# Programming Languages (Langages Evolués)

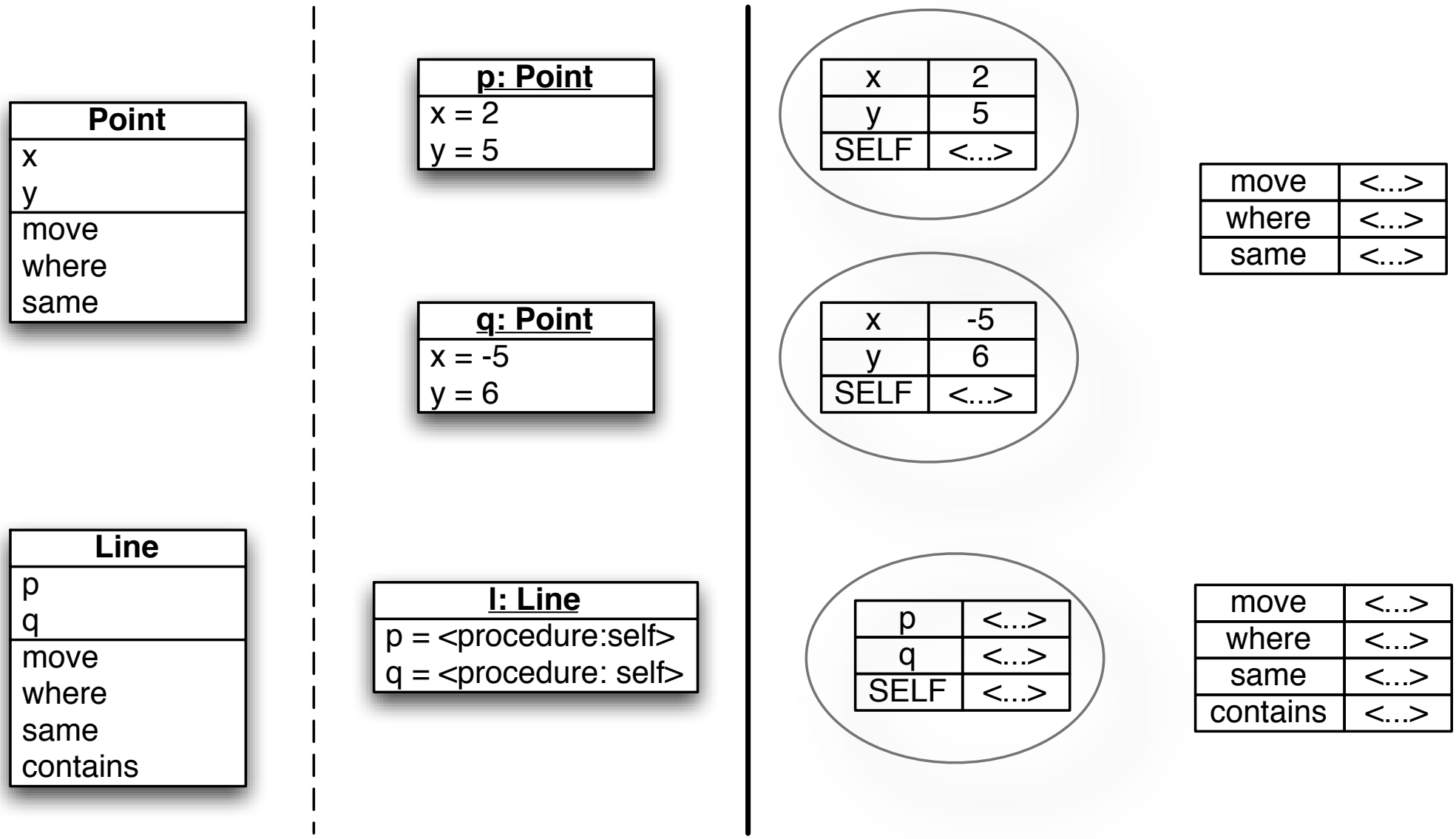
Roel Wuyts

OOP - Inheritance

# OOP Basics: Recap

- Last week we introduced a class-based object-oriented system (reentrant):
  - objects have encapsulated state
  - methods are shared between objects
  - accomplished by using a (hidden) context in methods, and supplying that to evaluate methods
    - So "Self" (this) is dynamic!

# Graphically...



# Initialization & Evaluation

- Initialization when constructing object from class:
  - copy "template" table of instance variables
  - evaluate expressions to get initialized object
- Evaluation
  - lookup method body in (shared!) table
  - supply context (containing args and self) and eval body in that context
    - so method body is shared between objects

# Inheritance

- In class-based object-oriented languages, classes allow methods to be shared among objects
- Furthermore, inheritance allows methods to be shared amongst classes
  - reuse of code (and design...)
- Will introduce single inheritance now
  - and other mechanisms later

# Inheritance

- ☑ built on top of reentrant system
- ☑ variable templates are copied
- ☑ unknown messages are delegated
- ☑ delegation is at meta-level
- ☑ context is passed around
- ☑ root class is required

In

```
(define («TABLE»)
  (define tab '())
  (lambda (op . rest)
    (case op
      ((instantiate)
       (let ((table («TABLE»)))
         (for-each
          (lambda (elt)
            (table 'put (car elt) (cdr elt)))
          tab)
         table))
      ((copy)
       (let ((table («TABLE»)))
         (for-each
          (lambda (elt)
            (table 'put (car elt) (cdr elt)))
          tab)
         table))
      ((get)
       (let* ((key (car rest))
              (entry (assoc key tab)))
         (if entry
             (cdr entry)
```

(<table> 'copy)
returns (deep) copy

In

```
        tab)
        table))
  ((get)
   (let* ((key (car rest))
          (entry (assoc key tab)))
     (if entry
         (cdr entry)
         #f)))
  ((put)
   (let* ((key (car rest))
          (entry (assoc key tab))
          (value (cadr rest)))
     (if entry
         (error "duplicate name" key)
         (set! tab (cons (cons key value) tab)))))
  ((replace)
   (let* ((key (car rest))
          (entry (assoc key tab))
          (value (cadr rest)))
     (if entry
         (set-cdr! entry value)
         (error "undefined name" key))
     #t)))))
```

returns false if message not found

generates error if found/not found

# Inheritance

```
(define Root
  (lambda (MSG . ARGS)
    (case MSG
      ((new)
       (error "cannot instantiate root class"))
      ((«EVAL»)
       (let*
           ((msg (cadr ARGS)))
         (error "method not found " (car msg))))
      ((«COPY»)
       (let ((tab («TABLE»)))
         (tab 'put '«SELF» '())
         tab))
      (else
       (error "invalid class messa
```

instantiation class method

evaluator class method

copy class method

« ... » means hidden

```
(define-macro CLASS
  (lambda (super . defs)
    `(letrec
         ((«SUPER» ,super)
          («METHODS» («TABLE»))
          («VARS» («SUPER» '«COPY»))
          («CLASS»
           (lambda (MSG . ARGS)
             (case MSG
               ((new)
                (let*
                    ((context («VARS» 'instantiate))
                     (self
                      (lambda (msg . args)
                        («CLASS» '«EVAL» context msg args))))
                  (context 'replace '«SELF» self)
                  self))
               ((«EVAL»)
                (let*
                    ((context (co
                     (msg (cadr A
```

super refers to the superclass

class instantiation

method evaluation is delegated to the class

evaluation of message

```
((context («VARS» 'instantiate))
 (self
  (lambda (msg . args)
    («CLASS» '«EVAL» context msg args))))
 (context 'put '«SELF» self)
 self))
((«EVAL»)
 (let*
    ((context (car ARGS))
     (msg (cadr ARGS))
     (args (caddr ARGS))
     (entry («METHODS» 'get msg)))
   (if entry
       (apply entry (cons context args))
       («SUPER» '«EVAL» context msg args))))
((«COPY»)
 («VARS» 'copy))
(else
 (error "invalid class message" MSG))))))
,@defs
«CLASS»)))
```

immediate or delegated

copy class method

```
(define-macro SUPER
  (lambda (msg . args)
    `(«SUPER» '«EVAL» «CONTEXT» ',msg ,args)))
```

# Inherit

```
(letrec
  ((«SUPER» Root)
   («METHODS» («TABLE»))
   («VARS» («SUPER» '«COPY»))
   («CLASS»
     (lambda (MSG . ARGS)
       (case MSG
          ((new)
           (let*
             ((context («VARS» 'instantiate))
              (self (lambda (msg . args)
                      («CLASS» '«EVAL» context msg args))))
             (context 'replace '«SELF» self)
             self))
          ((«EVAL»)
           (let*
             ((context (car ARGS))
              (msg (cadr ARGS))
              (args (caddr ARGS))
              (entry («METHODS» 'get msg)))
             (if entry
                (apply entry (cons context args))
                («SUPER» '«EVAL» context msg args))))
          ((«COPY»)
           («VARS» 'copy))
          (else (error "invalid class message" MSG))))))
```

```
(define Counter
  (CLASS Root
         (VAR cou
         (METHOD

         (METHOD

         (METHOD

         (METHOD
    (VAR count 0)
    (METHOD incr ()
```

# Notes on inheritance

- We saw before: "Self" is dynamic
  - Only at runtime do you know what object will be sent a message
- Now we saw: "Super" is static
  - At compile time you know who will be called
- This enables Frameworks

# Frameworks

- Framework: application-independent implementation for a particular domain
- Applications instantiate the framework
  - fill in the application-specific parts
- Example: GUI Application Framework
  instantiated for text editor, UML editor, web browser, development environment, ...

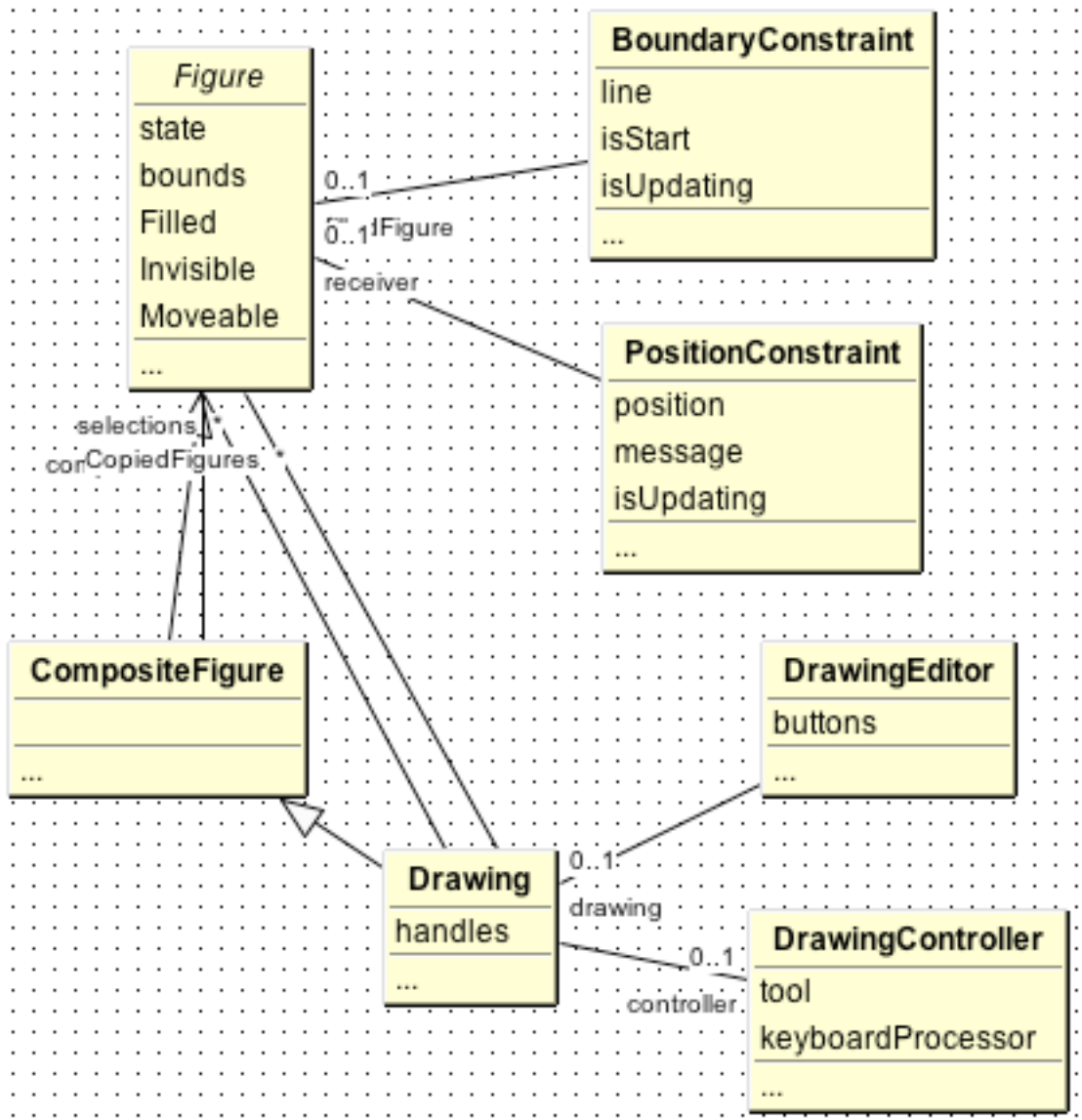
# Framework mechanisms

- Framework implementation mechanism:
  - Hook method: , the framework instantiator must overwrite this method if adding a subclass.
    - can be abstract, or can have default implementation
  - Template method: implements core behaviour, and calls hook methods

# Framework example

- HotDraw: graphical editor framework
  - Framework implements Figures, Tools, constraints, ...
  - One instantiation: state diagram editor: customizes with state figures, specific tools, ...
  - Other instantiation: class diagram editor: customizes with class figures, associations, ...
  - ...

# Part of HotDraw



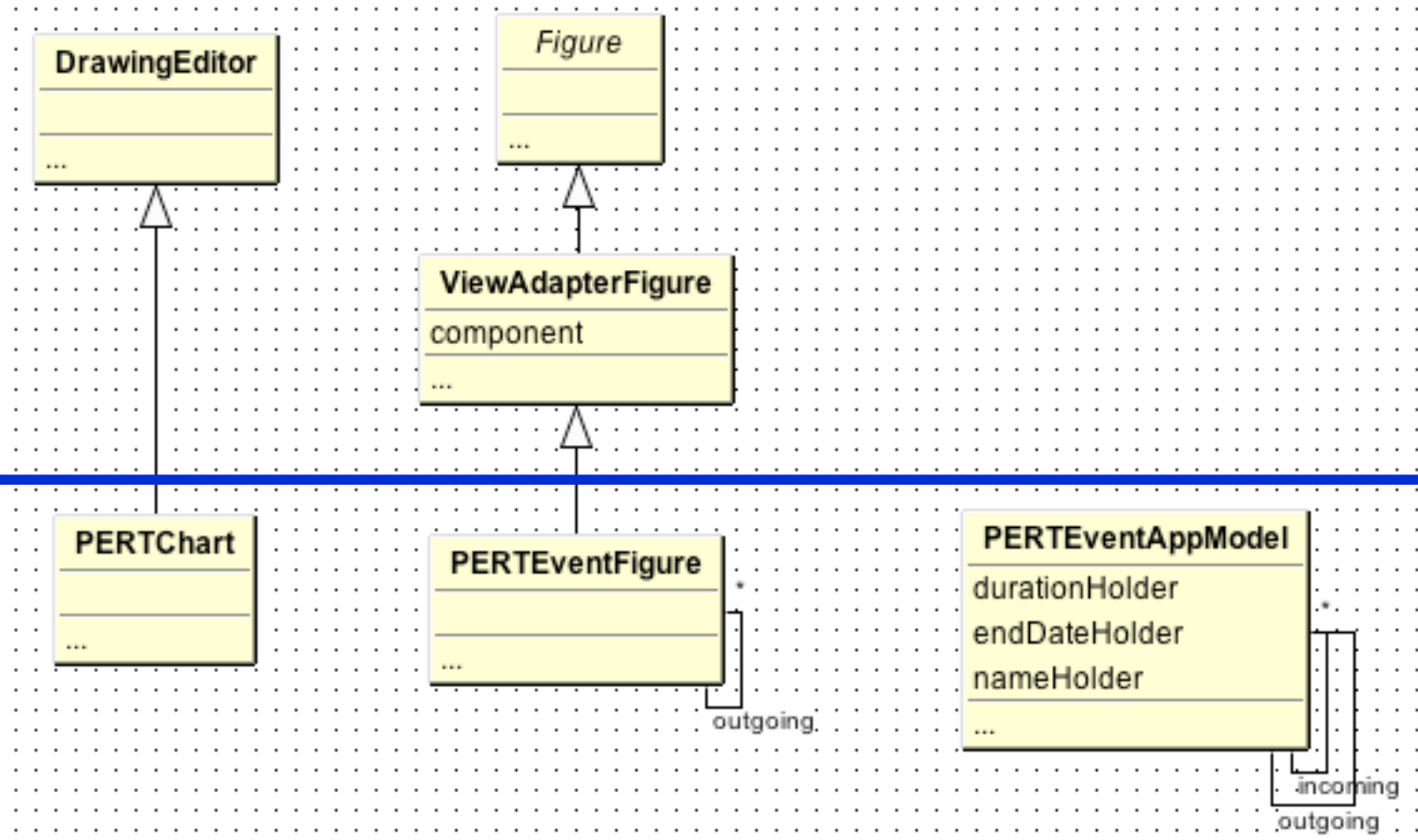
# Hook & Template example

```
Figure>>basicTranslateBy: aPoint
    "This method is called by #translateBy:. The #translateBy: method has
already moved our bounds. If we have other points that need to be moved, we
need to move them also."


Figure>>translateBy: aPoint
    "Move ourself by aPoint. Instead of overriding this method,
    subclasses will probably just need to override basicTranslateBy:"

    | oldBounds |
    self isMoveable ifFalse: [^self].
    aPoint = (0 @ 0) ifTrue: [^self].
    oldBounds := self preferredBounds.
    self translatePreferredBoundsBy: aPoint.
    self basicTranslateBy: aPoint.
    self changedPreferredBounds: oldBounds.
    self changed
```

# PERT Diagram Editor

HD

PERT



# Reuse

- White-box reuse: reuse where one has access to the internals of the reused entity
  - typically unanticipated
  - e.g. overriding framework method
- Black-box reuse: reuse of an entity of which you only have access to its interface
  - typically hooks are planned
  - e.g. "components", APIs, Netscape plug-in

# Overloading vs. Polymorphism

- Overloading: procedures/methods have the same name, but differ in number or types of arguments
  - have same return type
- Polymorphism: polymorphic methods have the same signature (name + types of arguments + return type), but are implemented on different classes
  - method overriding

# Methods vs. procedures

- In a message there is a designated receiver that accepts the message.
  - The interpretation of the message may differ depending upon the receiver
- A procedure has no such thing
  - Cannot use polymorphism
  - Overloading is no help

# Example

```
class Root
{
  public:
    virtual void t() { h(); };
    virtual void h() { /* Subclasses can override this */ };
};

class Sub
{
  public:
    virtual void h() { cout << "Something concrete." << endl; };
};
```

---

```
//ADT Root
procedure t(Root r) { h(r); }
procedure h(Root r) {}

//ADT Sub
procedure h ???
```

# Parametric Polymorphism

- C++ templates: compile-time parametric polymorphism
  - a.k.a. *parametric* or *generic* types
  - functional programming concept
    - function with argument of type <int>
    - function with argument of type list<int>
    - function with argument of type f(list<int>)
    - ...

# Stack example

```
template <class T>
class Stack
{
public:
   Stack(int = 10) ;
   ~Stack() { delete [] stackPtr ; }
   int push(const T&);
   int pop(T&) ;  // pop an element off the stack
   int isEmpty()const { return top == -1 ; }
   int isFull() const { return top == size - 1 ; }
private:
   int size ;  // Number of elements on Stack
   int top ;
   T* stackPtr ;
} ;
```

# Some Stack functions

```
//constructor with the default size 10
template <class T>
Stack<T>::Stack(int s)
{
        size = s > 0 && s < 1000 ? s : 10 ;
        top = -1 ;  // initialize stack
        stackPtr = new T[size] ;
}

// push an element onto the Stack
template <class T>
int Stack<T>::push(const T& item)
{
        if (!isFull())
        {
                stackPtr[++top] = item ;
                return 1 ;  // push successful
        }
        return 0 ;  // push unsuccessful
}

...
```

# Using the stack

```
//creating and using two stacks
typedef Stack<float> FloatStack ;
typedef Stack<int> IntStack ;

FloatStack fs(5) ;
float f = 1.1 ;
cout << "Pushing elements onto fs" << endl ;
while (fs.push(f)) {
              cout << f << ' ' ;
              f += 1.1 ; }
cout << endl << "Stack Full." << endl << endl << "Popping elements" << endl ;
while (fs.pop(f))
              cout << f << ' ' ;
cout << endl << "Stack Empty" << endl ;

IntStack is ;
int i = 1.1 ;
cout << "Pushing elements onto is" << endl ;
while (is.push(i)) {
              cout << i << ' ' ;
              i += 1 ; }
     cout << endl << "Stack Full" << endl ;
```

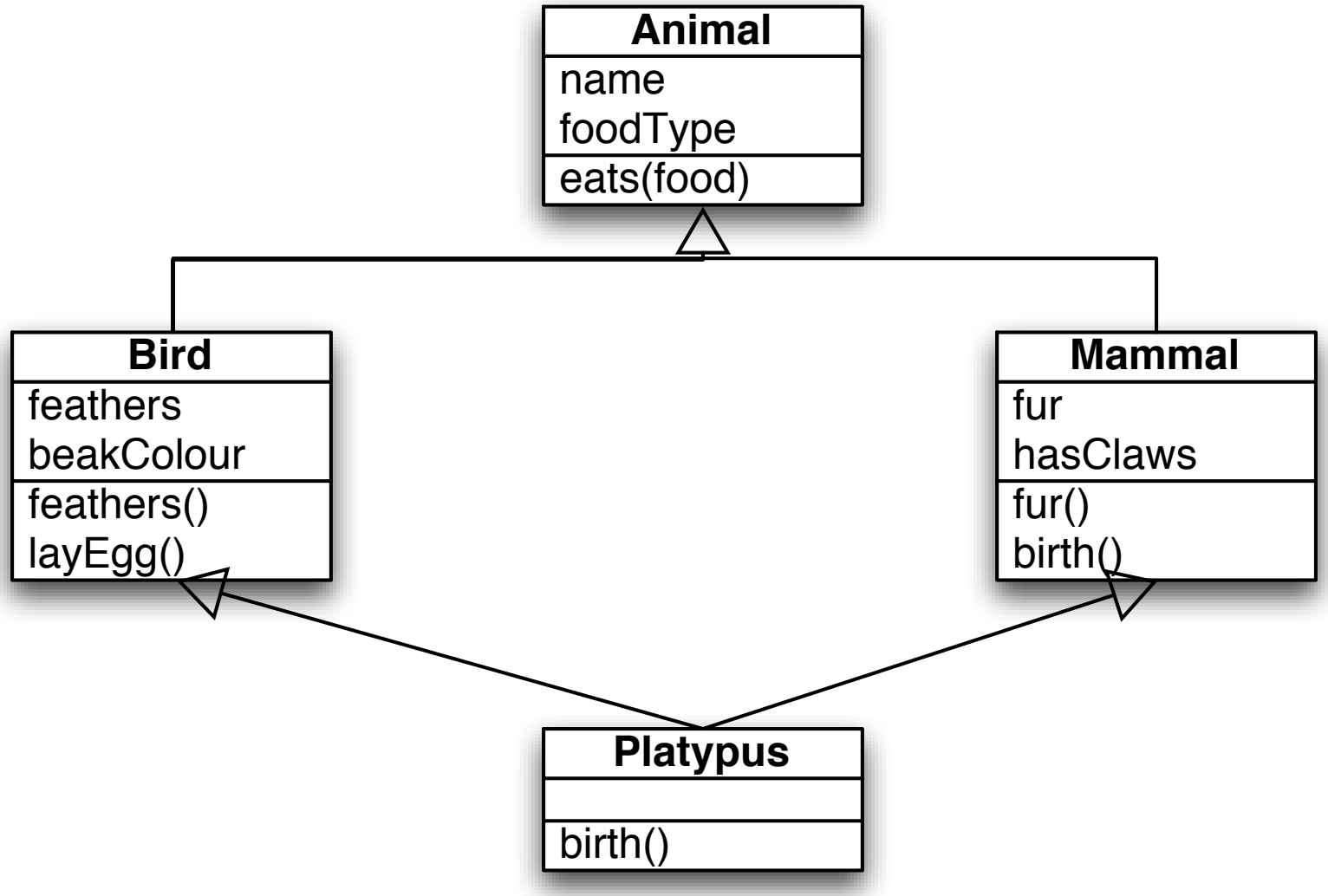
# Templates

- copy is created at compile time for each different type that is required
  - code gets "expanded"
  - not space efficient
- time efficient

# Multiple Inheritance

- Sometimes it is convenient to have a class which has multiple parents.
- Subclasses inherit instance variables and methods from all parents.
- Conflicts can occur:
  - methods that are inherited from multiple classes
  - instance variables inherited from multiple classes

# Multiple Inheritance Example



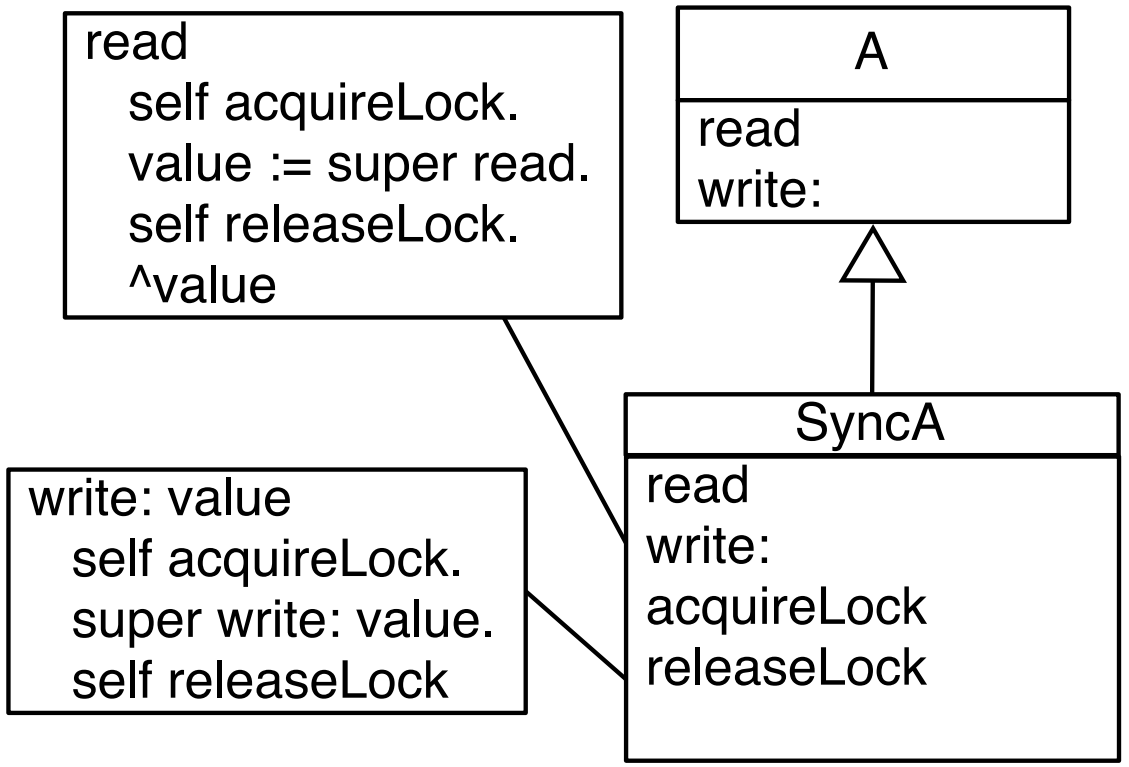

- Any Comments?

# Multiple Inheritance Problem

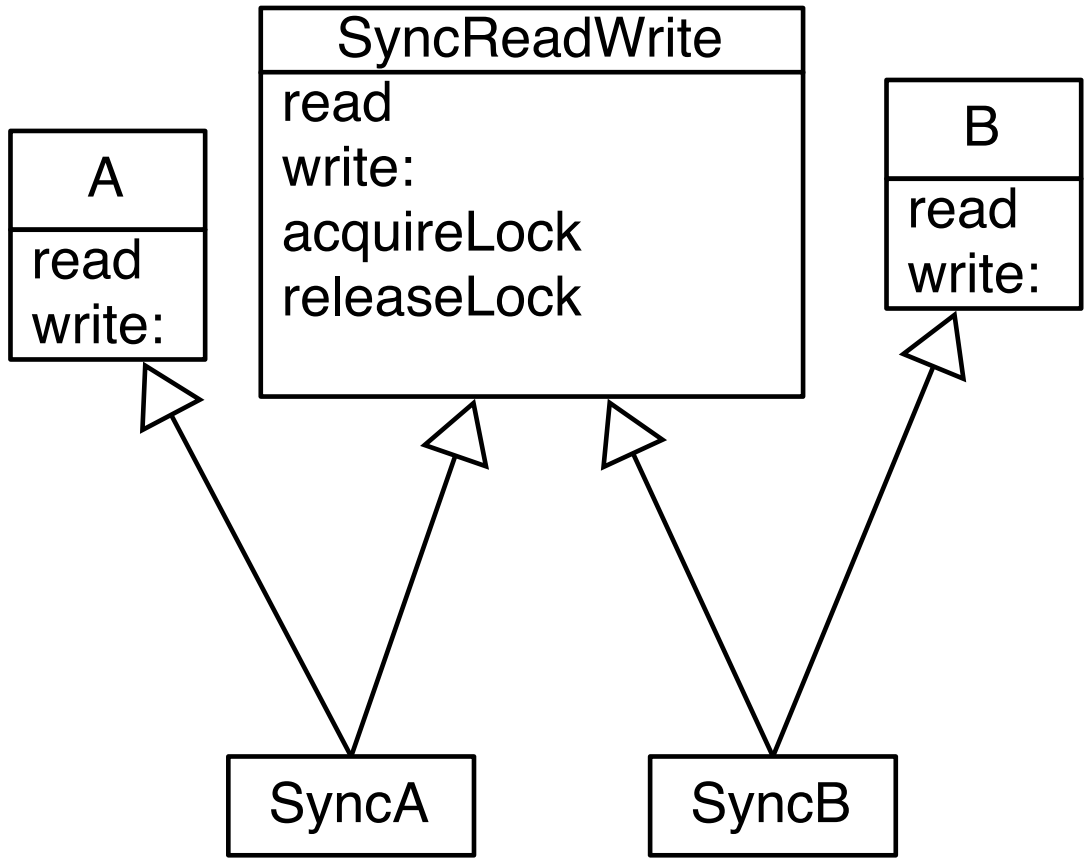
- Inheriting state and methods via multiple paths
  - duplicated methods
    - solve with overriding
  - duplicated state
    - harder to solve

# Yet another problem

- Suppose two classes: A and a synchronised A:



# Yet another problem

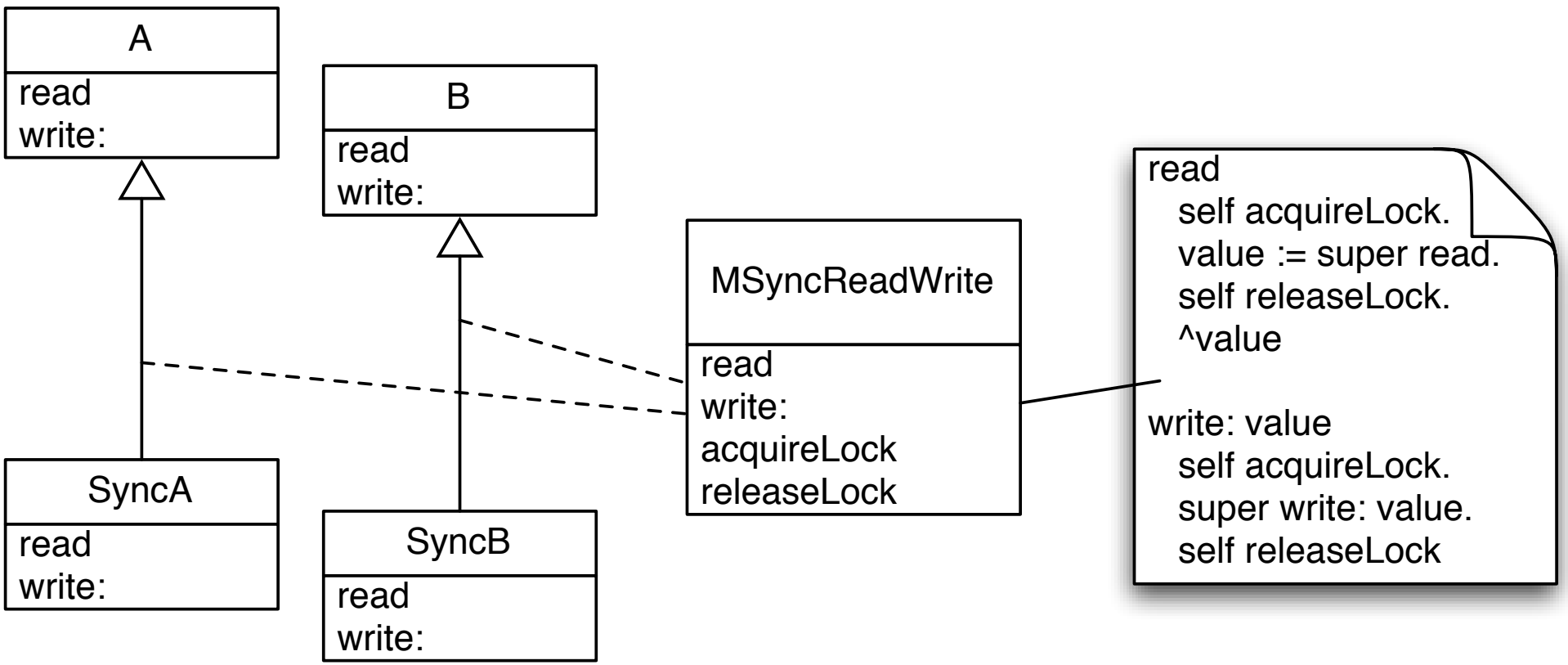



- Does not work. Why? Solution?

# Mixin Inheritance

- Factoring out the increment when subclassing.
- Result: class parametrized by its superclass.
- Put differently: operation that takes as input a class and produces another class
- How does it work?
  - mixin composition is typically linearized
    - so uses regular inheritance

# Mixins Example



# Ad-hoc mixins in C++

```
class Car
{
   public:
      virtual void honk() { cout << "Ran honk on Car\n"; };
      virtual void size() { cout << "Ran size on Car\n"; };
};


class Book
{
   public:
      virtual void numberOfPages() { cout << "Ran numberOfPages on Book\n"; };
      virtual void size() { cout << "Ran size on Book\n"; };
};
```

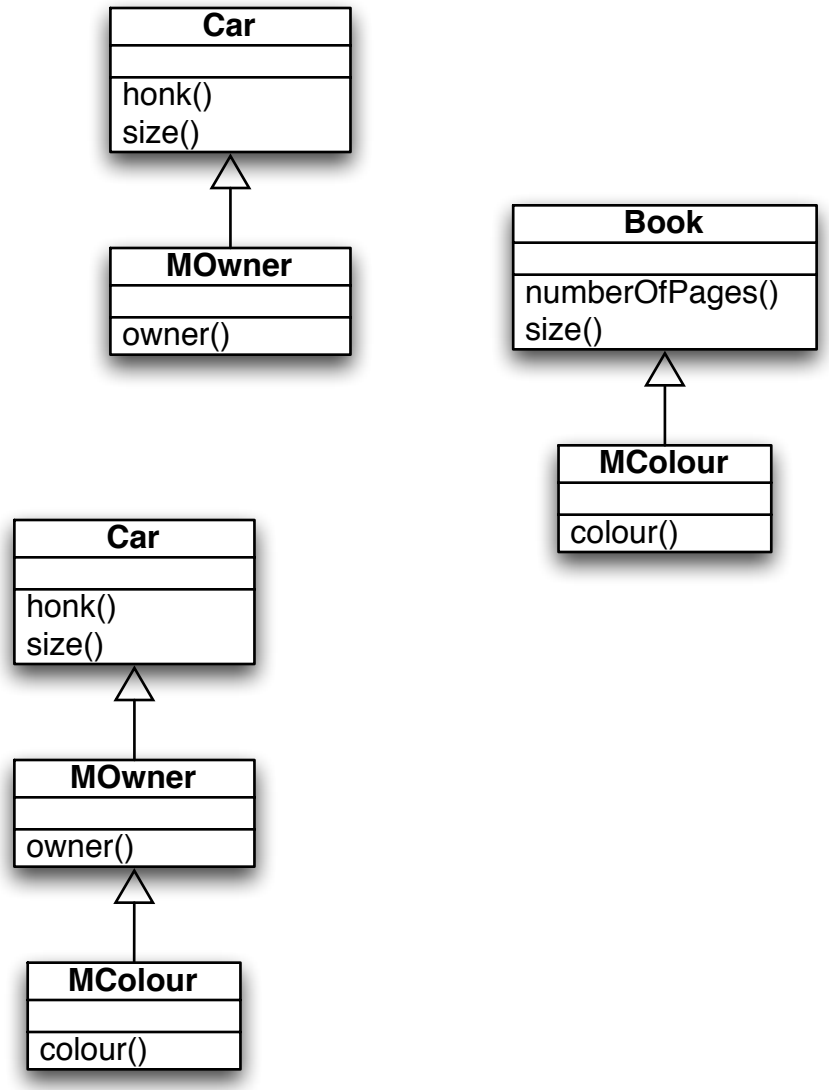
# Mixins

```
//Mixin: Colour
template <class T>
class MColour: public T
{
public:
        void colour() { cout << "Ran colour on MColour\n"; };
} ;


//Mixin: Owner
template <class T>
class MOwner: public T
{
public:
        void owner() { cout << "Ran owner on MOwner\n"; };
} ;
```

parametrized superclass

# Composing Classes and Mixins

```
typedef MOwner<Car> OwnedCar ;
typedef MColour<Book> ColouredBook;
typedef MColour<OwnedCar> ColouredOwnedCar;

OwnedCar roelsCar ;
ColouredBook blackBook;
ColouredOwnedCar blackStephaneCar ;

roelsCar.honk();
blackBook.colour();
blackStephaneCar.colour();
blackStephaneCar.honk();
```



# Composing mixins

```
//Mixin: ColouredOwner
template <class T>
class MColouredOwner: MOwner<T>, MColour<T>
{
public:
   void colour() { cout << "Ran colour on MColouredOwner - "; MColour<T>::colour();
};
   void owner() { cout << "Ran owner on MColouredOwner - "; MOwner<T>::owner(); };
} ;
```

super send in C++
(no keyword!)

# But...

```
//Mixin: Owner
template <class T>
class MOwner: public T
{
public:
      void owner() { cout << "Ran owner on MOwner\n"; };
      void address() {cout << "Ran address on MOwner\n"; };
} ;

typedef MColouredOwner<Book> ColouredOwnedBook;
ColouredOwnedBook redRoelBook;
redRoelBook.address();
```

does not work!

why?

```
typedef MColour<OwnedCar> ColouredOwnedCar;
ColouredOwnedCar blackStephaneCar;
blackStephaneCar.address();
```

works!

# Problems with Mixins

- Linearization causes problems
  - fragile
  - introducing extra mixin can change behaviour
  - composition order is important

# Wrap-up

- Self is dynamic & Super is static
  - and why!
  - reuse results
- Multiple Inheritance
  - the good, the bad & the ugly
- Mixins

# References

- http://www.ulb.ac.be/di/rwuyts/INFO020_2003/