - Interestingly, due to embedding, a derived prototype can embed a copy of its base prototypes
    - So one could use an embedded prototype to produce a respective object

Embedding (7/11)

- *class A*
  - *object A*
- *subclass B of A*
  - *object B = embed(partial(B), A)*
- *subclass C of B*
  - *object C = embed(partial(C), B)*
- *subclass D of A, C* ****assume C more recent*
  - *object D = embed(partial(D), embed(C,A))*
  - but what we do in case of multiple inheritance?

Embedding (8/11)

A	*{ x, y, f() }*
B : A	*{ w, g() }*
C : B	*{ x, f(), g() }*
D : A, C	*{ y, f(), w }*



super(x) ≡ x.begin().get()
x.super.f ≡ get_local(super(x), "f")

Embedding (9/11)

- $A[B_1,\ldots,B_n]$ where B_{i+1} more recent than B_i
- *embed*(A, *embed*(B_n, *embed*(B_{n-1},… ,*embed*(B_2,B_1)…))
- To avoid attribute renaming it is mandatory in embedding to implement an ordered sequential lookup with a list, rather than have traditional hashing
  - hashing requires unique names
  - thus embedding would have to rename base attributes recursively in case of name conflicts
  - that's why in the list implementation more recent attributes are closer to the beginning of the list
- In multiple inheritance, access to base attributes requires good knowledge of the embedding mechanism and the 'more recent' priorities

Embedding (10/11)

- We usually support only packing of objects but never unpacking
  - although with a few extensions unpacking is also feasible
- Dynamic attribute addition or removal is straightforward on the list
  - Usually we only allow editing of the most derived object
  - If we can extend or reduce selectively any base object
    - then the semantics of base methods involving local attribute access may break
    - but you may decide that such a discrepancy is a program design issue not a language flaw as such

Embedding (11/11)

- In embedding (concatenation) every object is self-contained by definition
  - being a flattened ordered collection of all inherited attributes (properties and methods)
  - thus allows objects live independently of any class objects carrying attributes specific to their class
- Due to this feature *embedding is appropriate for object structure emulation*
  - linking individual objects to their prototypes is impossible since a sharing mechanism is lacking
  - but it can be emulated with explicit selective or global propagation mechanism (e.g. broadcast method ***m*** to all objects of set ***s***)

Delegation (1/10)

- We have seen delegation before, so we will skip the details of the method
- Since delegation is a sharing mechanism we study the way we can implement the basic inheritance patterns
  - class graph emulation
    - intuitively, this seems to be a perfect match for delegation since class inheritance is by definition sharing
  - object structure emulation
    - intuitively, we can only think of modeling individual objects as webs of their subobjects

Delegation (2/10)

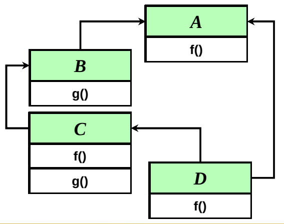
- *Class graph emulation (1/4)*
  - Since through class nodes we may only share behavior (methods), individual objects should carry their own state copy (properties, variables)
    - this is analogous to class-based inheritance were instances carry only data state
      - with the exception of pointers to vtables and to shared subobjects
  - Since a derived object inherits state we also need an appropriate way to package inherited state together
    - in a way that conflicts are avoided and access is preserved as with class-based inheritance

Delegation (3/10)

- *Class graph emulation (2/4)*
  - We assume ***ex nihilo*** object creation is supported via the dynamic field dictionary style
    `[ a:10, b:20, method f(){ return self.a + self.b; } ]`
  - the basic ***delegate*** and ***undelegate*** operations are offered
  - and ***super(i)*** returning ***delegate #i*** as if it was found through delegation lookup (i.e. ***self.super(0).f()*** still invokes ***f*** on ***self***)

A	*{ x, y, f() }*
B : A	*{ w, g() }*
C : B	*{ x, f(), g() }*
D : A, C	*{ y, f(), w }*

****Assume that any method of a class should access its attributes but also inherited attributes and methods (i.e., qualified super access).*



Delegation (4/10)

- *Class graph emulation (3/4)*

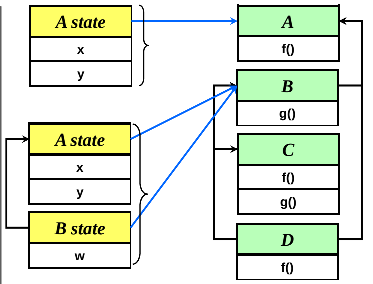
class-based design → ***method B::g() { A::f();}***
object-based implementation → ***method g() {***

```
  self.f();                       ☒ f is late bound
  self.super(0).f();              ☒ f still late bound
  self.B.super(0).f();            ☒ f called on B, not self
  self.super(1).super(0).f()      ☑ self.B.A.f called on self
}
```

****Without our extra requirement that **super.** preservers binding of **self** upon method invocation to the original **self**, the previous function cannot be implemented!*

*---The way the **B::g** is implemented is not derivation enabled since, while **self** is essentially late bound in delegation, **B::g** involves **super** path access that assumes **self** (start) is always a **B** object!*

*---Access via **super** paths is painful, unreadable and requires expressions whose size depends on the distance of the class owning the method (e.g. **A**) from the class access the method (e.g. **B**).*



Delegation (5/10)

- *Class graph emulation (4/4)*

class-based design → ***method B::g() { A::x;}***
class-based design → ***method C::g() {C::x; $x^{\text{late bound}}$; }***
object-based implementation → ***method B::g() {***

```
  self.x;                         ☒ for self:C is C::x
  self.super(0).super((0).x;      ☒ works only for self:C
  self.A.x;                       ☑ yes, explicit up refs
}
```

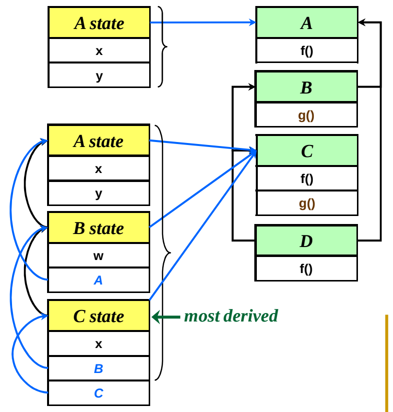
object-based implementation → ***method C::g() {***
Implementing C::x

```
  self.x                          ☒ works only for self:C
  self.C.x;                       ☑ yes, to ensure from C
```

Implementing $x^{\text{late bound}}$

```
  self.x                          ☒ only on most derived
  self.most_derived.x             ☒ rather impractical
}
```

****With delegation we cannot have late binding for data attributes (fields). Whether this is really an inadequacy of the language for object-oriented programming we do not know yet.*

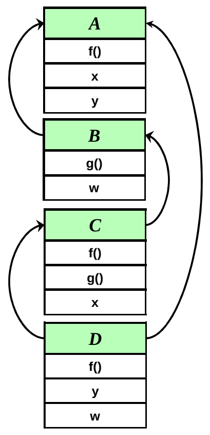


Delegation (6/10)

- Summary for class graph emulation:
  - For qualified base-method invocation ***super*** must be part of the language with delegation lookup semantics
  - With multiple delegates the ordering must be well-defined and be part of the language semantics so that ***super(i)*** is unambiguous
  - *A better alternative is a design pattern: client programmers introduce explicit base reference fields in derived objects named as the designed classes*
    - In this case ***self*** binds to the method owner
    - The pattern allows also qualified access to base-data members
  - Due to parental lookup, late binding for data members is possible only when a reference to the most derived subobject is used
    - *Thus late binding for data members can't be supported*

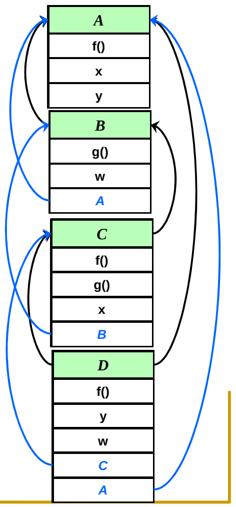
Delegation (7/10)

- *Object structure emulation (1/2)*
  - We extend the model of linking data subobjects by incorporating method addresses inside subobjects
  - There are two main consequences:
    - *Qualified base-method invocation* must be revisited
      - Since we do not have separate class objects on which we may refer
    - *Late bound method invocations* using base subobjects must be asserted
      - As method lookup no longer relies on the class node graph



Delegation (8/10)

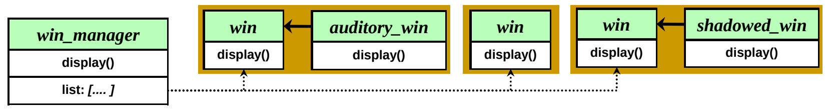
- *Object structure emulation (2/2)*
  - For qualified access to base methods we apply the same trick as with qualified base data
    - ask programmers resolve this through *extra base references inside objects*
  - Late bound method invocation with a base-subobject reference suffers with the same issue as late bound data members
    - *Parental lookup disables resolution downwards* the subobject structure



Delegation (9/10)

```
win_manager = object {
        method display() { for each w in local list do w.display(); },
        method add(w)    { add w in a local list } … };
win_proto = object {
        x, y, w, h:0,
        method new()     { w = clone self; win_manager.add(w); return w; },
        method display() { display a normal window } … };
shadowed_win_proto = object {
        shadow_w, shadow_h : 1,
        method new() { w = clone self; delegate(w, win_proto.new()); return w; },
        method display() { super(0).display(); display the shadow }… };
```

When a shadowed window is created, will its shadow be displayed after `win_manager.display()` *is invoked?*



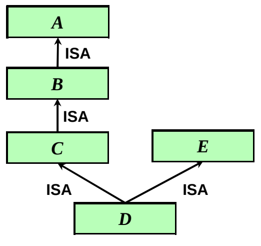
Delegation (10/10)

- Summary for object structure emulation:
  - Client programmers should handle qualified access to base attributes writer
    - *We may agree that this is an acceptable overhead*
  - Lookup for data members (properties) is late bound only when the most derived object is involved
    - The analogy to class-based inheritance it that when doing upcasting we loose the dynamic dispatching behavior
    - *We may agree that this loss is acceptable* when it comes to data members as most language don't support late bound variables
    - But since lookup for methods is similar, late method binding is not guaranteed for base subobjects
    - *Thus can't allow polymorphic functions, being unacceptable*
- *Thus object structure emulation is not feasible in delegation*

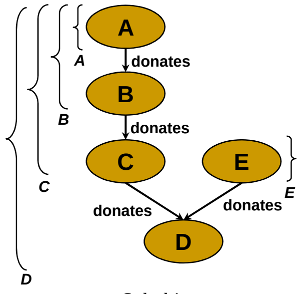
Subobject trees (1/15)

- The idea is to make an object model in which derived objects are made up with a structure similar to the respective class tree
  - besides other advantages, *it is easier for programmers to assimilate a structure being mostly isomorphic to the class hierarchy*

Class scheme
A *{ x,y,f() }*
B : A *{ x,f(),g() }*
C : B *{ z,g(),h() }*
E *{ m() }*
D : C, E *{ m(),f(),a }*



Class tree



Subobject tree

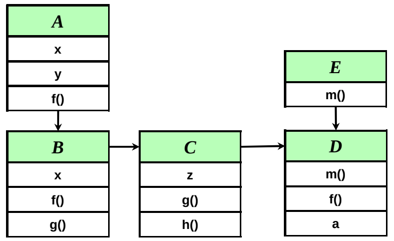
Subobject trees (2/15)

- The tree is composed of distinct nodes with every node corresponding to a class and encompassing all members (data and methods) appearing in the 'body' of its respective class
  - *meaning inside a node we never add what is inherited*
  - *a node is a partial object for subclasses and a fully working object for top level classes*

Class scheme

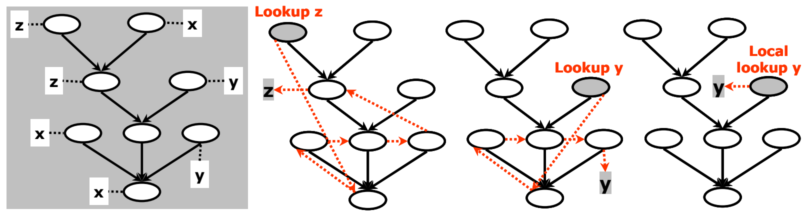
A	*{ x,y,f() }*
B : A	*{ x,f(),g() }*
C : B	*{ z,g(),h() }*
E	*{ m() }*
D : E, C	*{ m(),f(),a }*

****assume C more recent than E in D subclass*



Subobject trees (3/15)

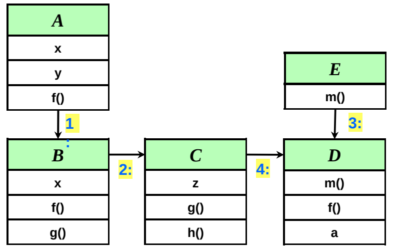
- The lookup semantics are very simple
  - the search order is a breadth-first search, left to right for siblings, *always starting from the root*
    - notably, the search order does not depend on the node from which a lookup is invoked
    - thus, for a given index, the lookup returns always the same result from any node of the tree



Subobject trees (4/15)

- Assuming the root at ***level* `0`** (leaves at the top) the lookup algorithm visits nodes with the following rules:
  - the nodes of ***level* `n`** are visited before the nodes of ***level* `n+1`**
    - the root being at ***level* `0`** is always visited first
  - a node at ***level* `m`** is visited before its rightmost siblings at the same level
- For nodes ***β*** at ***level* `n+1`** and ***α*** at ***level* `n`** where ***α*** is linked with ***β*** we denote $\alpha \swarrow \beta$
  - we say ***α*** *inherits from* ***β*** and ***β*** *is a direct base of* ***α***
  - we may refer to ***β*** *as an inheritance parent of* ***α***, or simply a parent, with a <u>meaning on inheritance and not on tree connectivity</u>

Subobject trees (5/15)



1:	*inherit(B,A)*	*B↙A*
2:	*inherit(C,B)*	*C↙B*
3:	*inherit(D,E)*	*D↙E*
4:	*inherit(D,C)*	*D↙C*

Lookup (visit) ordering: D→C → E → B → A

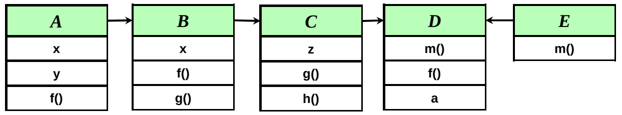
inherit(α, β)	Add α↙β with β *leftmost* parent of α, return α
inheritredirect(α, β)	Cancel any α′↙β if it exists, then *inherit(α,β)*
uninherit(α, β)	Cancel α↙β and return β
isderived(α, β)	Returns true if α←β else false
getbases(α)	Returns list of α parents ordered left to right
getderived(α)	Returns derived object if any, *nil* if none
getmostderived(α)	Returns most derived object (can be α itself)

Subobject trees (6/15)

Rule. *In invocations **self** is always the method owner.*
Axiom. *Late binding applies for methods and properties.*
class-based design → ***method A::f() { x $^{\text{early bound}}$; y $^{\text{late bound}}$; }***
object-based implementation → ***method A::f() {***
  self..x; ☑ ***local access via self guarantees absolute addressing***
  self.y; ☑ ***late bound by definition***
}

class-based design → ***method B::g() { A::f() $^{\text{base method invocation}}$;}***
object-based implementation → ***method B::g() {***
  getbases(self)[0]..f(); ☑ ***works due the rule: we get the bases of B self***
  self..A..f(); ☑ ***alternatively, handled explicitly by clients***

****In our last case we use **self..A** and not merely **self.A** to ensure we get the **local A** of B just in case there is another **most derived A** field in B derivatives.*
}



***record** Method {*
  Address address;
  Object owner;
}

•In methods we set the ***owner*** field only once upon construction of the object actually owning the method.

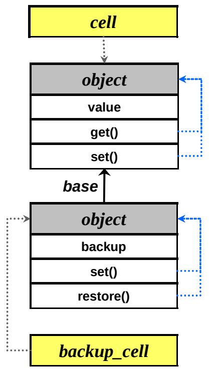
•When a method is invoked, ***self*** binds always to its owner.

•The language may allow ***self*** to be mutable for advanced programming patterns.

Subobject trees (7/15)

Assume that *@<id>* is syntactic sugar for *self..<id>* (local access):

```
cell = object {
        value:0,
        method get()  { return @value; },
        method set(x) { @value = x; },
};

backup_cell = object {
        backup:0,
        method set(x)
          { @backup = self.value;
            getbases(self)[0]..set(x); },
        method restore() { self.value = @backup; }
};
inherit(backup_cell, cell);
cell.restore();             Well defined.
backup_cell.set(23);        Well defined.
backup_cell.restore();      Well defined.
```



Subobject trees (8/15)

- The fact that a lookup in a tree returns the same result regardless of the subobject from which it is initiated is an interesting property
- ***subobject substitutability***
  - Assume a tree T of subobjects $\alpha_1,\ldots,\alpha_N$ and an expression E involving any subobject $\alpha_j \in T$
  - Let ***eval*** denote the evaluation of E at runtime.
    - We require that during the evaluation of E, T may be changed only after the last access of α_j is performed
  - We define $E(\alpha_j \text{ by } \alpha_\kappa)$ to be the expression gained when substituting every occurrence of α_j in E by another $\alpha_k \in T$
  - *Then the following holds:*

Subobject trees (9/15)

$$\begin{aligned} \textit{eval } E = \quad & \textit{eval } E(\alpha_j \textit{ by } \alpha_1) &= \\ & \textit{eval } E(\alpha_j \textit{ by } \alpha_2) &= \ldots \\ & \textit{eval } E(\alpha_j \textit{ by } \alpha_{j-1}) &= \\ & \textit{eval } E(\alpha_j \textit{ by } \alpha_{j+1}) &= \ldots \\ & \textit{eval } E(\alpha_j \textit{ by } \alpha_N) & \end{aligned}$$

- In other words, for functions involving only late-bound member access, thus not applying qualified local access, the subobjects of the same tree behave as synonyms to each other
- Thus they are referentially equivalent, since we can substitute any subobject with another in an expression and still gain the same evaluation outcome
- This feature, not supported in delegation, is crucial to allow implement generic / polymorphic functions

Subobject trees (10/15)

- Now lets explain why a tree structure is an appropriate and adequate model for subclass instances
- We define the opposite of *inheritance*, being *donation* as a relationship among objects in the context of subclassing
- Let $B : A_1,\ldots,A_n$ be a subclassing scheme T
  - Then instB contains $instA_1,\ldots,instA_n$ and the members introduced explicitly in B
    - ***donates($instA_j \rightarrow instB$ in T)***
  - Additionally, instB contains $instS_1,\ldots,instS_k$ for S_i a shared base class in the inheritance scheme of any Aj
    - ***donates($instS_j \rightarrow instB$ in T)***

Subobject trees (11/15)

- *Lemma*. Given *donate(instX→instY in T)* there can be no $instZ \neq instX$ where *donate(instX→instZ in T)*
- *Proof*.
  - (α) If X is a shared base class of Y and Z then *instX* appears only once by definition, thus we come to contradiction.
  - (β) Let $Y{:}X$ aT_1 subschema of T, $Z{:}X$ *a* T_2 subschema of T. Since X is repeatedly inherited it holds:
    - $donate(instX' \rightarrow instY \text{ in } T_1) \wedge donate(instX'' \rightarrow instZ \text{ in } T_2)$ for distinct X objects $instX' \neq instX''$ *(by repetition semantics)*
    - By hypothesis $\Rightarrow instX = instX' \wedge instX = instX'' \Rightarrow instX \neq instX$, thus we come to contradiction

Subobject trees (12/15)

- Our proof essentially states that in object structure models, defining how subclass instances are composed, *base objects donate only once to their derived objects*
- Subobject trees impose that a base object has only one outgoing donation link
  - i.e., can to donate only to one object
- Thus subobject trees are sufficient to model the object structure of subclass instances
  - i.e., the single derivation link is not restrictive

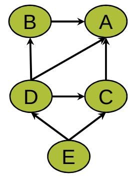
Subobject trees (13/15)

- Shared base objects will have to appear only once in a subobject tree
  - *but to which subobject in the tree they donate?*
  - we define the donation rules in a way preserving the lookup semantics of class-based shared inheritance
- Lets assume *S* a shared base class in the inheritance schema *T* of a subclass
  - Let $A_1,\ldots,A_n$ be all subclasses of *S* in *T* so $A_i{:}S$
  - Assume A_r be the more recent of those in *T*
  - Then *donate(instS→inst*A_r *in T)*

Subobject trees (14/15)

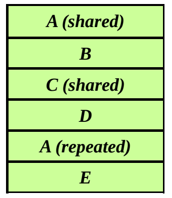
Class scheme (linearization exists)

A	*{ x, y, f() }*	***A***
B: shared A	*{ x, f(), g() }*	***BA***
C: shared A	*{ z, g(), h() }*	***CA***
D: A, B, shared C	*{ m() }*	***DCBAA***
E: shared C, D	*{ m(),f(),a }*	***EDCBAA***



class graph

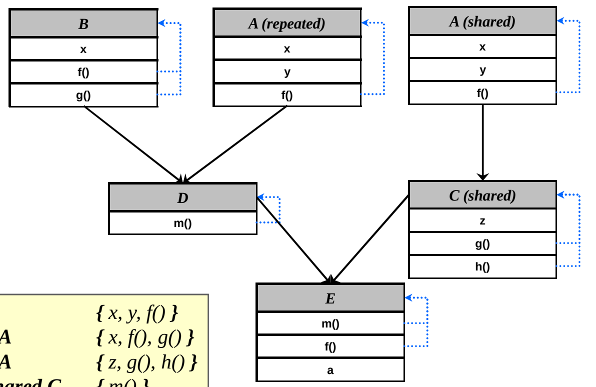
Hint: Never use the class graph to extrapolate ordering (it says nothing about that). Refer to the subclass definitions and the language rules about the ordering semantics when specifying the list of base class names.



A (shared)
B
C (shared)
D
A (repeated)
E

object model

Subobject trees (15/15)



A shared object donated to the subobject of the most recent subclass of A being C

C shared object donated to the subobject of the most recent subclass of C being E

A	*{ x, y, f() }*
B: shared A	*{ x, f(), g() }*
C: shared A	*{ z, g(), h() }*
D: A, B, shared C	*{ m() }*
E: shared C, D	*{ m(),f(),a }*

Lookup sequence: E→D→C→B→A→A

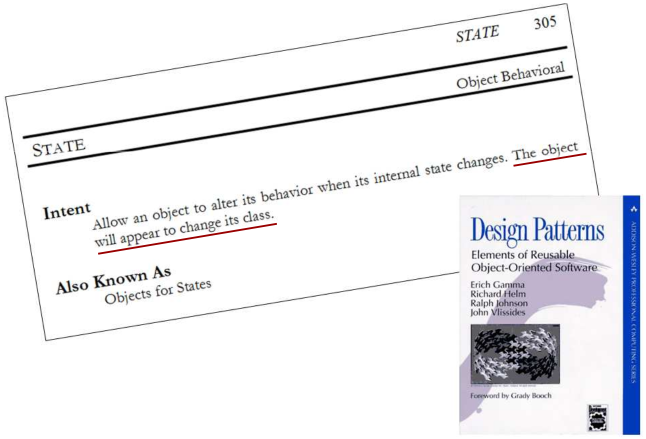
Issues (1/10)

- *Uninheritance (1/4)*
  - Since inheritance is applied at runtime we could be able to undo it – at least in principle
  - In case of object structure emulation, where objects are self-contained, uninheritance seems to be well defined
    - could imply unpacking or unlinking of subobjects
    - undo effect is local on the subject object with no other implications on the rest of the program
  - Uninheritance at the level of class objects, with graph structure emulation, requires special attention
    - effect on behavior would apply on all related instances
    - but uninheritance on state requires per-object intervention

Issues (2/10)

- *Uninheritance (2/4)*
  - *But is uninheritance needed in real practice anyway?*
    - In general, need-driven criteria are dangerous as they imply a language feature is unnecessary unless it address a known demand of the current practice
    - In some cases features do appear from current practice but in other cases they shape out future practices
      - like function-call operator overloading of C++ (clearly, one may live without it, but its use leads to very interesting patterns)
      - or imagine a language designer in the early sixties 60's arguing "*who would need inheritance?*"
  - There is one well known design pattern that is strongly linked to object-based uninheritance
    - *The State pattern*

Intermezzo


STATE 305

Object Behavioral

STATE

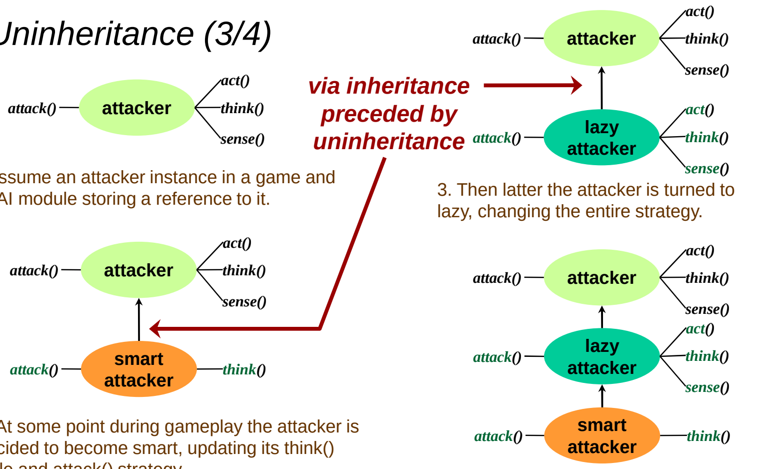
Intent Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Also Known As Objects for States

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch

Issues (3/10)

- *Uninheritance (3/4)*



1. Assume an attacker instance in a game and the AI module storing a reference to it.

2. At some point during gameplay the attacker is decided to become smart, updating its think() style and attack() strategy.

3. Then latter the attacker is turned to lazy, changing the entire strategy.

4. Finally, the lazy attacker gets smarter.

Issues (4/10)

- *Uninheritance (4/4)*

```
attacker = object
{       method sense(){…}, method think(){…}, method act(){…},
        method attack() {…}       };
AI.register(attacker);
smart_attacker = object
{       method think(){… getbases[0]..think(); },
        method attack(){… getbases[0]..attack(); } };
lazy_attacker = object
{       // Assume all super calls are done
        method sense(){…}, method think(){…}, method act(){…},
        method attack() {…}       };

// Notice that derived objects need not know if they inherit directly from
// 'attacker' or another object already inheriting from 'attacker'.
inherit(smart_attacker,   getmostderived(attacker));
uninherit(smart_attacker, getmostderived(attacker));
inherit(lazy_attacker,    getmostderived(attacker));`
```

Issues (5/10)

- *Non-monotonic evolution (1/5)*
  - An instance may dynamically deviate from with the object model of its respective class due to:
    - addition or removal of attributes
    - inheritance
      - getting extra heritage comparing to what is defined in its class
    - uninheritance
      - abandoning part of the heritage defined in its class
  - When the latter takes place an instance may sometimes offer less than its original object model
  - In this case the object is said to evolve non-monotonically
    - i.e. not preserving subclass ordering

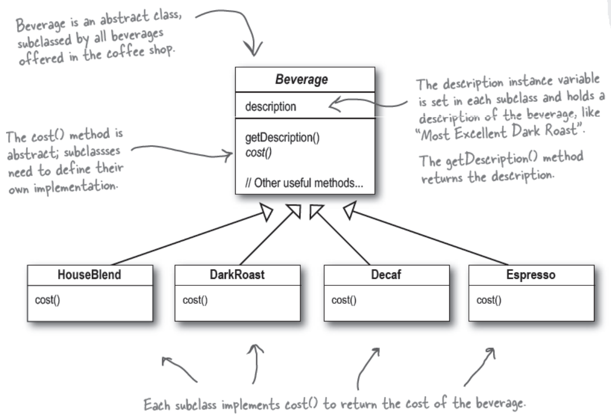
Issues (6/10)

- *Non-monotonic evolution (2/5)*
  - To support *monotonic evolution* an object may only '*expand*' but never '*shrink*'
    - with monotonic evolution an object can still be used in ways compliant to its subclass
  - But one should not so easy assume *non-monotonic evolution* to be an evil design idea
    - Consider a program in which instances are manipulated through superclass interfaces
    - Then it will not necessarily break if objects evolve non-monotonically regarding their subclasses
    - But always guarantee to offer their superclass goodies
      - like *the State pattern*

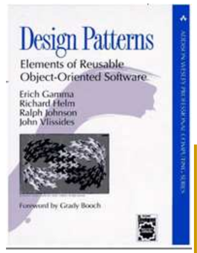
Issues (7/10)

- *Non-monotonic evolution (3/5)*
  - Furthermore, a program may be designed to handle safely non-monotonic evolution
    - An algorithm may conditionally perform actions depending on presence of certain attributes or heritage from specific objects
  - In other cases adding or removing layers of inheritance is the actual design intent
    - Like dynamic decorators, e.g. having dynamic shadows or special effects added or removed on windows
      - the *Decorator pattern*
    - Recall the shadowed window example: we could simply uninherit the shadowing part and have a working program on a valid state
    - We could even further donate shadowed behavior to another instance

Intermezzo


Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.
Beverage
description
getDescription()
cost()
// Other useful methods...
The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".
The getDescription() method returns the description.
The cost() method is abstract; subclassses need to define their own implementation.
HouseBlend
cost()
DarkRoast
cost()
Decaf
cost()
Espresso
cost()
Each subclass implements cost() to return the cost of the beverage.

Decorator – Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Issues (8/10)

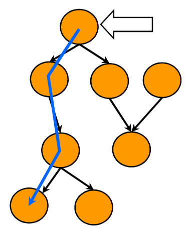
- *Non-monotonic evolution (4/5)*

```
uninherit(shadow, win);                        Case 1: if we have both
inherit(shadow, another);

uninherit(shadow = getderived(win), win);      Case 2: if we have the 'win' (super)
uninherit(shadow, another);

uninherit(shadow, getbases(shadow)[0]);        Case 3: f we have the 'shadow' (derived)
uninherit(shadow, another);
```

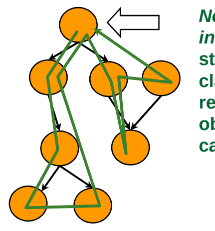
  - The general situation in which an object freely changes its type during runtime is called ***dynamic reclassification***
  - ***Monotonic evolution*** is the special case of reclassification where an object may only change types down its inheritance tree, meaning it changes to a subtype, thus preserving the subtype ordering
  - Monotonic evolution can be implemented via successive dynamic inheritance operations. It is like decoration, though with absolutely safe upcasting.

Issues (9/10)

- *Non-monotonic evolution (5/5)*



***Monotonic evolution*: starting from an initial class, only downwards evolution is allowed (i.e. can't uninherit)**



***Non-monotonic inheritance evolution*: starting from an initial class, any evolution resulting in a subclass object is allowed (i.e. can uninherit).**

Issues (10/10)

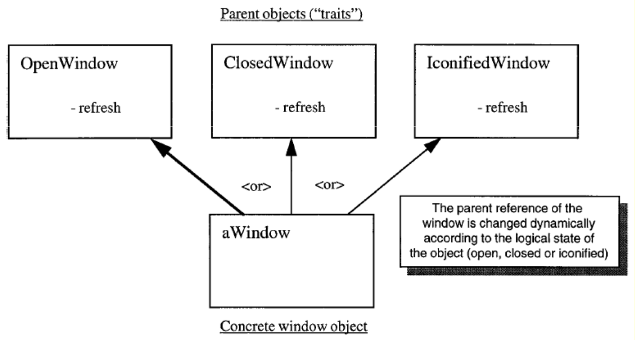
- *Mode switching (1/3)*
  - Consider that an object is allowed to change its base type during runtime
    - it looks similar to the State pattern, but the idea is more general as we can think of altering any class in the class tree of an object
  - In this scenario a class is replaced by another offering at least the same attributes
    - but with alternative semantics
  - Such a feature would allow modularly switch the behavior of objects from one mode to another
    - and is called *mode switching*

Issues (11/10)

- *Mode switching (2/3)*
  - The technique could be implemented per object with all three mechanisms we have studied
    - *embedding*: by substitution of base packets or overwriting
    - *delegation*: by reparenting
    - *subobject trees*: by uninheritance
  - Notably, delegation could allow mode switching to be applied globally at the class level
    - but if the state model (data) changes massive reconstruction of affected instances is needed (how is this done?)
  - In any case, mode switching requires copy or convert object state from the previous mode to the new mode
    - An example will clarify this issue

Issues (12/12)

- *Mode switching (3/3)*


Parent objects ("traits")
OpenWindow
- refresh
ClosedWindow
- refresh
IconifiedWindow
- refresh
<or>
<or>
aWindow
The parent reference of the window is changed dynamically according to the logical state of the object (open, closed or iconified)
Concrete window object

When changing window modes via type change, some state of the old subobject (e.g. an **OpenWindow** subobject) may have to be transferred or converted to the new subobject (e.g., a **ClosedWindow** subobject).

But there is no guarantee that the two classes reveal their intrinsic state so as to make such state propagation possible. Usually the latter is bypassed as most object-based languages have no access qualifiers.