Vrije Universiteit Brussel
Faculteit Wetenschappen

Building frameworks through specialisable nested objects.

Marc Van Limberghen

Techreport vub-prog-tr-96-08

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

Building frameworks through specialisable nested objects.

Marc Van Limberghen
Department of Computer Science
Faculty of Sciences
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, BELGIUM
E-mail: mvlimber@vnet3.vub.ac.be

Abstract

If a framework that internally creates objects is to be specialisable, then not only procedure calling must be subject to late-binding, but also object creation. In OO languages with compile-time classes, complicated extra code is needed that only serves to bypass the static character of class instantiation. The programmer has to maintain this code during framework specialisation. In languages with first-class classes or with prototypes, a structure of classes respectively prototypes has to be constructed and maintained during specialisation, an equally complicated task.
This paper proposes specialisable nested objects to deal directly with late bound object creation in frameworks. Nesting is exploited between a framework and the objects that belong to it. Object-based encapsulation is provided to develop the framework and its specialisations more independently.

1 Introduction

A framework is a family of collaborating classes, of which some may be abstract. A specialised version of the framework may make some of the abstract classes concrete or may refine or extend classes [Steyaert 96]. Amongst other things, the class collaboration can consist of the instantiation[1] of framework classes from within the framework itself. The instantiating code must transparently instantiate specialised versions of the classes. This software prerequisite has been called the Factory Method design pattern [Gamma 94]. In order to be able to perform the instantiations, the classes of the framework need to refer to each other somehow. In class-based languages with compile time classes, as C++ [Stroustrup 91] or Eiffel [Meyer 88], the programmer has to explicitly implement this reference structure and reconstruct it for each framework specialisation, cluttering the code that represents the essential framework behaviour.

Also external clients must be able to transparently instantiate specialised versions of a framework, a prerequisite that corresponds to the Abstract Factory design pattern [Gamma 94]. Usually this is implemented by providing the client with an object of some artificial class containing a create method for each framework class. The programmer has to build such an artificial class for each framework specialisation.

If the classes of the framework are instantiated from within the framework as well as by an external client, then it is required to combine the implementations of the Factory Method and Abstract Factory design patterns, making things even more complicated.

First-class classes, as in Smalltalk [Lalonde 90], or clonable prototypes, as in Self [Ungar 87], do not

[1] A concrete version of an abstract class is sometimes called an "instantiation" of that abstract class. But in this paper we will reserve the term instantiation for the creation of an object corresponding to a class.

alleviate the problems: In order to allow different specialised versions of a framework to coexist, an entire new set of interconnected classes (or prototypes) has to be constructed for each framework specialisation.

So, Factory Methods and Abstract Factories tend to require complicated implementations in current OO languages. Nevertheless these design patterns express an essential OO idea: late (binding of) object creation. It seems rather natural for an OO framework to comply with both design patterns. Therefore object creation should be subject to the same late-binding mechanism as method invocation, with the known advantages of polymorphism and genericity as a consequence.

This paper offers a thorough discussion of object creation problems in frameworks, and provides a solution to these problems under the form of "specialisable nested objects". This language concept is presented as a characteristic of an experimental OO language called LENS (Late-bound Encapsulated Name Spaces). A LENS program consists of nested name spaces that can be encapsulated and refined dynamically. Late object creation is obtained by directly coupling object creation to method invocation. Object creation methods are put in a name space representing the framework. Nested scoping allows an object belonging to the framework to easily create any other kind of framework objects. The framework is not a module or library, but an object. This allows different versions of the framework to coexist. Encapsulation of names is provided to hide implementation details of the framework.

Problems with object creation in frameworks have been addressed before in [Kiczales 93] and in [Riehle 95]. Section 5 contrasts LENS with these approaches, and also discusses nesting in other OO languages, especially in the OO language Beta [Madsen 93].

The rest of this paper is organised as follows. Section 2 describes an example of a framework complying simultaneously with the Abstract Factory and Factory Method design patterns, namely the abstract grammar of a programming language. Section 3 thoroughly discusses the problems when this framework is implemented in current OO paradigms. Section 4 explains the basic concepts of the OO language LENS and shows how the framework implementation problems are solved in it. Section 5 discusses related work in detail, section 6 presents future perspectives and section 7 concludes.

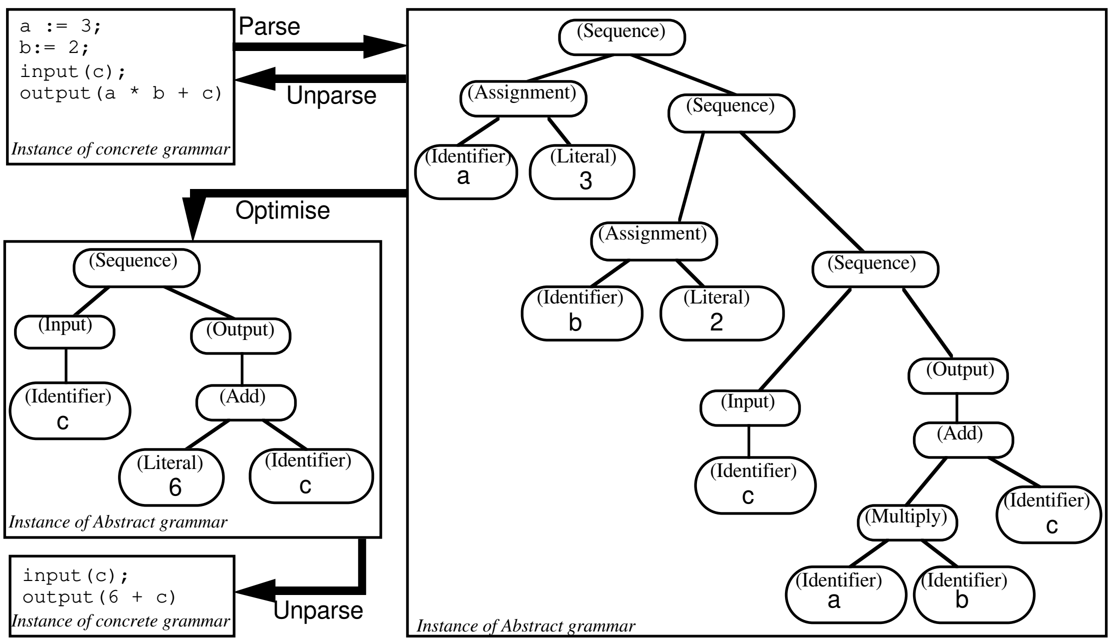

Figure 1: Implementing a programming environment by means of an abstract grammar

2 A Framework example: abstract grammar

This section presents a programming environment as an example framework that is used throughout the paper. In the implementation of a programming environment, it is common to represent a program (an instance of the concrete grammar) internally by a tree of nodes. This accords to the Interpreter design pattern [Gamma 94]. The internal representation, often called abstract grammar, is easy to interpret, compile, type check, optimise, etc., independently of the syntactical peculiarities of the concrete grammar. Figure 1 shows a program and its internal representation generated by the parser. The unparser and the optimiser perform actions on instances of the abstract grammar. The optimiser detects which variables have a constant value and generates a more efficient internal representation.

In an OO approach the abstract grammar can be implemented by representing each kind of node by a class. Each node-class will provide a method for each of the necessary activities: unparsing, optimising, evaluating, etc. The node classes of the abstract grammar together constitute a framework that complies with the Factory Method design pattern. Indeed, the optimising methods generate a new instance of the abstract grammar, i.e. they instantiate the node-classes of the abstract grammar. Our abstract grammar also complies with the Abstract Factory Method design pattern. The external client that instantiates the node-classes is the parser. It should be possible to use the same parser code for every abstract grammar specialisation: a variant of the defined unparsing or optimisation, or even an extra facility, like program execution.

Figure 2 shows an OMT-like scheme representing a basic abstract grammar framework offering optimisation, and a framework specialisation that also offers unparsing. Besides the object creation issue, also an encapsulation issue is manifested in our example: in Figure 2, only the items represented with thick lines should be visible outside their surrounding rectangle. The optimiser and the unparser both use their own implementation details to implement their behaviour. The optimiser needs to access some *optimiserTable* that holds the constant values. It also uses its own intermediate classes. The class *RuntimeExpression* for instance is merely an implementation aspect of the optimiser process. The unparsing process, depicted in Figure 2 in

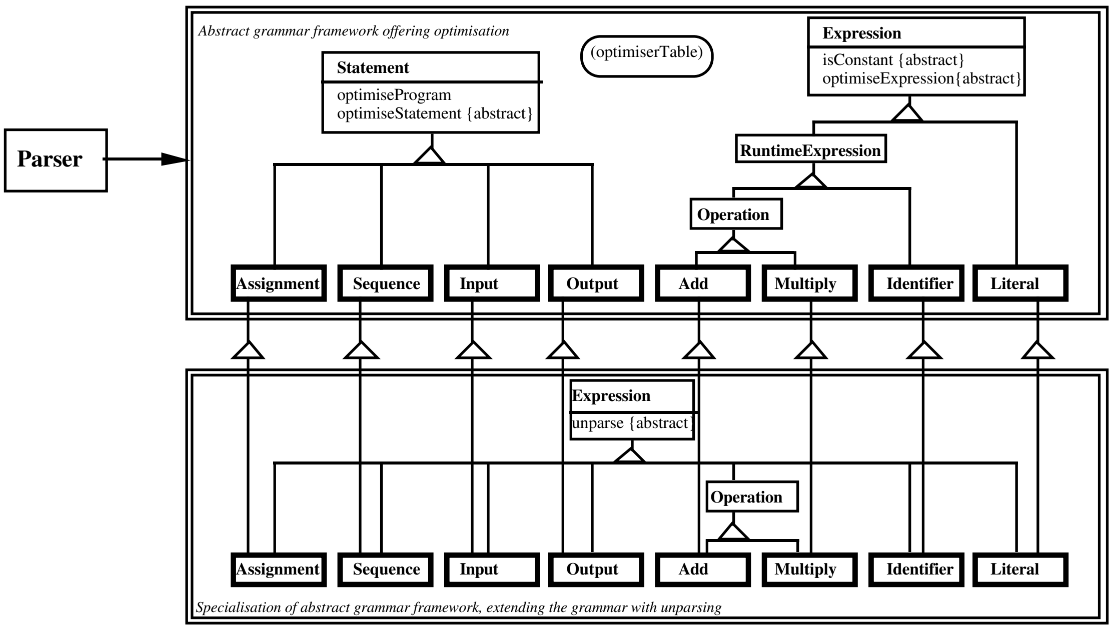

Figure 2: OMT-like scheme of the abstract grammars

the bottom rectangle, has its own, different, private inheritance structure. Such implementation details should be hidden from framework users and also from framework specialisers.

Different research papers indicated the need to separate typing and code reuse in OO languages (e.g. [Canning 89]). Therefore we consider inheritance as a pure code-reuse mechanism. That 's why the entire optimiser inheritance tree, in the upper rectangle of Figure 2, is hidden. If type-information is desired, then the framework should declare some (visible) statement-type.

3 Framework implementation problems

This section shows why current OO languages are inadequate to build the abstract grammar framework. The first part elaborates upon problems of late object creation, the second part upon the visibility issue. It should be clear that the discussion is not particularly oriented to abstract grammars, but applies to the framework problems in general.

3.1 Object creation problems

This section discusses the problems one encounters when implementing simultaneously the Abstract Factory and Factory Method design patterns in a framework. In order to implement the Factory Method aspect, some classes of the framework need to know (i.e. need to refer to) some other classes, namely those they want to instantiate. But while implementing the unparser specialisation of our abstract grammar example of Figure 2, the programmer does not (want to) know which internal instantiations are performed by the optimiser. So, the specialisation has to foresee a *total* cross reference between the node-classes, in order to make the implementation of the specialisation (the unparser) independent of the implementation of the original (the optimiser). Thus each specialisation needs its own cross-reference. The following two subsections reveal the problems when trying to achieve these cross-references. The first subsection acts on languages with compile time classes and the second on languages with prototypes or first-class classes.

3.1.1 Object creation problems in a language with compile time classes

[Gamma 94] describes separately how to implement in C++ the Abstract Factory and Factory Method design patterns. In the case of an abstract grammar framework, we have to combine both implementation strategies, and construct a total cross reference for the Factory Method (for the reason mentioned earlier). Figure 3 shows this combination. The grey zones implement the cross-reference between node-classes and should be private. Note that only the methods that serve to implement the design patterns are present: the real behaviour of the abstract grammar itself is even not drawn! The slightest specialisation, for instance only unparsing assignments in a different way, requires a total new cross reference to be constructed.

In C++, it is impossible to directly inherit the create methods from a common super class due to multiple inheritance problems: each create method will be inherited via different paths making it still necessary to specify the correct one. The same problem occurs in almost all existing class-based languages since usually (even in the *rename* clause of Eiffel) the class name is used somehow to disambiguate between multiple inherited homonymous attributes [Carré 90] [Van Limberghen 96].

Can we at least avoid the ponderous code duplication in the grey zones? The node-classes can directly refer to their Abstract Factory class, but they unfortunately need an *instance of* their Abstract Factory class instead of that class itself.
We present the different alternatives to obtain an instance of the Abstract Factory:

- Creating a new Abstract Factory object, i.e. instantiating the (known) Abstract Factory class, each time when a node is instantiated. This causes a very expensive runtime overhead.
- Putting an instance of the Abstract Factory class in a global variable. The framework

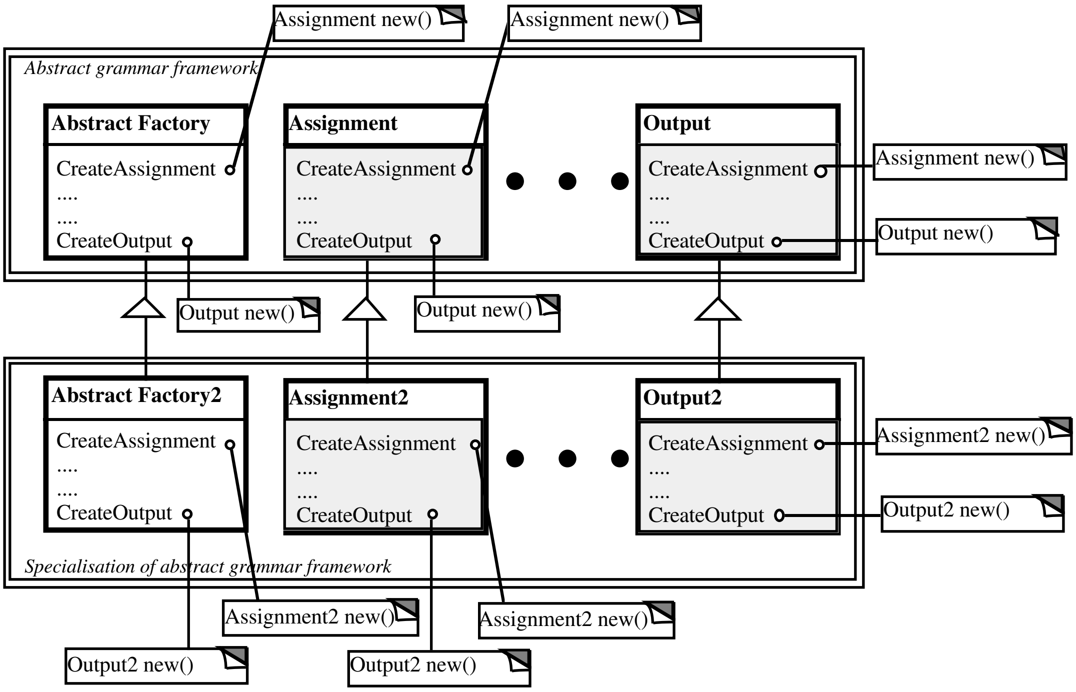

Figure 3: The abstract grammar framework in a language with compile time classes

must then be delivered with an initialisation procedure to fill in this global variable. But each specialisation requires its proper installation code. Porting the framework becomes more error-prone. Moreover initialisation from the outside exposes implementation variables: they can be reassigned (or reinitialised) anytime. And, last but not least, a global variable makes it is impossible to run different specialisations simultaneously.

- Adding a parameter in each node-creation method. This parameter can then be filled in with the Abstract Factory object. This is probably the best alternative implementation to avoid the code duplication in the grey zones, but the code of the node-classes is troubled by the existence of and the communication with this extra parameter that is not part of the essential node behaviour.

Either way, the Abstract Factory classes have to be built and the artificial communication with it has to be programmed: a (too) complex task only for obtaining late creation.

3.1.2 Object creation problems in a language with prototypes or first-class classes

In languages with first-class classes, as in Smalltalk, classes can contain state and classes can be stored in variables. A node class can then contain every other node class in a variable[2]. The resulting scheme would resemble Figure 3, except that the create methods would be replaced by variables holding the appropriate node class. Analogously to avoiding the duplication of the create methods in the grey zones of Figure 3, it is now possible to avoid duplication of variables by holding only one variable in each node-class, namely an instance of the Abstract Factory class.

In order to allow the coexistence of different specialised versions, the specialisation of only one kind of node still necessitates to build a new class for *every* kind of node: the cross-reference has to be reconstructed entirely. So, compared to the solutions of the previous section, first-class classes do not ease the implementation of late creation.

[2] In Smalltalk instance variables of class objects must be used, and these variables must be made accessible for the instances of the classes. Class-variables are inappropriate since they are shared with subclasses,. That sharing would exclude the coexistence of different specialisations.

An additional Smalltalk-specific problem is that, due to the absence of C++ -like constructor methods, the variables holding the classes have to be initialised from the outside. Again, porting the framework is more complicated and the variables, albeit this time class object variables, are exposed.

Clonable prototypes are sometimes proposed as alternative object creation mechanism. But linking the right prototypes together is exactly equivalent to correctly filling in the Smalltalk class object variables. The scheme of related prototypes will be isomorphic with the corresponding Smalltalk class structure. Moreover prototypes give rise to other problems: the deep versus shallow clone dilemma; the variable exposing problem (a uniform parameterless clone operator forces initialisation from the outside, exposing even more implementation variables); and an artificial prototype only serving to be cloned can be inconsistent with the data-model (for instance which license plate should we give to an artificial car only serving to be cloned?).

3.2 Framework visibility through object-based encapsulation

As explained in section 2, the classes of our abstract grammar need common access to names representing state or classes. Some of these names represent implementation details. In order to obtain some independence in the development of the different frameworks in a system, and in the development of different specialisations of a single framework, some visibility mechanism is needed. Sharing a name between a family of classes that are not necessarily in subclass relationship with each other, and hiding the name from other classes, poses a problem in many object oriented languages. Similarly, hiding names from the specialisation of the framework and vice versa is often impossible.

A few object-oriented languages include some form of separate module system to regulate visibility. In C++ files can be used as modules. [Wirfs-Brock 88] introduced modules in Smalltalk. But modules are inappropriate to hide framework implementation details for the very same reason as they are inappropriate to hide object implementation details. The difference between modules and objects is just that different objects of the same kind can coexist, whereas a module represents a unique set of its items since it is a compile time aspect. As a consequence running simultaneously different specialisations of a framework with state, or providing simultaneously different implementations of the same framework functionality, requires the framework to be an object instead of a module. Perhaps coexistence of different specialisations is not directly a realistic need in the case of an abstract grammar, but it certainly is in the application-document example of a Factory Method in [Gamma 94]. So also object-based encapsulation instead of module-based encapsulation is needed to hide names, not only in simple objects but also in frameworks.

4 The abstract grammar in LENS

Figure 4 contains the implementation of the abstract grammar in LENS. Before discussing how the framework problems are solved, we briefly introduce the syntax and basic concepts of LENS necessary to understand Figure 4.

4.1 Syntactical conventions

All identifiers and operators in bold are predefined. Identifiers that would correspond to a class in a class-based language, are capitalised. But this is only a convention for clarity. The following message passing notation is adopted:

- Messages with an explicit receiver are denoted by writing the receiver followed by the message-selector. Arguments are added between parentheses, as in line 16: *optimiserTable atPut(to, whatOptimised)*.
- Receiverless messages represent self sends, e.g. *optimiseStatement* in line 8 and *myOperation(..,...)* in line 65.
- 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

```
[
/* Abstract grammar framework offering optimisation  */
 [
  optimiserTable variable;

  STATEMENT method(
    [  optimiseProgram method(
         [ optimiserTable!(DICTIONARY);
           optimiseStatement])
    ]);

  ASSIGNMENT(to, what) method(
    [ STATEMENT;
      optimiseStatement method(
        [ whatOptimised temporary(what optimiseExpression);
          whatOptimised isConstant if (
            optimiserTable atPut(to, whatOptimised),
            ASSIGNMENT(to, whatOptimised))]])
    ]);

  SEQUENCE(first, second) method(
    [ STATEMENT;
      optimiseStatement method(
        [ firstOptimised temporary(first optimiseStatement);
          (firstOptimised = nil) if(
            second optimiseStatement,
            SEQUENCE(firstOptimised, second optimiseStatement))
        ])
    ]);

  INPUT(to) method(
    [ STATEMENT;
      optimiseStatement method(
        [ optimiserTable removeKey(to);
          INPUT(to)])
    ]);

  OUTPUT(what) method(
    [ STATEMENT;
      optimiseStatement method(OUTPUT(what optimiseExpression))
    ]);

  LITERAL(val) method(
    [ isConstant method(true);
      value method(val);
      optimiseExpression method(LITERAL(val))
    ]);

  RUNTIME_EXPRESSION method(
      isConstant method(false)
    );

  IDENTIFIER(label) method(
    [ RUNTIME_EXPRESSION;
      optimiseExpression method(
        optimiserTable atIfAbsent(label, IDENTIFIER(label)))
    ]);

  OPERATION(left, right) method(
    [ RUNTIME_EXPRESSION;
      optimiseExpression method(
        [ leftOptimised temporary(left optimiseExpression);
          rightOptimised temporary(right optimiseExpression);
          ((leftOptimised isConstant) & (rightOptimised isConstant)) if(
            LITERAL(calculate(leftOptimised value,rightOptimised value)),
            myOperation(leftOptimised,rightOptimised))]])
    ]);

  ADD(left, right) method(
    [ OPERATION(left, right);
      calculate(left, right) method(left + right);
      myOperation(left, right) method(ADD(left, right))
    ] encaps(calculate, myOperation));

  MULTIPLY(left, right) method(
    [ OPERATION(left, right);
      calculate(left, right) method(left * right);
      myOperation(left, right) method(MULTIPLY(left, right))
    ] encaps(calculate, myOperation))

 ] encaps(STATEMENT, RUNTIME_EXPRESSION,
          OPERATION, optimiserTable);
```

```
/*Specialisation of abstract grammar,
  extending  the grammar with unparsing. */
 [
 ASSIGNMENT(to, what) method(
   [ super(to, what);
     leftHand constant(IDENTIFIER(to));
     unparse method(
       [ leftHand unparse;
         " := " print;
         what unparse])
   ] encaps(leftHand));

  SEQUENCE(first, second) method(
   [ super(first, second);
     unparse method(
       [ first unparse;
         "; " println;
         second unparse])
   ]);

  INPUT(to) method(
   [ super(to);
     toIdent constant(IDENTIFIER(to));
     unparse method(
       [ "input(" print;
         toIdent unparse;
         ")" print])
   ] encaps(toIdent))

  OUTPUT(what) method(
   [ super(what);
     unparse method(
       [ "print(" print;
         what unparse;
         ")" print])          ]);

  LITERAL(val) method(
   [ super(val);
     unparse method(val print)
   ]);

  IDENTIFIER(label) method(
   [ super(label);
     unparse method(label print)
   ]);

  OPERATION(left, right) method(
     unparse method(
       [ "(" print;
         left unparse;
         printOperator;
         right unparse;
         ")" print
       ]
   ));

  ADD(left, right) method(
   [ super(left, right);
     OPERATION(left, right);
     printOperator method(" + " print)
   ] encaps(printOperator));

  MULTIPLY(left, right) method(
   [ super(left, right);
     OPERATION(left, right);
     printOperator method(" * " print)
   ] encaps(printOperator));

 ] encaps(OPERATION);
```

```
 DICTIONARY method(    ...............  );

 parse method(
   SEQUENCE(
     ASSIGNMENT("a", LITERAL(3)),
      SEQUENCE(
        ASSIGNMENT("b", LITERAL(2)),
         SEQUENCE(
           INPUT("c"),
           OUTPUT(
             ADD(MULTIPLY(IDENTIFIER("a"),
                          IDENTIFIER("b")),
                 IDENTIFIER("c")))))))

 ] parse optimiseProgram unparse
```

Figure 4: the abstract grammar in LENS

ASSIGNMENT method in line 86, we will simply write ***super**(to,what)* to call the super variant of the *ASSIGNMENT* method.

LENS provides three kinds of slot declarations:

- *calculate(left, rigtht)* ***method**(body)* in line 70, declares a method slot with selector *calculate* and the two parameters *left* and *right*. *body* is evaluated each time the *calculate* message is sent.
- *optimiserTable* ***variable*** in line 3, declares a variable. This declaration introduces two slots: a retrieve slot *optimiserTable* and an update slot *optimiserTable!*.
- *leftHand* ***constant**(value)* ,in line 87, declares a read-only instance variable by only introducing the retrieve slot *leftHand* .

temporary declarations (*leftOptimised* and *rightOptimised* in lines 61 and 62) do not introduce object slots, but only introduce a local variable in the lexically surrounding method *optimiseExpression*.

New objects can be created from scratch by denoting them as a block (i.e. a name space) using brackets *[...]*. For instance the *LITERAL* method in line 42 creates a new literal node object consisting of the three slots declared in lines 43 to 45. The three slots are actually composed using inheritance, denoted by the ';' operator. Note that the outermost block of Figure 4, beginning at line 0 and ending at line 168, also denotes an object. We sent the message *parse* to it in line 168.

The *optimiseProgram* method declared in line 6 is an example of a more ordinary method invoking two statements. Since this method is denoted using brackets, it also creates an object. The convention is that assignments, for instance that in line 7, return an empty object. Composing this empty object with the result of the second statement, the method invocation in line 8, eventually yields the wanted result. Syntactically this is similar to writing the result of a method as its last statement.

4.2 Basic concepts of LENS

LENS is slot-based because instance variables are only accessible through a couple of retrieve and update methods. Slot-based instance variables blend state and behaviour, even towards inheritors, enhancing the degree of encapsulation [Ungar 87].

The *STATEMENT* method declared in line 5 creates an incomplete (or abstract) object only consisting of the method *optimiseProgram* that calls through a self send the undefined (or abstract) method *optimiseStatement*. Incomplete objects can be passed around. Wrong 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[3] to avoid such wrong uses at compile time.

Inheritance can be applied dynamically: inheritance hierarchies can be built at run time. In line 12 for example the *ASSIGNMENT* method, adds by means of inheritance the *optimiseStatement* method to a (newly created) statement object. Figure 5 presents an OMT-like [Rumbaugh 91] diagram of this situation. The diagram is OMT-incorrect because in OMT only classes can be specialised. 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 92]. But LENS is not prototype-based in the sense that its primitive object creation mechanism does not rely on the cloning of prototypes.

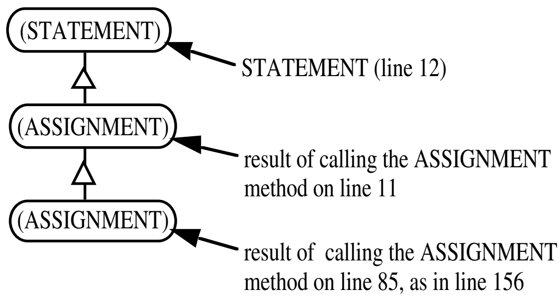

Figure 5: inheritance on abstract grammar node objects

[3] Theoretical foundations for typing dynamic inheritance can be found in [Lucas et al. 95]

In LENS a sub-object does not declare the class of its super-object: sub- and super-objects can be freely combined. Therefore the inheritance of LENS corresponds to mixin-based inheritance [Bracha 90].

In line 72, the ***encaps*** operator hides the slots *calculate* and *myOperation* from further clients (inheritors as well as senders[4]). ***encaps*** corresponds to the 'hide' operator of [Bracha 92]. This object-based encapsulation mechanism can be used as visibility boundary between sender and receiver or between original and specialisation. The combination of mixin-based inheritance and this kind of encapsulation is an innovative way to look at multiple inheritance [Van Limberghen 96].

A LENS program can contain nested name spaces. Nesting gives rise to a kind of closure: an assignment node for instance will implicitly retain a reference to its surrounding context, i.e. its creator. In this way it can refer to the more global variable *optimiserTable*.

We conclude this section by summarising the characteristics of LENS:

- object creation from scratch, no class instantiation, no prototype cloning
- dynamic mixin-based inheritance
- slot-based access to methods and variables
- nesting of objects

4.3 Solution of the framework problems

In contrast with the complicated implementations of our abstract grammar framework in section 3, the LENS program directly reflects its class design: the left respectively right rectangle in Figure 4 correspond to the upper and bottom rectangles in Figure 2. The abstract grammar indeed complies with

- Factory Method: during optimisation, the operation nodes *ADD* and *MULTIPLY* for instance create a *LITERAL* when both operands are constant (line 64).
- Abstract Factory: the *parse* method (line 154, in the program here only generating the example code of Figure 1) can be used to generate any specialisation of the abstract grammar.

The Factory Method aspect is accomplished so easily mainly thanks to <u>object creation from scratch</u>. In C++ Factory Methods are implemented by introducing (overridable) methods that perform the creation. The code of these methods consists of the instantiation of a class (see figure 3). Object creation from scratch avoids this intermediate step: in LENS these methods directly contain the object to be created. As a consequence the double administration of object creating methods and their corresponding classes is no longer necessary.

A second benefit of object creation from scratch with regard to Factory Methods concerns nesting. In C++, classes can be nested, but the induced nested scoping of identifiers cannot be used because the instance of a nested class can exist without any instance of the surrounding class. This is a typical problem when introducing nesting in an OO language with compile time classes. Object creation from scratch bypasses this problem: a nested object can only be created by a surrounding object. As a consequence the induced nested scoping is valid.

The total cross reference between the node classes, needed for the Factory Method aspect as explained in section 3.1, is obtained thanks to this <u>nested scoping</u>. By nesting the node objects in the same surrounding object, they can call each other. The surrounding object represents the framework. This way we obtain the total cross reference without having to write any supplementary code. We only have to implement the essential behaviour of the abstract grammar.

Most OO languages heavily emphasise object classification, but neglect object composition. Nevertheless object composition, especially part-whole composition, is an important method of apprehending real world objects during analysis. Moreover part-whole composition can also be

[4] LENS also contains a ***preventSpecialisation*** operation to hide a slot only from inheritors.

employed as a method of structuring software objects during design [Civello 93]. The nesting of LENS introduces a part-whole relationship between the nested object and the lexically surrounding object. In Figure 4 the whole is the abstract grammar framework and the parts are the grammar nodes.

Note that the nested scoping of LENS is not entirely lexical. The effective surrounding whole can be a specialised version of the statically surrounding one. This feature allows the whole to be specialisable, an essential requirement for the abstract grammar framework. In general the whole could even be abstract, i.e. some parts still must be specified by means of inheritance.

In section 3.1.2 we showed that storing classes or prototypes in variables is an inferior strategy to obtain framework genericity. Since <u>a class instantiation or prototype cloning mechanism is absent</u> in LENS, the programmer is discouraged from passing around first-class object templates. Instead, LENS advocates to uniformly use inheritance as genericity mechanism.

In LENS <u>inheritance</u> is applied <u>on objects</u>. This feature allows objects to be built incrementally. Consequently the initialisation of an object created by a framework can be distributed over the different parts of that framework, e.g. the optimiser and the unparser. In a class-based approach on the contrary, initialisation has to be done at once at instantiation time for the entire new object. Possibly some initialisation values, specific for an implementation of one part of the framework, have to be passed as arguments to the class instantiation procedure. This violates the independence between implementation and use of the parts of a framework.

By <u>encapsulating</u> the intermediate classes in lines 80 and 150, the internal inheritance structures for the optimiser and unparser are totally hidden, even towards each other. Consequently, the original framework and its specialisation(s) can be developed more independently. But, if desired, one can of course let part of the intermediate class structure be visible, in order to reuse the inheritance structure in framework specialisations. We could for instance let class OPERATION be visible since it represents the same classes in the parser and the optimiser.

The program in Figure 4 only invokes one version of the abstract grammar. But running different specialisations simultaneously could easily be obtained by giving the specialisations a (method-) name. In such a case the client has to qualify the desired framework anyway, so why not by means of message passing: the parser would for instance make an assignment node by sending *SEQUENCE* to the appropriate framework object instead of performing a self send as in Figure 4 (line 155). Since frameworks are objects, each version would have its own state. It would consequently also be possible to provide simultaneously different implementations of the optimiser framework.

5 Related work

5.1 Other research on late creation.

[Riehle 95] introduces late creation and class tree encapsulation in frameworks by only granting access to classes through a class specification system. A client can only retrieve classes by providing a logic expression describing the desired classes. The graphical editor in [Riehle 95] for instance finds all graphical classes with the class specification *isGraphical and isConcrete*. This way the editor can provide a button for each class of graphical objects that is defined in the actual framework. The framework implementor has to provide its classes with (meta) methods to answer on the retrieve specifications. Such a mechanism is useful when an external client needs to find a set of present classes, as in the graphics editor example. We would need a similar specification mechanism to solve this particular problem in LENS.

To cope with Factory Methods, [Riehle 95] additionally introduces a special kind of class specification to refer to the current computing context. In the case of our abstract grammar, this means that the implementor would specify something like *isAssignment and ofCurrentContext* to create an assignment node from within the specialisable framework. Again, the framework implementor has to provide the class-information. In this case he must ensure that the specification is not ambiguous since we expect only a single class. Furthermore the class tree has to be traversed, a rather expensive operation for simply indicating one specific concrete subclass. Another problem with this approach is that specialising only one abstract grammar node class (providing another unparsing for assignments for instance), still needs a whole set of new node-classes to be introduced. The issue of frameworks containing state, as with our *optimiserTable*, is not handled in [Riehle 95]. So, the approach of logic class specifications is not really appropriate for the problems presented in our paper, i.e. to simply obtain a late bound reference to the desired subclass.

[Kiczales 93] tackled yet another object creation problem emerging when specialising a framework, namely when the behaviour of internally created objects depends on the initialisation values. For instance when extending a framework of graphical objects with *optional* moving behaviour, a polygon should only be "movable" (i.e. contain a *move* method) when every of it lines is "movable", and a line on its turn should only be "movable" when both its end-points are "movable". [Kiczales 93] introduced "traces" to easily tackle this propagation phenomenon, without the programmer having to implement it. The concept of traces is orthogonal to the concept of late creation we proposed: LENS could be orthogonally extended with traces.

5.2 Nesting in other OO languages

Most OO languages do not support nested scoping. Smalltalk and Eiffel do not offer nesting. In C++, classes can be nested. But as explained in section 4.3, the induced nested scoping of identifiers cannot be used.

The only prototype-based language with nested scoping that we know of is Agora [Steyaert 93]. Nesting is conceived there as a kind of subclassing restriction (a task that we would like to delegate to an optional and more powerful classification system, see our future work).

Beta [Madsen 93] is a class-based language with nested scoping. In Beta, a nested class can only be accessed by sending a message to *an instance of* the surrounding class. This allows nested scoping to be introduced but necessitates at the same time that classes are first-class values. LENS deliberately omitted first-class object templates. Just as in LENS, the Beta nested scoping is not lexical for identifiers denoting overridable methods. The advantage of inheritance on objects, as in LENS, with respect to inheritance on classes, as in Beta, concerns initialisation, as explained in section 4.3.

5.3 Other related research

[Kiczales 92] identified problems in extending traditional class libraries due to overridden methods that are called from within the framework. The specialiser does not know where or when these calls happen in the library. For this reason we provided a *total* cross reference between the framework "classes" by implementing the Factory Method design pattern (see section 3.1). [Kiczales 92] criticises non-overridable methods as solution because the framework has to anticipate them. This criticism is not valid in LENS because the encapsulation operators can also be performed afterwards by the specialiser. But, apart from this detail, LENS starts from the philosophy that a self send to a method intends to be possibly overridden. If not, the called behaviour is no longer worth to be called a method and should be encapsulated (possibly only towards inheritors), transforming the method internally in a procedure. In this sense we believe that a library designer certainly should keep specialisations in mind by indicating what is overridable and what isn't, and by creating and invoking intermediate methods containing replaceable code.

[Kühne 95] identifies cases where ordinary parametrisation is a more appropriate software adaptability mechanism than inheritance. The enhanced inheritance mechanism of LENS omits different of the flaws of traditional inheritance enumerated in [Kühne 95] and allows inheritance to be applied more uniformly as adaptability mechanism.

[Ossher 95] agree with us that there is a need for unanticipated extension and composition, for decentralised development and for grouping methods by functionality instead of by class. They propose subject-orientation as a solution. [Harrison 93], their preceding paper on subject-oriented programming, even uses as main example a tree with different separate functionalities, just as our abstract grammar tree. We also agree that traditional OO is too firmly connected to the notion of *identity*. In LENS the inheritance mechanism was detached from the identity of the self object: self sends are used instead of self references. We try to identify some differences with our approach without judging:

- [Ossher 95] does not seem to propose anything like late object creation.
- In LENS classes are constructed by means of inheritance and encapsulation. [Ossher 95] proposes a set of composition rules to merge existing classes.

Contracts [Holland 92] are yet another approach to group code by functionality instead of by class. Contracts emphasise rather on specifications to which the collaboration must comply, whereas LENS deals with building the collaboration.

In [Andersen 92] separate aspects of objects are represented in a role model, containing the roles the objects 'play' in the context of that aspect. The notion of role models, emerging from the design world, very much resembles a set of collaborating mixins as in LENS.

6 Future work

LENS was created with the aim to express OO design ideas more directly. A comparison with the other design patterns in [Gamma 94] strengthens us in the approach taken. LENS is not innovative in implementing design patterns that are not involved with inheritance, for instance the Iterator and Command design patterns that are also valid in software without inheritance. But when inheritance is involved, LENS scores well. The Visitor design pattern for instance deals with different classes that have to be iterated, possibly adapting state during traversal. An abstract grammar is actually a perfect example. In other languages an artificial visitor class and the communication with the visitor object has to be programmed. The combination of dynamic inheritance and encapsulation also allows to directly obtain the software requirements indicated by the design patterns Decorator, Bridge and Strategy.

A possible way to make LENS even more design oriented is to restrict the dynamic inheritance with static classification. Mixins are chunks of code that can be freely combined. Therefore mixin-based inheritance is sometimes criticised to be a mere code-sharing mechanism without conceptual meaning. In traditional class-based systems the class hierarchy partially fulfils a combination restricting role, but still has to be enhanced with extra restricting capabilities. Multiple inheritance is less expressive than it appears, essentially in its lack to put constraints on multiple inheritance from different classes. For example we should be able to prevent a class to inherit from the classes Male and Female simultaneously. To this extent, [Hamer 92] include *classifiers* in their class hierarchy. We are thinking about a similar static classification mechanism especially destined for mixins, preserving its characteristic of parametrical super binding. [Lucas 95] suggested to use a classification system as to make their type system less verbose. Unifying static classification and static typing sounds very reasonable and is one of the following challenges for LENS.

Currently we execute LENS programs with a bluntly implemented interpreter: we did not pay attention to efficiency matters in the current implementation. LENS and the prototype-based language Self both contain a dynamic inheritance mechanism and both are dynamically typed. Therefore we expect that the results of the efforts undertaken to optimise Self [Chambers 91] can be transported to LENS and will give it an acceptable performance. But avoiding runtime method lookup in our dynamically typed language seems a hard problem. Static method lookup can be obtained in an efficient way in dynamically typed languages, as long as the inheritance structure remains static [Driesen 95]. But for languages with dynamic inheritance, static method lookup techniques are as yet unavailable. It has still to be investigated if the static type and classification system we are thinking of, can and should be used for efficiency matters.

7 Conclusions

Object orientation is conceived to specify behaviour incrementally. But current OO languages do not support the incremental specification of a particular kind of behaviour, namely object creation. However the design patterns Abstract Factory and Factory Method demand late bound object creation to be considered as a high level concept in object oriented frameworks.

LENS was presented as the orthogonal combination of existing OO aspects: late-binding (by means of message passing), pure (i.e. mixin-based or code-reuse) inheritance and object-based encapsulation. This combination was shown to fit the needs for framework construction. The object creation mechanism did not consist of class instantiation, nor prototype cloning. Instead, objects were created from scratch. This way we used the same mechanism to subject both object creation and procedure calls to late-binding. Consequently we did not need to introduce new language concepts to obtain late creation.

Object composition, another important constituent of design, has been neglected in most OO languages. In LENS it was present by means of nesting, modelling part-whole relationships between software components. Because objects, especially nested objects, were specialisable, it was easy to construct specialisable frameworks. Object-based encapsulation was used to selectively restrict specialisation.

The Abstract Factory and Factory Method design patterns did no longer have to be implemented: only the essential behaviour of the abstract grammar framework was written down. The code directly reflected the design of the framework. All of this made us believe that LENS narrows the gap between OO design and implementation and that it constitutes a valuable basis for an OO software design language.

Acknowledgements

I am much indebted to Tom Mens for experiments in and thorough discussions about LENS. I also thank Kris De Volder, Kim Mens, Niels Boyen and Thomas Kühne for comments on earlier versions of this paper.

References

[Andersen 92] E. P. Andersen and T. Reenskaug: System Design by Composing Structures of Interacting Objects. In Proc. of ECOOP’92, pp.133-152, Springer-Verlag.

[Bracha 90] G. Bracha and W. Cook: Mixin-based Inheritance. In Proc. of OOPSLA/ECOOP’90, pp.303-311, ACM Press.

[Bracha 92] G. Bracha and G. Lindstrom: Modularity meets Inheritance. In Proc. of International Conference on Computer Languages, 1992, IEEE Computer Society, pp. 282-290. Also available as Technical report UUCS-91-017.

[Canning 89] P. S. Canning, W. R. Cook, W. L. Hill & W. G. Olthoff: Interfaces for Strongly-Typed Object-Oriented Programming. In Proceedings of OOPSLA '89, pp. 457-467, ACM Press.

[Carré 90] B. Carré and J. Geib: The Point of View notion for Multiple Inheritance. In joint OOPSLA/ECOOP ’90 Conference Proceedings, pp. 312-321, ACM Press.

[Chambers 91] C. Chambers, D. Ungar: Making Pure Object-Oriented Languages Practical. In Proceedings of OOPSLA '91, pp. 1-15, ACM Press.

[Civello 93] F. Civello: Roles for composite objects in object-oriented analysis and design. In Proc. of OOPSLA'93, pp. 376-493, ACM Press.

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

[Driesen 95] K. Driesen, U. Hölzle: Minimising Row Displacement Dispatch Tables. In Proc. of OOPSLA'95, pp. 141-154, ACM Press.

[Gamma 94] 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.

[Hamer 92] J. Hamer, J. G. Hosking, and W.B. Mugridge: Static Subclass Constraints and Dynamic Class Membership Using Classifiers. Technical Report (1992), University of Auckland, Computer Science Department.

[Harrison 93] W. Harrison and H. Ossher: Subject-Oriented Programming. In Proc. of OOPSLA'93, pp. 411-429, ACM Press.

[Holland 92] I. M. Holland: Specifying reusable components using Contracts. In Proc. of ECOOP'92, pp.287-308, Springer-Verlag.

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

[Kiczales 92] G. Kiczales and J. Lamping: Issues in the Design and Specification of Class Libraries. In Proc. of OOPSLA'92, pp. 435-451, ACM Press.

[Kiczales 93] G. Kiczales: Traces: A cut at the 'make isn't generic' problem. ISOTAS '93 Conference Proceedings, LNCS 742, Springer-Verlag.

[Kühne 95] T. Kühne: Parametrisation versus inheritance. In Proceeding of TOOLS Pacific (TOOLS 15), 1995, Prentice Hall.

[Lalonde 90] W. R. Lalonde and J. R. Pugh: Inside Smalltalk. ISBN 0-13-468430-3, 1990, Prentice-Hall.

[Madsen 93] O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Object-Oriented Programming in the Beta Programming language. ISBN 0-201-62430-3, Addison Wesley, 1993.

[Meyer 88] B. Meyer: Object-oriented Software Construction. ISBN 0-13-629031-0, 1988, Prentice Hall.

[Ossher 95] H. Ossher, M. Kaplan, W. Harrison, A. Katz and V. Kruskal: Subject-Oriented Composition Rules. In Proc. of OOPSLA'95, pp. 235-250, ACM Press.

[Riehle 95] D. Riehle: How and Why to Encapsulate Class Trees. In Proc. of OOPSLA'95, pp. 251-264, ACM Press.

[Rumbaugh 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen: Object-Oriented Modeling and Design. ISBN 0-13-630064-5, 1991, Prentice Hall.

[Steyaert 93] P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, M. Van Limberghen: Nested Mixin-Methods in Agora. In ECOOP '93, pp. 197-219, Springer-Verlag .

[Steyaert 96] P. Steyaert, C. Lucas, K. Mens, T. D'Hondt: Reuse Contracts: Managing the Evolution of Reusable Assets. To appear in OOPSLA '96, ACM Press.

[Stroustrup 91] B. Stroustrup: The C++ Programming Language. Second edition, ISBN 0-201-53992-6, 1991, Addison-Wesley.

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

[Van Limberghen 96] M. Van Limberghen and T. Mens: Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems. Object Oriented Systems journal volume 3 number 1, 1996, pp.1-30, Chapman & Hall.

[Wirfs-Brock 88] A. Wirfs-Brock, B. Wilkerson: An Overview of Modular Smalltalk. In Proc. of OOPSLA '88, pp. 123-134, ACM Press.