Vrije Universiteit Brussel
Faculteit Wetenschappen

A lens on the encapsulation operator

Marc Van Limberghen & Tom Mens

Techreport vub-prog-tr-96-14

Programming Technology Lab
PROG(WE)
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3525
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

A lens on the encapsulation operator

Tom Mens & Marc Van Limberghen

Programming Technology Lab
Department of Computer Science
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussel - BELGIUM

Abstract. In [VanLimberghen&Mens96] we showed how an orthogonal combination of two operators, inheritance and encapsulation, provided a solution for multiple inheritance problems in class-based systems. In this document we elaborate further upon this idea in two ways. First we present a small prototype-based language called LENS incorporating the two operators. Secondly we focus on the encapsulation operator, and argument a further decomposition of this operator in two more primitive operators, namely preventInvocation and preventSpecialisation.

1. LENS, a simple but full fledged OO language

The name LENS is an abbreviation of Language with Encapsulated Name Spaces. Indeed, a LENS program consists of nested name spaces that can be encapsulated and refined dynamically. Name spaces are first class: they represent objects created ex nihilo.

To illustrate the syntax of LENS, consider the program of Figure 1. All identifiers in bold are reserved words. The Robot method (line 5) creates a robot with some move operations. The StepRobot method (line 19) takes a robot as argument and delivers a specialised view on it by decomposing its move operations into smaller steps.

In LENS, the following message passing notation is adopted:

- Messages with an explicit receiver are denoted by writing the receiver followed by the message-selector. Arguments can be passed by putting them behind the selector between parentheses. E.g., on line 42, *myRobotStepByStep move(2,4)* sends the message *move* to *myRobotStepByStep* with arguments *2* and *4*.
- Receiverless messages represent self sends, e.g. *moveHorizontal(dx)* on line 9 means *self moveHorizontal (dx)* and on line 12 *x* means *self x.*
- Super calls are denoted with the ***super*** keyword. We use super calls only to refer to the method we are currently overriding. Consequently it is redundant to specify which super method is activated. In other words when for example overriding the *moveHorizontal* method, we will simply write ***super****(d)* to call the super variant of the *moveHorizontal* method (line 28).

LENS provides different kinds of control structures. For example, a conditional ***if*** statement is given on line 32. It expects two arguments. The first argument is evaluated if the receiver evaluates to true, otherwise the second argument is evaluated.

LENS provides four kinds of slot declarations:

- *move(dx,dy)* ***method****(body)* on line 8 declares a method slot with label *move* and the two parameters *dx* and *dy*. *body* is evaluated each time the move message is sent.
- *x* ***variable****(initialx)* on line 6 declares a variable with initial value *initialx*. This declaration introduces two slots: a retrieve slot *x* and an update slot *x!*. Slot-based instance variables blend state and behaviour, even towards inheritors, enhancing the degree of encapsulation [Snyder87] [Ungar&Smith87].
- *screen* ***constant****(value)* on line 2 declares a read-only instance variable by only introducing the retrieve slot *screen.*
- The ***temporary*** declarations (lines 40 and 41) introduce a local variable in the surrounding method *main*.

```
01    [
02      screen constant(
03          print(text,value) method([text print;value println]));
04
05      Robot(initialx, initialy) method(
06        [ x variable(initialx);
07          y variable(initialy);
08          move(dx,dy) method(
09            [ moveHorizontal(dx);
10              moveVertical(dy)]);
11          moveHorizontal(d) method(
12            [ x!(x + d);
13              screen print(" my new x is ",x)]);
14          moveVertical(d) method(
15            [ y!(y + d);
16              screen print(" my new y is ",y)]);
17        ] encaps(x!,y!));
18
19      StepRobot(robot, horizontalStep, verticalStep) method(
20        [ robot;
21          [ x variable(horizontalStep);
22            y variable(verticalStep);
23            scaleSteps(factor) method(
24              [ x!(x * factor);
25                y!(y * factor)]);
26            moveHorizontal(d) method(
27              (d < x) if(
28                super(d),
29                [ super(x);
30                  moveHorizontal(d - x)]));
31            moveVertical(d) method(
32              (d < y) if(
33                super(d),
34                [ super(y);
35                  moveVertical(d - y)]))
36          ] encapsExcept(moveHorizontal, moveVertical, scaleSteps)
37        ]);
38
39      main method(
40        [ myRobot temporary(Robot(0,0));
41          myRobotStepByStep temporary(StepRobot(myRobot, 0.5, 0.5));
42          myRobotStepByStep move(2,4)])
43
44    ] main
```

2. Orthogonal Concepts of LENS

2.1. Dynamic Mixin-Based Inheritance

Inheritance is denoted by the ';' operator, and can be applied dynamically: inheritance hierarchies can be built at run time. In line 20 for example the *StepRobot* method specialises its *robot* argument by means of inheritance. Figure 2 presents a diagram of this situation. An apparent characteristic of LENS is life-time sharing between child-objects and their common parent object, a specific property of prototype-based languages [Dony&al92]. But in all prototype-based languages that we know of, cloning of objects belongs to the primitive object creation mechanism. In LENS on the contrary, we deliberately omitted the cloning of prototypes from the primitive object creation mechanism.

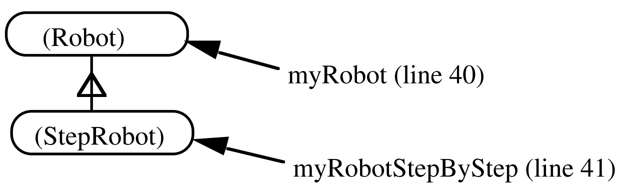


Figure 2: a robot object and its specialisation

The *Robot* method itself constructs a robot object ex nihilo by composing different atomic objects, only consisting of a method or variable. Such incomplete (or abstract) objects can be passed around. Incorrect use of incomplete objects raises a runtime "message not understood" error. We do not consider this as a problem because we feel it is the task of an optional static type system[1] to avoid such wrong uses at compile time.

The *StepRobot* specialisation code (lines 21 to 36) can also be attached to other kinds (classes) of robots: its super parameter (in this case *robot* on line 20) is filled in by the inheritance operator. Therefore the inheritance of LENS corresponds to mixin-based inheritance [Bracha&Cook90]. Whereas traditional class-based inheritance is a troika of code-reuse, classification and encapsulation, mixin-based inheritance extracts the code-reuse aspect. Mixins are chunks of code that can be freely combined by inheritance. Therefore mixin-based inheritance is sometimes criticised to be a mere code-sharing mechanism without conceptual meaning. Mixin-based inheritance should however be seen as a basis that can be *orthogonally* extended with suitable software engineering tools that reinforce reliability.

2.2. Encapsulation

In line 17, the ***encaps*** operator hides the slots *x!* and *y!* towards further clients (inheritors as well as message senders). This way clients are unable to directly alter the position of a robot. ***encaps*** corresponds to the *hide* operator of [Bracha&Lindstrom92]. ***encapsExcept***, as in line 36, is its complement: it hides all the slots except the specified ones. Note that the *x* slot of the *StepRobot* code (declared in line 21), representing the horizontal step size, does not collide with the *x* slot of its parent code (declared in line 6), representing the abscissa of the robot. Indeed, *myRobotStepByStep* (line 41) still responds to *x* by returning its abscissa. This illustrates that subclass and parent class can be developed more independently thanks to this kind of encapsulation.

Nesting gives rise to a kind of closure: a robot for instance will implicitly retain a reference to its surrounding context. This way a robot can reference to the global constant *screen*.

Note that the ***constant*** and ***temporary*** declarations are not essential since they can be expressed using ***encaps***:

`[ ...;  x temporary(1); ...]` and `y constant(1)`

are respectively the shorthand notations for

`[ ...;  x variable(1); ...] encaps(x)` and `[y variable(1)] encaps(y!)`

2.3. Nesting

A LENS program can contain nested name spaces, giving rise to a kind of closure. Nested scoping is lexical in LENS, except for identifiers denoting overridable methods. The fact that nested methods are overridable makes LENS ideal for constructing specialisable frameworks [VanLimberghen96].

Most other object-oriented languages, like Smalltalk and Eiffel, do not offer nesting. In C++, classes can be nested, but the nested scoping of identifiers is not totally satisfactory [VanLimberghen96]. Beta [Madsen93] is a class-based language with nested scoping. Like in LENS, the Beta nested scoping is not lexical for identifiers denoting overridable methods. However, a nested class can only be accessed by sending a message to an *instance of* the surrounding class. This necessitates classes to be first-class values.

2.4. Self sends

In traditional object-oriented languages self sends are too firmly connected to the notion of object identity. Most object-oriented languages provide a self pseudo-variable that serves two purposes: invokimg methods through self sends and returning a self reference. Unfortunately, this double functionality is incompatible with object-based encapsulation, as extensively discussed in [Mens&VanLimberghen95]. For this reason, the inheritance mechanism in LENS is detached from the identity of the self object. Receiverless self sends are used instead of self references. For more details about the self send mechanism, we refer to [Mens&VanLimberghen95].

[1] Theoretical foundations for typing dynamic mixin-based inheritance can be found in [Lucas&al95]

3. Encapsulation revisited

3.1. preventInvocation and preventSpecialisation

The encapsulation mechanism used so far always hides an attribute from inheriting and sending clients simultaneously. Sometimes it can be useful to split the encapsulation operator into two more primitive orthogonal operators, called ***preventInvocation*** and ***preventSpecialisation*** (and the complementary actions ***preventInvocationExcept*** and ***preventSpecialisationExcept***). The ***preventInvocation*** operator prevents message senders to invoke an attribute, but an inheritor is still allowed to specialise it. The ***preventSpecialisation*** operator prevents inheritors to specialise an attribute of an object, while it can still be invoked. The encapsulation operator ***encaps*** is the combination of the more primitive operations ***preventInvocation*** and ***preventSpecialisation***.

The following example serves to explain more exactly the behaviour resulting from the use of these encapsulation operators. It illustrates that a method to which ***preventSpecialisation*** is applied can no longer be specialised, and a method to which ***preventInvocation*** is applied cannot be invoked anymore from the outside.

```
[
  SwedishCook method(
    [ bork method("I bork" print);
      prepareDinner  method(bork)
    ] preventInvocation(bork)
      preventSpecialisation(prepareDinner));

  "bork is present in the specialisation interface,
   but not in the sending interface" comment;
  "prepareDinner is present in the sending interface,
   but not in the specialisation interface" comment;

  DanishCook method(
    [ SwedishCook;
      bork method(
        [ super;
          " in Danish" println ])
    ]);

  tryToInvokeBork method(
    [ bork method("A cook only borks while preparing dinner" println);
      SwedishCook;
      bork method([ super ]);
      invokeBork method([ bork ])
    ]);

  tryToSpecialisePrepareDinner method(
    [ prepareDinner method("One can not specialise prepareDinner" print);
      SwedishCook;
      prepareDinner method(
        [ super;
          " as I tried to do" println ])
    ]);

  main method(
    [ DanishCook prepareDinner;
    " 'I bork in Danish' is printed" comment;
      tryToInvokeBork invokeBork;
    " 'A cook only borks while preparing dinner' is printed" comment;
      tryToSpecialisePrepareDinner prepareDinner;
    " 'One can not specialise prepareDinner as I tried to do' is printed" comment
    ])

] main
```

Note that

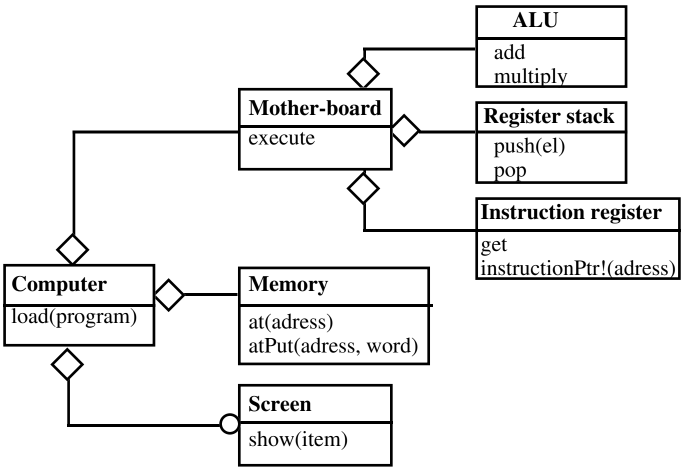
- *prepareDinner* is really hidden from specialisers. Specialisers will specialise the version of the super-class, if present. If no other version of *prepareDinner* exists, a "message not understood" runtime error will occur.
- Similarly, *bork* is not invokable from the outside. Inheritors can specialise it, but are not allowed to invoke it.

If we prevent specialisation of all attributes of an object with ***preventSpecialisationExcept***, the object becomes non-specialisable. Note that such an object can still be used in combination with the ';' operator. For instance, the expression

```
[ object1 preventSpecialisationExcept;
  object2 preventSpecialisationExcept;
  object3 preventSpecialisationExcept;
  object4 preventSpecialisationExcept
]
```

is a direct implementation of the ChainOfResponsability design pattern [Gamma&al94]: a request is passed along the chain (object4 to object1) until an object handles it. There is no late binding between the objects.

The following computer simulation program is a more realistic example of ***preventInvocation***: the LENS program simulates the following computer diagram in OMT:




```
[
Computer method(
 [
  motherBoard constant(
    [ alu constant(
        [ add method(registers push((registers pop) + (registers pop)));
          multiply method(registers push((registers pop) * (registers pop)))
        ]);

      registers constant(
        [ .....
        ] encapsExcept(pop, push));

      instructionRegister constant(
        [ instructionPtr variable;
          get method([
            instructionPtr!(instructionPtr+1);
            memory at(instructionPtr)])
        ] encaps(instructionPtr));
          "instructionPtr! remains visible to implement jump" comment;
```

```
      executeInstruction(instruction) method(
              (instruction = pushConstant) if(
                 registers push(instructionRegister get),
              (instruction = valueToMemory) if(
                 [ holdAdress temporary(registers pop);
                   (holdAdress = videoAdress) if(
                      screen show(registers pop),
                      memory atPut(holdAdress, registers pop))],
              (instruction = add) if(
                 alu add,
              (instruction = multiply) if(
                 alu multiply,
                 "system error: illegal instruction" println)))));

      execute method(
        [ instructionRegister instructionPtr!(-1);
          instruction temporary(instructionRegister get);
          (instruction >= 0) while(
           [ executeInstruction(instruction);
             instruction!(instructionRegister get)
           ])
        ])
    ] preventInvocation(executeInstruction));

  memory constant(Dictionary);

  screen variable(show(value) method(nil));

  load(program) method(
    [ loadAdress temporary((program size) - 1);
      program doUntilFalse(execute(instruction) method(
         [ memory atPut(loadAdress, instruction);
           loadAdress!(loadAdress - 1);
           true]));
      motherBoard execute
    ])

 ] encaps(screen));
"Initially there is no screen, but a screen can be connected afterwards
 using the label screen!  " comment;


FloatComputer method(
  [ Computer;
    motherBoard constant(
      [ super;
        alu constant(
          [ super;
            addFloats method(registers push((registers pop) + (registers pop)));
            multiplyFloats method(
               registers push((registers pop) * (registers pop)))
          ]);

        executeInstruction(instruction) method(
              (instruction = addFloats) if(
                 alu addFloats,
              (instruction = multiplyFloats) if(
                 alu multiplyFloats,
                 super(instruction))))
        ])
  ]);


Dictionary method(
  [ ........
  ] encapsExcept(at, atPut));
```

```
"the machine code instructions are:" comment;
stop method(-1);
pushConstant method(0);
valueToMemory method(1);
add method(10);
multiply method(11);
addFloats method(12);
multiplyFloats method(13);
videoAdress method(2000);

main method(
  [ ...  ])

] main
```

In this example, *executeInstruction* is a procedural parameter of the motherboard: it can be specialised but it cannot be invoked from the outside. Indeed, this is what happens in *FloatComputer*: a specialised computer that deals with floating point numbers is created by inheriting from *Computer*, extending its *alu*, and specialising the *executeInstruction* method.

Note that the read-slot *screen* is encapsulated in *Computer* while the write-slot *screen!* is visible: it simulates a connector where a screen can be attached. This illustrates the usefulness of treating read- and write-accessor methods orthogonal to encapsulation. This is opposed to the approach taken in Omega [Blaschek94] where write-access to variables is considered as a kind of privilege that is superior to read-access: in Omega having write-access implies having read-access.

3.2. Implementation Issues

We do not yet have a mathematical definition for the operators ***preventSpecialisation*** and ***preventInvocation***. Given our actual object model the mathematical definitions are not straightforward due to our particular treatment of super calls. For example ***preventSpecialisation*** differs from non-virtual methods in C++ because non-virtual methods can still be specialised by simulating super calls through qualified message passing. This is illustrated in the following C++ program resulting in writing 100 instead of 200. We want to avoid such situations in our system, because we feel it does not match the idea of method specialisation. ***preventSpecialisation*** differs from the freeze operator in [Bracha&Lindstrom92] for similar reasons.

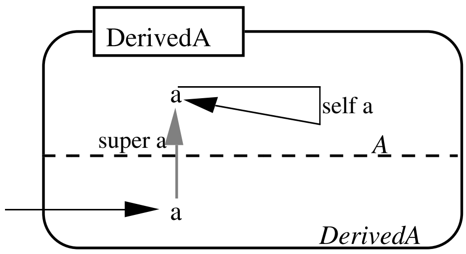
```
#include <iostream>

class A{
public:
int a(int x){if (x==0) {
                return 100;}
             else {
                return a(0);};
            };
};

class DerivedA : public A{
public:
int a(int x){if (x==0) {
                return 200;}
             else {
                return A::a(x);};
            };
};

void main()
{
   DerivedA* anObject = new DerivedA;
   cout << anObject->a(1) << endl;

}
```



Although ***preventSpecialisation*** and ***preventInvocation*** are not yet defined mathematically, we have proven with our implementation that a safe use of these operators can be implemented. However, the implementation of ***preventInvocation*** is runtime expensive. The reason for this is that we want to prevent inheritors from bypassing the invocation restriction by specialising the method and subsequently invoking the specialisation (as illustrated in our *SwedishCook* example). Preventing this behaviour is currently implemented by some shallow-copy operation. Maybe it would be more efficient to conceive ***preventInvocation*** as a kind of (inheritance-)type information. Inheritance should be obstructed when the extension code contains a self send to the slot that may not be invoked. This is possible by including information about self sends in the type system, as proposed in [Lucas&al95].

4. Conclusion

In [VanLimberghen&Mens96] we explained that the entire multiple inheritance dilemma can and should be seen as a conflict between inheritance and data encapsulation only. Orthogonalising these two concepts in a mixin-based framework provided an appropriate solution to multiple inheritance name collisions. A formal model was proposed, which can be seen as the foundation of the current LENS language. LENS is an object-oriented language in which the language features are as orthogonal as possible. The most important orthogonal language features are:

- dynamic mixin-based inheritance
- late bound object creation without need for cloning
- receiverless self sends and super calls at method level
- nested methods
- slot-based access to instance variables
- an encapsulation mechanism, that can be further decomposed in two more primitive operators, namely preventInvocation and preventSpecialisation.

In [VanLimberghen96] the orthogonal combination of these languages features was shown to be beneficial for object-oriented framework construction, and removed the need for several design patterns.

References

[Blaschek94] G. Blaschek: Object-Oriented Programming with Prototypes. ISBN 3-540-56469-1, 1994, Springer-Verlag.

[Bracha&Cook90] G. Bracha and W. Cook.: Mixin-based Inheritance, joint OOPSLA/ECOOP ’90 Conference Proceedings, ACM Press, pp. 303-311.

[Bracha&Lindstrom92] G. Bracha and G. Lindstrom: Modularity meets Inheritance, Proc. of International Conference on Computer Languages, 1992, IEEE Computer Society, pp. 282-290. Also available as Technical report UUCS-91-017.

[Dony&al92] C. Dony, J. Malenfant and P. Cointe: Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. Proceedings of OOPSLA ‘92, pp. 201-217, ACM Press.

[Gamma&al94] E. Gamma, R. Helm, R. Johnson and J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. ISBN 0-201-63361-2, 1994, Addison-Wesley.

[Lucas&al95] C. Lucas, K. Mens and P. Steyaert: Typing Dynamic Inheritance, a Trade-Off between Substitutability and Extensibility. Technical Report vub-prog-tr-95-03.

[Madsen93] O. L. Madsen, B. Møller-Pedersen and K. Nygaard: Object-Oriented Programming in the Beta Programming Language, Addison-Wesley, 1993.

[Mens&VanLimberghen95] T. Mens and M. Van Limberghen: Self sends as primary construct: a criterion for object-oriented language design. Technical Reports vub-prog-tr-95-08.

[Snyder87] A. Snyder.: Inheritance and the Development of Encapsulated Software Components, B. Shriver and P. Wegner, Research Directions in Object-Oriented Programming, MIT Press, pp. 165-188.

[Ungar&Smith87] D. Ungar and R. Smith: SELF: The power of simplicity, OOPSLA '87 Conference Proceedings, ACM Press, pp. 227-242.

[VanLimberghen96] M. Van Limberghen: Building frameworks through specialisable nested objects. Proceedings of TOOLS USA, 1996.

[VanLimberghen&Mens96] M. Van Limberghen and T. Mens: Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems. OOS Journal, Chapmann & Hall, March 1996.