Programming Language Concepts

Prof. Dr. Klaus Ostermann

Aspect-Oriented Programming Group
Darmstadt University of Technology

Inheritance and beyond

Inheritance

- Different interpretations of inheritance have already been discussed
  – conceptual specialization
  – code reuse/sharing
  – subtyping
- In the following: Forms of inheritance
  – single inheritance
  – multiple inheritance
  – mixin inheritance
- Focus on code sharing
  – "implementation inheritance" not "interface inheritance"

Descendant-driven Inheritance in Java

```
class Person {
  String name;
  void display() { print(name); }
}
class Graduate extends Person {
  String degree;
  void display() { super.display(); print(degree); }
}
```

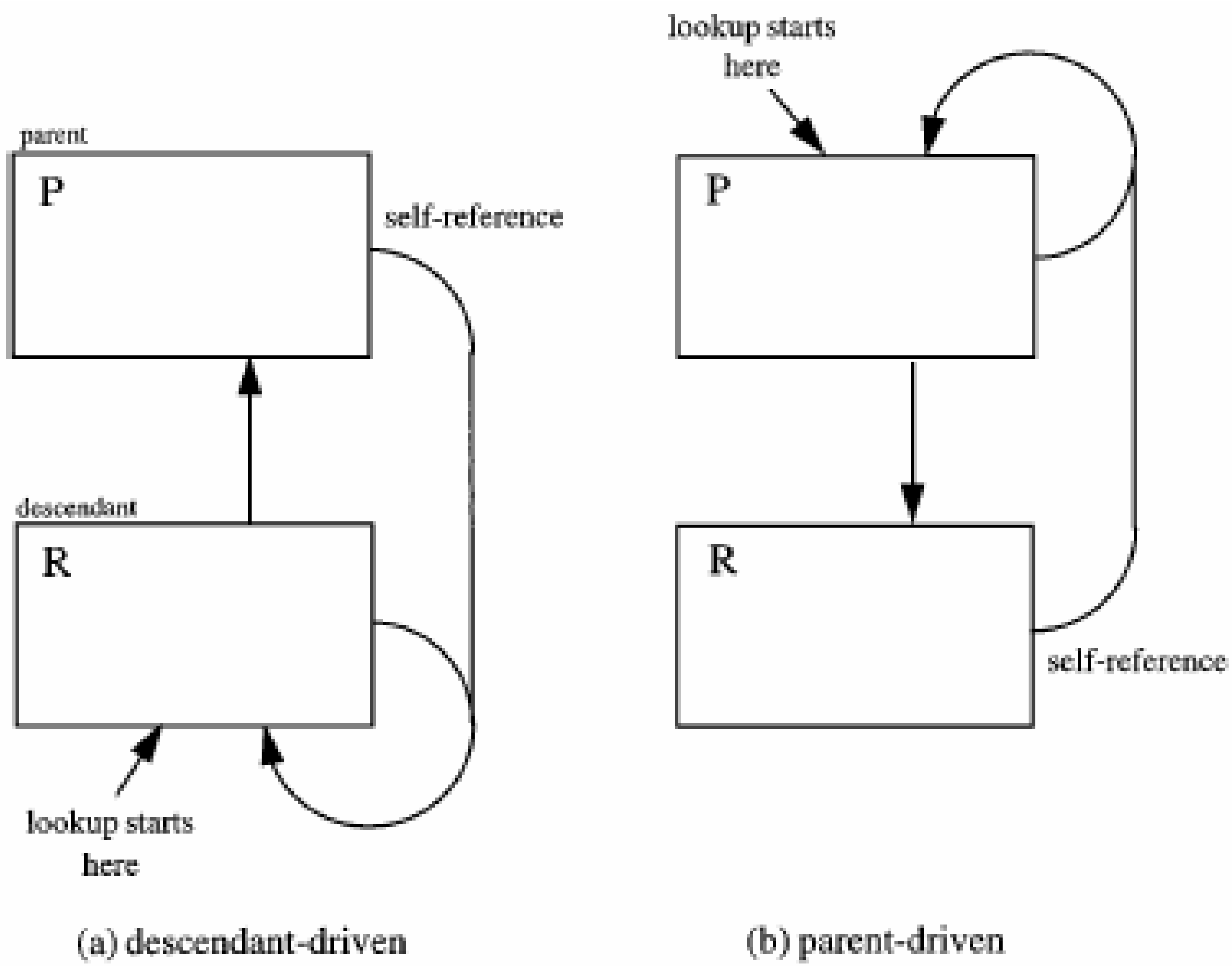
- The subclass defines a delta: $C = \Delta(P) + P$
- The subclass is in control:
  - there is no way one can prohibit the subclass to redefine display such that it displays the time
  - The subclass decides whether degree should be printed first or last

Parent-Driven Inheritance in Beta

```
Person: class
(# name: string;
    display: virtual proc
      (# do name.display; inner #);
#);
Graduate: class Person
(# degree: string;
    display: extended proc
    (# do degree.display; inner #);
#)
```

- The superpattern is parameterized with a subpattern
- The superpattern is in control:
  - It is impossible to print “Dr.” *before* the name, design decision in Person
- Better control over behavioral compatibility
- Less possibilities to “patch” in unanticipated ways

Descendant- vs. Parent-driven Inheritance



Single Inheritance

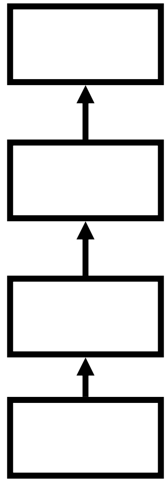
- Every class has at most one superclass
  - languages with “root”: exactly one superclass
- Superclass structure forms a tree (common root) or a set of disjoint trees (no common root)
- Advantage: Semantically and operationally simple
  - unique path from every class to its root, path = linear order for method dispatch
- Difficult to encode multiple classification hierarchies

Multiple Inheritance

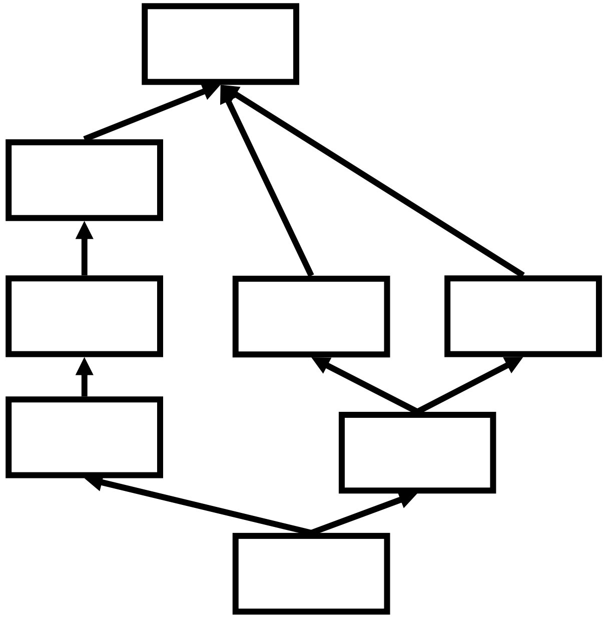
- Sometimes, single inheritance seems to be insufficient
  - WaterSkiingAccount extends Accountant, Waterskier
- MI: Classes can have more than one superclass
- Inheritance Structure no longer a tree but a directed acyclic graph (DAG)
  - no unique path to root
- “Multiple Inheritance is good but there is no good way to do it” (Alan Snyder)
- Languages with MI: Eiffel, C++, CLOS

Multiple Inheritance

- Single Inheritance: Use unique path to root for method dispatch
- Multiple Inheritance: Need means to search inheritance DAG: which path to follow?



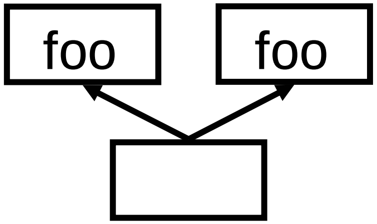
Single Inheritance



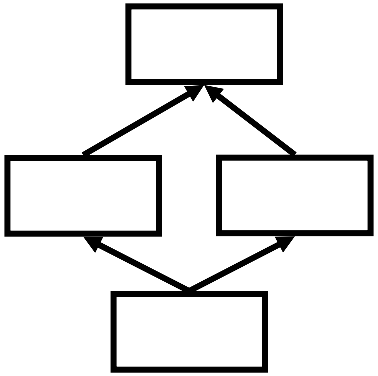
Multiple Inheritance

Multiple Inheritance

- Two basic problems:
  - name clashes
  - diamond inheritance
    - a.k.a. repeated inheritance, fork-join inheritance, common roots
- No generally accepted solution available
  - controversial subject in PL research
  - each PL has a different solution



name clash



diamond inheritance

Name Clashes

- Inheritance of two independent methods that – incidentally – have the same name
- No deep semantic conflict, but need to be resolved
  - either by the class introducing the conflict
  - or by clients of the class

Resolving Name Clashes in C++

```
class Cowboy {
  public:
  virtual void draw();
};
class Displayable {
  public:
  virtual void draw();
}
class DisplayableCowboy: public Cowboy, public Displayable {…}

void f(DisplayableCowboy *c) {
  c->draw(); // ambiguous
  c->Cowboy::draw();  // OK
  c->Displayable::draw(); // OK
}
```

Resolving Name Clashes in C++

- Either explicit disambiguation by client...
- or class must override both methods and somehow combine the results
  – only an option if sensible combination exists
- Breaks encapsulation
  – client needs to know about inheritance structure
- Fragile
  – adding a method may break many clients

Resolving Name Clashes in Eiffel

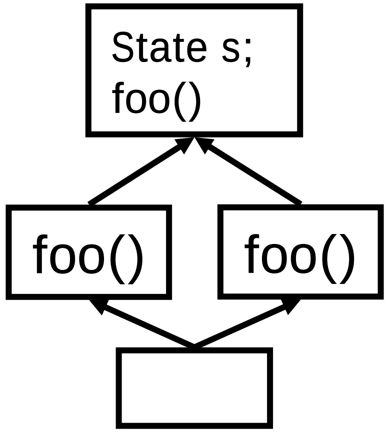
```
class DisplayableCowboy inherit
  Displayable
     rename draw as disp_draw end;
  Cowboy
     rename draw as cowb_draw end;
...
end
...
c: Cowboy; d: Displayable; dc: DisplayableCowboy

dc.disp_draw(); // OK
dc.cowb_draw(); // OK
dc.draw(); // error
c= dc;
c.draw(); // OK
d = dc;
d.draw(); // OK
```

Resolving Name Clashes in Eiffel

- Renaming better solution than in C++
  - can be completely resolved in inheriting class, clients do not need to bother
- Fragility problem less severe
  - can rename new method

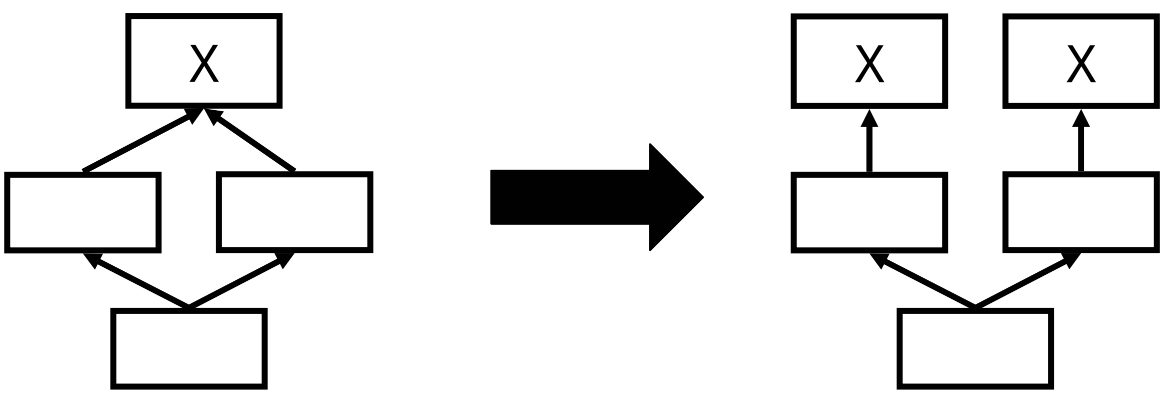
Diamond Inheritance

- Not a syntactic problem (as name conflicts) but semantic problem
- What happens to the state that is inherited twice?
  - one shared copy? two copies?
- How are method conflicts resolved?
  - renaming not sufficient



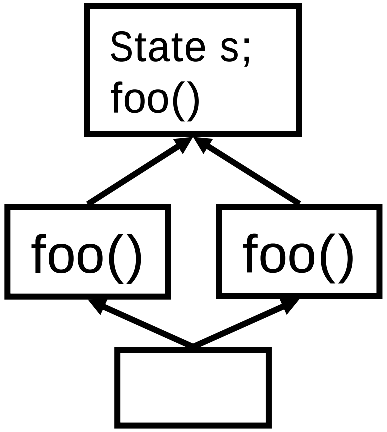
Diamond Inheritance: Approach 1: Convert to Tree

- DAG is converted into tree by **duplicating** shared nodes
- Remaining naming conflicts need to be resolved
- Duplicate parent not always desired semantics
  - e.g., Point, HistoryPoint, BoundedPoint
- Default Behavior in C++
  - use scope resolution operator :: to disambiguate



Diamond Inheritance: Approach 2: Deal with graph

- Sometimes, replicated parents are not an option
  - Points, HistoryPoints, BoundedPoints
- Shared parents
  - need to deal with method conflicts
  - need to deal with shared state
    - how to maintain invariants in shared ancestor
- Exposes inheritance structure
  - class needs to know inheritance structure of its ancestors

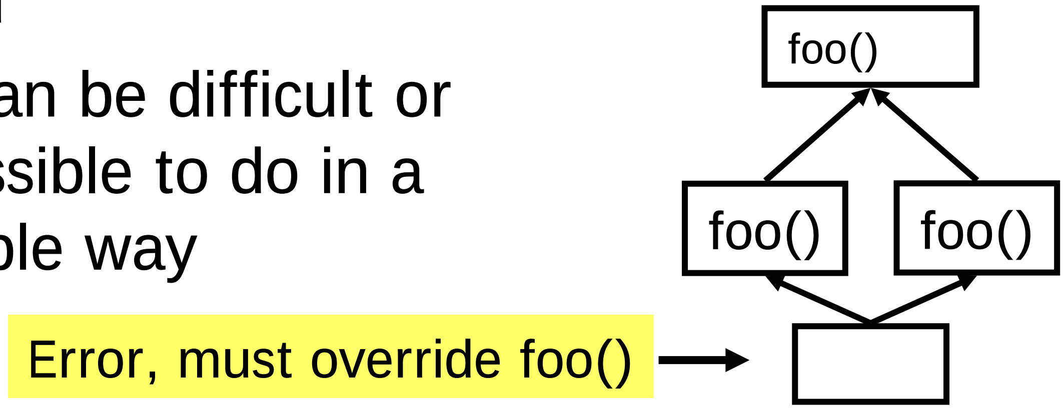


Maintaining invariants in shared ancestors

```
class X {
  public:
  int i;
  X(int j) { i=j;}
};

class Y: public virtual X {
  public:
  Y(): X(3) {} // Y assumes X.i is 3
};
class Z: public virtual X {
  public:
  Z(): X(5) {}  // Z assumes X.i is 5
};
class XYZ: public Y, public Z {
{
  public:
  XYZ(): X(2) {}
};
```

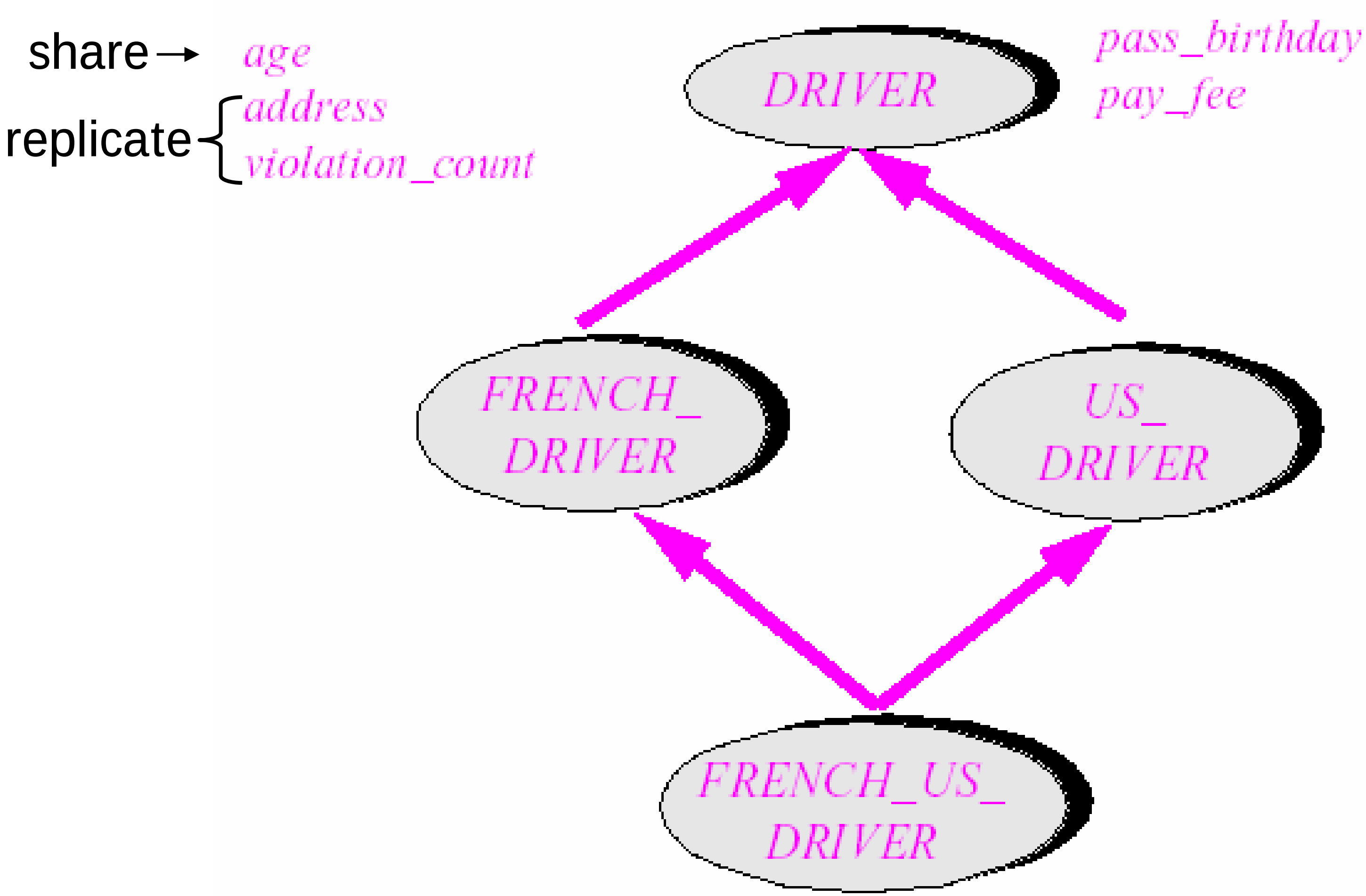
Virtual Base Classes in C++

- Classes can mark base classes as “virtual”
  public MyClass: public **virtual** MyBase {…}
- Meaning: Every virtual base of a derived class is represented by the same (shared) object
- Method conflicts are resolved by the obligation to implement a most specific method
  – this can be difficult or impossible to do in a sensible way



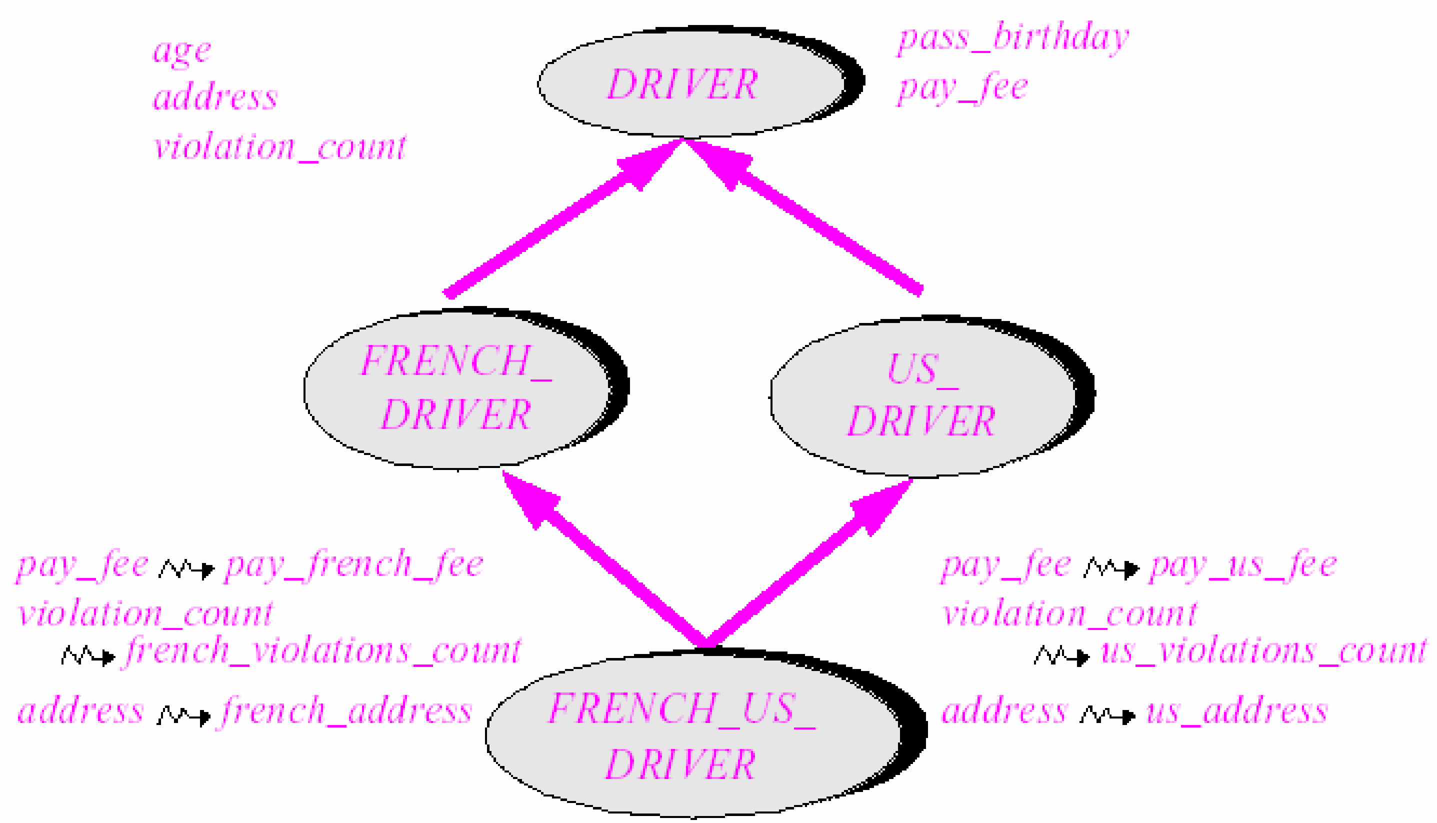
Multiple Inheritance in Eiffel

- Sharing/Replication more fine-grained: can be different for individual inherited variables (!)



Multiple Inheritance in Eiffel

- Repeated Inheritance Rule:
  – Versions of a repeatedly inherited slot inherited under the same name represent a single feature (sharing)
  – Versions inherited under different names represent separate features (replication)

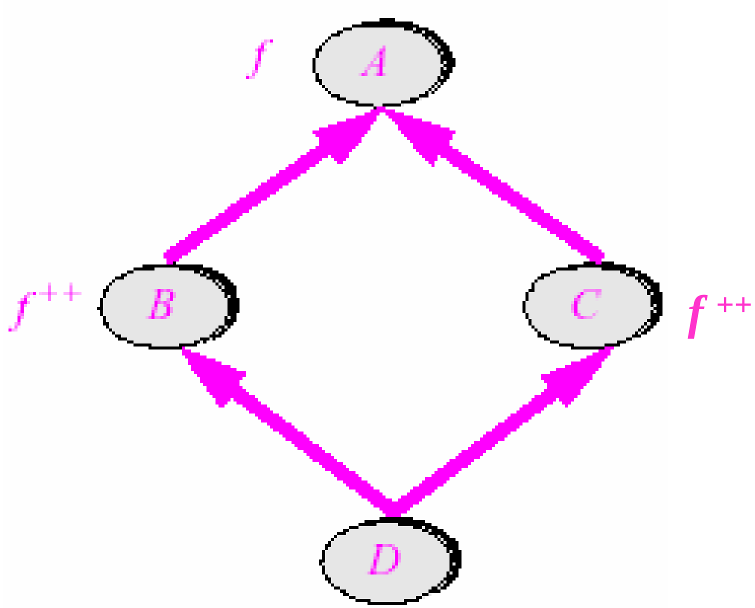


Multiple Inheritance in Eiffel

```
class French_US_Driver inherit
    French_Driver
        rename
            address as french_address,
            violations as french_violations,
            pay_fee as french_pay_fee
        end
    US_Driver
        rename
            address as us_address,
            violations as us_violations,
            pay_fee as us_pay_fee
        end
feature     . . .  // shared:  age, name, ...
end;
```

Multiple Inheritance in Eiffel

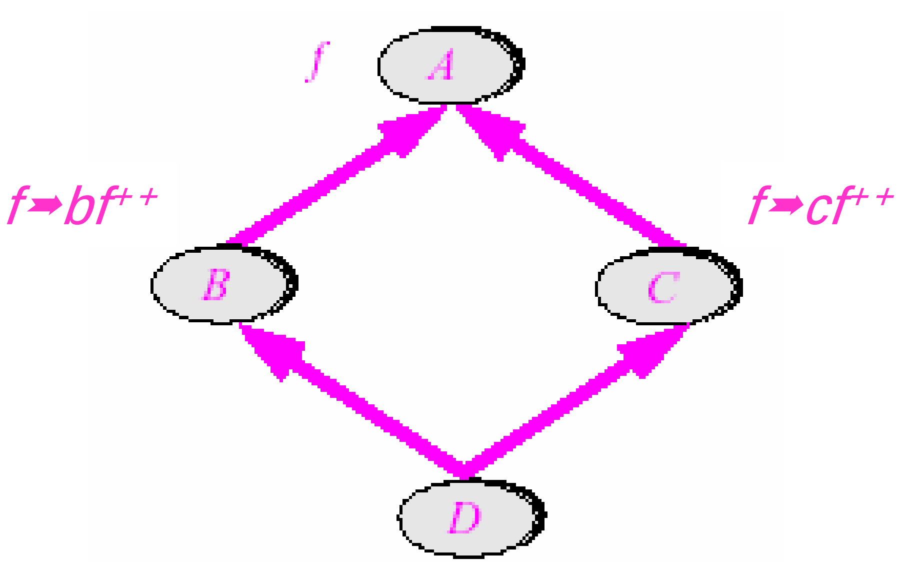
- Conflicts under sharing
  - Resolve with undefine




```
class D inherit
        B
        C
              undefine f end
feature
        ...
end
```

Multiple Inheritance in Eiffel

- Conflicts under Replication
  - Resolve with “select”



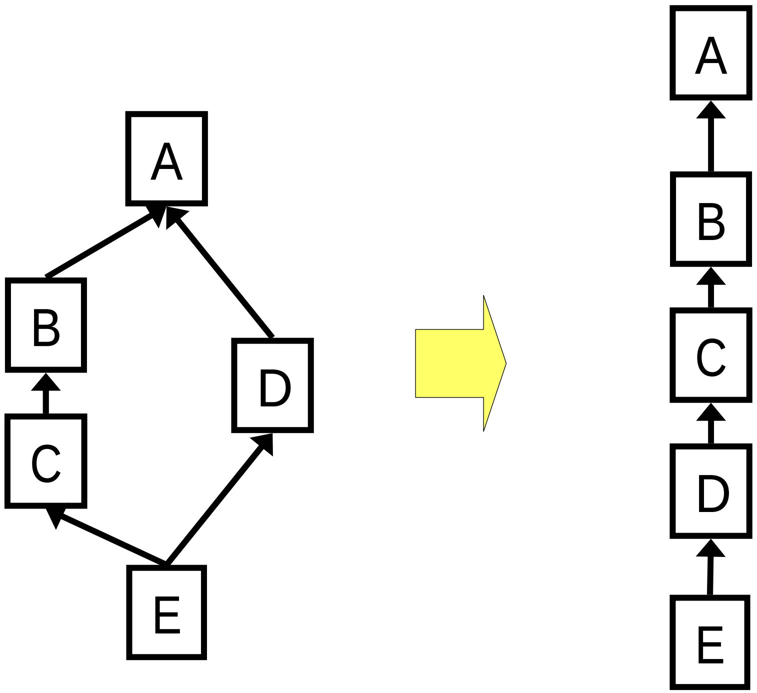

```
class D inherit
      B
              select bf end
      C
feature
      ...
end
```

Linearized Inheritance

- Tree Inheritance: Copy
- Graph Inheritance: Sharing, deal with conflicts
- C++, Eiffel: Mixture of Tree/Graph Inheritance
- Linearized Inheritance: flatten inheritance structure into linear chain without duplicates
  - search this chain in order to find slots
  - works via topological sort of Inheritance DAG

Linearized Inheritance

- Create linear order, e.g., via depth-first search
- Linear order determines method dispatch order
- Advantage:
  - no ambiguities
- Disadvantages:
  - total order is not unique
  - linearization inserts unrelated classes between related ones
  - Methods may be overriden accidently



```
(defclass Person ( ) (name))
(defmethod display (( self Person))
  (display (slotvalue self ' name)))
```

inherits from   list of inst. vars

```
defclass Graduate (Person) (degree))
(defmethod display ((self Graduate))
    (call-next-method)
    (display (slot-value self ' degree)))
```

(= super along linearized inh. chain)

```
(defclass Doctor (Person) ( ))
(defmethod display (self Doctor))
    (display "Dr.")
    (call-next-method))
```


```
(defclass Research-Doctor (Doctor Graduate) ( ))
```

- Linearization algorithm takes order of superclass declaration into account

Multiple Inheritance: Summary

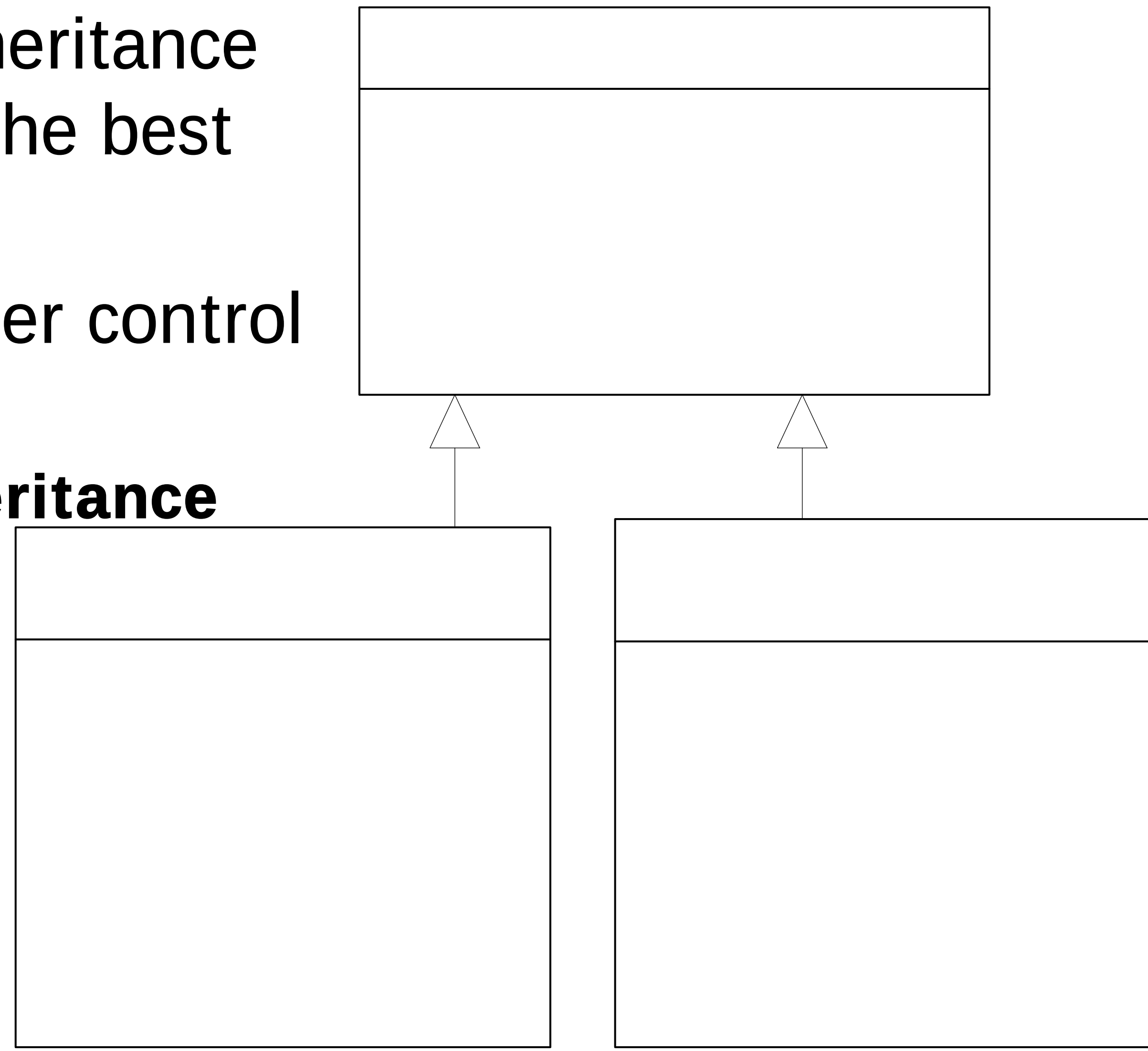
- Still controversial whether multiple inheritance is necessary
- MI frequently misused to model multiple independent classification hierarchies
  - wait for next generation aspect-oriented languages ☺
- Different strategies have different strenghts
  - one strategy cannot easily simulate another one
  - Tree inheritance: semantically easy, not appropriate if replication is not an option
  - Graph inheritance: semantically complicated, programmer has to deal with problem
  - Linearized inheritance: semantically easy, powerful method combination semantics, may be surprising

From Multiple- to Mixin-Inheritance

Want BufferedCompressedOS

- Linearized inheritance seems to be the best solution
- But want better control over ordering

  ⇒ **Mixin Inheritance**



Mixin Inheritance

- Idea: Parameterize class with its superclass
  - actual superclass can be subclass of static superclass

  a la

  BufferedCompressedOS = BufferedOS<CompressedOS>;
  - available as “pattern” in CLOS
  - can be simulated with C++ templates
    - superclass = template parameter
  - can be approximated with decorator pattern
    - no late binding, but dynamic
  - type-safe variants without accidental overriding exist
    - MixedJava, JAM

F-bounded parametric polymorphism

What is it?

- Polymorphism
  - “Works” for things of different types.
  - Some type information under-specified.
- Subtype Polymorphism
  - The “different types” must be related by the “subtype” relation (usually hierarchical).
- Parametric Polymorphism
  - The “different types” may be unrelated.

A Simple Example ...

Without Param. Poly.

```
// create a collection
// just holds “objects”
Vector v = new Vector();

// add an object
v.add(new String(“hello”));

// retrieve the object
// requires a cast
String s = (String) v.get(0);
```

With Param. Poly.

```
// create a collection
// holds Strings! (expressiveness)
Vector<String> v =
  new Vector<String>();

// add an object
v.add(new String(“hello”));

// retrieve the object
// no cast required! (safety)
String s = v.get(0);
```

Why is it good?

- Static type safety
- Expressive power
- Reduced code maintenance
- Execution efficiency (potentially)

Language Design

- Principles
  - Covariance & Contravariance
  - Constrained vs Unconstrained (F-bounds)
  - “Binary” methods
  - Reference types vs Primitive types
- Proposals
  - Pizza, GenericJava (GJ), NextGen, PMG, PolyJ
  - Virtual Types & Structural Virtual Types

Parameterized Types

```
Set<E> {
    insert(element : E) {...}
}
```

// Set<Integer> is a type!

// Set is not a type, it is a **function** on types

```
Set<Integer> s = new Set<Integer>();
```

Parametric Polymorphism and Subtyping

$$\begin{tikzcd}
\text{Collection}\textcolor{blue}{<\text{Vehicle}>} & \text{Collection}\textcolor{blue}{<\text{Car}>} \arrow[l] \\
\text{Set}\textcolor{blue}{<\text{Vehicle}>} \arrow[u] & \text{Set}\textcolor{blue}{<\text{Car}>} \arrow[l] \arrow[u, "\text{Structural Subtyping}"']
\end{tikzcd}$$

Covariant Subtyping

Bounded Type Parameters

```
class Pair<A> { A x; A y; }
class Map<E extends Pair> { ... }
```

- Type parameters: A and E
- A is *unbounded*
- E is *bounded* by Pair

- lessThan is a “binary” method
- x.lessThan(y)
- x and y of same type!
- How to type-check that x and y have the same type?

- think of equals(Object obj)
- usually you use an instanceof test on obj

```
interface Ordered<A> {
    boolean lessThan(A obj);
}

class OrderedInt implements
    Ordered<OrderedInt> {
    int i;
    boolean lessThan(OrderedInt j) {
        return i < j.i;
    }
}
```

F-bounds

- B implements Ordered<B>
- B appears in it's own bound!
- That's an "f-bound"
- Necessary to type-check *x.lessThan(y)*
- In general, f-bounds require structural subtyping.

- You may not yet see why the f-bound is necessary ...
- So we'll try to do without it and see where we run into problems ...

```
class Pair<B implements Ordered<B>> {
    B x; B y;
    B min() {
        if (x.lessThan(y)) return x;
        else return y;
    }
}
```

Life without F-bounds

- B implements Ordered<C>
  - B's not in the bound
  - no longer an f-bound
- We know that Pair.x and Pair.y are of the same type.
- But we don't know if that type can be compared to itself!
- e.g. Int can only be compared to Real; not to Int
- e.g. Real can be compared with itself, so it's ok
- We can't type-check *x.lessThan(y)*

```
class Pair<B implements Ordered<C>> {
    B x; B y;
    B min() {
        if (x.lessThan(y)) return x;
        else return y;
    }
}

class Int implements Ordered<Real> {...}
class IntPair extends Pair<Int>

class Real implements Ordered<Real> {...}
class RealPair extends Pair<Real> {}
```

Will be available in Java

- Java Community Process JSR-14
- GenericJava (GJ), Pizza, NextGen
- It's different than C++ templates!
  - C++ templates are just macro-expansion
    - in comparison ...
    - the main problem with C++ is that type-checking is done after expansion
- Some changes to compiler front-ends
- No changes to JIT's, optimizers, etc.