Typology of programming languages

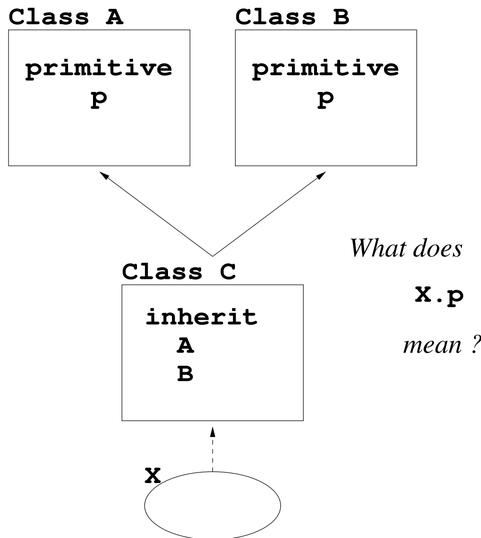
~ Handling inheritance ~

Problem Statement

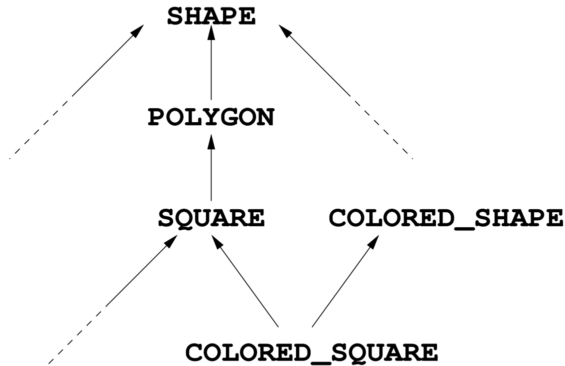
- **Simple Inheritance**: a class may herit at most from only one class
- **Multiple Inheritance**: more powerful than the simple inheritance **but** introduces problems.
  Eiffel proposes the adaptation clauses to solve these problems.

Problem Statement

- **Simple Inheritance**: a class may herit at most from only one class
- **Multiple Inheritance**: more powerful than the simple inheritance **but** introduces problems.
  Eiffel proposes the adaptation clauses to solve these problems.

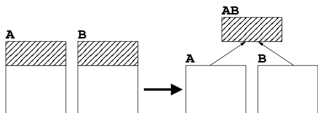


Multiple Inheritance is Sometimes Necessary

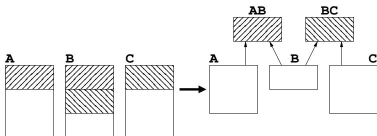


Inheritance for factorization

Simple inheritance helps to factorization:



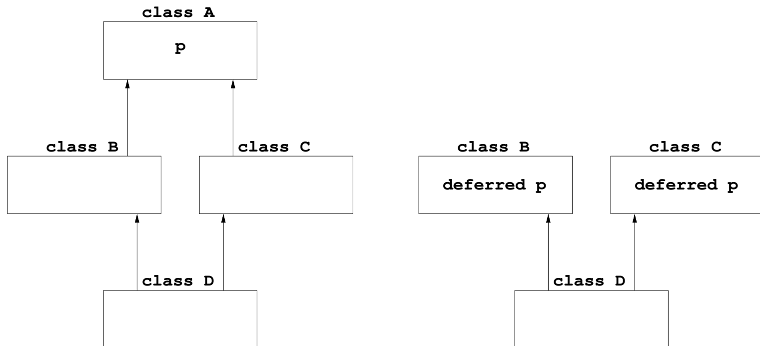
And multiple inheritance is sometimes mandatory



Smalltalk, Java, ... only propose a solution for modelisation while Eiffel also solves the factorization problems.

Jointure of primitives

Two corner cases **deferred** is an Eiffel keyword meaning **virtual** in C++:



Quick Overview of the Other Languages

- Multiple inheritance is forbidden because it raises numerous problems and it is not necessary.
  $\Rightarrow$ Java, Smalltalk, Ada
- Chosse a lookup strategy and the programmer must conform it:
  $\Rightarrow$ C++
- Propose tools (in the language) for solving problems related to multiple inheritance
  $\Rightarrow$ Eiffel's inheritance adaptation clauses.

Adaptation Clauses

Features:

- Rename inherited primitives
- Modify Visibility of inherited primitives
- **A-definition** inherited primitives (make a primitive virtual)
- Redefine inherited primitives
- Selection clauses

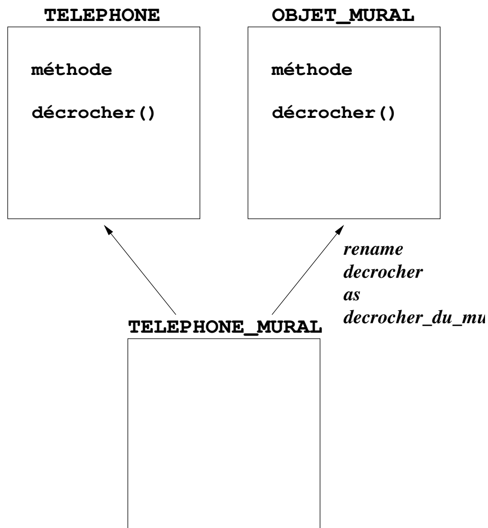
With these operations, we can resolve all problems related to multiple inheritance.

Renaming Clauses

```
class SQUARE

inherit
  SHAPE
     rename
        make as make_shape
     end ;

feature
  width : INTEGER ;
  make(x,y : INTEGER ;
         w : INTEGER) is
     do
        make_shape(x,y) ;
        width := w ;
     end ;

end -- class SQUARE
```

- The renamed primitive is still accessible but with a different name.
- The original name can then be used for another primitive even with a different signature.

(French) Example



Visibility Filter

```
class SQUARE

inherit
   SHAPE
      rename make as make_shape
      export {NONE} make_shape
      end ;

feature

   width : INTEGER ;

   make(x,y : INTEGER ;
          w : INTEGER) is
      do
         make_shape(x,y) ;
         width := w ;
      end ;
end -- class SQUARE
```

- `make_shape` was accessible without reasons in class SQUARE
- May help to mask inherited primitive

Access Restrictions

`feature` ou `feature{ANY}`
: primitives with default access value
(All objects derive from ANY)

`feature{A,B,C,...}`
: primitives with access restricted only to some classes A, B, C

`feature{}` ou `feature{NONE}`
: unreachable primitives
(NONE : no instance from this classe)

Redefinition Clauses

```
class SQUARE

inherit
  SHAPE
     rename make as make_shape
     export {NONE} make_shape
     redefine draw
     end ;

feature

  draw(g : GRAPHICS) is
     do
        ...
     end ;
  ...
```

- Constraints on redefintions
- Each redefinition must be declared
- Redefined methods are targetted by the dynamic lookup

Keep and redefine

Redefinition to support dynamic lookup

Here, we loose dynamic lookup

```
class B

inherit
  A
   rename p as pa end;

feature

   p ... is ...
```

Keep and redefine

Redefinition to support dynamic lookup

Here, we loose dynamic lookup

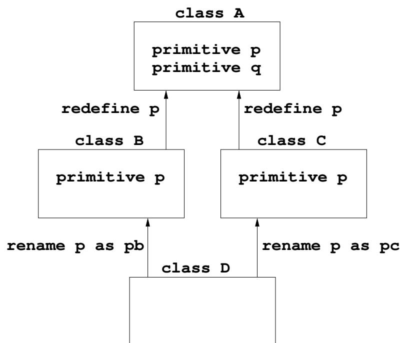
```
class B

inherit
  A
   rename p as pa end;

feature

   p ... is ...
```

Here, dynamic lookup will work:

```
class B

inherit
   A
    rename p as pa end;
   A
    redefine p end;

feature

   p ... is ...
```

Selection Clauses

How to resolve this problem:



Given x : A, what does x.p means? If x references an instance of class A, it is the primitive p from A. Same thing happens for an object of B ou C. What about instances of class D ?

Example:

```
q() is do p() end ;
```

A-definition

The A-définition allows to undefine methods

Useful to ”delete” methods that don’t make sense anymore.

```
class TELEPHONE_MURAL
inherit
   TELEPHONE ;
   OBJET_MURAL
         undefine decrocher
end ;
...
```

Summary