Symbol Table Construction and Name Lookup in ISO C++

James F. Power
Department of Computer Science
National University of Ireland
Maynooth, Co. Kildare
Ireland
james.power@may.ie

Brian A. Malloy
Computer Science Department
Clemson University
Clemson, SC 29634
U.S.A.
malloy@cs.clemson.edu

Abstract

In this paper, we present an object-oriented model of symbol table construction and name lookup for ISO C++ using the Unified Modeling Language (UML). Our use of UML class, activity and sequence diagrams serves to explicate our model and our use of patterns such as decorator *and* facade *increase the understandability of the model. Clause three of the ISO C++ standard describes the procedures and rules for performing name lookup; our activity and sequence diagrams serve to simulate these procedures in graphical fashion. An advantage of our approach is that our model can increase C++ understandability for those practitioners with a working UML knowledge.*

An important contribution of our work is that our model forms the basis for construction of a parser front-end for ISO C++. Our explication of the name lookup problem represents a necessary first step in this construction and our component approach is independent of the compiler technology utilized. Our use of the UML in describing parser-driven applications demonstrates how front-end development can be integrated into an object-oriented framework. Construction of an ISO C++ front-end will help to increase the collection of tools for applications that use this popular language.

1: Introduction

As software developers shift their priorities to the construction of complex, large scale systems that are robust and easy to extend, traditional approaches and methodologies fall short. Object orientation makes it possible to model systems that are close to their real world analogues. The goal of object-oriented design is to accurately identify the principle roles in a process, assign responsibilities to these roles and encapsulate them in an object. The benefits of object technology are extensibility, ease of modification and ease of reuse.

Given the advantages of object technology, it is surprising that some practitioners have still not adopted the object-oriented approach. The most obvious difficulty is that software support for object technology is not uniform or widespread[8, 9]. Software tools are fundamental to the comprehension, analysis, testing and debugging of application systems. Tools can automate repetitive tasks and, with large scale systems, can enable computation that would be prohibitively time consuming if performed manually. In many cases, tool development requires a parser front-end to recognize the implementation language of the system under development. The lack of software support for applications using the C++ language is especially noteworthy; in fact, there is no tool available in the public domain that can accept applications written in ISO C++[1].

One explanation for the lack of software tools for C++ is the difficulty in constructing a front-end for the language, as described in references [3, 6, 7, 12, 13]. This difficulty results from both the complexity and scale of the language[11]. Many constructs in C++ cannot be recognized through syntactic considerations alone. For example, the difficulty in distinguishing a declaration from an expression can only be resolved by performing name lookup. Given the obvious mapping of objects to the roles in front-end construction such as a parser or scanner, it is unfortunate that the carry-over of object technology to parser development has been minimal.

In this paper, we present an object-oriented model of the name lookup problem for ISO C++ using the Unified Modeling Language (UML). Our use of UML class, activity and sequence diagrams serves to explicate our model and our use of patterns such as *decorator* and *facade* increase the understandability of the model. Clause three of reference [1] describes the procedures and rules for performing name lookup; our activity and sequence diagrams serve to simulate these procedures in graphical fashion. An advantage of our approach is that our model can increase C++ understandability for those practitioners with a working UML knowledge.

An important contribution of our work is that our model forms the basis for construction of a parser front-end for ISO C++. Our explication of the name lookup problem represents a necessary first step in this construction and our component approach is independent of the compiler technology utilized. Our use of the UML in describing parser-driven applications demonstrates how front-end development can be integrated into an object-oriented framework. Construction of an ISO C++ front-end will help to increase the collection of tools for applications that use this popular language.

The rest of this paper is organized as follows. In the next section we discuss the complexity of the problem that we consider, provide background and introduce terminology important to our work. In Section 3 we discuss the analysis of clause three of the C++ standard that allowed us to determine the information required for our model and in Section 4 we describe the subsystem of our model that initiates and choreographs name lookup. Section 5 describes the design of our symbol table and its use in resolving context-dependent ambiguity. Finally, in Section 6 we draw conclusions.

2: Overview of the Name Lookup Problem

In the next subsection, we define terms and introduce concepts important for understanding the problem that we consider. We then describe some of the difficulty involved in the construction of a parser front-end for C++, including the importance of a solution to the name lookup problem. In the second subsection, we overview our approach for constructing a symbol table for C++ and the technique that we use to perform name lookup.

2.1: Terminology and statement of the problem

Software developers need tools to facilitate the design, analysis and testing of the system under development. Many useful tools require a parser front-end to compute and store information about the program, and to perform the analysis needed. A *parser front-end* performs lexical and syntactic analysis, constructs a symbol table, performs semantic analysis and possibly generates an intermediate representation of the program. A *symbol table* is a data structure that stores information about the types and names used in the program.

Many programming language constructs have an inherently recursive structure that can be defined by context-free grammars, or CFGs[2]. Most of the constructs of languages such as Pascal and Ada can be specified by CFGs, and parser front-ends for these languages can base their recognition on syntactic considerations alone. An exception to this easy-parse rule can be found in the language C, where a declaration may not be easily distinguished from an expression. Consider the following code segment:

```
f (x);
```

Intuitively, the above code segment appears to be an expression involving an invocation of function `f` with parameter `x`. However, if the context includes the following declaration:

```
typedef int f;
```

then the code segment is actually a declaration of `x` as an integer variable with redundant parentheses. This *declaration/expression ambiguity* not withstanding, parser front-ends for the C language have not been difficult to construct. However, a parser front-end for the C++ language has proven elusive and the difficulties involved have been described in references [3, 6, 7, 11, 12, 13]. Currently, there is no parser front-end in the public domain that can parse the language described in the ISO C++ standard[1]. Many constructs in the C++ language cannot be recognized by syntactic consideration alone; these constructs not only include the *typedef* declaration/expression ambiguity of C, but C++ also includes context-dependent keywords for *namespace*, *class*, *enumeration* and *template* declarations[1, Appendix A].

To illustrate the declaration/expression ambiguity introduced by templates into C++, consider that for a code segment such as a < b, the name a must be looked up to determine whether the < is the beginning of a template argument list or a less-than operator [1, §3.4.5/1]. Thus, the disambiguation of many C++ constructs requires a solution to the name lookup problem. The *name lookup* problem is defined as follows: given the use of a name in the program, find the corresponding declaration of that name.

The GNU Free Software Foundation offers gcc, a public domain compiler for the C++ language. However, it is difficult to de-couple the parser front-end of gcc from the back-end of the compiler. Even if de-coupling were achieved, low-level access to the internals of gcc is not easily accomplished. Furthermore, gcc does not parse the language described in the C++ standard.

2.2: Overview of our approach

Figure 1 summarizes the design of our system to construct a parser front-end for ISO C++. The figure presents two subsystems, illustrated as tabbed folders and designated by the ≪subsystem≫ stereotype. The ProgramProcessor subsystem is shown on the left side and the Symbol Table subsystem is shown on the right side of Figure 1. The ProgramProcessor and Symbol Table subsystems are elaborated in Sections 4 and 5 respectively.

The ProgramProcessor subsystem includes a Scanner and Parser and is responsible for initiating and directing symbol table construction and name lookup. This responsibility includes two phases: (1) assembling the necessary information for creation of a NameOccurrence object, and (2) directing the search for a corresponding NameDeclaration object in the Symbol Table subsystem.

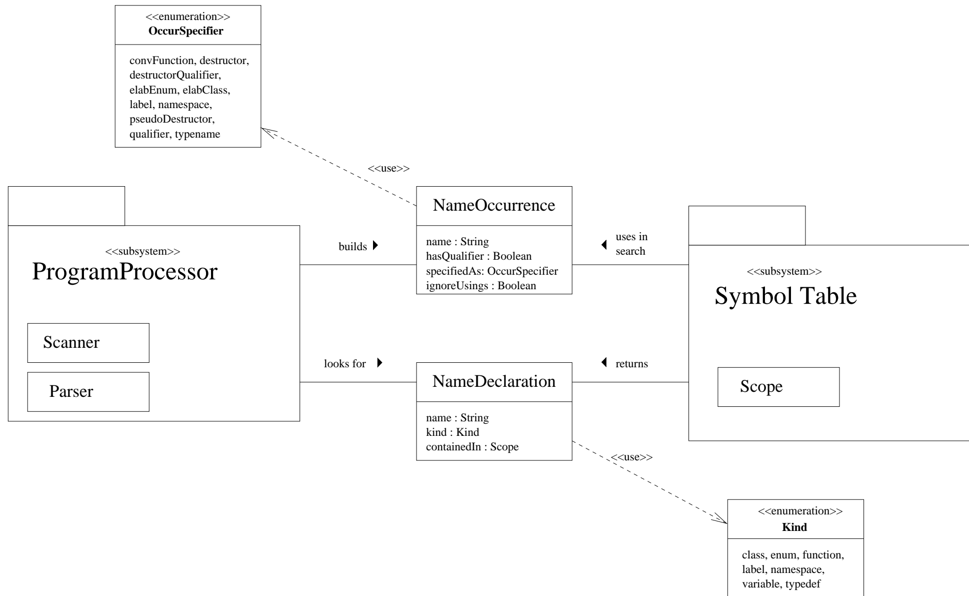

Figure 1. *System summary*. **This figure summarizes the design of our system to construct a parser front-end for ISO C++. The** ProgramProcessor **subsystem is responsible for initiating and directing symbol table construction and name lookup by marshaling information about the name in a** NameOccurrence **object and directing the search for a corresponding** NameDeclaration **in the** Symbol Table **subsystem.**

The NameOccurrence object encapsulates local information relevant to the lookup, including the String representation of the name, a boolean to indicate name qualification (by class or namespace), and an enumeration, OccurSpecifier, that captures lexical information about the context in which the name occurred. The NameDeclaration object includes the String representation of the name, an enumeration indicating the type of name, and a pointer to the enclosing scope. The NameOccurrence object is discussed further in the next section.

3: Structural Analysis

In this section we describe the analysis process carried out on clause three of reference [1] in order to determine the information required for the construction of our model. A use-case analysis is often employed to guide the design of high-level systems; however, the relatively low-level nature of designing a C++ front-end required that this approach be adapted somewhat.

Since the purpose of name lookup is to associate each usage of a name with its corresponding declaration, clearly the representation of both the usage and declaration of a name will be central to the design. The natural modularization of the design into the ProgramProcessor and SymbolTable subsystem creates a central role for these NameOccurrence and NameDeclaration objects as the principal means of communication between these subsystems.

Clause three of reference [1] describes the various scenarios relevant to name lookup,

Attribute	Provider	Location	Reference from [1]
elabEnum, elabClass	TD	NO	3.3.1/5, 3.4.4/3
ignoreUsings	CM	NO	3.4.2/2, 3.4.3.2/6
convFunction	P	NO	3.4.3.1/1
isDeclarator	P	LC	3.3.1/5, 3.3.1/6, 3.4.1/10, 3.4.1/12, 3.4.3.2/6, 3.4.4/3
destructor	TD	NO	3.4.3/5, 3.4.5/3
destructorQualifier	TD	NO	3.4.3/5
isFriend	CM	LC	3.3.1/6, 3.4.4/3
label	TD	NO	3.3.4
namespace	TD	NO	3.4.6
pseudoDestructor	TD	NO	3.4.3/5
qualifier	TD	NO	3.4.3/5, 3.4.5/4
memberOf	TD	LC	3.4.3/5, 3.4.5/1, 3.4.5/3, 3.4.5/4
prevDeclarator	CM	LC	3.4.1/10, 3.4.1/12, 3.4.3/3
qualifiedBy	TD	LC	3.3.1/5, 3.4.3/4, 3.4.3/5, 3.4.3.1/1, 3.4.3.2/6, 3.4.4/3, 3.4.5/4
searchWholeClass	CM	LC	3.4.1/7, 3.4.1/8, 3.4.1/12

Table 1. *Attributes required for name lookup*. **This table summarizes the analysis carried out on the relevant sections of clause three of the C++ Standard in order to decide on the nature of the attributes required for name lookup. In each case we specify the provider of the information as either the** TokenDecorator **(TD) or the** ContextManager **(CM), and also the location of the information as either in a** NameOccurrence **object (NO) or in the** ContextManager **(CM).**

---

both in terms of the structure of sample C++ programs, as well as the context of a given name occurrence (such as the presence of qualifiers, elaborations etc.). In all, sections 3.3 and 3.4 of reference [1] present 24 pieces of C++ code, containing roughly 65 instances of name occurrences. Our approach was to analyze each example in order to determine the information necessary to perform name lookup and, in particular, to distinguish each given case of name lookup from the others.

The results of this analysis, which determined the structure of the NameOccurrence class, are summarized in Table 1. In each case, it was necessary to determine whether the required information could be determined lexically, or whether a more global context was required. In addition, a distinction was made between the information needed to correctly initiate name lookup, and the information needed during the lookup process itself. These results also provided an initial analysis for the construction of classes within the ProgramProcessor and SymbolTable subsystems, since these subsystems are responsible respectively for constructing and for using instances of NameOccurrence.

The scenarios depicted in the examples given in clause three of reference [1] do not, of course, provide a complete specification of the name lookup process. They do, however, provide a set of scenarios necessary to both the analysis of the problem and to the validation of the model. In terms of the number of such scenarios, they provide a reasonable balance between case coverage and the feasibility of performing manual validation for each case.

It would be possible to construct a comprehensive set of test cases from the rules detailed throughout clause three. However, given the complexity of the name lookup problem, such a test suite would be quite large, and probably more suitable as a basis for the validation

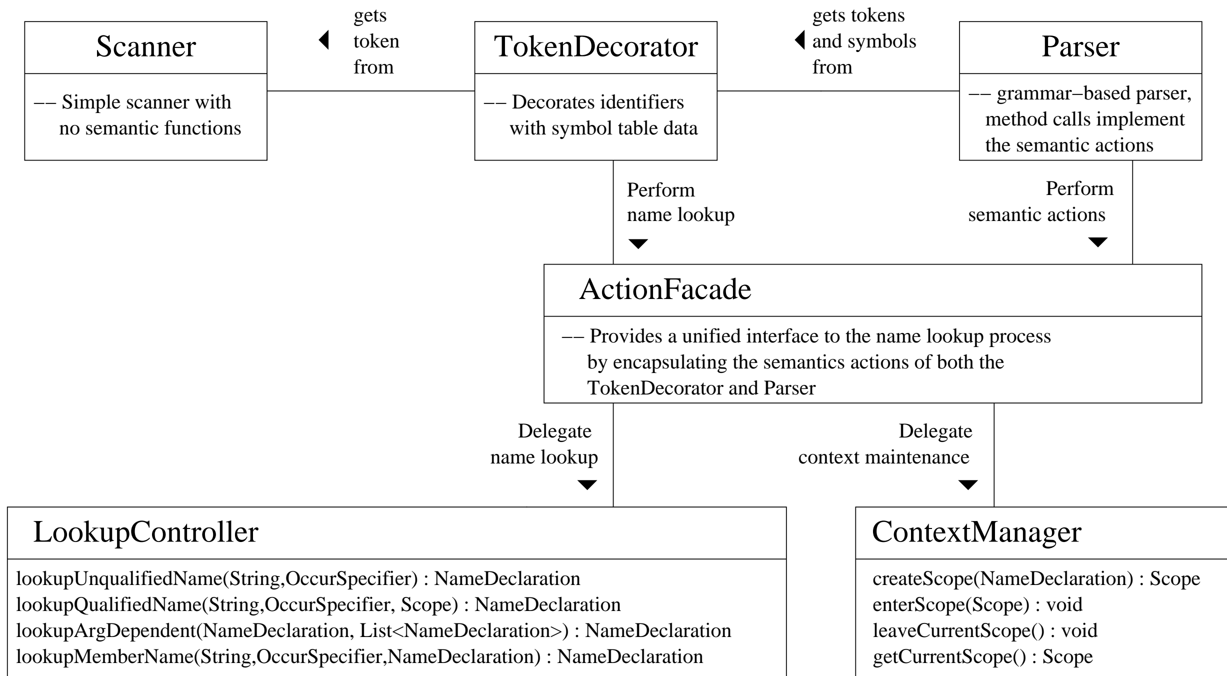

Figure 2. *Classes in the Program Processor Subsystem*. **This figure illustrates the relationships between the classes in the ProgramProcessor subsystem. The activity of the system is driven by the Parser, which interacts with the Scanner via the TokenDecorator, and with the LookupController and ContextManager via the ActionFacade.**

of a prototype implementation, where automatic testing techniques can be used.

4: The Program Processor Subsystem

In this section we describe the internal structure of the ProgramProcessor subsystem. This subsystem includes both the Scanner and the Parser, but is also responsible for initiating and directing name lookup. This task can be divided into two main phases: assembling the necessary information for the creation of a NameOccurrence object, and directing and collating the result of the search for a corresponding NameDeclaration object.

The class diagram in Figure 2 shows the main classes in the ProgramProcessor subsystem. The traditional view of the parsing process is represented by classes for the Scanner and Parser which communicate here via the TokenDecorator class. The ActionFacade class unifies the interface between these three classes and those which actually control the name lookup process, the LookupController and ContextManager.

The Scanner of Figure 2 is a simple lexical analyzer with no semantic content; in particular, identifiers are presented by a single *identifier* token, and not by their context-sensitive counterparts, such as *class-name*, *namespace-name*, etc. The task of mapping the identifiers to their context-sensitive equivalents is handled by the TokenDecorator class. This implements the *Decorator* pattern in the sense of reference [5] by "wrapping" identifier tokens as symbol table entries before passing them to the Parser. As such it is similar to the "post-lexical analysis pass" of the processor described in reference [12]. When the parser

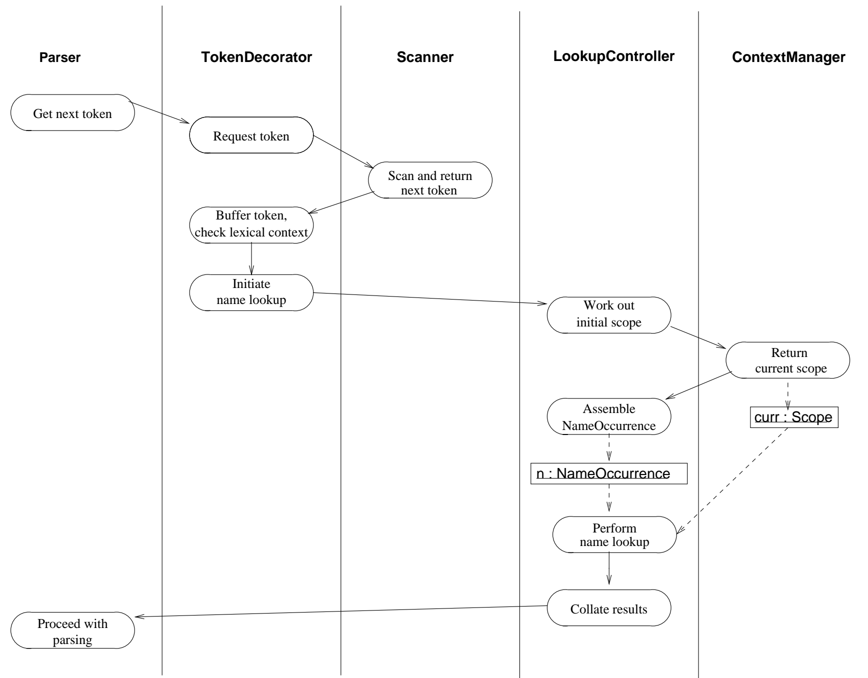

Figure 3. *Activity Diagram for Name Lookup***. This figure illustrates the sequence of high-level actions that take place during name lookup. The role of the** ActionFacade **has been omitted here for clarity.**

---

requests a token from the TokenDecorator, it will either receive a non-identifier token passed directly from the Scanner, or one of the context-sensitive identifier tokens, deduced from the corresponding symbol-table entry.

In order to match an identifier with the corresponding instance of NameDeclaration from the symbol table, the TokenDecorator must initiate a name lookup for that identifier. To do this it must assemble any local information relevant to the lookup, and pass control to the LookupController which will direct the search. The information collected by the TokenDecorator is limited to that which can be deduced from a few tokens of lexical context, as discussed in Table 1 of section 3, and is manifested in the choice of lookup function called. The sequence of actions involved here is summarized in the first three swim-lanes of the activity diagram shown in Figure 3.

All messages passed from the TokenDecorator or Parser that deal with symbol-table access or maintenance are routed through the ActionFacade. This is an example of the *Facade* pattern of [5], and is crucial to the design of our system. The ActionFacade does not deal directly with the symbol-table itself, but delegates these operations to either the Lookup-Controller or the ContextManager.

The chief purpose of the ActionFacade is to act as a repository for the semantic actions of the parser. The actions associated with each grammar rule may vary in complexity, but can involve a number of different semantic operations. For example, on encountering a new class definition, we will want to create a new symbol table entry for class name, a new Scope object for the class, and to inform the ContextManager that any subsequent declarations belong to this new Scope object. Placing all this information in the parser definition would

add complexity to the grammar definition, impede manipulation of the grammar rules and reduce modularity. Instead, complex series of semantic actions are represented by a single method in the ActionFacade, and it is a call to this method that is used alongside the grammar rules.

The use of the ActionFacade here also provides the usual advantage of de-coupling the Parser from the classes concerned with name lookup. Thus, changes to the parser need not impact either the LookupController or ContextManager classes. In particular, significant changes in the parsing algorithm (e.g. from top-down to bottom-up parsing) can be handled by appropriately sub-classing the ActionFacade class, rather than refactoring the entire design of the ProgramProcessor.

The ActionFacade class provides one more important function. While it is intended that a user of the C++ parser would mainly be interested in the output of the system as a whole, it may be necessary, for certain fine-grained operations, to access the operation of the system at a lower-level. Since the ActionFacade is at the heart of all significant semantic operations, a suitable interface to this class should provide a sufficiently low-level access to the system internals for most purposes.

Behind the ActionFacade are two classes - the LookupController and the ContextManager - that are responsible for interacting with the Scope hierarchy to perform name lookup. The ContextManager holds those aspects of the context needed for name lookup that cannot be determined by the TokenDecorator, as indicated in Table 1 of section 3. One of the most crucial pieces of information here is maintaining a link to the current Scope object, since this acts as the starting point for unqualified name lookup, and is the typical location of new declarations.

Changes to the context information in the ContextManager are directed by the Parser. The ContextManager is then queried for the relevant information by the LookupController before the control of name lookup is passed to the Scopes. The two swim-lanes on the right of the activity diagram of Figure 3 summarize the main activities involved here. For clarity, the ActionFacade has been elided, since its role is trivial during this operation.

As can be seen from Figure 3, the LookupController first consults the ContextManager and collects any relevant context information. The LookupController then creates a NameOccurrence object, mainly from the information received from the TokenDecorator. Name lookup is then performed by passing the NameOccurrence object to at least one relevant Scope - typically the current Scope returned by the ContextManager, or the specified scope in the case of qualified or class-member lookup.

Certain more complex cases may involve more than one name lookup call, such as argument-dependent name lookup [1, §3.4.2], or in the case of a qualified class-member [1, §3.4.5/3]. In these cases the LookupController has the responsibility of directing each lookup to the appropriate Scope, and collating the results of these lookups. In all cases a single NameDeclaration object is returned to the TokenDecorator, which then returns the appropriate token to the Parser. Should name lookup fail, a new instance of NameDeclaration can be constructed and returned; the Parser then has the responsibility for managing this object and, where appropriate, initiating the process of inserting it into the symbol table.

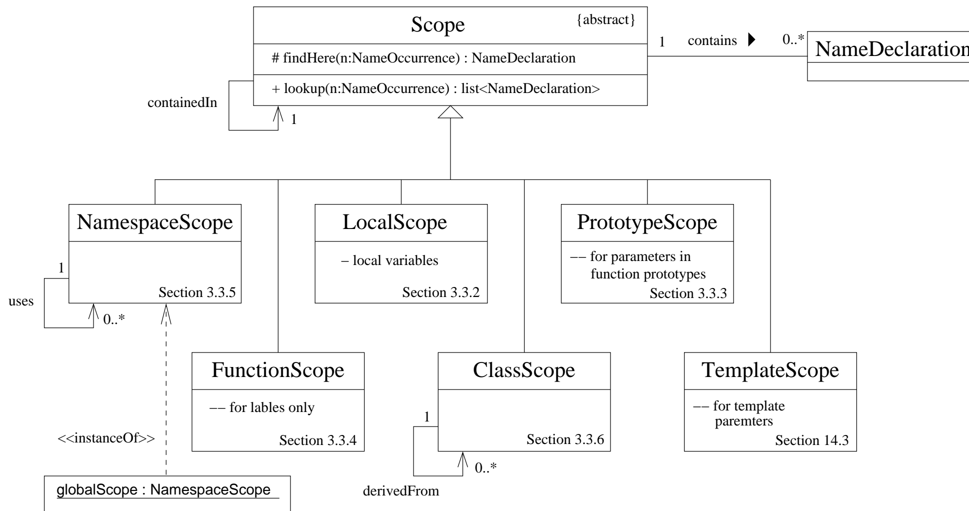

Figure 4. *Class Diagram for the Scopes Hierarchy*. **Each** Scope **object contains a list of** NameDeclaration **instances, along with name lookup functionality. The** Scope **class has six subclasses, as detailed in the referenced sections of the ISO standard.**

5: The Scope Subsystem

In this section we turn to the second subsystem depicted in Figure 1, the symbol table. In general, compilers are designed so that the symbol table can store considerable amounts of semantic information. For our purposes, however, we simply need to be able to determine the context-dependent keyword that corresponds to a given identifier in order to allow parsing to proceed. Thus, the standard entry in the symbol table is a NameDeclaration object, as described in section 3.

ISO C++ has a relatively complex system of scopes when compared to e.g. ISO C. In addition to the typical nested block structure of local declarations in functions, we must also consider classes and namespaces, along with the relationships between these via inheritance and `using` directives and declarations. Composing these scoping constructs, via nested namespaces, unnamed namespaces, nested and local classes and, of course, templates further complicates the process of name lookup.

The complete set of scoping constructs is shown in Figure 4. In addition to representations of namespace, class and local scopes, there are three other scope constructs for special cases: FunctionScope for labels, TemplateScope for template parameters, and PrototypeScope for the parameters in a function prototype. Each instance of a Scope consists of a set of NameDeclaration objects, representing the identifiers declared at that scope level.

Because of the complexity of the name lookup process, it would be undesirable to seek to assign responsibility for all of this functionality to the LookupController. Instead, as much of the searching and decision-making process as possible is delegated to each kind of scoping construct, represented here by the lookup method of the Scope class. This has the effect of reducing the complexity of the LookupController, and decreasing the coupling between the ProgramProcessor and SymbolTable subsystems. The pragmatics of this design are discussed further in reference [10].

For each instance of a subclass of Scope we need to specify the lookup procedure to be

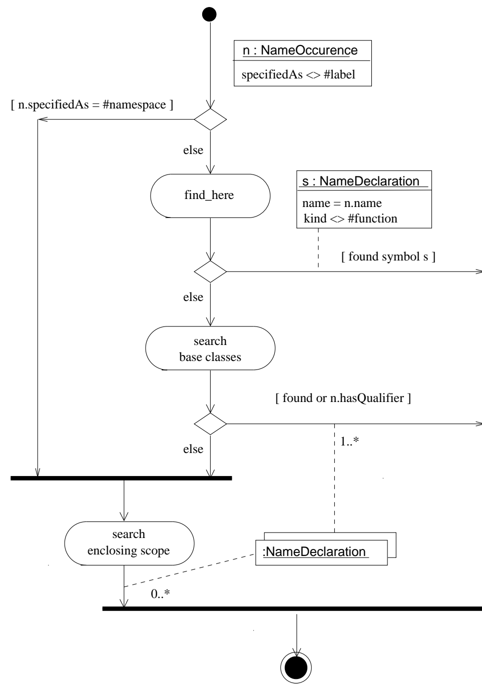

Figure 5. *Activity Diagram for Name Lookup in ClassScope*. **This is a description of the** lookup **method, which is specific to each of the six subclasses of the** Scope **class. We provide for a** *set* **of returned** NameDeclaration **objects to allow for function overloading.**

---

carried out. In each case, the method findHere is called to search the scope in question. This is a straightforward search, identical for all Scope classes, and takes into account the preferences expressed in the NameOccurrence object. For example, class or enum names are normally hidden by variable or function names, except in certain special cases where this is reversed, such as elaborated type specifiers [1, §3.4.2].

The lookup method is specific to each subclass of Scope, and implements the main decision-making involved in the name lookup for this kind of scope. As an example, the lookup method for instances of ClassScope is modeled in the activity diagram shown in Figure 5. Here we see the basic procedure expressed as: a search in the current scope, a search through the base classes (if any), followed by a search through enclosing scopes. In each case where the search is delegated to another instance of Scope, it is the delegate Scope that then takes control of the lookup process.

An example of the name lookup process is shown in the sequence diagram of Figure 6. This diagram is based on the example from [1, §3.4.5/4] of a class member which is explicitly qualified by a class name, in this case the expression `e.B::a` occurring in a local scope. In this example, `e` is an instance of a class `E`, `B` is a base class of `E`, and `a` is a member of a base class of `B`. In this case the qualifier name `B` must be looked up both in the scope of class `E`, and in the context of the whole expression.

The sequence diagram of Figure 6, showing the name lookup for the qualifier `B`, is particularly useful in demonstrating the division of responsibilities between the classes involved. The TokenDecorator is responsible for detecting that `B` is a qualifier, as well as noting that

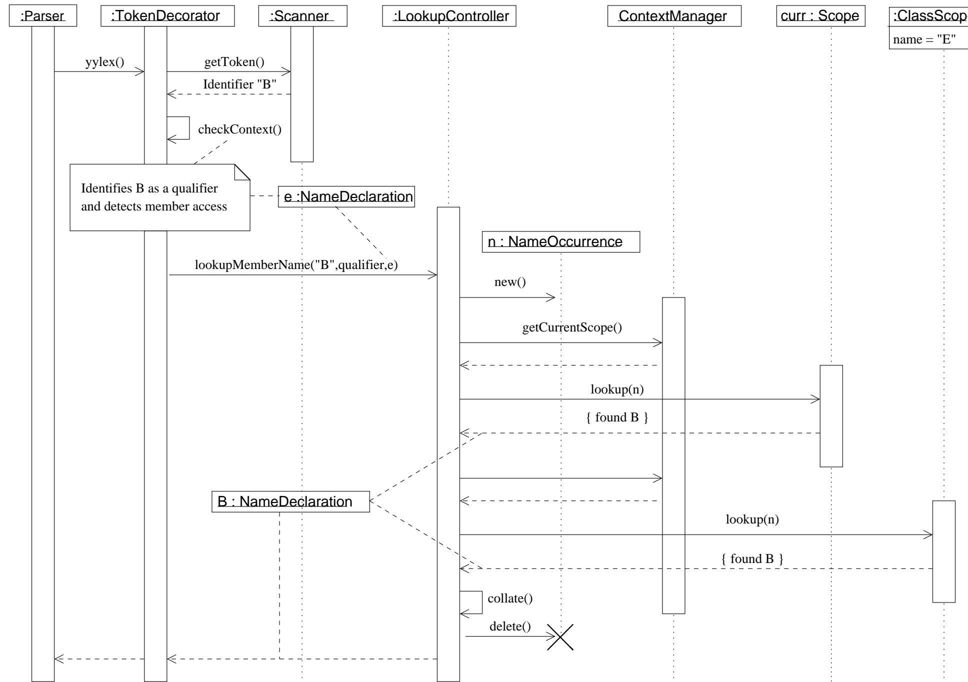

Figure 6. *Sequence Diagram for Name Lookup in* `e.B::a`**. This gives a detailed description of the interaction between the objects involved in the lookup for the qualifier `B`, based on an example in section 3.4.5/4 of the ISO standard.**

---

it occurs in the context of member access. The LookupController must then ensure that the lookup is initiated both in the current scope, and in the scope of the object e, both retrieved from the ContextManager. As well as creating and dispatching a suitable NameOccurrence object for the search, the LookupController is also responsible for collating the results of both searches, checking for consistency, and returning the appropriate NameDeclaration object to the TokenDecorator, which is then returned to the Parser.

The flexibility of the UML is particularly useful in dealing with the examples from clause three of the ISO standard. Since the text in the standard is often terse or vague, the examples play a crucial role in fleshing out the specification, as well as checking the completeness and consistency of the structural model. The web of interactions between the scopes in each example can be represented clearly using an object diagram, and each lookup can be represented as a sequence diagram. The construction of such diagrams for each example in clause three was of fundamental importance in both enhancing our understanding of the clause, and in increasing the accuracy and completeness of the model.

6: Concluding Remarks

In this paper we have presented an object-oriented model of the name lookup problem in ISO C++ using the UML. To do this we have carried out a case-based analysis of clause three of the ISO C++ standard, and have modeled aspects of this clause using UML class, activity and sequence diagrams.

The work presented here is similar in focus to that discussed in references [4] and [12],

although neither of these deal in detail with the object-oriented design of the program processor and its interaction with the symbol table. The only formal description dealing with aspects specific to the C++ programming language is reference [14], but this concentrates exclusively on aspects of the *dynamic* semantics of the language. In particular, none of the above approaches had the advantage of reference to the ISO standard, or deals with more recently-added features such as namespaces. For an analysis of the ambiguities associated with parsing C++, both references [6] and [13] are particularly useful.

As well as providing the basis for an implementation of a front-end for C++, we see the contribution of this work as being in three main areas. First, it provides an explication of clause three of the ISO standard, presenting a structural rather than purely procedural view of the issues involved. Second, it demonstrates the use of object-oriented techniques, and the UML in particular, in modeling semantic aspects of real-world programming languages. This is particularly relevant for the name lookup problem in C++ where the central issue is one of complexity and scale, rather than theoretical difficulty. Third, it provides an example of the use of the UML in describing parser-driven applications, and of how such applications can be integrated into an object-oriented framework.

References

[1] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. American National Standards Institute, first edition, September 1998.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *OON-SKI*, pages 122–136, Oregon, USA, 1994.

[4] S.C. Dewhurst. Flexible symbol table structures for compiling C++. *Software – Practice and Experience*, 17(8):503–512, August 1987.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] G. Knapen, B. Laguë, M Dagenais, and E. Merlo. Parsing C++ despite missing declarations. In *Proceedings of the 7th International Workshop on Program Comprehension*, Pittsburgh, PA, 1999.

[7] J. Lilley. PCCTS-based LL(1) C++ parser: Design and theory of operation. Ver. 1.5, February 1997.

[8] R. Malan, D. Coleman, and R. Letsinger. Lessons from the experiences of leading-edge object technology projects in Hewlett-Packard. In *Proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications*, pages 33–46, October 1995.

[9] C.M. Pancake. The promise and the cost of object technology: a five-year forecast. In *Communications of the ACM*, volume 38, pages 32–49, June 1995.

[10] J. F. Power and B. A. Malloy. An approach for modeling the name lookup problem in the C++ programming language. In *ACM Symposium on Applied Computing*, Como, Italy, March 2000.

[11] J. F. Power and B. A. Malloy. Metric-based analysis of context-free grammars. In *Proceedings of the 8th International Workshop on Program Comprehension*, Limerick, Ireland, June 2000.

[12] S.P. Reiss and T. Davis. Experiences writing object-oriented compiler front ends. Technical report, Brown University, January 1995.

[13] J.A. Roskind. A YACC-able C++ 2.1 grammar, and the resulting ambiguities. Independent Consultant, Indialantic FL, 1989.

[14] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.