INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

APPLYING REFLECTION IN OBJECT-ORIENTED SOFTWARE DESIGN

Yun Mai

A Thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science
Concordia University
Montréal, Québec, Canada

September 2001
© Yun Mai, 2001

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-64060-4

Canadä

Abstract

Applying Reflection in Object-Oriented Software Design

Yun Mai

Software systems evolve over time. They should be open to modifications in response to changing technology and requirements. Designing a system that meets a wide range of different requirements is a difficult task. A better solution is to specify an architecture that is open to modification and extension. The resulting system can then be able to adapt to changing requirements on demand.

Reflection is a process of reasoning about and acting upon itself. In the vocabulary of software development, it provides a mechanism for dynamically changing the structure and behavior of a software system. It supports the modification of some fundamental aspects like type structures, function call mechanisms, etc.

The thesis work performs a series of experiments on applying reflection technique to improve software design. First, a REFLECTIVE VISITOR pattern was captured to improve the traditional VISITOR pattern. Reflection enables a visitor to perform a run-time dispatch action on itself. The cyclic dependencies between the visitor structure and the element structure are broken thus both of them can be reused independently. Secondly, a parser framework was developed by applying several patterns. Especially, the REFLECTION pattern is used in the design of dynamically handling a parsing process by separating the system into two levels. The base-level defines the grammar rules. The meta-level handles the complex relationships of these rules. Reflection technique is used to discover grammar rules at run-time and determines the parsing order. Third, a dynamic object model was defined for a virtual machine that can support reflection. We demonstrated Forman's theory by developing a simplified object model based on a single inheritance system with the support of only one metaclass. Finally, an extensible and reusable compiler system (front-end) for the Decaf programming language was designed and implemented. It customizes the parser framework by defining concrete grammar rules for the Decaf language, constructs the virtual machine platform by extending the reflective class-based object model, and applies REFLECTIVE VISITOR to the code generation.

Acknowledgements

First of all, I would like to thank my mentor and supervisor, Dr. Michel de Champlain, for his patience and invaluable guidance. I really appreciate his dedicated mentoring and the timely and inspiring discussions he had with me. I am forever grateful to him for his inspiration, continuous encouragement, and step-by-step guidance.

I want to thank all professors who have taught me at Concordia University. Special thanks to Dr. Gregory Butler, Dr. V.S. Alagar, Dr. Joey Paquet, Dr. H. Rivard, and Dr. S. Amiouny.

I also want to thank Dr. Gregory Butler and Dr. Ferhat Khendek for their time and their patience to read this thesis, and giving me valuable comments.

I am grateful to Stephane Ducasse (our PLoP '2001 shepherd), Brian Marick (our EuroPLoP '2001 shepherd), Doug Lea, Kevlin Henney, Arno Haase, Joseph Bergin, Saluka R. Kodituwakku, and Alexander Horoshilov for their valuable comments for my pattern papers.

I gratefully acknowledge Concordia University for the research grant it offered to me and the Quebec government for all financial support I received during my studies at Concordia. I also want to thank EuroPLoP '2001 conference committee for the scholarship I received.

I would like to thank my wife, Jinmiao Li, for her guidance, support, encouragement, and love. I dearly thank my parents and my parents-in-law for their continuously encouraging me to achieve this challenging work.

Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Aim 2

1.2 Motivation 2

1.3 Contribution of the Thesis 4

1.4 Layout of the Thesis 6

2 Background 7

2.1 Reflection 7

2.1.1 What is Reflection ? 7

2.1.2 History of Reflection 8

2.1.3 Reflection Pattern 9

2.1.4 Java Reflection 10

2.1.5 Metaclass and Dynamic Object Model 15

2.2 Pattern 20

2.2.1 What is a Pattern ? 20

2.2.2 History of Patterns 23

2.2.3 Qualities of Patterns 23

2.3 Design Pattern 25

2.3.1 What is a Design Pattern ? 25

2.3.2 Document a Design Pattern 26

2.4 Pattern Language 27

2.4.1 What is a Pattern Language ? 27

2.4.2 Document a Pattern Language 28
2.5 Framework 30
2.5.1 What is a Framework ? 30
2.5.2 Framework vs. Software Pattern 32
2.5.3 Develop a Framework 33
2.6 Summary 34

3 Reflective Visitor: To Extend and Reuse Object Structure 35
3.1 Terms and Concepts 36
3.2 Reflective Visitor 37
3.3 A Pattern Language to Visitors 56
3.3.1 A Road Map to Visitors 56
3.3.2 A Simple Example 61
3.3.3 Double Dispatch 61
3.3.4 Easy Operation Adder 64
3.3.5 Catch-All Operation 72
3.3.6 Easy Element Adder 77
3.3.7 Easy Element and Operation Adder 85
3.4 Summary 89

4 Parsing with Reflection Pattern 90
4.1 A Pattern Language for Parsing 91
4.1.1 Overview 91
4.1.2 Parser Structure 92
4.1.3 Language Structure 95
4.1.4 ParserBuilder 97
4.1.5 MetaParser 101
4.2 Summary 105

5 Reflective Class-Based Object Model 106
5.1 General Principles 106
5.2 Class Hierarchy of the Object Model 107
5.3 Elements of the Object Model 108
5.3.1 Class Table 108
5.3.2 Object Reference 109

5.3.3 Class Reference 110
5.3.4 MetaClass 112
5.3.5 Field Reference 113
5.3.6 Method Reference 114
5.3.7 Code Pointer 115
5.3.8 Byte Code 115
5.3.9 ListValue 116
5.4 Summary 116

6 Decaf Compiler: An Application Example 118
6.1 Decaf Language Specification 118
6.1.1 Decaf Grammar 119
6.1.2 Virtual Machine 121
6.2 Architectural Design 123
6.2.1 Overview of the Subsystems 124
6.2.2 Dependency Relationships among the Subsystems 124
6.3 Lexical Analyzer 125
6.3.1 The Lexer Component: Facade of the Lexical Analyzer 125
6.3.2 The Token Type Hierarchy 128
6.3.3 The Reserved Word Table and the Flyweight Pattern 129
6.4 Customize the Parser Framework 129
6.4.1 Define the Language Structure 130
6.4.2 Define the Grammar Rules 138
6.5 Code Generation 143
6.5.1 Extend the Reflective Class-based Object Model for Decaf . . 144
6.5.2 Implement the Visitor Hierarchy 151
6.6 Summary 155

7 Conclusion and Future Work 157
7.1 Summary and Conclusions 157
7.2 Future Work 159

List of Tables

1 Problem/Solution Summaries for Visitors 59
2 Comparison on Visitor Patterns 60
3 Problem/Solution Summaries for Patterns in the Parsing 91
4 Conventions of Decaf Syntax 119
5 Instruction Set 122

List of Figures

1	A General Compiling Process	3
2	Class Diagram for Class java.lang.Class	11
3	Class Diagram for Package java.lang.reflect	13
4	Application Developed from a Framework	32
5	An Expression Hierarchy	38
6	Apply GoF Visitor Pattern to the Expression Example	40
7	Apply Reflective Visitor Pattern to the Expression Example	41
8	The Structure of the Reflective Visitor	43
9	The Sequence Diagram for the Visiting Process	45
10	Road Map for the Visitor Patterns	56
11	Structure for the Double Dispatch Pattern	62
12	Structure for the Easy Operation Adder Pattern	66
13	Structure of the Catch-All Operation Pattern	73
14	Structure for the Easy Element Adder Pattern	78
15	Structure for the Parser Structure	94
16	Structure for the Language Structure	96
17	Structure for the ParserBuilder	99
18	Sequence Diagram for the Parsing Process in the ParserBuilder	100
19	Structure for the MetaParser Pattern	102
20	Sequence Diagram for the Parsing Process in the MetaParser	104
21	Class Hierarchy of the Object Model	107
22	Class Diagram for ClassTable	109
23	Class Diagram for ObjectReference	109
24	Class Diagram for Class Reference	111
25	Class Diagram for MetaClass	112
26	Class Diagram for FieldReference	113

27 Class Diagram for MethodReference 114
28 Class Diagram for Code Pointer 115
29 Class Diagram for Byte Code 116
30 Class Diagram for ListValue 117
31 Class Hierarchy for Decaf Language 123
32 Package Diagram for Decaf Compiler 125
33 Separating the Lexical Analyzer Subsystem into Three Components . 126
34 Class Diagram for the Lexer Component 126
35 Sequence Diagram for a Tokenizing Process 127
36 The Token Type Hierarchy 128
37 Top Level Classes in the Expression Hierarchy 131
38 Class Diagram for Expressions with Two Operands 133
39 Class Diagram for Expressions with One Operand 134
40 Class Diagram for Primary Expressions 135
41 Class Diagram for Special Expressions 136
42 Class Diagram for Statement Hierarchy 137
43 Class Diagram for Class Declarations 138
44 Class Diagram for Member Declarations 139
45 Class Diagram for Rule Library 141
46 Class Diagram for Class DecafGrammar 142
47 Structure of the Memory Package 145
48 Class Diagram for DecafClassValue 146
49 Class Diagram for MethodValue 149
50 Class Diagram for Code Pointer 150
51 Build-in Classes 151
52 Structure of Code Generation 152
53 Class Diagram for Class CodeGeneration 153
54 Class Diagram for Classes ExpressionGenerator and StatementGenerator 154
55 Class Diagram for Class ClassGenerator 155

Chapter 1

Introduction

Systems evolve over time — new functionality is added and existing services are changed. During system design and implementation, customers may request new features, often urgently and at a late stage. The designer may also need to provide services that differ from customer to customer. Design for change is therefore a major concern when developing a software system.

Since reflection was first introduced by Smith [67] as a framework for language extension, the reflection technology has been a research topic and attracted the attention of researchers throughout computer science over the last twenty years. In the recent years, the reflection technique has been integrated into programming languages and has become an importance feature in most object-oriented programming languages such as Java [45, 32], Smalltalk [38], Oberon [77, 70], Scheme [28], etc.

Conceptually, reflection is the process of reasoning about and acting upon itself. In the vocabulary of software development, reflection provides a mechanism for changing structure and behavior of a software system dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms in a programming language.

This thesis work focuses on the application of reflection in object-oriented software design, especially on how reflection helps to increase the flexibility in compiler systems to maximize the benefits of reuse.

1.1 Aim

The thesis aims to experiment with the reflection technique in reusable and extensible software design. Specifically, the research focuses on three aspects:

- How can the reflection feature in an object-oriented programming language benefit the software design? In this aspect, we focus on applying Java Reflection to simplify the software design and improve the system's reusability.
- How to design an extensible software system with the reflection pattern? In this aspect,we focus on applying the reflection pattern to a syntactic analyzer (i.e. parser) in a programming language compiler system to adapt to changing the grammar at run-time.
- How to implement reflection in an object-oriented programming language? In this aspect, we focus on designing and implementing a dynamic object model on a virtual machine system to support reflection of an object-oriented programming language.

1.2 Motivation

Suppose that we are designing a compiler framework for compiling an object-oriented programming language to a virtual machine. Like some object-oriented programming languages such as Java, the compiled object-oriented programming language has reflection support as one of its basic features. This means that this programming language can obtain class information (e.g. fields, methods, parent of the class, etc.) on the fly and can even invoke methods that do not appear explicitly in the code, but are discovered and invoked at run-time.

The virtual machine for an object-oriented programming language generally defines an instruction set to support method compilation and a class object heap to keep information for each compiled class.

In general, a compiling process (front-end) has three basic phases: lexical analysis, syntactical analysis, and code generation. Figure 1 shows a compiling process in general. To simplify the problem context, some compiler phases such as code optimization will not be considered in this example.

In our compiler framework design, there are some design issues:

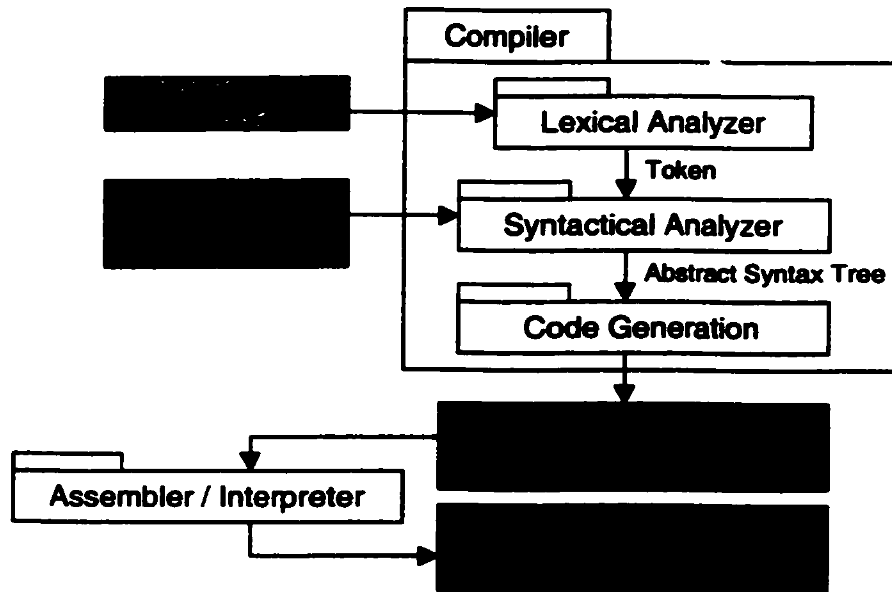

Figure 1: A General Compiling Process

- In order to support extension and modification of a programming language grammar, the framework should accept changing grammars in the parsing process. That is, the language grammar can be changed without changing the existing code. But, how to let the system determine the parsing process execution at run-time rather than at compiler-time?

- Code generation is a process determined by the parsing result and the specification of the virtual machine. But, how to decouple the code generation and parser so that the code generation process can be changed without changing the parsing process?

- How to define an object model for the virtual machine so that the programming language can support reflection and allow the virtual machine to be easily extended?

The contradictory considerations must be taken into account when choosing a solution to a problem. Forces reveal the intricacies of a problem and define the kinds of trade-offs that must be considered in the presence of the tension or dissonance they

create [56]. The forces that influence the design decisions of the compiler framework are:

- A programming language is determined by its grammar.
- The specifications of a language grammar and virtual machine determine what a compiler should do.
- The grammar of a programming language may change or can be extended.
- The virtual machine may change. That is, the instruction set and the class information structure may change.
- Extending a system without modifying the existing system will reduce some new bugs introduced by some modification of the existing code and avoid long recompilation time.
- To let the system be aware of the changes at run-time rather than at compile time will make the system more flexible and extensible.
- It is hard to reuse any module within a tightly coupled system. Reducing the couplings between the modules will improve the reusability of the system.
- We assume that the execution time is not a primary concern for a compiler system design.

1.3 Contribution of the Thesis

The thesis work performs a series of experiments on the application of the reflection technique to better improve the software design in some circumstances, that is, a software environment that can support reflection.

First, a REFLECTIVE VISITOR pattern [53] was captured to improve the structure of the VISITOR pattern [37] so that both the element hierarchy and visitor hierarchy are extensible. In our REFLECTIVE VISITOR pattern, a visitor can perform the run-time dispatch action on itself by using the reflection technique (e.g. Java Reflection). The cyclic dependencies between the visitor hierarchy and the element hierarchy are therefore removed. The REFLECTIVE VISITOR is thus more flexible and reusable.

Second, due to the high usage of the VISITOR pattern in the software design, a pattern language for Visitors [52] is presented to classify and organize the VISITOR variants. This pattern language will assist the application developer to choose the right VISITOR pattern that best suites the intended purpose by enumerating all important forces and consequences for each variant.

Third, a parser framework [51] was developed. A pattern language is presented for developing a framework for parsing in object-oriented compiler design based on the principle of the predictive recursive-descent parsing approach. It describes four patterns that address three design aspects in developing an object-oriented parser. Two alternative patterns are presented to provide alternative solutions to solve the recursion problem in the object-oriented software design. A two-level structure is defined for implementing the parsing process control based on the REFLECTION pattern [13, 34]. The base-level contains a set of classes, where each represents a grammar rule. The meta-level handles the complex relationships of the rules that are maintained in a hash table. Reflection is used to discover rules at run-time and determine the parsing order. The base-level delegates the dynamic dispatch to a meta-level object.

Fourth, a dynamic object model is defined for a virtual machine that can support reflection. We demonstrate Forman's theory [33] by developing a simplified object model based on a single inheritance system with the support of only one metaclass.

Finally, we designed and implemented an extensible and reusable compiler system (front-end) for the Decaf programming language, which is an extensible tiny object-oriented programming language. This compiler system customizes the parser framework by defining concrete grammar rules for the Decaf language, constructs virtual machine platform by extending the reflective class-based object model, and applies REFLECTIVE VISITOR for implementing the code generation.

The Decaf language itself can be further refined as a framework for experimental object-oriented languages so that a more complex language can be developed based on it without changing the existing system.

1.4 Layout of the Thesis

Chapter 2 presents the thesis background including an introduction to the reflection technology, pattern, design pattern, pattern language, and framework.

Chapter 3 introduces the Reflective Visitor pattern, which is an improvement of the "Gang of Four" VISITOR pattern [37] based on the programming language reflection feature (i.e. Java reflection). The Reflective Visitor pattern can lead to a framework that is easy to extend and reuse the object structure. A pattern language to visitors is presented to introduce and compare the visitor variants.

Chapter 4 presents a parser framework design, which is documented as a pattern language. This pattern language consists of a set of patterns that address the architectural design and component design for a parser. The MetaParser pattern is based on the REFLECTION pattern [13, 34].

Chapter 5 presents a reflective class-based object model, which can be extended to support reflection feature on most of the object-oriented programming language.

Chapter 6 shows a concrete application example of customization. The example applies the techniques in chapter 3, 4 and 5 to develop a compiler system.

Chapter 7 concludes this thesis with an outline of its contributions and gives suggestions for future work.

Chapter 2

Background

This chapter presents the background of the thesis. In general, there are three trends in object-oriented design. The first trend is the reuse of the source code. Roughly, before 1993, people focused on improving the coding quality to ensure that some pieces of code are good enough and general enough to be reused in some other similar circumstances.

After that, people found that to only reuse source code was not enough, they also wanted to reuse the software analysis and design, especially those precious experience from domain experts. There came the second trend. Patterns are used to capture successful software designs, pattern languages are used to better document the successful experience, and frameworks are used to support software reuse of the software analysis, design, and source code.

The third trend is reflection and meta-programming. An object's run-time behavior can be better controlled and manipulated if it is self-aware and self-discoverable.

Section 2.1 reviews the notion of reflection and its three major applications. Then the concepts of pattern, design pattern and pattern language are introduced in Section 2.2, Section 2.3 and Section 2.4. Section 2.5 introduces the framework concept.

2.1 Reflection

2.1.1 What is Reflection ?

Software systems evolve over time. They must be open to modifications in response to changing technology and requirements. Designing a system that meets a wide range

of different requirements a priori can be an overwhelming task. A better solution is to specify an architecture that is open to modification and extension. The resulting system can then be adapted to changing requirements on demand. In other words, we want to design for change and evolution [13].

A software system has dynamic adaptability if it can adapt itself to dynamically changing runtime environments [6]. Conceptually, reflection is the process of the system reasoning about and acting upon itself [68, 48, 80]. Intuitively, a reflective computational system allows computations to observe and modify properties of their own behavior, especially properties that are typically observed only from some external, meta-level viewpoint [69]. In the vocabulary of software development, reflection provides a mechanism for dynamically changing the structure and behavior of a software system. It supports the modification of fundamental aspects, such as type structures and function call mechanisms in a programming language.

A system design with reflection maintains information about itself and uses this information to remain changeable and extensible. In particular, a reflection system opens its implementation to support adaptation, change, and extension of specific structural and behavioral aspects such as type structures, function call mechanisms or implementations of particular services [13].

2.1.2 History of Reflection

Since reflection was first introduced by Smith [67] as a framework for language extension, reflection technology has been a research topic and has attracted the attention of researchers throughout computer science over the last twenty years. Many books [15, 46, 82], conferences, and on-line sources on reflection have pointed out the growing interest and importance of reflection and meta-level architecture in the fields of software engineering and programming languages and systems. Object Management Group(OMG) published a Meta Object Facility (MOF) specification [59] in 1999, which defines a MOF meta-data architecture based on the traditional framework for meta-modeling of four layer architecture: User Object layer (data), Model layer, Meta-model layer, and Meta-meta-model layer. In the recent years, reflection has been integrated into programming languages and has become an important feature in most object-oriented programming languages such as Java [45, 32], Smalltalk [38], Oberon [77, 70], Scheme [28], etc.

The following sections focus on the discussion of the REFLECTION pattern, Java Reflection, and reflective class-based object model.

2.1.3 Reflection Pattern

To make the software self-aware and to make selected aspects of its structure and behavior accessible for adaptation and change, an architecture can be split into two major parts: a meta-level and a base-level.

The REFLECTION pattern, also known as META-LEVEL ARCHITECTURE or OPEN IMPLEMENTATION, provides a mechanism for dynamically changing the structure and behavior of a software system. It supports the modification of fundamental aspects, such as type structure and function call mechanisms. In this pattern, an application is split into two parts. A meta-level provides information about selected system properties and makes the software self-aware. A base-level includes the application logic. Its implementation builds on the meta-level. Changes to information kept in the meta-level affect subsequent base-level behavior.

The meta-level provides a self-representation of the software to give it knowledge of its own structure and behavior, and consists of so-called metaobjects. Metaobjects encapsulate the software properties. Only system details that are likely to change or which vary from customer to customer should be encapsulated by metaobjects.

The base-level defines the application logic. The components of the base-level represent the various services the system offers as well as their underlying data model. The base-level also specifies the fundamental collaboration and structural relationships between the components it includes. In general, its implementation uses the metaobjects to remain independent of those aspects that are likely to change. For example, base-level components may only communicate with each other via a metaobject that implements a specific user-defined function call mechanism. Changing this metaobject changes the way in which base-level components communicate, but without modifying the base-level code.

Normally, an interface is specified for manipulating the metaobjects. It is called the metaobject protocol (MOP). This metaobject protocol allows clients to specify particular changes. The metaobject protocol itself is responsible for checking the correctness of the change specification, and for performing the change. Every manipulation of metaobjects through the metaobject protocol affects subsequent base-level

behavior.

The general structure of a reflective architecture is very similar to a layered system. The meta-level and base-level are two layers. The meta-level consists of a set of metaobjects. Each metaobject encapsulates selected information about a single aspect of the structure, behavior, or state of the base-level. All metaobjects together provide a self-representation of an application. A metaobject does not allow the base-level to modify its internal state. Manipulation is possible only through the metaobject protocol or by its own computation. The base-level uses the information and services provided by the metaobjects. This allows the base-level to remain flexible — its code is independent of aspects that may be subject to change and adaptation. Base-level components are either directly connected to the metaobjects on which they depend, or submit requests to them through special retrieval functions, which are also part of the meta-level. Since the base-level implementation explicitly builds upon information and services provided by metaobjects, changing them has an immediate effect on the subsequent behavior of the base-level. In conventional modification, the system can be changed without modifying based-level code. The metaobject protocol (MOP) services as an external interface to the meta-level, and makes the implementation of a reflective system accessible in a defined way. Clients of the metaobject protocol can specify modifications to metaobjects or their relationships using the base-level. This metaobject protocol itself is responsible for performing these changes. This provides a reflective application with explicit control over its own modification [13].

2.1.4 Java Reflection

2.1.4.1 Overview

Java Reflection refers to the ability of Java classes to reflect upon themselves. It lets the software designer easily write Java code to discover class information at run time. The designer can even invoke methods that do not appear directly in the code, but are discovered and invoked at run time.

The Java Core Reflection API [71] supports introspection about the classes and objects in the current Java Virtual Machine. The API can be used to:

- construct new class instances and new arrays.
- access and modify fields of objects and classes.

- invoke methods on objects and classes.
- access and modify elements of arrays.

Generally, Java Reflection is provided by three components:

- method *getClass* in class `Object`
  The method *getClass* defined in the class `Object` returns the `Class` object associated with any `Object`. The class `Object` is the root class in Java. All classes are subclasses of `Object`, and thus all objects can obtain its class object by invoking the *getClass* method.
- `java.lang.Class` class

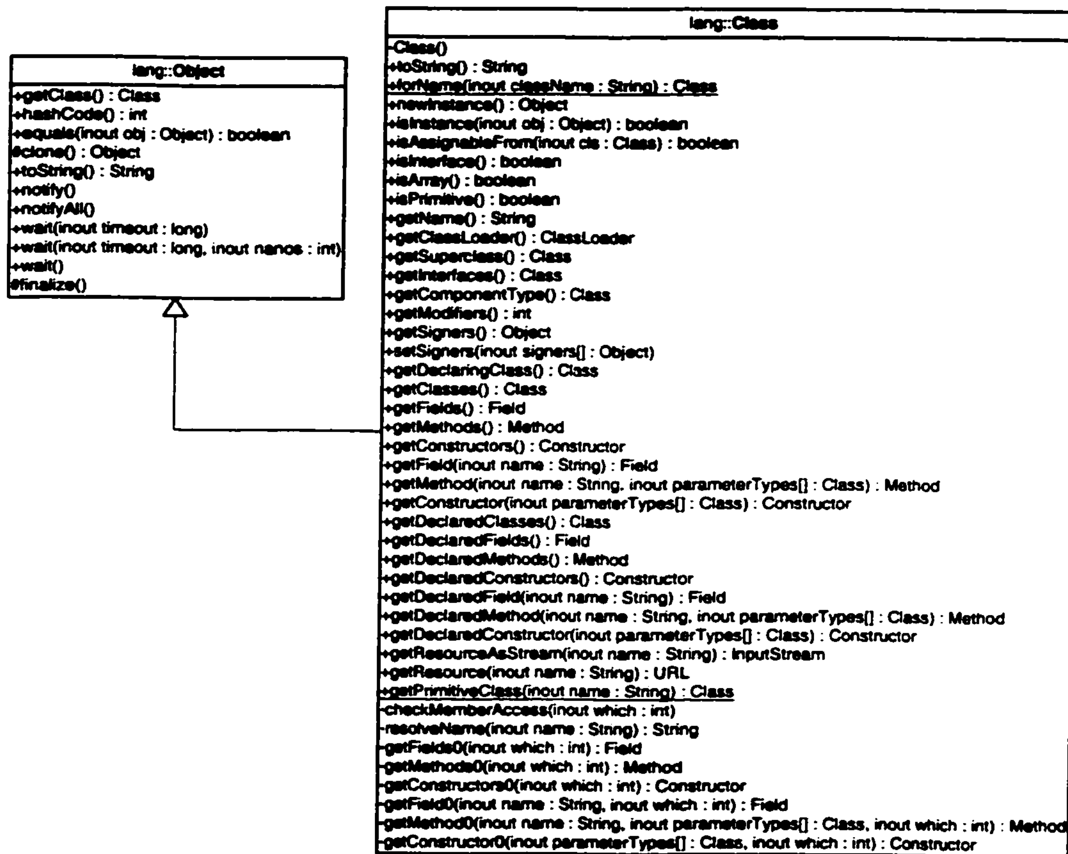

Figure 2: Class Diagram for Class java.lang.Class

This class represents a Java class or interface, or any Java type. There is one `Class` object for each class that is loaded into the Java Virtual Machine, and there are special `Class` objects that represent the Java primitive types. `Array` types are also represented by `Class` objects. There is no constructor for this class. The designer can obtain a `Class` object by calling the *getClass* method of any instance of the desired class. Figure 2 shows the members declared in the `java.lang.Class`.

- `java.lang.reflect` package
  The Java reflection model defines three final classes `Field`, `Method`, and `Constructor`. Only the Java Virtual Machine can create instances of these classes. These instances(objects) are used to manipulate the underlying object, that is, to:
  - get reflection information about the underlying member or constructor;
  - get and set field value;
  - invoke methods on objects or classes;
  - create new instances of classes.

Figure 3 shows the class diagram for the classes defined in the `java.lang.reflect` package.

2.1.4.2 Obtaining Class Information

To find out information about a class or interface, the designer first needs to get the class's `java.lang.Class` object and then query it for the desired information [78].

The following shows some important methods defined in class `Java.Lang.Class`. These methods are frequently used in the thesis work.

isAssignableFrom

```
public boolean isAssignableFrom(Class fromClass)
```

This method tests whether the type represented by the specified `Class` parameter can be converted to the type represented by this `Class` object via an identity conversion or a widening reference conversion. It returns true if so, false otherwise. If

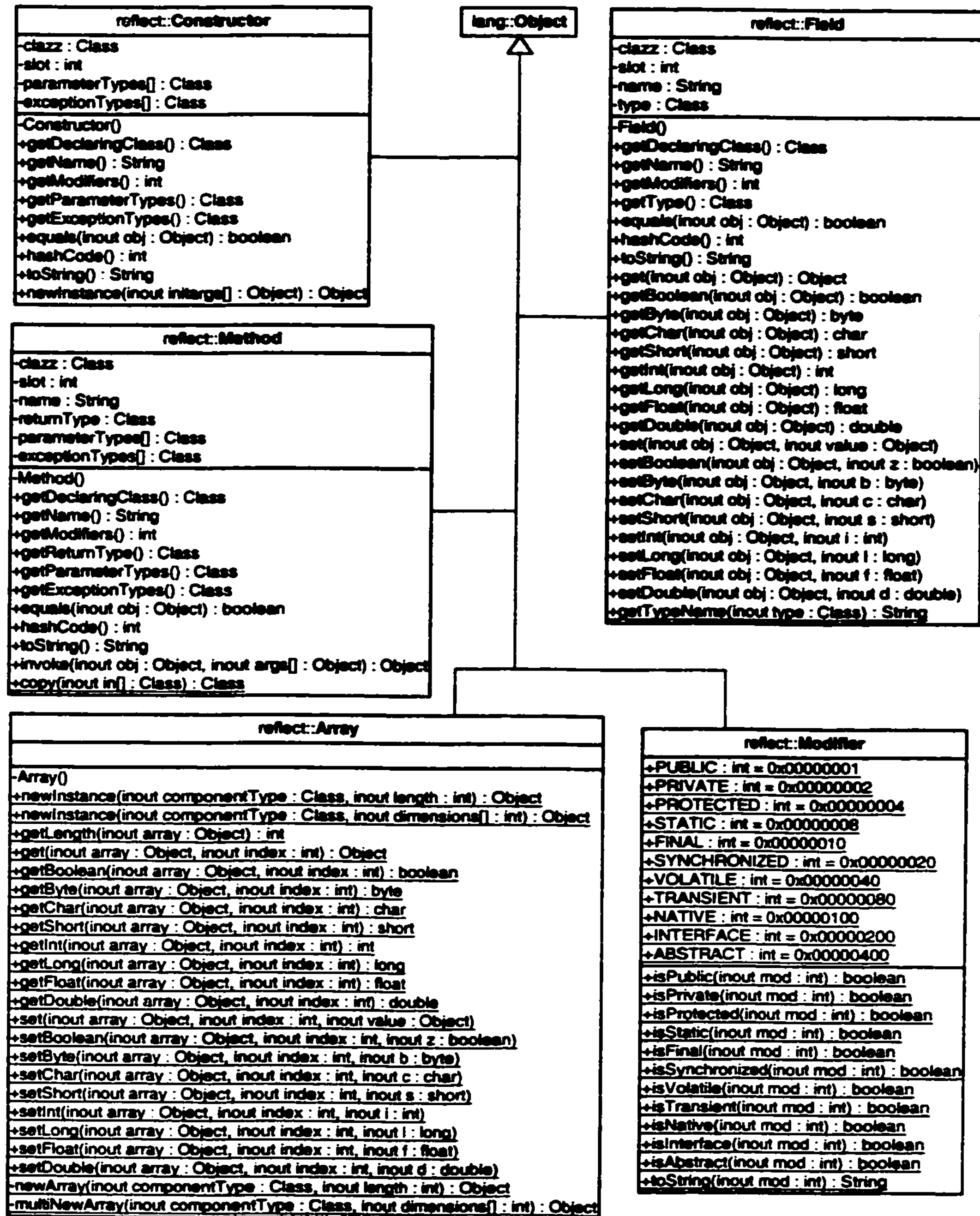

Figure 3: Class Diagram for Package java.lang.reflect

this `Class` object represents a primitive type, the method returns true if the specified `Class` parameter is exactly this `Class` object. Otherwise it returns false.

This method throws a `nullPointerException` if the specified `Class` parameter is null.

getSuperClass

```
public Class getSuperClass()
```

If this `Class` object represents a class other than `Object`, the method returns the `Class` that represents the superclass of the class. It returns "null" if this `Class` represents the class `Object` or if it represents an interface type or a primitive type.

getDeclaredMethod

```
public Method getDeclaredMethod(String name, Class[]paramterTypes)
        throws NoSuchMethodException, SecurityException
```

Returns a `Method` object that reflects the specified declared method of the class or interface represented by this `Class` object. The *name* parameter is a string that specifies the simple name of the desired method, and the *parameterTypes* parameter is an array of `Class` objects that identify the method's formal parameter types, in declared order.

The method throws a `NoSuchMethodException` if a matching method is not found. The method throws a `SecurityException` if access to the underlying method is denied.

forName

```
public static Class forName(String className)
        throws ClassNoFoundException
```

Given the fully-qualified name for a class, this method attempts to locate, load, and link the specified class. If it succeeds, it returns the `Class` object representing the class. Otherwise, it throws a `ClassNoFoundException`.

`Class` objects for array types may be obtained by this method. These `Class` objects are automatically constructed by the Java Virtual Machine. `Class` objects that represent the primitive Java types or *void* cannot be obtained by this method.

2.1.4.3 Invoking Methods

With reflection, the designer can construct objects and invoke methods at run time without performing compile-time type checking.

```
public Object invoke(Object obj, Object[] args)
        throws NullPointerException, IllegalArgumentException,
            IllegalAccessExcption, InvocationTargetException
```

The method invokes the underlying method represented by this method object on the specified object with the specified parameters. Individual parameters are automatically unwrapped to match primitive formal parameters. Both primitive and reference parameters are subject to widening conversions as necessary. The value returned by the underlying method is automatically wrapped in an object if it has a primitive type.

2.1.5 Metaclass and Dynamic Object Model

In object-oriented programming, objects are created as instances of classes. Ira R. Forman and Scott H. Danforth systematically present a theory in their book "Putting Metaclasses to Work" [33] on how to construct a reflective class-based model. In this object model, classes are themselves objects. That is, classes are created as instances of other classes.

In this section, we briefly introduce the theory with a simplified object model, which is based on a single inheritance system with only one supported metaclass. First, we will explain some fundamental concepts in the reflective object model. Followings are the introductions of structures of some major elements in the object model.

2.1.5.1 Fundamental Concepts

Object Reference

Each member of the nonempty, finite set of objects is identified by a value called an object reference.

Class Reference

Each object has a uniquely associated entity called a class. Each class is an object. In other words,

$\{\text{Set of Classes}\} \subset \{\text{Set of Objects}\}$

Hence, class reference indicates the object reference of a class.

Metaclass

Since a class is an object, a class must have a class, called metaclass. A metaclass is an object whose instances are classes. In a single metaclass object model, there is only one metaclass named `Class`. The metaclass `Class` is a class and it has a class that is itself, namely, `Class`. In other words,

$\text{MetaClass} \in \{\text{Set of Classes}\}$

2.1.5.2 Data Table

Data table contains the attribute information defined in a class. Since objects of a class need these attribute information to illustrate their states, the data table is also used in illustrating the structure of an object reference. The content of a data table for an object is determined by its class.

A data table can be constructed as a dictionary, where keys are class references and slot values are dictionaries in which keys are field names (strings). The general structure of a data table can be represented as:

```
{ classRef = { "field1" = field1_initValue;
               "field2" = field2_initValue;
               ... ... }
  ... ...
}
```

2.1.5.3 Method Table

Method table is used to contain the method information defined in a class. The methods to which an object responds are determined by its class.

The method table is constructed as a dictionary, where keys are class references and slot values are dictionaries in which keys are method names (strings) and values are code pointers. The code pointer identifies an executable procedure. It is a value

that can be used to start the execution of a procedure. The general structure of a method table can be represented as:

```
{ classRef = { "method1" = method1_CodePtr;
               "method2" = method2_CodePtr;
               ... ... }
  ... ...
}
```

2.1.5.4 Object Reference

The structure of an object reference includes a reference to its class and state information of the attributes that are declared in the class. An object is constructed as a dictionary merged from a slot named class (whose value is the class reference of the object) and a data table. That is:

{ "class" = classRef } $\lhd$ DataTable

The symbol "$\lhd$" means that two dictionaries merge into one. The data table (DataTable) defines the values of the attributes in the class and the values of the attributes inherited from its ancestor classes.

The general structure of an object reference can be represented as:

```
{ "class" = classRef;
  classRef      = { "field1" = field1_initValue;
                    "field2" = field2_initValue;
                    ... ... }
  superClassRef = { "superField1" = super1_initValue;
                    "superField2" = super2_initValue;
                    ... ... }
  ... ...
}
```

For example, an object **aString** of the class **String** can be represented as:

```
{ "class" = String;
  String  = { "_mnemonic" = "DEFAULT";  }
}
```

2.1.5.5 Class Reference

In a single metaclass system, each class object has a unique metaclass: `Class`. The structure of a class reference is determined by the attributes of its metaclass `Class`. The basic structure of a class reference looks like:

```
{ "class" = Class;
   Class = { "ivdefs" = { ... ... }
             "mdefs"  = { ... ... }
             "parent" = superClassRef;
             "ivs"    = { ... ... }
             "mtab"   = { ... ... }
           }
}
```

In this structure, there are two data tables for the definition of the attributes in a class: one is *ivdefs*, the other is *ivs*. These two data tables contain the attribute declarations and initial values of these attributes in the class. The *ivdefs* contains the information of all attributes that are declared in the class. The *ivs* contains the information of attributes that are declared in the class and attributes that are inherited from the its ancestor classes. The attribute information contained in the *ivdefs* is a subset of that in the *ivs*. The attribute information in a class reference can be constructed as:

```
{"class" = Class;
 Class = {"ivdefs" = { classRef={"field1" = field1_initValue;
                                 "field2" = field2_initValue;
                                 ... ... }
          "ivs" = { classRef = { "field1" = field1_initValue;
                                 "field2" = field2_initValue;
                                 ... ... }
                    superClassRef={"superField1" = super1_initValue;
                                   "superField2" = super1_initValue;
                                   ... ... }
                    ... ... }
          ... ... }
```

}

The *mdefs* and *mtab* define the method tables for the definition of the methods in a class. Both *mdefs* and *mtab* are attributes determined by the metaclass `Class`. The *mdefs* contains the information of all methods that are declared in the class including the methods that override that of its ancestor classes. The *mtab* contains all the method information that are declared in the *mdefs* and the methods that are inherited from its ancestor classes. The behavior information contained in the *mdefs* is a subset of that in the *mtab*.

The behavior information in a class structure can be constructed as:

```
{"class" = Class;
 Class={"mdefs"={classRef = { "method1" = method1_CodePtr;
                              "method2" = method2_CodePtr;
                              ... ... }
                 superClassRef={"overridedMethod1" = overrided1_CodePtr;
                                "overridedMethod2" = overrided2_CodePtr;
                                ... ... }
                 }
        "mtab"={classRef = { "method1" = method1_CodePtr;
                             "method2" = method2_CodePtr;
                             ... ... }
                superClassRef={"superMethod1" = method1_CodePtr;
                               "superMethod2" = method2_CodePtr;
                               "overridedMethod1" = overrided1_CodePtr;
                               "overridedMethod2" = overrided2_CodePtr;
                               ... ... }
                ... ... }
        ... ... }
}
```

The *parent* contains a reference to its superclass.

2.1.5.6 Metaclass

All classes are instances of the metaclass `Class`, and the metaclass `Class` is also a class itself. In a single inheritance system, the root class `Object` is the superclass of

the metaclass `Class`.

As mentioned before, the **ivdefs**, **ivs**, **mdefs**, **mtab**, and *parent* determine the structure of a class. They are all attributes of the metaclass Class. The methods declared in the metaclass `Class` construct the metaobject protocol of the system.

The basic structure of the metaclass `class` can be constructed as:

```
{ "class" = Class;
   Class = { "ivdefs" = { Class = { "ivdefs" = { }
                                    "mdefs"  = { }
                                    "parent" = { }
                                    "ivs"    = { }
                                    "mtab"   = { }  }
             "ivs"    = { Class = { "ivdefs" = { }
                                    "mdefs"  = { }
                                    "parent" = { }
                                    "ivs"    = { }
                                    "mtab"   = { }  }
             "mdefs"  = { Class = { "newInstance" = newInstance_CodePtr;
                                    ... ... }  }
             "mtab"   = { Class = { "newInstance" = newInstance_CodePtr;
                                    ... ... }
                          Object = {... ... }  }
             "parent" = Object;
           }
}
```

In the following sections, we will introduce some software design concepts including pattern, design pattern, pattern language, and framework.

2.2 Pattern

2.2.1 What is a Pattern ?

Patterns have become a popular way to reuse software design in the object-oriented community. The goal of patterns within the software community is to help software

developers resolve common difficult problems encountered throughout all of software engineering and development [8].

A pattern captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces [63]. A pattern is an essay that describes a problem to be solved, a solution, and the context in which that solution works [41]. A pattern involves a general description of a recurring solution to a recurring problem replete with various goals and constraints. But a pattern does more than just identify a solution, it also explains why the solution is needed [17].

In the spirit of Alexander's patterns, software patterns make up for lapses in the memory of the contemporary software design culture, and capture structure not immediately apparent from the code or from most system design documents [17]:

- Patterns capture obscure but important practice: Patterns work at many levels of detail. They capture established practices that remain obscure in the broad practices of a given domain. A pattern is abstract because it approaches the problem at a suitable general level, although the solution may entail detail.

- Patterns capture hidden structure: Patterns cut across the predominant partitionings of the subject area. Good software patterns address system problems and relationships that are obscured by a perspective from inside any of the parts. Patterns complement object-oriented design methods to capture the important constructs that cut across objects.

Patterns are usually concerned with some kinds of architecture or organization of constituent parts to produce a greater whole. Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves [36]. A good pattern will do the following [16]:

- It solves a problem: Patterns capture solutions, not just abstract principles or strategies.

- It is a proven solution: Patterns capture solutions with a track record, not theories or speculation.

- The solution is not obvious: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns generate a solution to a problem indirectly – a necessary approach for the most difficult problems of design.
- It describes a relationship: Patterns do not just describe modules, but describe deeper system structures and mechanisms.

In general, a software pattern can be categorized as an architectural pattern, an analysis pattern, a design pattern, or a programming pattern.

- **Analysis Pattern**
  Analysis patterns are groups of concepts that represent a common construction in application domain modeling. Analysis patterns involve looking behind the surface requirements to come up with a conceptual model of what is going on in the problem [34].
- **Architectural Pattern**
  An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them [13, 19].
- **Design Pattern**
  A design pattern is a pattern whose form is described by means of software design constructs, for example objects, classes, inheritance, aggregation and use-relationship [63]. A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context [13].
- **Programming Pattern (Coding Pattern, Idiom)**
  A programming pattern is a pattern whose form is described by means of programming language constructs [63]. It is a low-level pattern specific to a programming language. A programming pattern describes how to implement particular aspects of components or the relationships between them using the features of the given language [13].

Analysis patterns are based upon metaphors in a restricted application domain. Architectural patterns are high-level strategies that concern large-scale components and the global properties and mechanisms of a system. They have wide-sweeping implications which affect the overall skeletal structure and organization of a software system. Design patterns are medium-scale tactics that flesh out some of the structure and behavior of entities and their relationships. They do not influence overall system structure, but instead define micro-architectures of subsystems and components. Programming patterns are paradigm-specific and language-specific programming techniques that fill in low-level internal or external details of a component's structure or behavior [63].

2.2.2 History of Patterns

The term "pattern" is derived from the writings of the architect Christopher Alexander who has written several books on the topic as it relates to urban planning and building architecture [1, 2, 4, 3].

Patterns have been used for many different domains. Software patterns first became popular with the wide acceptance of the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (frequently referred to as the Gang of Four or just GoF) [37]. At present, the software community is using patterns largely for software architecture and design, and (more recently) software development processes and organizations. Each year, new patterns are published in the conference series of "Patterns Languages of Program Design" (currently this conference series includes PLoP, EuroPLoP, ChilliPLoP, and KoalaPLoP). Pattern books "Pattern Languages of Program Design" [18, 76, 55, 40] contain selected papers from conferences on Patterns Languages of Program Design (PLoP).

2.2.3 Qualities of Patterns

A well written pattern should exhibit several desirable qualities. Ideally, pattern entries have the following properties [47, 8]:

- **Encapsulation**
  Each pattern encapsulates a well-defined problem/solution. Patterns are independent, specific, and precisely formulated enough to make clear when they apply and whether they capture real problems and issues. Patterns should provide crisp, clear boundaries that help crystallize the problem space and the solution space by parceling them into a lattice of distinct, interconnected fragments.

- **Generativity**
  Each entry contains a local, self-standing process prescription describing how to construct realizations. Pattern entries are written to be usable by all development participants, not merely trained designers.

- **Equilibrium**
  Each pattern must realize some kind of balance among its forces and constraints. This may be due to one or more invariants or heuristics that are used to minimize conflict within the solution space. The invariants often typify an underlying problem solving principle or philosophy for the particular domain, and provide a rationale for each step/rule in the pattern. The aim is that each pattern describes a whole that is greater than the sum of its parts, due to skillful choreography of its elements working together to satisfies all its varying demands.

- **Abstraction**
  Patterns represent abstractions of empirical experience and everyday knowledge. They are general within the stated context. They serve as abstractions which embody domain knowledge and experience, and may occur at varying hierarchical levels of conceptual granularity within the domain.

- **Openness and Variability**
  Each pattern should be open for extension or parametrization by other patterns so that they may work together to solve a larger problem. A pattern solution should be also capable of being realized by an infinite variety of implementations (in isolation, as well as in conjunction with other patterns).

- **Composibility**
  Patterns are hierarchically related. Coarse grained patterns are layered on top

of, relate, and constrain fine grained ones. Pattern entries are arranged conceptually as a language that expresses this layering. Because the forms of patterns and their relations to others are only loosely constrained and written entirely in natural language, the pattern language is merely analogous to a formal production system language, but has about the same properties, including infinite nondeterministic generativity.

2.3 Design Pattern

2.3.1 What is a Design Pattern ?

A design pattern is a reusable implementation model or architecture that can be applied to solve a particular recurring class of problem. Each pattern focuses on a particular object-oriented design problem or issue. It captures expertise and make it accessible to non experts [74]. Design patterns allow software designers to learn from and apply the experience of other designers. Hence, software designers can apply their experience with past problems and solutions to new, similar problem [16].

More specifically, design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. They describes how methods in a single class or sub-hierarchy of classes work together. It also shows how multiple classes and their instances collaborate [37].

Design pattern makes it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns make them more accessible to developers of new systems. Design patterns help us choose design alternatives that make a system reusable and avoid alternatives that compromise reusability.

Design patterns have a different emphasis than most reuse programs or design catalogues: they tend to capture broader abstraction. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design "right" faster [37].

2.3.2 Document a Design Pattern

The goal of design pattern is to capture design experience in a form that people can use effectively. The design pattern identifies the participating classes and instances, their roles and collaborations and the distribution of responsibilities. It describes when it applies, whether it can be applied in view of other design constraints and the consequences and trade-offs of its use.

Alexander says that "every pattern we define must be formulated in the form of a rule which establishes a relationship between a context, a system of forces which arises in that context, and a configuration, which allows these forces to resolve themselves in that context." [3, 4]. Several different formats have been used for describing patterns. The format used in the book "Design Pattern" [37] is referred to as "GoF format". The following essential components should be clearly recognizable upon reading a pattern [37]:

- **Pattern Name and Classification**
  The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of our design vocabulary.

- **Intent**
  A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

- **Also Known As**
  Other well-known names for the pattern, if any.

- **Motivation**
  A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help us understand the more abstract description of the pattern that follows.

- **Applicability**
  What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can we recognize these situations?

- **Structure**
  A graphical representation of the classes in the pattern using a notation based on the Unified Modeling Language (UML) [12] [65].

- **Participants**
  The classes and/or objects participating in the design pattern and their responsibilities.

- **Collaborations**
  How the participants collaborate to carry out their responsibilities.

- **Consequences**
  How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

- **Implementation**
  What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

- **Sample Code**
  Code fragments that illustrate how you might implement the pattern.

- **Known Uses**
  Examples of the pattern found in real systems.

- **Related Patterns**
  What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

2.4 Pattern Language

2.4.1 What is a Pattern Language ?

A pattern language is a collection of patterns that build on each other to generate a system. The term "pattern language" comes from building architecture and has been popularized by Alexander [4]. A pattern language is a piece of literature that

describes an architecture, a design, a framework, or other structure. It has structure, but not the same level of formal structure the one finds in programming language [17].

A pattern in isolation solves an isolated design problem, but a pattern language builds a system. Pattern languages place individual patterns in context. Generally, a pattern language is applied under the context that the developers are to trying to use the "pattern form" to describe a procedure with many steps or a complex solution to a complex problem. Some of the steps may only apply in particular circumstances. There may be alternate solutions to parts of the problem depending on the circumstances. A single pattern is insufficient to deal with the complexity at hand. To easily digest and use parts of the solution in different circumstance, a pattern language factors the overall problem and its complex solution or procedure into a number of related problems with their respective solutions. The pattern language captures each problem/solution pair as a pattern, which solves a specific problem within the shared context of the language [56].

2.4.2 Document a Pattern Language

A pattern language consists of a list of patterns, where each pattern can be organized in a structure [56] as follows. The reason why we use different formats to document a pattern in a pattern language rather than that of an individual pattern structure in Section 2.3.2 is that, the patterns in a pattern language exist in the global problem context and cooperate with other related patterns.

- **Name**
  A name by which the problem/solution pair can be referenced. Good pattern names form a vocabulary for discussing conceptual abstractions.

- **Aliases**
  Other names by which this pattern might be known.

- **Context**
  The circumstance in which the problem is being solved imposes constraints on the solution. It implies the pattern's applicability. It can be thought of as the initial configuration of the system before the pattern is applied to it.

- **Problem**

  The specific problem that needs to be solved. It describes the intent: the goals and objectives it wants to reach within the given context and forces.

- **Forces**

  The often contradictory considerations that must be taken into account when choosing a solution to a problem. Forces reveal the intricacies of a problem and define the kinds of trade-offs that must be considered in the presence of the tension or dissonance they create. A good pattern description should fully encapsulate all the forces which have an impact upon it.

- **Solution**

  The proposed solution to the problem. The most appropriate solution to a problem is the one that best resolves the highest priority forces as determined by the particular context. The description may encompass diagrams and prose which identify the pattern's structure, its participants and their collaborations, to show how the problem is solved.

- **Rationale**

  A justifying explanation of steps or rules in the pattern, and also of the pattern as a whole in terms of how and why it resolves its forces in a particular way to be in alignment with desired goals, principles, and philosophies. It explains how the forces and constraints are orchestrated in concert to achieve a resonant harmony. This tells us why it works, and why it is "good". The solution component of a pattern may describe the outwardly visible structure and behavior of the pattern, while the rationale provides insight into the deep structures and key mechanisms that are beneath the surface of the system.

- **Code Samples**

  Sample code shows how to implement the pattern. Examples help the reader understand the pattern's usage and its applicability. Visual examples and analogies can often be especially illuminating. An example may be supplemented with a sample implementation to show one way the solution might be realized.

- **Resulting Context**

  Resulting context refers to the state or configuration of the system after the

pattern has been applied. It includes the consequences (both the goodness and the badness) of applying the pattern and new problems and new patterns that may arise in this context. It describes the postconditions and side-effects of the pattern. It also describes which forces have been resolved, which ones remain unresolved, and which patterns may now be applicable.

- **Related Patterns**
  The static and dynamic relationships between this pattern and others within the same pattern language or system. Related patterns often share common forces. They also frequently have an initial or resulting context that is compatible with the resulting or initial context of another pattern. Such patterns might be predecessor patterns whose application leads to this pattern; successor patterns whose application follows from this pattern; alternative patterns that describe a different solution to the same problem but under different forces and constraints; and codependent patterns that may (or must) be applied simultaneously with this pattern.

- **Known Uses**
  Describes known occurrences of the pattern and its application within existing systems. This helps to validate a pattern by verifying that it is indeed a proven solution to a recurring problem.

2.5 Framework

2.5.1 What is a Framework ?

A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact. It is the skeleton of an application that can be customized by an application developer.

Frameworks are an object-oriented reuse technique. They take advantage of all three of the distinguishing characteristics of object-oriented programming: data abstraction, polymorphism, and inheritance. Like an abstract data type, an abstract class represents an interface behind which implementations can change. Polymorphism is the ability for a single variable or procedure parameter to take on values of

several types. Object-oriented polymorphism lets a developer mix and match components, lets an object change its collaborators at run-time, and makes it possible to build generic objects that can work with a wide range of components. Inheritance makes it easy to make a new component.

A framework describes the architecture of an object-oriented system; the kinds of objects in it and how they interact. It focuses on how a particular kind of program is decomposed into a set of interacting objects [42].

A framework is characterized by [62]:

- Partial design;
- Incomplete implementation;
- Inversion of system control, and it contains the part of control that invokes the methods supplied by the user;
- Reuse arises in all stages of system analysis, design and implementation.

An application developed from a framework can be identified by several different parts as shown graphically in Figure 4 [81, 35].

- Framework Core: The core of the framework defines the generic structure and behavior of the framework, and forms the basis for the application developed from the framework. It generally consists of abstract classes, but can also contain concrete classes that are meant to be used as is in all applications built from the framework.
- Framework Library: Extensions to the framework core consisting of concrete components that can be used with little or no modification by applications developed from the framework.
- Application Extensions: Application specific extensions made to the framework.
- Application: in terms of the framework, the application consists of the framework core, the used framework library extensions, and any application specific extensions needed.
- Unused Library classes: Typically, not all of the classes within a framework will be needed in an application that can be developed from the framework.

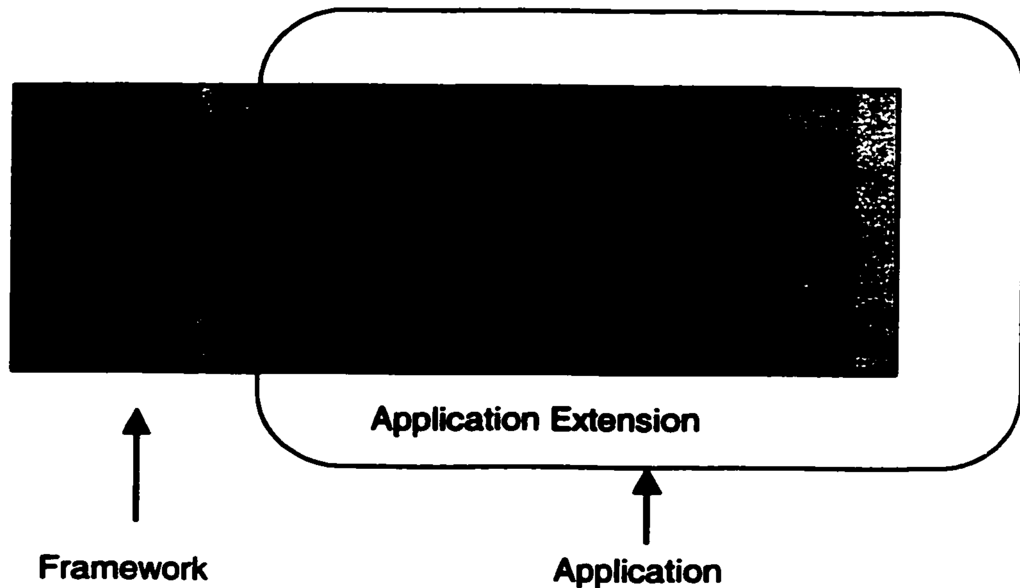

Figure 4: Application Developed from a Framework

Roberts and Johnson present the framework's development process as a pattern language [64], which includes nine patterns: THREE EXAMPLES, WHITE-BOX FRAMEWORK, COMPONENT LIBRARY, HOT SPOTS, PLUGGABLE OBJECTS, FINE-GAINED OBJECT, BLACK-BOX FRAMEWORK, VISUAL BUILDER, and LANGUAGE TOOLS.

2.5.2 Framework vs. Software Pattern

A framework describes how a system behaves. A framework can be considered to be a type of a pattern. In fact, a framework can be viewed as an architectural pattern that offers an extensible template [12].

The design patterns are closely related to frameworks. Design patterns were discovered by examining a number of frameworks, and were chosen as being representative of reusable, object-oriented software. In general, a framework may contain many design patterns, but a design pattern does not contain frameworks. Moreover, frameworks are more specialized than design patterns. Frameworks are built for a

particular application domain, and describe an application architecture for that domain. Frameworks are at a different level of abstraction than design patterns. A framework is a pattern arising at the system architectural level and design patterns are the architectural elements of frameworks [42].

A framework can be designed and documented in terms of patterns, where each pattern describes how to solve a small part of the larger design problem. Each pattern describes a problem that occurs over and over again in the problem domain of the framework, and then describes how to solve that problem. Thus, the idea is that someone has solved a specific problem once and documented the solution as a pattern, we can use this information in order to solve same kinds of problems [41].

2.5.3 Develop a Framework

A framework may begin as an application that evolves to a framework, and other applications are developed to confirm the reusability of this framework before it is rolled out for general use [14].

A framework evolves over time. Uses of a framework may expose some insufficiency and incompleteness in the framework design. The framework is then refined to accommodate the new raised issues and the old ones. A framework evolves as a wider application domain is covered, hot spots [62] are more precisely identified, customization is concisely specified, and all the jargons are clearly defined.

The major steps in developing an application framework can be summarized as [43, 72]:

1. Identify and analyze the application domain and identify the framework. If the application domain is large, it should be decomposed into a set of possible frameworks that can be used to build a solution. Analyze existing software solutions to identify their commonality and the differences.

2. Identify the primary abstractions. Clarify the role and responsibility of each abstraction. Design the main communication protocols between the primary abstractions. Document them clearly and precisely.

3. Design how a user interacts with the framework. Provide concrete examples of the user interaction, and provide a main program illustrating how the abstract classes are related to each other and to the classes for user interaction.

4. Implement, test, and maintain the design.

5. Iterate with new applications in the same domain.

The design and implementation of frameworks relies heavily on abstract classes, inheritance, and polymorphism.

2.6 Summary

This chapter introduces the basic concepts for reflection, design pattern, pattern language, and framework. These concepts closely relate to our research.

The following chapters presents the thesis work. Chapter 3 presents a REFLECTIVE VISITOR, an improved VISITOR variant based on the reflection technique. Chapter 4 is the design of a parser with the reflection pattern. We will show how can REFLECTION pattern benefit the object-oriented design of a predictive recursive-descent parser. Chapter 5 presents a framework of the reflective class-based object model. Chapter 6 is a customization example of the frameworks presented in the previous chapters.

Chapter 3

Reflective Visitor: To Extend and Reuse Object Structure

The VISITOR pattern [37] wraps associated operations that are performed on the elements of an object structure into a separate object. It allows the software designer to define new kinds of operations over the object structure without changing the classes of this structure. But a well-known drawback of the standard visitor structure is that extending the object structure is hard.

Since Gamma et al. first published the VISITOR design pattern [37] in 1995, there have been proposed several variations in the design pattern literature. In this chapter, we first present the design and implementation of a new variation of VISITOR pattern based on the reflection technique, we call it REFLECTIVE VISITOR [53]. This VISITOR pattern allows the software developer to extend an object structure and define new operations over this object structure without modifying the existing system.

This chapter also presents a pattern language to variations of VISITORs [52] to assist the application developer to choose the right VISITOR pattern that best suites the intended purpose by enumerating all important forces and consequences for each variation.

Section 3.1 introduces the terms and concepts used in this chapter. Section 3.2 presents the design and implementation of the REFLECTIVE VISITOR pattern. Section 3.3 is a pattern language to VISITORs.

3.1 Terms and Concepts

Visitor

A visitor implements behaviors for traversing a set of element objects and assigning responsibilities to these elements. It encapsulates operations to be performed over these elements and wraps them in a class separated from these elements. In general, there are two hierarchies to be defined when a visitor is implemented. One is the element hierarchy representing the objects that will be visited. The other is the visitor hierarchy representing operations to be performed on the elements. The use of visitor lets the developer easily change the behaviors that would otherwise be distributed across classes without modifying these classes.

Element

An element is a class whose instance belongs to a set of a fair number of instances of a small number of classes [16] that will be visited.

Object Structure

An object structure refers to a fair number of instances of a small number of classes [16]. Especially in a VISITOR pattern, an object structure contains a set of instances that will be visited.

Cyclic Dependency

A component of a system is said to depend on another component if the correctness of the first component's behavior requires the correct operation of the second component. A dependency relationship is said to be acyclic if it forms a tree. That is the set of possible dependencies in a system are considered to form an acyclic graph. It is possible, however, for a dependency relationship to cycle back upon itself. A cyclic dependency relationship is one that cannot be described as part of a tree, but rather must be described as part of a directed cyclic graph [39].

Element Adder

An Element Adder is a kind of solution that can easily add new elements to an element hierarchy without modifying the existing program.

Operation Adder

An Operation Adder is a kind of solution that can easily add new operations over an element hierarchy without modifying the existing program.

3.2 Reflective Visitor

The REFLECTIVE VISITOR is a new variant of the VISITOR pattern. Its design and implementation are based on the reflection technique. With the power of the reflection technique, the REFLECTIVE VISITOR pattern can extend an object structure and define new operations over this object structure in a much easier way without changing the existing system. The REFLECTIVE VISITOR pattern can be used in an environment that the implementation language supports reflection and the execution time is not a major concern.

Intent

Define a new operation over the object structure without changing the classes of the elements on which it operates, while in the meantime, allow the element classes in the object structure to be extended constantly without changing the existing system.

Also Know As

Easy Element and Operation Adder

Motivation

Consider the code generation design in a compiler framework. The responsibility of the code generation is to generate a target code list as the output of the compiler. The format of the target code list is specified by the system requirements, which may require the target code list to be compatible with different operation platforms. In

order to support the cross-platform features, different code generation operations need to be co-exist and allow easily switching from one to another. On the other hand, the code generation process have to depend on the parser result. The parser result, normally in terms of an abstract syntax tree, can be represented as a compound data structure, each of whose elements is constructed from the language structure. The language structure is a composite hierarchy consisting of a set of terminals and non-terminals, which can be extracted from the language grammar specification. Since the operation of code generation is actually performed on the abstract syntax tree, its design should accommodate any potential changes on the language structure. For a compiler framework, the design of the code generation needs to:

1. Prepare for changes of the generated code format.

2. Prepare for modifications in the language grammar.

3. Reduce the coupling between the language structure and the code generator in order to promote the system reusability.

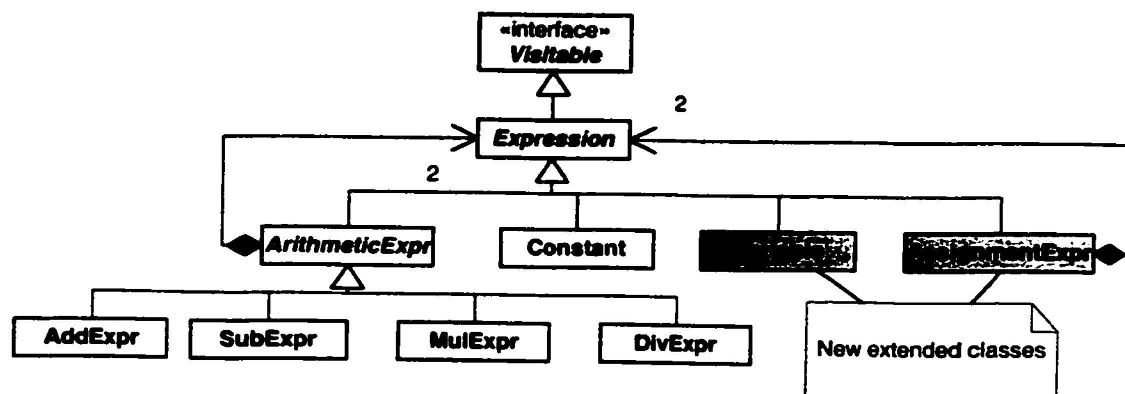

Figure 5: An Expression Hierarchy

Given a simple expression example, suppose it supports arithmetic expression such as addition, subtraction, multiplication, and division for constants. Figure 5 shows the language structure hierarchy for this expression example. The language (expression) structure hierarchy is organized as a composite structure and can be implemented by a COMPOSITE design pattern [37]. The abstract syntax tree therefore is represented

as a composite object, which is recursively constructed with the instances of the node classes in the expression structure during the parsing. The code generation process then performs the code generation operations over this abstract syntax tree.

Basically, there are two kinds of potential extensions to the above example: one is the changing of the expression structure, the other is generating different code formats. For example, the expression structure can be extended with supports of variable and assignment expression that will be used to assign the value or expression to the variable. As shown in Figure 5, they are represented as two extended classes in gray. On the other hand, the code generation may require target code to be generated in different code format according to the design requirements. It may also require easy switching from one format to another and easy addition of new kind of output code format. For simplicity, we suppose the code generation for above example should support the two different virtual machines, VM1 and VM2.

The basic design issues in the design of code generation for this expression example are:

1. Both the code generations for VM1 and VM2 should be represented as different operations that are performed on the abstract syntax tree.
2. The code generation should support easy switching between the VM1 output format and the VM2 output format. Any future extension of the output format can be easily added without modifying and re-compiling the existing system program.
3. The addition of the variable and assignment expression needs not affect the rest of the system.
4. The language (expression) structure can stand alone and has no knowledge about the code generator.

As Gamma et al. pointed out in their Design Patterns book [37], the VISITOR pattern is suitable to represent an operation to be performed on the elements of an object structure. We refer this VISITOR PATTERN as GOF VISITOR pattern in this thesis. The GOF VISITOR pattern lets the designer define a new operation over the object structure without changing the elements of that structure. Figure 6 shows the design of the above expression example with the GOF VISITOR pattern.

By applying GoF VISITOR pattern, the code generation can be easily changed or extended to support different kinds of output code formats, for example, switching between the VM1 output format and the VM2 output format. But we can also see that the extension of the expression structure becomes difficult. In our example, to support the addition of the variable and assignment expression, the GoF VISITOR pattern requires the new code generation methods, such as *visitVariable(Variable)* and *visitAssignmentExpr(AssignmentExpr)*, to be added across the visitor hierarchy (the `Visitor` interface the `VM1CodeGenVisitor` class and the `VM2CodeGenVisitor` class in Figure 6). Therefore, all classes in the visitor hierarchy need to be modified and re-compiled due to changing of the expression structure. Obviously, the GoF VISITOR pattern could not fit our system design requirement and it is not suitable for the code generation design in a compiler framework.

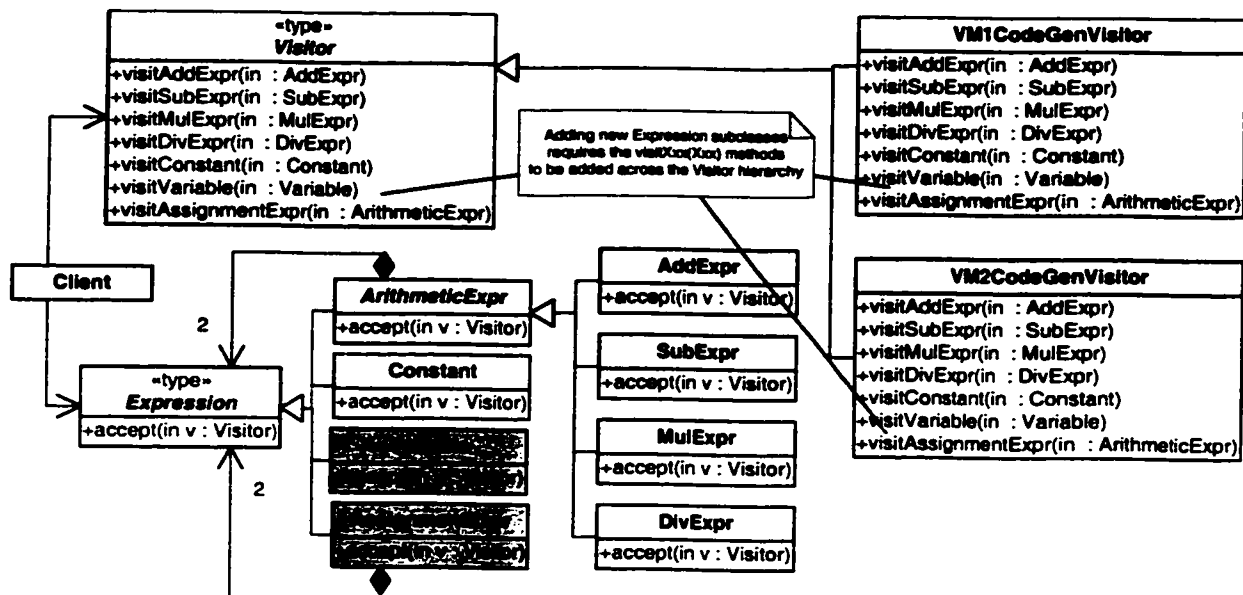

Figure 6: Apply GoF Visitor Pattern to the Expression Example

There are several variations of the VISITOR pattern [52] intended to overcome this shortcoming so that the VISITOR pattern can also be used in an environment that the object structure changes often. A brief summary of them is mentioned in the Related Patterns section of this section.

The REFLECTIVE VISITOR pattern introduced in this paper supports both the changes of the generated code format and the changes of the language grammar without changing the existing classes. It achieves this goal by performing the dynamic operation dispatch in the `Visitor` class through reflection. The `Visitor` class declares a *visit* method to be responsible for the dynamic dispatch. The corresponding operation can be invoked automatically at run-time. Therefore the *accept* methods are no longer needed and the cyclic dependencies are removed. This *visit* method is defined as the only interface visible to the outside of the system so that detailed implementation of code generation is hidden from the outside of the system.

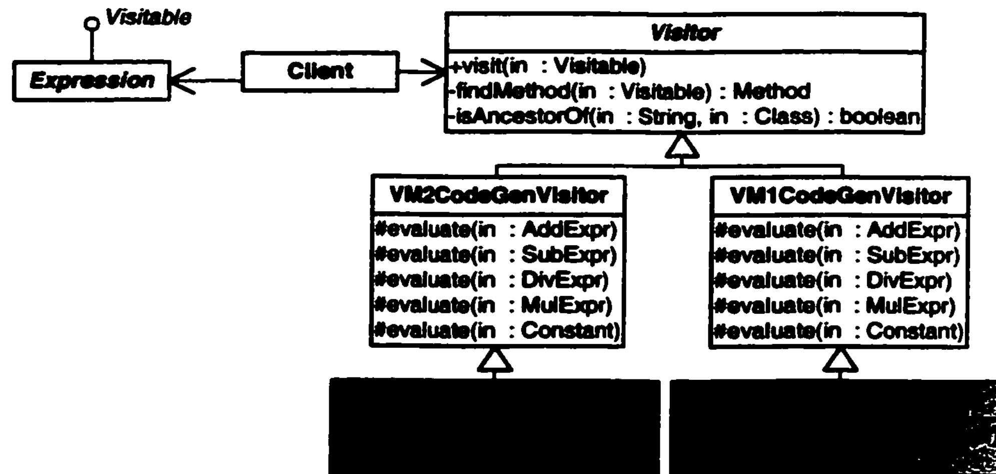

Figure 7: Apply Reflective Visitor Pattern to the Expression Example

Figure 7 shows the solution for the above expression example by applying the REFLECTIVE VISITOR pattern to the code generation. The *visit* method declared in the `Visitor` class takes the concrete `Expression` object as argument. It queries the concrete `Expression` class information through reflection to find the *evaluate* method based on the concrete `Expression` object and then invokes the *evaluate* method to perform the operation. In our example, to support two code generation formats for the virtual machine VM1 and VM2, we define two concrete visitors `VM1CodeGenVisitor` and `VM2CodeGenVistor` respectively. The addition of the variable and assignment expression in the expression structure only requires two new visitor classes (`ExtendVM1CodeGenVistor` and `ExtendVM2CodeGenVisitor`) to be added in the visitor hierarchy and their *evaluate* methods to be implemented. All existing

classes in both the expression structure and the visitor hierarchy need not to be modified and re-compiled. All the *evaluate* operations in the visitor hierarchy are declared protected so detailed implementation information of code generation is encapsulated.

With the REFLECTIVE VISITOR pattern, the system designer can easily add new operations to the object structure by simply defining new concrete `Visitor` classes, as what the GoF VISITOR pattern does. On the other hand, the designer can also easily add new concrete `Element` classes by simply defining new `Visitor` subclasses in the visitor hierarchy. The *visit* method is the only visible interface of the visitor hierarchy. The client only needs to invoke this method to perform any desired operation on the object structure. Since the interface and the implementation of the operations on the object structure are separated, the client is shielded from any potential changes of the implementation details.

Applicability

The REFLECTIVE VISITOR pattern can be applied when:

1. The programming language that the designer uses to implement the REFLECTIVE VISITOR design pattern should support reflection. For example, Java.
2. An object structure contains many classes of objects with differing interfaces, and the designer performs operations on these objects that depend on their concrete classes [37].
3. Distinct and unrelated operations need to be performed on objects in an object structure [37].
4. The object structure may be changed often to fit changing requirements. The designer do not want to redefine the interface and recompile all existing classes.
5. The designer may need to reuse the object structure in the future and thus wants to break the cyclic dependencies and de-couple the object structure and the visitor hierarchy.
6. The designer wants to define a unified stable operation interface for the client and to encapsulate the implementation details.
7. The run-time efficiency is not a major concern in the design.

Structure

Figure 8 shows the structure of the REFLECTIVE VISITOR design pattern.

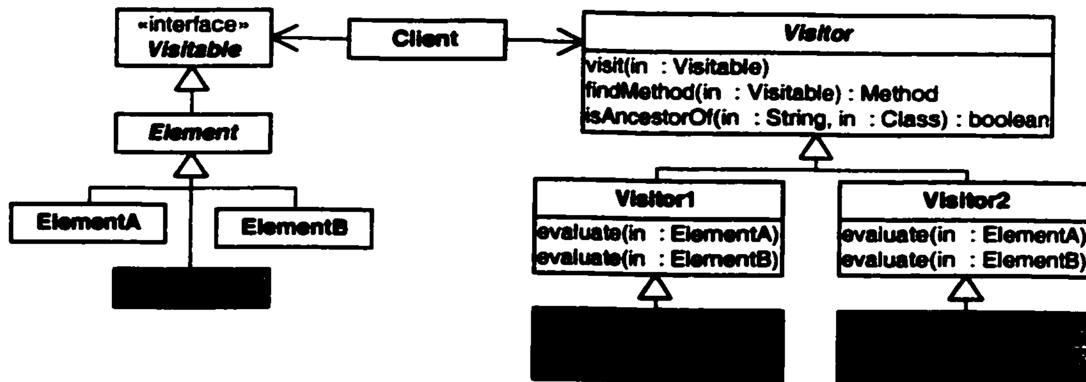

Figure 8: The Structure of the Reflective Visitor

Participants

Visitor (Visitor)

1. The abstract class `Visitor` is the facade and the root of the visitor class hierarchy. All the concrete `Visitor` classes are derived from it.

2. The `Visitor` class defines a public *visit* operation, which is the unified operation interface for the `Visitor` class. The client invokes the *visit* method to execute the corresponding operations on the object structure.

3. The *visit* method takes a `Visitable` interface object as argument. It performs the dynamic dispatch for the concrete `Element` object. That is, the *visit* method finds the corresponding concrete *evaluate* operation from the `Visitor` hierarchy and invokes it at run time.

Visitor1 (VM1CodeGenVisitor, VM2CodeGenVisitor)

1. The `Visitor1` defines a set of *evaluate* operations, each implements the specific behavior for the corresponding concrete `Element` class.

2. The *evaluate* operations are declared as protected so that the implementation information can be hidden from the outside of the system.

Visitable

The interface `Visitable` is the interface for all the classes that can be visited. It is an empty interface and provides the run time type information for the `Visitor`.

Element (Expression)

The class `Element` is the root of the element class hierarchy to be visited. It implements the `Visitable` interface. All concrete `Element` classes derive from the `Element` class and they have no knowledge about the `Visitor`.

ElementA (AddExpr, SubExpr, MulExpr, DivExpr)

The class `ElementA` is a descendant of the `Element` class. The `Element` class and all the concrete `Element` classes construct the element class hierarchy.

Collaborations

A client who uses the Visitor pattern must create a concrete `Visitor` object (e.g. `Visitor1`) and pass the concrete `Element` object (e.g. `ElementA`) to the `Visitor` for visiting.

The `Visitor` uses reflection to query the `ElementA` class information and finds the corresponding *evaluate* method whose argument type is same as that of the `ElementA`. The search process begins from the `Visitor1` class, and then traces up its ancestors until it reaches the root of the visitor hierarchy. If the method is found, it is invoked. Otherwise, we assume that this *evaluate* method is defined for the ancestor classes of the `ElementA`, so the search process repeats for these ancestors. If all the ancestors of the `ElementA` have been tried and the corresponding *evaluate* method can not be found, an error is thrown. Figure 9 is the sequence diagram of the *visit* method.

Consequences

Some benefits of the REFLECTIVE VISITOR pattern are:

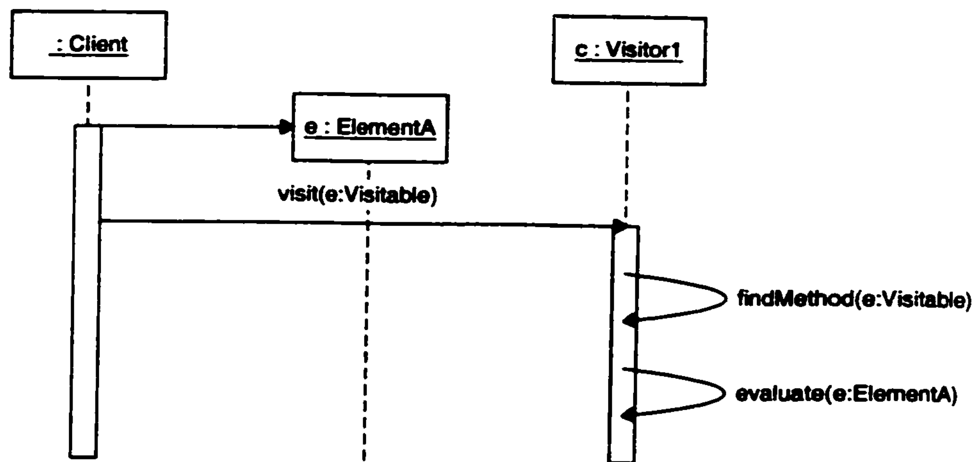

Figure 9: The Sequence Diagram for the Visiting Process

1. As that of the GoF VISITOR pattern, adding a new operation is easy. The existing code can be avoided from modifying by simply subclassing the visitor hierarchy if a new operation over an object structure is to be added.

2. Adding a new element class `ElementC` is easy. Since the `Visitor` is responsible for the dynamic dispatch, any operation operating on this new `ElementC` can be defined within a new subclass of the `Visitor1` without modifying the existing codes. The system's extensibility is then improved.

3. The cyclic dependencies are broken and the coupling between the object structure and the visitor hierarchy is reduced. As the key of the standard VISITOR pattern, the double-dispatch technique is used to bind the operation with the concrete element in the object structure at run time. But this technique reduces the system's reusability. With the reflection technique, the REFLECTIVE VISITOR pattern can avoid the cyclic dependencies by performing the dynamic dispatch within the Visitor class. Since the `Visitor` is responsible for the dynamic dispatch, the element hierarchy has no knowledge about the visitor. Hence the system's reusability is improved. On the other hand, the visitor can visit any object that has a corresponding *evaluate* operation in the visitor hierarchy only if this object has a `Visitable` interface.

4. The *visit* method is the only visible interface of the visitor hierarchy. The client only needs to invoke this method to perform any desired operation on the object structure. Since the interface and the implementation of the operations on the object structure are separated, the client is shielded from any potential changes of the implementation details.

Some liabilities of the REFLECTIVE VISITOR pattern are:

1. The name of the operation needs to be fixed. The system designer should follow the name convention and keeps all the operations named *evaluate*. Since the *evaluate* is only visible within the visitor hierarchy, there is no direct influence to other parts of the system.

2. The programming languages that used to implement this REFLECTIVE VISITOR pattern need to support reflection. This limitation lets some languages, like C++, can not be used as the implementation language for the REFLECTIVE VISITOR PATTERN.

3. The use of reflection imposes a significant performance penalty and reduces the system efficiency [60]. This pattern can be considered to be used only in time non-critical systems.

Implementation

The abstract class `Visitor` declares a unique method *visit* that takes a concrete `Element` object for visiting. This *visit* method invokes the *findMethod* operation to fetch the corresponding *evaluate* method object through reflection. Then the *visit* method invokes the *evaluate* method object to execute the operation related to the concrete `Element` object.

The *findMethod* takes a `Visitable` interface object as argument. It queries the corresponding *evaluate* method object based on the method name "evaluate" and the type of the concrete `Element` object. The search process starts from the current concrete `Visitor` class and traces up until it reaches the root class (`Visitor`) in the visitor hierarchy. If the corresponding *evaluate* method is found, the *findMethod* returns the method object. Otherwise, the search process repeats for the ancestors of this concrete `Element` until it reaches the root class (`Visitable`) in the element

hierarchy. If all the ancestors have been tried and the corresponding *evaluate* method can not be found, an error is thrown.

There is a nested loop statement in the method *findMethod*. The inner loop is used to search for an *evaluate* method with a given `Element` object as parameter. The outer loop assigns the `Element` object to the inner loop for search. The assignment principle is that the `Element` object to be visited is tried first, then the `Element` object whose declare type is the superclass of the current `Element` is tried until the corresponding *evaluate* method is found or an error is thrown if the search reaches the root interface `Visitable`. The nested loop statement guarantees that the searches trace up over the visitor hierarchy for the concrete `Element` object and all its ancestors until the corresponding *evaluate* method is found.

The `Visitor` class would be declared in Java like:

```
abstract class Visitor {
    public void visit(Visitable v) throws NoSuchMethodException {
        Method m = findMethod(v);
        try {
            m.invoke(this, new Object[] { v });
        }
        catch ( IllegalAccessException    e1 ) { /* code handling */ }
        catch ( InvocationTargetException e2 ) { /* code handling */ }
    }

    private Method findMethod(Visitable v)
                                        throws NoSuchMethodException {
        String methodName = "evaluate";
        Class visitable = v.getClass();
        while ( isAncestorOf("Visitable", visitable) {
            Class visitor = getClass();
            while ( isAncestorOf("Visitor", visitor) {
                try {
                    Method m = visitor.getDeclaredMethod(
                                    methodName,new Class[]{visitable});
                    return m;
```

```
            } catch ( NoSuchMethodException e ) {
                visitor = visitor.getSuperclass();
            }
        }
        visitable = visitable.getSuperclass();
    }
    String errMsg = "put error message here";
    throw new NoSuchMethodException(errMsg);
}

private boolean isAncestorOf(String ancestorName,Class descendant)
{
    try {
        return Class.forName(ancestorName).isAssignableFrom(
                                                descendant);
    }
    catch ( ClassNotFoundException e ) { /* code handling */ }
    return false;
}
}
```

The concrete Visitor class (e.g. class Visitor1) derives from the Visitor class. It declares an *evaluate* operation for each concrete class of Element that need to be visited. Each *evaluate* operation in the Visitor1 takes a particular concrete Element as argument. The visitor accesses the interface Element directly, and the visitor-specific behavior for that corresponding concrete Element class (e.g. ElementA) is executed.

```
class Visitor1 extends Visitor {
    protected void evaluate(ElementA e1) {
        // perform the operation on ElementA;
    }
    protected void evaluate(ElementB e2) {
        // perform the operation on ElementB;
    }
```

```
}
```

Sample Code

We'll use the Expression example defined in the Motivation section to illustrate the REFLECTIVE VISITOR pattern. Instead of generating code, we implement the example as a calculator that calculates the arithmetic expression for integers. The variables and assignment expressions are added as extensions.

Expression Hierarchy

Figure 5 is the class diagram for the Expression hierarchy. The interface `Visitable` may be declared like:

```
interface Visitable { }
```

The `Expression` is an abstract class implementing the `Visitable` interface:

```
abstract class Expression implements Visitable { }
```

The classes `ArithmeticExpr`, `AddExpr`, `SubExpr`, `MulExpr`, `DivExpr`, and `Constant` are defined as:

```
abstract class ArithmeticExpr extends Expression {
    protected ArithmeticExpr(Expression left, Expression right) {
        this.left  = left;
        this.right = right;
    }
    public Expression getLeft()  { return left;  }
    public Expression getRight() { return right; }

    private Expression left;
    private Expression right;
}

class AddExpr extends ArithmeticExpr {
    public AddExpr(Expression left, Expression right) {
        super( left, right );
```

```
    }
}

class SubExpr extends ArithmeticExpr {
    public SubExpr(Expression left, Expression right) {
        super( left, right );
    }
}

class MulExpr extends ArithmeticExpr {
    public MulExpr(Expression left, Expression right) {
        super( left, right );
    }
}

class DivExpr extends ArithmeticExpr {
    public DivExpr(Expression left, Expression right) {
        super( left, right );
    }
}

class Constant extends Expression {
    public Constant(int value) { this.value = value; }
    public int getValue() { return value; }

    private int value;
}
```

Then we add two extended expressions to the Expression hierarchy. They are classes `Variable` and `Assignment` and can be declared like:

```
class Variable extends Expression {
    public Variable(String id) {
        this.id    = id;
        this.value = 0;
    }
```

```
    public int getValue() { return value; }
    public void setValue(int value) { this.value = value; }
    public String getId() { return id; }

    private String id;
    private int value;
}

class Assignment extends Expression {
    protected Assignment(Expression lvalue, Expression rvalue) {
        this.lvalue = lvalue;
        this.rvalue = rvalue;
    }
    public Expression getLvalue() { return lvalue; }
    public Expression getRvalue() { return rvalue; }

    private Expression lvalue;
    private Expression rvalue;
}
```

Visitor Hierarchy

The implementation of the abstract class `Visitor` has been showed in the Implementation section. The `CalculationVisitor` is defined to perform a calculation operation on the expressions. Its declaration may like:

```
class CalculationVisitor extends Visitor {
    protected void evaluate(AddExpr expr)
                                throws NoSuchMethodException {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        visit(left);
        int leftResult = result;
        visit(right);
        result = leftResult + result;
```

```
}
protected void evaluate(SubExpr expr)
                          throws NoSuchMethodException {
    Expression left = expr.getLeft();
    Expression right = expr.getRight();
    visit(left);
    int leftResult = result;
    visit(right);
    result = leftResult - result;
}
protected void evaluate(MulExpr expr)
                          throws NoSuchMethodException {
    Expression left = expr.getLeft();
    Expression right = expr.getRight();
    visit(left);
    int leftResult = result;
    visit(right);
    result = leftResult * result;
}
protected void evaluate(DivExpr expr)
                          throws NoSuchMethodException {
    Expression left = expr.getLeft();
    Expression right = expr.getRight();
    visit(left);
    int leftResult = result;
    visit(right);
    result = leftResult / result;
}
protected void evaluate(Constant c) {
    result = c.getValue();
}
public int getResult() { return result; }
protected int result;
```

```
}
```

In order to adapt to the changing of the Expression hierarchy, a concrete `Visitor` class `ExtendCalculationVisitor` is defined to perform calculation operation on the newly added `Expression` classes. The class `ExtendCalculationVisitor` is an immediate subclass of the `CalculationVisitor` and can be declared like:

```
class ExtendCalculationVisitor extends CalculationVisitor {
    protected void evaluate(Variable var) {
        result = var.getValue();
    }
    protected void evaluate(Assignment expr)
                                    throws NoSuchMethodException {
        Expression lvalue = expr.getLvalue();
        Expression rvalue = expr.getRvalue();
        visit(rvalue);
        if ( lvalue instanceof Variable);
            ((Variable)lvalue).setValue(result);
    }
}
```

Client Code

For example, to calculate the expression x= 2*y+3, a client method *calculate* can be written as:

```
void calculate() {
        Expression expr = new Assignment(
                            new Variable("x"),
                            new AddExpr(new MulExpr(new Constant(2),
                                                    new Variable("y")),
                                        new Constant(3) ) );
        ExtendCalculationVisitor calculator
                            = new ExtendCalculationVisitor();
        try {
            calculator.visit(expr);
```

```
            System.out.println( calculator.getResult() );
        }
        catch ( NoSuchMethodException e ) { /* code handling */ }
}
```

Known Uses

The REFLECTIVE VISITOR pattern is applied to a compiler framework developed by the authors [50]. This framework is implemented in Java. The code generation part of the compiler framework is implemented using the REFLECTIVE VISITOR pattern. The code generation operation is started with a direct call to the `Visitor`. The abstract syntax tree that generated by the syntactical analyzer is passed to the code generation (i.e. the `Visitor`). The later recursively visit each node in the abstract syntax tree to generate the corresponding code. With the REFLECTIVE VISITOR pattern, the dispatch action is done by the `Visitor` itself. The abstract syntax tree includes no accept method and thus it can stand alone, which improve the reusability of the system.

The REFLECTIVE VISITOR pattern is also used in the design and implementation of an extensible one-pass assembler developed by the authors [49]. This assembler is based on a virtual micro assembly language under a simple virtual processor (SVP) system and is implemented in Java.

Martin E. Nordberg III [58] describes an EXTRINSIC VISITOR pattern, which focuses on breaking the cyclic dependencies between the visitors and the elements.

Jens Palsberg and C. Barry Jay [60] use the Java reflection technique in the VISITOR pattern to break the double-dispatch between the dynamic linked list and the visitor Walkabout.

Jeremy Blosser [11] and Jeanne Sebring [66] also use the Java reflection to gain the flexibility to extend the object structure (element hierarchy) in the VISITOR pattern.

Related Patterns

COMPOSITE pattern [37]: The REFLECTIVE VISITOR pattern can be used to recursively execute operations over a composite object implemented in the COMPOSITE pattern.

INTERPRETER pattern [37]: The REFLECTIVE VISITOR pattern can work with the INTERPRETER pattern to do the interpreter.

GoF VISITOR pattern [37] is used to represent an operation to be performed on the elements of an object structure. It is most likely to be used in an environment that this visited object structure is stable.

VLISSIDES VISITOR pattern [74] defines a catch-all operation in the Visitor class to perform the run-time type tests that ensure the correct code generation operations to be invoked. It is best suitable in a situation where occasional extensions are occurred to the visited object structure.

VISSER VISITOR [73] is a variation on the Vlissides Visitor framework [75]. It defines generic counterparts **AnyVisitor** and **AnyVisiable** for visitor and element hierarchies respectively.

SABLECC VISITOR pattern [29] allows the visited object structure to be extended without any limitation by performing a downcasting in the object structure. However, this approach introduces a deeper binding between the object structure and the Visitor hierarchy. It is used in situations where reusability of the object structure is not a major concern to the designer.

ACYCLIC VISITOR [54] breaks the cyclic dependency. It allows new elements to be added without changing the existing classes. This is done by defining individual Visitor interface for each Element to provide the operation interface. A dynamic cast is needed in the *accept* method to cast the Visitor parameter to its corresponding Visitor interface.

WALKABOUT VISITOR pattern [60] removes the cyclic dependency between the elements and the visitor hierarchy by using the Java reflection technique to perform the dispatch action. Its drawback is that it can not visit a complex multi-level composite hierarchy. The Reflective Visitor pattern can replace Walkabout Visitor pattern wherever it is used.

BLOSSER VISITOR pattern [11] and JEANNE SEBRING VISITOR pattern [66] also implements the dispatch action with Java reflection. It supports re-dispatch actions so that it can visit a complex multi-level composite hierarchy. The **accept** method is still used to implement the recursive traversal in this pattern. The Blosser Visitor pattern can be replaced by the Reflective Visitor pattern when the designer wants to remove the cyclic dependencies and to define a unified operation interface and

encapsulate the implementation details.

EXTRINSIC VISITOR pattern [58] removes the cyclic dependencies between the element hierarchy and the visitor hierarchy by defining a *dispatch* method in the visitor to perform the dispatch action dynamically. Although the Extrinsic Visitor Pattern reduces the coupling between the elements and visitors, adding new element classes is hard because all related visitor classes have to redefined. The Extrinsic Visitor Pattern is limited to be implemented under a C++ development environment.

3.3 A Pattern Language to Visitors

There are a number of approaches that have been proposed to handle a class of complex and related objects and their related operations. The baseline approach is the VISITOR design pattern proposed by Gamma et al [37]. But this approach gains flexibility in some aspects and loses it in others. Variations are proposed aiming for some improvement to the baseline approach.

This section classifies the major variations of the VISITOR pattern and organizes them into a pattern language. It first provides general descriptions about the basic concepts of visitors, a road map to visitors, how to use these patterns and a comparison on these visitors. The following sections describe each visitor in details.

3.3.1 A Road Map to Visitors

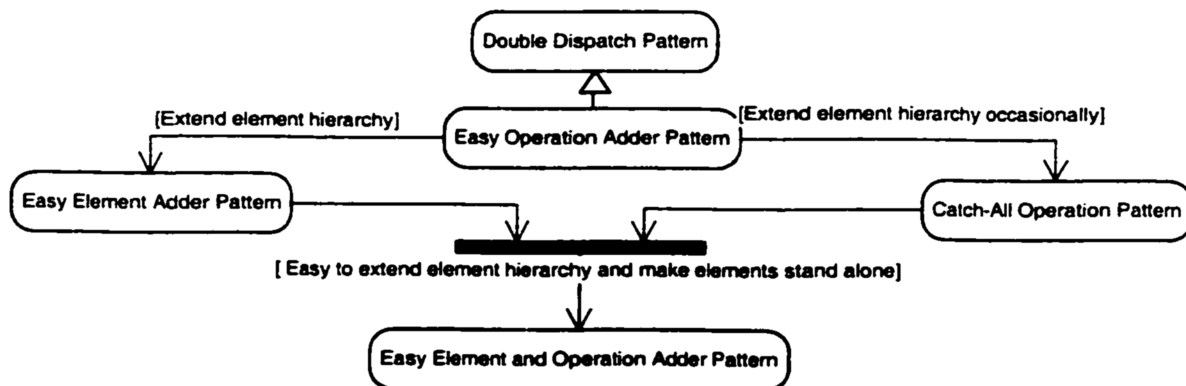

Figure 10: Road Map for the Visitor Patterns

This section provides a road map for the Visitors. It also gives a short description on how to use these patterns and a comparison on these Visitors and a guidance that assists the application developer to make the best choice.

Figure 10 shows the road map for major variations of the VISITOR pattern. The topmost pattern in Figure 10 is the DOUBLE DISPATCH pattern [37, 5, 16] and the following is the EASY OPERATION ADDER pattern [37] introduced by the GoF book. The DOUBLE DISPATCH pattern is a kind of generalization of the EASY OPERATION ADDER pattern and the EASY OPERATION ADDER pattern is the baseline approach of all Visitors. The followings are two VISITOR patterns: the CATCH-ALL OPERATION pattern [75] introduced by John Vlissides and the EASY ELEMENT ADDER pattern [29] developed by Etienne Gagnon in his Sablecc Compiler framework. Both the CATCH-ALL OPERATION pattern and the EASY ELEMENT ADDER pattern use some kinds of run-time type checking and down-casting to enable the addition of the new elements to the element hierarchy easier. But they do not break the cyclic dependency between the visitor hierarchy and the element hierarchy, so the two hierarchies have to know each other. The last approach named EASY ELEMENT AND OPERATION ADDER pattern [53], is characterized by the use of the reflection technique. It not only makes the addition of both new operations and new element classes easy, but also breaks the cyclic dependency between the visitors and the elements so that the elements have no knowledge about the visitors. Furthermore, it hides the implementation details of the operations from the clients and thus simplifies the usage of the visitors. But this VISITOR pattern achieves its simplicity in the expense of performance.

There is not a right approach for the Visitors. The vitality of the Visitors is that it provides a choice that can robustly apply in certain circumstances.

Pattern Language Summary

This pattern language includes five patterns. They are described in the following order:

1. The DOUBLE DISPATCH pattern
   This pattern [37, 5, 16] is not a kind of VISITOR but a generalization of the most Visitors. It exists in the context that the execution of an operation depends on the kind of request and the types of two receivers, the dispatcher and the

element. The DOUBLE DISPATCH pattern lets the dispatcher request different operations on each class of element without modifying the existing classes.

2. The EASY OPERATION ADDER Pattern

   This pattern was first introduced in the Design Pattern book [37]. It separates the unrelated operations from the element hierarchy and wraps these operations into another class hierarchy. By using the EASY OPERATION ADDER pattern, you can define new operations over the elements by simply adding a new visitor. So the existing elements can remain unchanged. But adding new element classes is hard because all related visitor classes have to be redefined.

3. The CATCH-ALL OPERATION Pattern

   This pattern [75] is an improvement to the EASY OPERATION ADDER pattern for occasional extension of the element hierarchy. It defines a catch-all operation in the visitors and allows new element classes to be occasionally added without modifying the existing visitor interfaces. The cyclic dependencies still exist between the visitors and the elements.

4. The EASY ELEMENT ADDER Pattern

   This pattern [29] is also an improvement to the EASY OPERATION ADDER pattern in the situation where the element hierarchy is changed often. It allows new element classes to be added without any limit. But this approach introduces a deeper binding between the element hierarchy and the visitor hierarchy. It can be used where reusability of the element hierarchy is not a major concern to the designer.

5. The EASY ELEMENT AND OPERATION ADDER Pattern

   This pattern is proposed by us[53]. It takes advantages of the reflection technique to simplify its structure and implementation. The cyclic dependency between the visitors and the elements is broken and the implementation details of the operations are encapsulated. Both addition of new operations and new element classes become easy.

Table 1: Problem/Solution Summaries for Visitors

Problem	Solution	Pattern Name
How to accept additional types of arguments in a method without modifying the existing code of the method?	Shift responsibility from the class that performs the operation to a class hierarchy, where any element may appear as an argument to the operation.	Double Dispatch
How to define new operations on classes over time without changing these class interfaces?	Package the operations in a separate hierarchy and define the *accept* method in the elements to perform dispatch.	Easy Operation Adder
How to prevent the modification of the existing visitor classes while allowing new element classes to be added occasionally?	Define a catch-all operation in the `Visitor` class and override it in the concrete visitors to perform the run-time type checking.	Catch-All Operation
How to easily add new element or new operations without modifying the existing interfaces?	Redefine a complete interface for all visitors and perform a down-casting in the elements.	Easy Element Adder
How to enable both addition of new operations and new element classes easy, while at the same time, make the element hierarchy stand alone?	Remove the *accept* from the elements and let the visitor to elegantly handle the dispatch action based on reflection.	Easy Element and Operation Adder

Table 2: Comparison on Visitor Patterns

Pattern Name	Add New Operation	Add New Element	Coupling	Efficiency
Easy Operation Adder	Easy	Hard	Tight	High
Catch-All Operation	Easy	Easy for small extension	Tight	High
Easy Element Adder	Easy	Easy	Tight	High
Easy Element and Operation Adder	Easy	Easy	Loose	Low

How to Use These Patterns

The reader who searches for a solution to a visitor problem may resort to Table 1 and Table 2. If a pattern is of particular interest, Context, Forces, Rationale and Resulting Context sections can be examined to determine the applicability of this pattern in the target circumstance. Once a pattern is chosen, the Solution and Code Samples sections can help the reader to implement the chosen pattern in the target system. Table 1 summarizes the pairs of problem and solution for the patterns.

A Comparison on Visitors

Table 2 compares the VISITOR patterns based on easy addition of new operations, easy addition of new elements, coupling between the visitor hierarchy and the element hierarchy, and the run-time efficiency:

- All VISITOR patterns can easily add new operations because they all define two class hierarchies.

- The EASY ELEMENT ADDER pattern and the EASY ELEMENT AND OPERATION ADDER pattern allow new element classes to be easily added. The latter is superior to the previous due to the simplicity of its implementation. The CATCH-ALL OPERATION pattern allow a small number of element classes to be added, otherwise, the programming style will degrade into tag-and-case statements. Use the EASY OPERATION ADDER patterns to extend the element classes is hard because all related visitors need to be modified to incorporate new element types.

- The Easy Element and Operation Adder patterns breaks the cyclic dependency between the visitors and the elements so that they have loose coupling. But other patterns do not.

- Due to the use of reflection technique, the run-time efficiency of the Easy Element and Operation Adder pattern is low.

There is not a right Visitor pattern all the time. All patterns presented in this section are selectively applied in certain circumstances. From Table 2, we can also see that a system's extensibility is always traded with its efficiency. For example, the Easy Element and Operation Adder pattern has a very superior structure and good extensibility, but its efficiency is very slow.

3.3.2 A Simple Example

Because all patterns presented in this section are closely related, we will put all of them together to highlight their similarities and differences. We'll re-use the simple expression example that is shown in Section 3.2 to illustrate their implementations. This simple expression supports arithmetic expressions such as addition, subtraction, multiplication, and division for constants. Figure 5 shows the language structure hierarchy for this expression example. The expression structure hierarchy is organized as a composite structure and can be implemented by a Composite design pattern [37]. The abstract syntax tree therefore is represented as a composite object, which is recursively constructed with the instances of the node classes in the expression structure during the parsing. The code generation or calculation process then performs the code generation operation or calculation over this abstract syntax tree.

3.3.3 Double Dispatch

Alias

Visitor Essence.

Context

The behavior of a method depends not only on the class that implements the method but also on the classes of the method's arguments as well.

Problem

How to accept additional types of arguments in a method without modifying the existing code of the method?

Forces

- Using a case-like statement to perform type-checking on the types of arguments makes the system difficult to extend.
- Determining the argument types at run time rather than at compile-time makes the system more flexible.

Solution

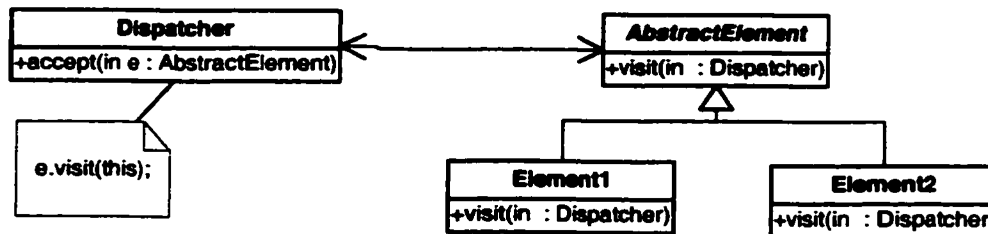

Figure 11: Structure for the Double Dispatch Pattern

Shift responsibility from the class that performs the operation to a class hierarchy, where any element may appear as an argument to the operation. All these elements share an identical operation interface.

Figure 11 shows the structure of the DOUBLE DISPATCH pattern. There are two groups defined in the design. One is the class `Dispatcher`. The other is the element hierarchy. The `Dispatcher` defines a method *accept* that can accept a concrete element instance. The implementation of the *accept* method calls the related method *visit* in the element hierarchy. The method *accept* is a double-dispatch operation. It depends on both the types of the `Dispatch` and the concrete element.

Rationale

Treating different types of receivers the same and letting the program determine the concrete types at run time help to avoid degrading the program into a long tag-and-case statement in order to distinguish different types of receivers. For instance, in Figure 11, we assume that the class `Element2` is a newly added concrete element class. It defines a *visit* method to implement the corresponding operation that should be executed during the *accept* method's execution. When the *accept* method is called on an instance of `Dispatcher` with a given instance of `Element2` as argument, it invokes the polymorphic *visit* method on the instance of class `Element2`(the argument of the method *accept*) and supplies itself as a parameter. The *visit* method of class `Element2` can call back to any method defined in the `Dispatcher`. As a consequence, any concrete element type can be accepted by the `Dispatcher` without modifying the invoking method defined in the `Dispatcher`.

Resulting Context

- Accept additional types of arguments in a method without modifying the existing code of the method. New type argument can be added in the element hierarchy by inheritance and the method can accept the instance of this new type and dispatch the operation to this instance by polymorphism.
- There is no need to write a case-like statement in the invoking method (eg. *accept*) to perform type-checking on the types of arguments and add additional type-checking statements once a new parameter type is added. Instead, the argument types can be determined at run time dynamically. With DOUBLE DISPATCH, the parameter type is determined at run-time, the behavior of the invoking method is hidden at compile time.
- The code is distributed in several classes so that locating and understanding the intending behavior becomes hard.

Code Samples

We focus on the `Constant` class in the simple expression example and show how to implement the DOUBLE DISPATCH pattern, The following shows the definitions of the related classed and the implementation in Java.

```
class Constant{
    public void accept(CodeGen gen) {
        gen.visit(this);
    }
}

abstract class CodeGen {
    abstract public void visit(Constant expr);
}

class VM1CodeGen extends CodeGen {
    public void visit(Constant expr) {
        // code generation for virtual machine VM1 ;
    }
}

class VM2CodeGen extends CodeGen {
    public void visit(Constant expr) {
        // code generation for virtual machine VM2 ;
    }
}
```

Related Patterns

The DOUBLE DISPATCH pattern is a kind of generalization of the EASY OPERATION ADDER pattern.

3.3.4 Easy Operation Adder

Aliases

GoF Visitor, Visitor Pattern, Standard Visitor.

Context

There are a fair number of instances of a small number of classes that are rarely changed, and you are expecting to perform new operation that involves all or most of them [16].

Problem

How to define a new operation on classes over time without changing the classes of the elements on which it operates?

Forces

- Codes that change often will introduce new bugs that are hard to locate and fix.
- If the number of classes is large, adding new operations to these classes needs a significant overhead of recompile.
- Grouping distinct and unrelated operations in a class may lead to a solution that is hard to understand and maintain.
- A volatile interface is hard to use and maintain because the client code needs to change often.

Solution

Define two hierarchies. Related operations are grouped into a hierarchy called the visitor hierarchy, and the other hierarchy includes all elements and is called the element hierarchy. A method *accept* is defined across the element hierarchy. An operation is performed on an element object by a call of the *accept* on the element object and a supply of the corresponding visitor as argument that represents the desired operation. Thus, in the EASY OPERATION ADDER pattern, the dispatch action is performed by the element object. A concrete element object knows which operation associated with it, so it dispatch a call to the corresponding visitor object by supplying itself as the parameter. Figure 12 shows the structure of the EASY OPERATION ADDER pattern.

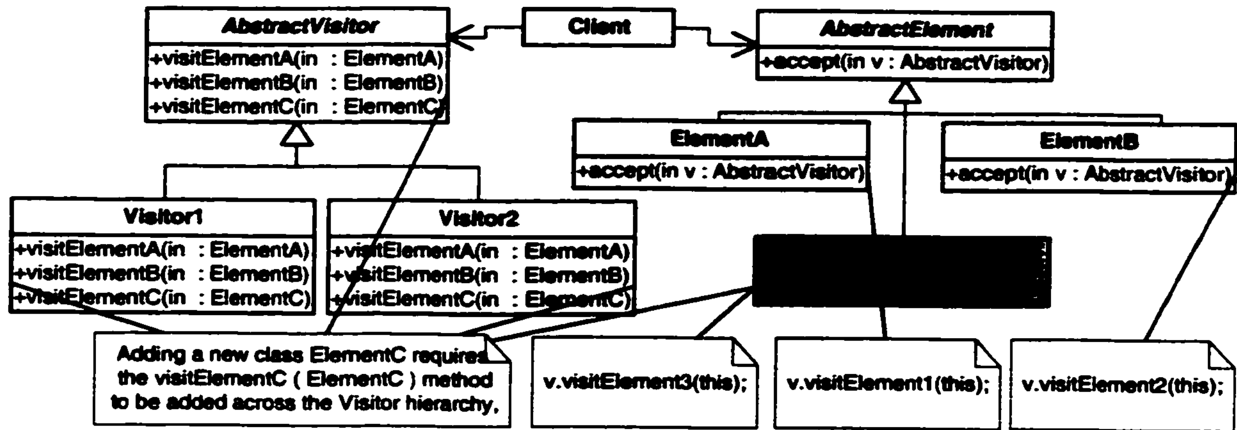

Figure 12: Structure for the Easy Operation Adder Pattern

Rationale

If an object is too complex to understand, it is better to separate it into smaller objects that are less complex. Isolating the changeable parts in an object helps to leading to a system that is easy to maintain and extend. On the other hand, removal of unrelated operations will make the elements more cohesive. It is also desirable to separate code that changes from the code that does not. Adding new operations will only happen within the scope of the visitor hierarchy, while the interface of the element hierarchy keeps unchanged. If any new bugs are introduced due to the adding of new operations, the bugs are easily located by examining the newly added operations in the visitor hierarchy. This solution structure also supports incremental programming because new operations to an object structure can be added incrementally.

Resulting Context

- A new operation can be easily added by simply adding a new visitor. Any code in the elements needs not to be changed.
- adding a new concrete element class is hard. Any addition of a new concrete element class **Xxx** requires a *visitXxx(Xxx)* method to be defined as abstract in the abstract visitor root class and implemented in all concrete element classes.

Code Samples

We will use the simple expression example to illustrate this VISITOR pattern. Instead of generating code, we implement the example as a calculator that calculates the arithmetic expression for integers. The variables and assignment expressions are added as extensions.

Expression Hierarchy

Figure 5 is the class diagram for the Expression hierarchy. The `Expression` is an interface:

```
interface Expression {
    public void accept(Visitor visitor);
}
```

The classes `ArithmeticExpr`, `AddExpr`, `SubExpr`, `MulExpr`, `DivExpr`, and `Constant` are defined as:

```
abstract class ArithmeticExpr implements Expression {
    protected ArithmeticExpr(Expression left, Expression right) {
        this.left  = left;
        this.right = right;
    }
    abstract public void accept(Visitor visitor);
    public Expression getLeft()  { return left;  }
    public Expression getRight() { return right; }

    private Expression left;
    private Expression right;
}

class AddExpr extends ArithmeticExpr {
    public AddExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
```

```
        visitor.visitAddExpr(this);
    }
}

class SubExpr extends ArithmeticExpr {
    public SubExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
        visitor.visitSubExpr(this);
    }
}

class MulExpr extends ArithmeticExpr {
    public MulExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
        visitor.visitMulExpr(this);
    }
}

class DivExpr extends ArithmeticExpr {
    public DivExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
        visitor.visitDivExpr(this);
    }
}

class Constant implements Expression {
    public Constant(int value) { this.value = value; }
    public int getValue() { return value; }
    public void accept(Visitor visitor) {
```

```
        visitor.visitConstant(this);
    }

    private int value;
}
```

Then we add two extended expressions to the Expression hierarchy. They are classes `Variable` and `Assignment` and can be declared like:

```
class Variable implements Expression {
    public Variable(String id) {
        this.id    = id;
        this.value = 0;
    }
    public void accept(Visitor visitor) {
        visitor.visitVariable(this);
    }
    public int getValue() { return value; }
    public void setValue(int value) { this.value = value; }
    public String getId() { return id; }

    private String id;
    private int value;
}

class AssignmentExpr implements Expression {
    protected AssignmentExpr(Expression lvalue, Expression rvalue) {
        this.lvalue = lvalue;
        this.rvalue = rvalue;
    }
    public void accept(Visitor visitor) {
        visitor.visitAssignmentExpr(this);
    }
    public Expression getLvalue() { return lvalue; }
    public Expression getRvalue() { return rvalue; }
```

```
    private Expression lvalue;
    private Expression rvalue;
}
```

Visitor Hierarchy The visitor hierarchy encapsulates calculation operations performed over the expression.

The following implementation also shows that a concrete element can also be a composite object.

```
interface Visitor{
    public void visitAddExpr(AddExpr expr);
    public void visitSubExpr(SubExpr expr);
    public void visitMulExpr(MulExpr expr);
    public void visitDivExpr(DivExpr expr);
    public void visitConstant(Constant expr);

    // newly added methods due to the extension of the expression
    public void visitAssignmentExpr(AssignmentExpr expr);
    public void visitVariable(Variable expr);
}

class CalculationVisitor implements Visitor {
    public void visitAddExpr(AddExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult + result;
    }
    public void visitSubExpr(SubExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
```

```
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult - result;
    }
    public void visitMulExpr(MulExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult * result;
    }
    public void visitDivExpr(DivExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult / result;
    }
    public void visitConstant(Constant expr) {
        result = expr.getValue();
    }

    // newly added methods due to the extension of the expression
    public void visitAssignmentExpr(AssignmentExpr expr) {
        Expression lvalue = expr.getLvalue();
        Expression rvalue = expr.getRvalue();
        rvalue.accept(this);
        if ( lvalue instanceof Variable);
            ((Variable)lvalue).setValue(result);
    }
```

```
    public void visitVariable(Variable expr) {
        result = expr.getValue();
    }
    public int getResult() { return result; }

    protected int result;
}
```

Related Patterns

Visitor pattern can be used to perform operations on a composite object defined in the Composite pattern [37]. It can also be applied to perform the interpretation in the Interpreter pattern [37].

3.3.5 Catch-All Operation

Aliases

Vlissides Visitor, Restricted Element Adder Visitor.

Context

You want to use visitors and you need to occasionally add new element classes. But you do not want to change the interfaces of the visitors once they have been defined.

Problem

How to prevent the modification of the existing visitor classes while allowing new element classes to be added occasionally?

Forces

- It is hard to modify an interface once it has been built in a framework.
- The structure proposed in the EASY OPERATION ADDER pattern separates operations from elements. Adding new operations are easy because it needs not modify the existing interfaces of elements. But adding a new element class is

hard. All interfaces of related visitor classes must be modified to incorporate a method to visit the newly added element class.

- A structure with changing interface is fragile.
- An interface can be easily extended by inheritance without modifying the existing interface.
- If a method could not be added to the existing classes, it can be added to the extended classes. But since the definition of this method is missing in the ancestors, a type casting is mandatory if the message receiver has a declared type of the ancestors.
- If the implementation of a method in a class could not be changed, it can be overridden in the extended classes. An overridden method is only necessary if it has a distinct behavior that the original method could not handle.

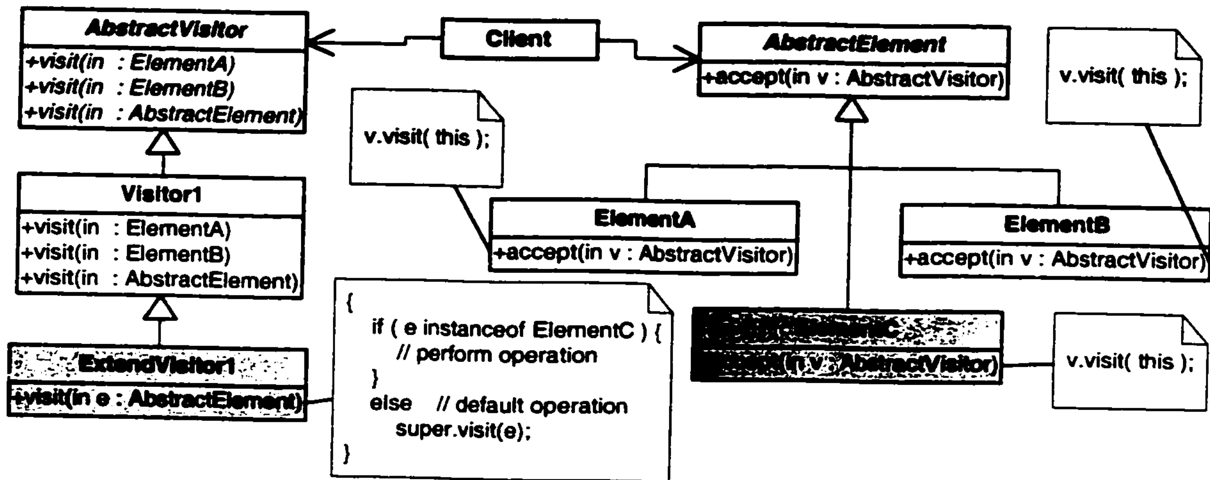

Figure 13: Structure of the Catch-All Operation Pattern

Solution

Similar with the Easy Operation Adder pattern, the structure of the pattern defines two class hierarchies, the element hierarchy and the visitor hierarchy. A catch-all operation is defined in the base class **AbstractVisitor** and it is overridden in its

descendants. If a new element class is added in the element hierarchy, new visitor classes are defined to extend the existing concrete visitor classes and the catch-all operation is re-written to perform the run-time type test on the newly added element class.

Figure 13 shows the structure for the CATCH-ALL OPERATION pattern. The newly added classes are adorned in gray. The `ElementC` is a newly added element class. A `ExtendVisitor1` class is defined to subclass the `Visitor1` and the catch-all operation *visit* is overridden to perform the run-time type checking on the `ElementC`. If a element to be visited is a newly added element, specific operation related to the new class is performed. Otherwise, the method demonstrates its previous behavior.

Rationale

Two simple class hierarchies as that defined in the EASY OPERATION ADDER pattern have many advantages including one that allows the new operation to be added easily without recompiling the element hierarchy. A catch-all operation is so blurring that it leaves margin to allow the new elements to be easily added and handled. Occasional addition of the element classes will not degrade the programming style of the catch-all operation because the tag-and-case statement is very short.

Resulting Context

- New element classes can be added occasionally without any modification of the existing visitor classes.
- If new element classes are constantly added, CATCH-ALL OPERATION will degrade into a tag-and-case-statement style of programming.

Code Samples

We'll still use the expression example to calculate the arithmetic expression for integers. The variables and assignment expressions are added as extensions.

Element Hierarchy

The classes defined in the element hierarchy are the same as the class declarations in the EASY OPERATION ADDER.

Visitor Hierarchy

```
interface Visitor{
    public void visit(AddExpr expr);
    public void visit(SubExpr expr);
    public void visit(MulExpr expr);
    public void visit(DivExpr expr);
    public void visit(Constant expr);

    // catch-all operation
    public void visit(Expression expr);
}

class CalculationVisitor implements Visitor {
    public void visit(AddExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult + result;
    }
    public void visit(SubExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult - result;
    }
    public void visit(MulExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
```

```
        int leftResult = result;
        right.accept(this);
        result = leftResult * result;
    }
    public void visit(DivExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult / result;
    }
    public void visit(Constant expr) {
        result = expr.getValue();
    }
    // catch-all operation
    public void visit(Expression expr) { }
    public int getResult() { return result; }

    protected int result;
}
```

Operations for newly added elements are encapsulated in the extended concrete visitor class.

```
class ExtendCalculationVisitor extends CalculationVisitor {
    // catch-all operation
    public void visit(Expression expr) {
        if ( expr instanceof AssignmentExpr )
            visit((AssignmentExpr)expr);
        else if ( expr instanceof Variable )
            visit((Variable)expr)
        else
            super.visit(expr);
    }
```

```
    public void visit(AssignmentExpr expr) {
        Expression lvalue = expr.getLvalue();
        Expression rvalue = expr.getRvalue();
        rvalue.accept(this);
        if ( lvalue instanceof Variable);
            ((Variable)lvalue).setValue(result);
    }
    public void visit(Variable expr) {
        result = expr.getValue();
    }
}
```

Related Patterns

This pattern is an improvement to the EASY OPERATION ADDER pattern [37].

Visser's Visitor [73] is a variation on the Vlissides Visitor framework [75]. It defines generic counterparts `AnyVisitor` and `AnyVisitable` for visitor and element hierarchies respectively.

3.3.6 Easy Element Adder

Aliases

Sablecc Visitor.

Context

You have visitor that works well. You are expecting to add element classes in the future, but you are unable to change the existing class interfaces.

Problem

How to easily add new element or new operations without modifying the existing interfaces?

Forces

- The EASY OPERATION ADDER pattern makes the addition of new operations easy, but adding new element classes is hard because it needs to modify the existing interfaces of the visitors.

- The CATCH-ALL OPERATION pattern is not suitable because constantly adding new element classes will degrade the implementation of the catch-all operation to be a tag-and-case style of programming.

- Inheritance provides a good means to extend the existing interface without actually modifying it.

- If the inheritance tree is high, a structure is hard to understand because the underneath classes could not be understood without resorting to its ancestors.

- A type casting can satisfy the compiler because it can precisely refer to a method defined in some classes but not in others. But the type casting is unsafe. It always depends on the good wills of the programmer.

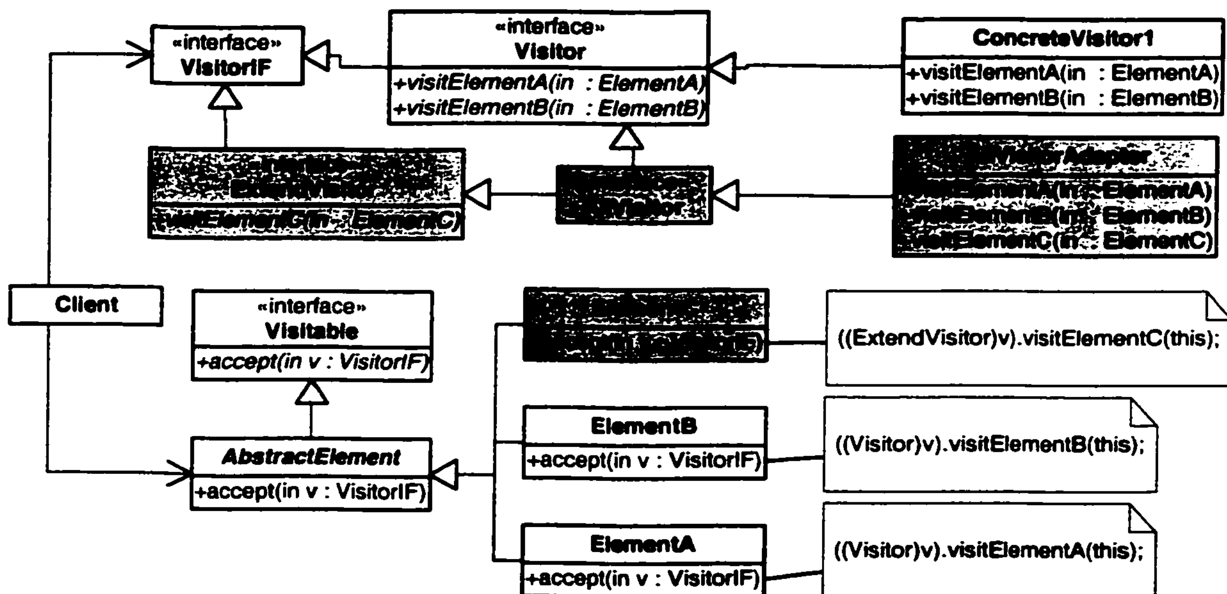

Figure 14: Structure for the Easy Element Adder Pattern

Solution

Separate the elements and their operations in two class hierarchies, one is the Visitor, and the other is the Element. An interface `VisitorIF` is defined on the top of the visitor hierarchy. New operations can be easily added by defining concrete visitor classes in the visitor hierarchy. New element classes can be added by extending existing interfaces in the Visitor hierarchy and defining concrete visitor to implement the newly added interface.

Figure 14 shows the structure for the EASY ELEMENT ADDER pattern. It is a snapshot after adding a new element class `ElementC`. The new added classes are adorned in gray. The interfaces of the visitors are extended by introducing two new interface classes: `ExtendVisitor` and `AllVisitor`. The previous extends the root interface `VisitorIF` by defining a new visit operation for the newly added element class. The latter represents the complete interface for the visitors due to the changing of the subject elements. Class `AllVisitorAdapter` is a new visitor class that implements the complete interface `AllVisitor`. In order to adapt to the changing interfaces in the visitor hierarchy, the *accept* methods that are defined in the element classes must indicate which visitor class it dispatches to by performing down-casting.

Rationale

An existing interface is hard to change, but it is easy to extend. A redefinition of the concrete visitors due to any addition of new element class is easy to understand without resorting to the ancestors. In general, both hierarchies are developed or maintained by the same person. So a type casting in the *accept* can be considered to be safe.

Resulting Context

- The elements and their operations can be easily extended without any restriction and without modifying the existing class interfaces.
- Frequently adding new classes makes the class hierarchy too complicated to understand and implement.
- The type-casting in the *accept* method makes the programming unsafe and tightly coupled with the concrete element types.

Code Samples

The following shows the Java implementation on how to applying EASY ELEMENT ADDER pattern to the calculation for the simple expression example.

Expression Hierarchy

```
interface Visitable { }

interface Expression extends Visitable {
    public void accept(Visitor visitor);
}
```

The classes ArithmeticExpr, AddExpr, SubExpr, MulExpr, DivExpr, and Constant are defined as:

```
abstract class ArithmeticExpr implements Expression {
    protected ArithmeticExpr(Expression left, Expression right) {
        this.left  = left;
        this.right = right;
    }
    abstract public void accept(Visitor visitor);
    public Expression getLeft()  { return left;  }
    public Expression getRight() { return right; }

    private Expression left;
    private Expression right;
}

class AddExpr extends ArithmeticExpr {
    public AddExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
        ((Visitor)visitor).visitAddExpr(this);
    }
}
```

```
class SubExpr extends ArithmeticExpr {
    public SubExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
       ((Visitor)visitor).visitSubExpr(this);
    }
}

class MulExpr extends ArithmeticExpr {
    public MulExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
        ((Visitor)visitor).visitMulExpr(this);
    }
}

class DivExpr extends ArithmeticExpr {
    public DivExpr(Expression left, Expression right) {
        super( left, right );
    }
    public void accept(Visitor visitor) {
        ((Visitor)visitor).visitDivExpr(this);
    }
}

class Constant implements Expression {
    public Constant(int value) { this.value = value; }
    public int getValue() { return value; }
    public void accept(Visitor visitor) {
        ((Visitor)visitor).visitConstant(this);
    }

    private int value;
```

```
}
```

Two extended expressions, classes **Variable** and **AssignmentExpr** can be added to the Expression hierarchy as following:

```
class Variable implements Expression {
    public Variable(String id) {
        this.id    = id;
        this.value = 0;
    }
    public void accept(Visitor visitor) {
        ((ExtendVisitor)visitor).visitVariable(this);
    }
    public int getValue() { return value; }
    public void setValue(int value) { this.value = value; }
    public String getId() { return id; }

    private String id;
    private int value;
}

class AssignmentExpr implements Expression {
    protected AssignmentExpr(Expression lvalue, Expression rvalue) {
        this.lvalue = lvalue;
        this.rvalue = rvalue;
    }
    public void accept(Visitor visitor) {
        ((ExtendVisitor)visitor).visitAssignmentExpr(this);
    }
    public Expression getLvalue() { return lvalue; }
    public Expression getRvalue() { return rvalue; }

    private Expression lvalue;
    private Expression rvalue;
}
```

Visitor Hierarchy For extending the new elements without modifying the existing visitors, the new visitor interfaces are defined to support the extension.

```
interface VisitorIF { }

interface Visitor extends VisitorIF {
    public void visitAddExpr(AddExpr expr);
    public void visitSubExpr(SubExpr expr);
    public void visitMulExpr(MulExpr expr);
    public void visitDivExpr(DivExpr expr);
    public void visitConstant(Constant expr);
}

interface ExtendVisitor extends VisitorIF {
    public void visitAssignmentExpr(AssignmentExpr expr);
    public void visitVariable(Variable expr);
}

interface AllVisitor extends Visitor, ExtendVisitor { }

class CalculationVisitor implements Visitor {
    public void visitAddExpr(AddExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult + result;
    }
    public void visitSubExpr(SubExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
```

```
        right.accept(this);
        result = leftResult - result;
    }
    public void visitMulExpr(MulExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult * result;
    }
    public void visitDivExpr(DivExpr expr) {
        Expression left = expr.getLeft();
        Expression right = expr.getRight();
        left.accept(this);
        int leftResult = result;
        right.accept(this);
        result = leftResult / result;
    }
    public void visitConstant(Constant expr) {
        result = expr.getValue();
    }
    public int getResult() { return result; }

    protected int result;
}

class ExtendCalculatorVisitor extends CalculatorVisitor
                              implements AllVisitor {
  public void visitAssignmentExpr(AssignmentExpr expr) {
      Expression lvalue = expr.getLvalue();
      Expression rvalue = expr.getRvalue();
      rvalue.accept(this);
```

```
        if ( lvalue instanceof Variable);
            ((Variable)lvalue).setValue(result);
    }
    public void visitVariable(Variable expr) {
        result = expr.getValue();
    }
}
```

Related Patterns

This pattern is an improvement to the EASY OPERATION ADDER pattern [37].

Acyclic Visitor [54] allows new elements to be added without changing the existing classes. It defines an individual visitor interface for each element to provide the operation interface. A dynamic cast is needed in the *accept* method to cast the visitor parameter to its corresponding visitor interface.

3.3.7 Easy Element and Operation Adder

Aliases

Reflective Visitor.

Context

You want to use visitors and you want to make both addition of new operations and new elements easy. You are also expecting to reuse the element hierarchy.

Problem

How to enable both addition of new operations and new element classes easy, while at the same time, make the element hierarchy stand alone?

Forces

- A structure that proposed in the EASY OPERATION ADDER pattern makes adding new operations easy, but adding new element classes is hard and reusing these elements is also hard.

- Structures proposed in the the CATCH-ALL OPERATION pattern and the EASY ELEMENT ADDER pattern support the extension on both the element hierarchy and visitor hierarchy, but reuse elements is hard because these elements depend on the visitors.
- Breaking the cyclic dependency and letting the element hierarchy stand alone, the system can reuse the elements hierarchy easily.
- If the dependency is removed on the side of elements, the Visitors must carry out the dispatch action that requires the type information about the related elements.
- The reflection technique provides an easy way to locate a method if its naming convention is known in advance. But the use of reflection gains simplicity in the expense of performance.
- A unified simple interface for operations is easy to use and maintain.
- To modify an interface is hard, but extend it is easy.

Solution

Separate operations from the elements. Objects of the elements to be visited are specified as `Visitable`. All *accept* methods are removed from the element hierarchy. Method *visit* in the root class `AbstractVisitor` is the only visible method in the visitor hierarchy and invokes the *findMethod* method to perform the dispatch operation. The method *findMethod* uses reflection technique to locate the desirable methods for the supplied parameter. Various *evaluate* methods are defined in the concrete visitors to perform specific operations on the related elements. Figure 8 shows the structure for the EASY ELEMENT AND OPERATION ADDER pattern.

Rationale

A common interface `Visitable` enables distinct elements to be built in different element hierarchies to share a common ancestor. The method *visit* implemented in the `AbstractVisitor` class is the only public method in the visitor hierarchy. It takes a role of a dynamic dispatcher by invoking the corresponding *evaluate* method found

by *findMethod* method. The reflection is used by the method *visit* and *findMethod* to support the method finding and method invocation dynamically. Because the dispatch operation is performed by the `Visitor` class, the *accept* methods can be removed from the element hierarchy and thus the developer can reuse these elements without the visitors' supports. The method *evaluate* can visit a composite object recursively because an *evaluate* method is invoked by the *visit* method and it can also make a call to the method *visit* if needed. The use of reflection will lead to a severe performance penalty, but it can still be accepted if performance is not a major concern in the system design. Any addition of new operations only needs to define a new concrete visitor class in the visitor hierarchy. Any addition of new element class only needs to extend the related concrete visitor classes and define new *evaluate* methods in the new visitor classes. Existing classes are thus kept from any potential modification.

Resulting Context

1. As that of the EASY OPERATION ADDER pattern, adding a new operation is easy. The existing code can be avoided from modifying by simply subclassing the visitor hierarchy when a new operation over the element hierarchy is added.

2. Adding a new element class is easy. Since the `AbstractVisitor` is responsible for the dynamic dispatch, any operation operating on this new element can be defined within a new visitor subclass without modifying the existing codes. The system's extensibility is then improved.

3. The cyclic dependencies are broken and the coupling between the element hierarchy and the visitor hierarchy is reduced. As the key of the traditional VISITOR pattern, the double-dispatch technique is used to associate the operation with the concrete element at run time. But this technique reduces the system's reusability. With the reflection technique, the VISITOR pattern can avoid the cyclic dependencies by performing the dynamic dispatch within the `AbstractVisitor` class. Since the visitor is responsible for the dynamic dispatch, the element hierarchy has no knowledge about the visitor. Hence the system's reusability is improved. On the other hand, the visitor can visit any object that has a corresponding *evaluate* operation in the visitor hierarchy only

if this object has a `Visitable` interface.

4. The *visit* method is the only visible interface of the visitor hierarchy. The client only needs to invoke this method to perform any desired operation on the visitable elements. Since the interface and the implementation of the operations on the elements are separated, the client is shielded from any potential changes of the implementation details.

5. The name of the operation needs to be fixed. The system designer should follow the name convention and keeps all the operations named *evaluate*. Since the *evaluate* is only visible within the visitor hierarchy, there is no direct influence to other parts of the system.

6. The programming languages that used to implement this VISITOR pattern need to support reflection. This limitation lets some languages, like C++, can not be used as the implementation language for this pattern.

7. The use of reflection imposes a significant performance penalty and reduces the system efficiency [60]. This pattern can be considered to be used only in time non-critical systems.

Code Samples

The Java implementation of the EASY ELEMENT AND OPERATION ADDER for the simple expression example is shown in Section 3.2.

Related Patterns

This pattern is an improvement to all other VISITOR patterns presented in this section and is applied when the programming environment supports reflection and efficiency is not a major concern. Section 3.2 presents a set of patterns that are related to this pattern.

3.4 Summary

The REFLECTIVE VISITOR improves the extensibility and reusability features upon the earlier implementations of the VISITOR pattern. The reflection technique enables the visitor to perform the run-time dispatch action on itself. The separation of the run-time dispatch action from the object structure makes any extension to the object structure become easy. It also removes the cyclic dependencies between the visitors and the object structure, so the reusability and extensibility of the system are improved.

This chapter also presents a pattern language to VISITORs that have been proposed since 1995. The pattern language can assist the application developer to better understand the circumstance that a VISITOR pattern is applied and their pros and cons so that a right decision can be made. However, this pattern language does not come to the end. As long as new VISITOR patterns continue to emerge, this pattern language will evolve with them.

The next chapter will present the design of a parser with the reflection pattern. We will show how can REFLECTION pattern benefit the object-oriented design of a predictive recursive-descent parser.

Chapter 4

Parsing with Reflection Pattern

Parsing is the core of the front end of a compiler. The predictive recursive-descent parsing approach is most widely used in a traditional compiler design. It is straightforward and easy to implement. But since predictive recursive-descent parsing degrades into structured program, it results in a parser that is very hard to change, extend and maintain.

A pattern language is a set of related patterns that solve a common problem in a problem domain. This chapter presents a pattern language for developing a framework for parsing in object-oriented compiler design based on the principle of the predictive recursive-descent parsing approach. It describes four patterns that address three design aspects in developing an object-oriented parser. Two alternative patterns are presented to provide alternative solutions to solve the recursion problem in the object-oriented software design. One is based on the Builder design pattern, and the other is based on the meta-programming technology. The parsers developed from this pattern language are easy to implement, easy to extend, and easy to maintain. This pattern language is intended to express a flexible and extensible design for parsing that can accommodate variations to its most extent. It is presented in a pattern language format [56] as described in Section 2.4.

Section 4.1 presents a pattern language for parsing. Section 4.1.1 is the overview of this pattern language. Section 4.1.2 to Section 4.1.5 describe patterns PARSER STRUCTURE, LANGUAGE STRUCTURE, PARSERBUILDER, and METAPARSER in details.

Table 3: Problem/Solution Summaries for Patterns in the Parsing

Problem	Solution	Pattern Name
How to define an extensible architecture to maximize accommodation of various hot spots for the design of a parser?	Separate grammar rules from the language structure.	Parser Structure
How to represent the language structure to anticipate the changing formats of the target languages?	Organize the language structure with the COMPOSITE design pattern.	Language Structure
How to assemble the loose coupling components in the parser, while at the same time, allow it to be easily extended without modifying the existing code?	Define a common parsing interface with a hook method and let a concrete class implement this hook method and wrap the parsing process for the corresponding target language.	ParserBuilder
How to encapsulate the application logic and build a self-manageable and intelligent parsing processing mechanism?	Define the base-level for the application logic and the meta-level to reflect the base-level and control the parsing process.	MetaParser

4.1 A Pattern Language for Parsing

4.1.1 Overview

As the use of pattern has injected insight in the analysis of a problem and its solutions, pattern is increasingly important in software design and presentation. A pattern language is a set of related patterns that solve a common problem in a problem domain. It is particular effective at addressing certain recurring problems.

The syntactic analyzer, or the parser, is the core of the front end of the compiler. Its main task is to analyze the program structure and its components [61]. In general, the design of a parser is changing due to the changing of the target language's definition. However, for various compiled languages, all parsing processes share the major commonalty, that is, they follow the same operation pattern.

This section presents a pattern language for developing a framework for parsing

in object-oriented compiler design based on the principle of the predictive recursive-descent parsing approach. It contains four patterns, each is described in a pattern style, where its context, problem, forces, solution, etc, are discussed. The target audience is the framework designer who intends to develop an extensible architecture for parsing or the application developer who needs to better understand the framework in order to customize it for a specific application.

This pattern language contains four different patterns to address three aspects of a framework design for the syntactic analysis in a compiler. These patterns are:

- An analysis pattern: Parser Structure, which addresses the architectural aspect of a parser.
- A structural pattern: Language Structure, which addresses the static representation of the target language.
- Two creational patterns: ParserBuilder and MetaParser, which address the dynamic aspects of the parsing process.

Table 3 is the problem/solution summaries for the patterns presented in the section. It can be used as a guidance and quick reference to the use of the patterns.

4.1.2 Parser Structure

Context

You have decided to develop a framework for syntactic analysis.

Problem

How to define an extensible architecture to maximize accommodation of various hot spots?

Forces

- To anticipate the unanticipated is hard. The definition of the target language is vague when the framework is building.
- The grammar rules and the elements of the language structure are embedded in the language definition, which implies the parsing process. Any changes of

the grammar rules or the language structure will cause the parsing process to change accordingly.

- A structure is easy to maintain if the code that is frequently changed is separated from that is not.
- The language definition contains so much information that it is too complex to handle. A number of simple problems are easier to solve than a complex one.
- To mix the processing of an object structure with its representation will make a system hard to understand and maintain.
- The user need not understand the implementation details of a parser. A simple interface is always preferable than a complex one because the complexity of a system is hidden.
- Successful examples often inject insight into the solutions for a recurring problem. Reuse of successful experience can minimize the potential risk.

Solution

Apply the ACCOUNTABILITY analysis pattern [34]. Separate grammar rules from the language structure and make the language structure stand alone. A grammar rule encapsulates the application logic and will drive the parsing process. It represents the dynamic aspect of the language definition. A language structure is only a representation of the target language. It represents the static aspect of the language definition.

Define a simple interface, `ParserHandler`, to simplify the use of the system. It provides the least and exact information that the user needs to know.

Structure Figure 15 shows the structure of the PARSER STRUCTURE.

The PARSER STRUCTURE contains three packages: `Parser Handler`, `Grammar Rules`, and `Language Structure` . Note that the packages in gray do not belong to this pattern. But since they are parts of the compiler design, they have direct dependency relationships with the parser.

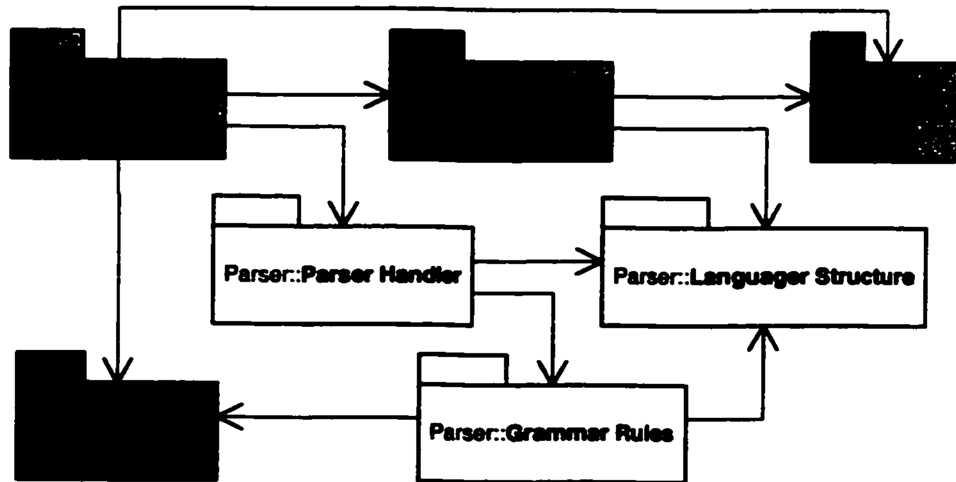

Figure 15: Structure for the Parser Structure

Participants

- `Parser Handler`
  Declares the interface for the syntactic analysis.

- `Grammar Rules`
  Encapsulates the grammar rules for the target languages and defines the execution sequence of the parsing process.

- `Language Structure`
  Defines the elements that make up of the target language and shows the static view of the relationships among the elements.

Consequences

The separation of the grammar rules from the language structure has the following implicit advantages:

- The static representation of the target language is separated from its potential processing. The grammar rules and the language structure have different roles to play and serve for different purpose. The architecture becomes less coupling and more cohesive.

- Both the grammar rules and the language structure are simple to handle than the one as a whole. The separation helps to reduce the complexity of the system.
- A loose coupling structure is easy to develop, extend, and maintain.

In addition, the `ParserHandler` provides a simple and stable interface to the user. The user is shielded from any potential changes of the grammar rules and the language structure.

Related Patterns

The ACCOUNTABILITY analysis pattern [34] provides similar solution to separate rules from the organization structure.

4.1.3 Language Structure

Context

You are defining the language structure and have applied the PARSER STRUCTURE.

Problem

How to represent the language structure to anticipate the changing formats of the target languages?

Forces

- To define a unified language structure for all potential target languages is hard and impossible. A reasonable representation of the language structure is a general abstraction of most frequently used target languages.
- An organized structure is easier to understand and maintain than a number of discrete objects. An organized structure offers a hierarchy that can benefit from some design techniques such as inheritance, which promotes software reuse and extensibility.
- A component of the language structure can be primitive or composite. To differentiate their processing is tedious and error-prone.

- The parsing output is a syntax tree. The representation of the language structure should allow the syntax tree to be easily built and processed.

Solution

Define an interface class **Language** to encapsulate the language abstraction. The language structure is organized using the COMPOSITE design pattern [37]. The syntax tree is represented as the object structure. It is a tree made up of objects of the language structure that are created at run-time.

Structure Figure 16 shows the structure of the LANGUAGE STRUCTURE.

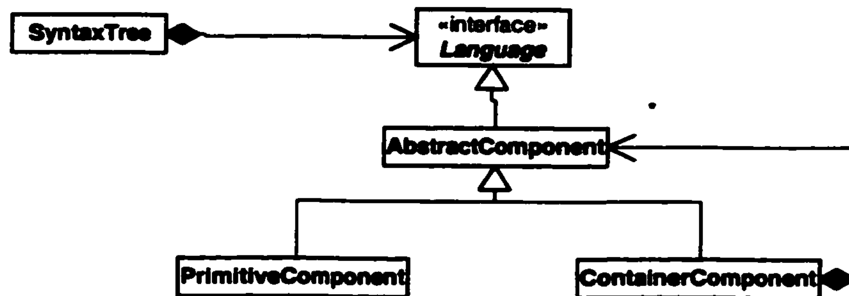

Figure 16: Structure for the Language Structure

Participants

- `SyntaxTree`
  A composite object structure that can be used to enumerate its elements.
- Language
  An interface for all components of the target language.
- `AbstractComponent`
  A place holder to group the related components into a hierarchy according to their semantics. It allows the hierarchy to be easily extended.
- `PrimitiveComponent`
  Represents an atomic component that does not contain any other components.

- `ContainerComponent`
  Represents a component other than the primitive component. It can contain primitive components and even container components.

Consequences

- The use of the **Language** interface allows different target languages to extend and prevents the client code from changing.
- A composite element can be made up of primitive elements or composite elements. The `AbstractComponent` treats elements uniformly. The language structure is easy to extend through inheritance.
- The syntax tree can be used to easily enumerate its element objects without knowledge of their concrete types.

Related Patterns

The COMPOSITE design pattern [37] treats all primitive and composite objects uniformly and define a structure that is easy to extend.

The REFLECTIVE VISITOR Pattern [53] or other variation of the Visitor pattern [52] can work with LANGUAGE STRUCTURE to perform operations (for example, code generation) on the elements in the LANGUAGE STRUCTURE.

4.1.4 ParserBuilder

Context

You are working towards the parsing process and you have applied the LANGUAGE STRUCTURE.

Problem

How to assemble the loose coupling components in the parser, while at the same time, allow it to be easily extended without modifying the existing code?

Forces

- A structure that is hard to or is restricted to modify can be extended through inheritance.
- The rule set encapsulates the application logic. If the rule set is changed or new rules are added, the parsing process needs to be changed accordingly. A changing procedure is hard to maintain and evolve.
- A stable interface can hide the implementation details and allows the implementation to change without changing the client code.
- If the parser is tightly bounded to the rule set, the parser is only meaningful when the corresponding rule set is in use. This makes the system hard to change.

Solution

Define a common parsing interface with a hook method and let a concrete class implement this hook method and wrap the parsing process for the corresponding target language. Apply the BUILDER design pattern [37] to separate the parsing process from the representation of the target language. A hook method *parse* is defined in the interface class and will be overridden in the concrete builder class. Processing of each rule is defined as a method in the concrete builder class.

Structure Figure 17 shows the structure of the PARSERBUILDER.

Participants

- `ParserBuilder`
  A class that plays the role of the `Parser Handler` and defines a hook method *parse* that needs to be overridden by the `ConcreteLanguageBuilder` to perform the actual parsing.
- `ConcreteLanguageBuilder`
  Encapsulates the grammar rules and implements the *parse* method to perform the parsing in a sequence that determined by the rules.

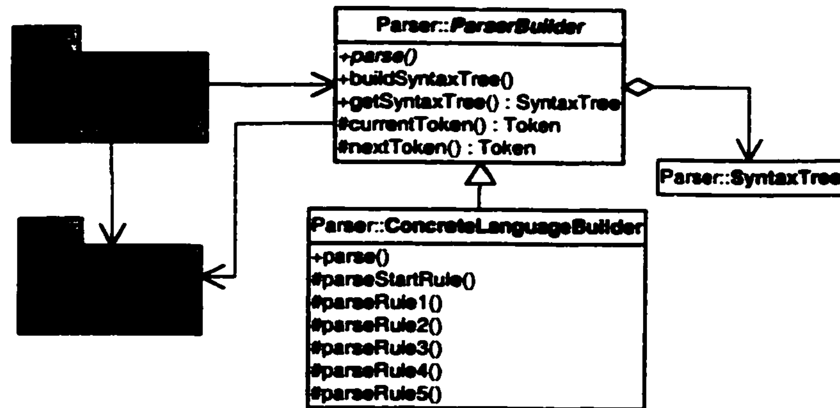

Figure 17: Structure for the ParserBuilder

- **`SyntaxTree`**
  A composite object structure that represents the parsing result and can be used to enumerate its element objects.

Collaborations Figure 18 shows the sequence diagram for the parsing process in the PARSERBUILDER.

- An object of the `ConcreteLanguageBuilder` is created for a specific target language.
- The client, compilerHandler, invokes the *parse* method on an object of class `ConcreteLanguageBuilder` to start the parsing process.
- The parsing method for each grammar rule is recursively invoked. It may need to interact with the `Lexical Analyzer` package to get tokens.
- The parsing result is added to the syntax tree.

Consequences

- Because of the use of the hook method *parse* in the interface, the client is unaware of whatever changes that may be made to the rule set and its implementation.

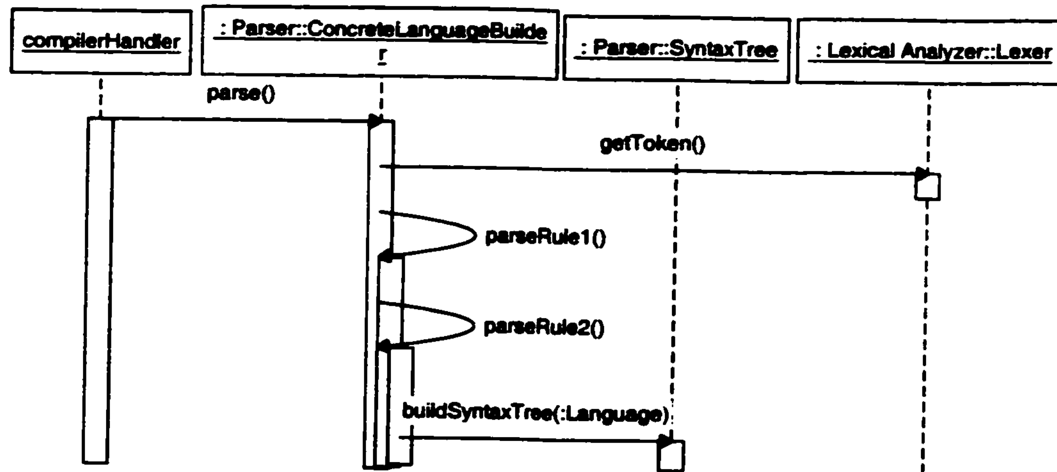

Figure 18: Sequence Diagram for the Parsing Process in the ParserBuilder

- A rule is easy to change or add by subclassing the `ConcreteLanguageBuilder`. But removing a rule will cause its corresponding method obsolete and redundant.
- The `ConcreteLanguageBuilder` will become too complex to understand and maintain if the rule set becomes large.
- It is hard to debug and maintain the rule parsing methods due to the recursive invocations among them.

Related Patterns

The BUILDER design pattern [37] separates the construction process from the object structure so that the same construction process can create different representations of the same object structure.

The METAPASER pattern that will be presented in Section 4.1.5 provides a more flexible structure for parsing.

4.1.5 MetaParser

Context

You are working towards the parsing process and you have applied the LANGUAGE STRUCTURE. You want a more flexible parser that supports its own modification at run-time.

Problem

How to encapsulate the application logic and build a self-manageable and intelligent parsing processing mechanism?

Forces

- The application logic encapsulates the changing rule set. A changing component will have limited impact on the rest of the system if it is wrapped into a separated component.
- When the rules are constantly added or are changed often, their relationships become unwieldy. A separate component may be necessary to control the spreading complexity.
- Changing software is error-prone and expensive. A desirable result is to let the software actively control its own modification.
- Changes to rules vary according to the target language. A uniform handling mechanism can lead to a system that is easy to understanding and maintain.

Solution

Apply the REFLECTION pattern [13] and define two levels in the system. The base-level contains a set of classes, where each represents a grammar rule. The meta-level handles the complex relationships of the rules that are maintained in a hash table. Reflection is used to discover rules at run-time and determines the parsing order. The base-level delegates dynamic dispatch to a meta-level object.

Structure Figure 19 shows the structure of the METAPARSER pattern. The gray area represents the meta-level of the system. The packages in gray belong to a compiler design and have direct interaction with the parser.

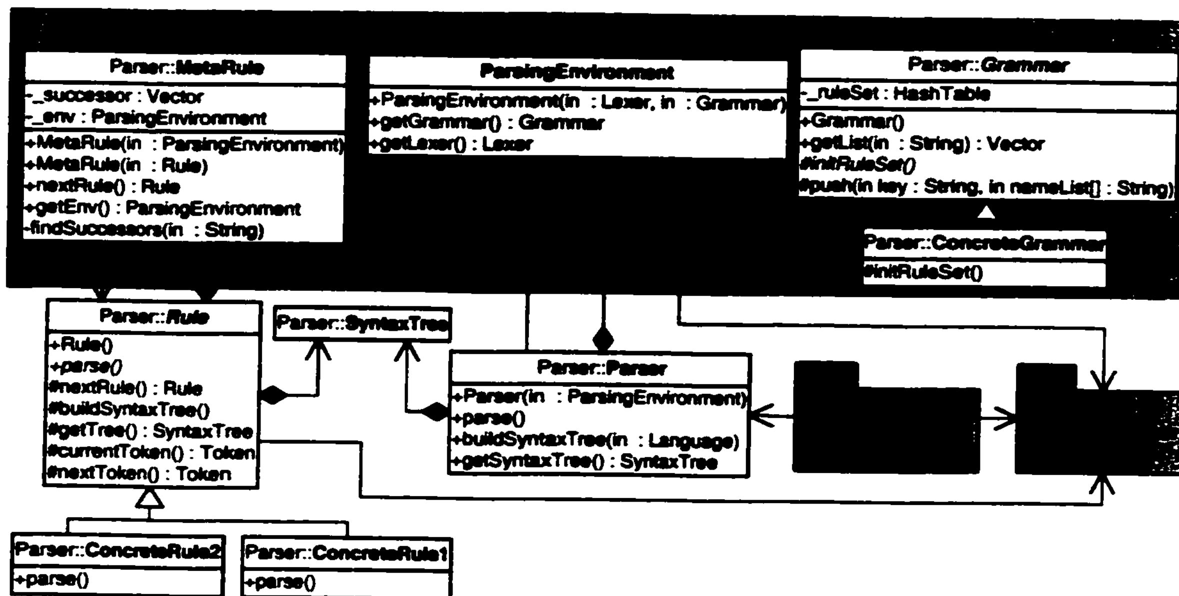

Figure 19: Structure for the MetaParser Pattern

Participants

- `Parser`

  A class that plays the role of the `Parser Handler`. The client can directly invoke its method *parse* to start the parsing process.

- `Rule`

  Defines a common interface for all grammar rules.

- `ConcreteRule`

  A concrete grammar rule defined in a target language. All grammar rules compose the rule library that can be reused over time.

- `MetaRule`

  Defines the properties of the rule. Each grammar rule class has a corresponding

metaobject whose declare type is **MetaRule**.

- **ParsingEnvironment**
  Encapsulates the parsing related information used by the **Rule**. It is managed by the **MetaRule** and shared by all **MetaRule** objects.

- **Grammar**
  Defines a common interface for all grammar rules in the potential target languages. It contains a hash table that defines the relationships of the grammar rules.

- **ConcreteGrammar**
  Represents a grammar rule in the target language. It needs to initialize the hash table by specifying the actual grammar rules in use and their relationships.

- **SyntaxTree**
  A composite object structure that represents the parsing result.

Collaborations Figure 20 shows the sequence diagram for the rule execution.

- The client invokes the *parse* method on the **Parser** to start the parsing process.

- The **Parser** initializes the **MetaRule** with the **ParsingEnvironment** object and invokes the *nextRule* method on its own **MetaRule** object to start the parsing. This **MetaRule** object then searches the hash table defined in the **ConcreteGrammar** to locate the start rule and creates the corresponding metaobject for the start.

- Once the **Parser** get the start **Rule** object from its **MetaRule**. It calls the *parse* method on that **Rule** object.

- When a rule is executed, it asks its own **MetaRule** object for the successors by invoking the *nextRule* method on this **MetaRule** object. The **MetaRule** object searches the **ConcreteGrammar** for the **Rule**'s successors. A *parse* method is then called on the successors.

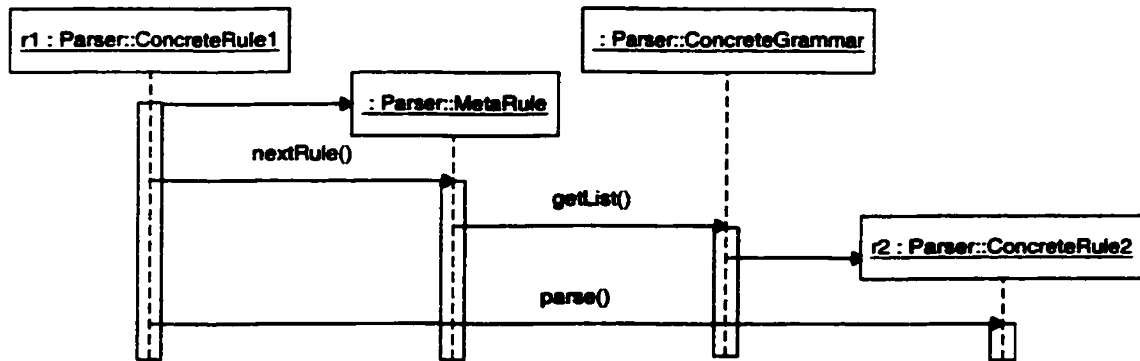

Figure 20: Sequence Diagram for the Parsing Process in the MetaParser

Consequences

- There is no need to explicitly modify the source code. Any potential changes are implicitly handled by the meta-level.
- The complexity of the system is reduced because the many-to-many relationships among the rules are changed to many-to-one relationship between the rules and the meta-level.
- The hash table that encapsulates the relationships of the rules can be modified or extended, the corresponding parsing logic and priority are then changed dynamically.
- A pool of grammar rules can be created and maintained, and optionally selected by the meta-level at run-time. The design promotes the reuse of the grammar rules even if they are defined for different target languages.
- A graded meta objects can be created to accommodate a graded complexity of the application logic. It is especially useful in incremental system development and testing.
- The design is more extensible and flexible. The grammar rules can be easily changed or extended without changing the existing classes. The hash table is

free to add, delete, or modify an entry. The debug and test become easier. Any combination of the grammar rules can be set up in the hash table for different debugging purpose.

- There two major liabilities in the design. One is that the run-time efficiency is low due to the use of the reflection technique. The other is that the increased number of classes because each rule needs be represented as an individual class.

Related Patterns

The REFLECTION pattern [13] is used to discover the grammar rules at run-time.

The ACCOUNTABILITY analysis pattern [34] defines a knowledge level (meta-level) and an operational level (base-level) to reduce the complexity of the system.

4.2 Summary

This chapter intends to address the extensibility of the parser. The patterns presented can be easily used to build an extensible parser framework. We have used them to build a compiler framework [50], which is implemented in Java. These patterns were also used in an extensible one-pass assembler developed by us [49]. This assembler is based on a virtual micro assembly language under a simple virtual processor (SVP) system and is implemented in Java. We agree that there exist different implementations of a parser in the compiler community, such as the table-driven parser [61, 57, 7], etc. The recursive-descent parser was chosen because it is one of the most difficult to extend in compiler design, we limit our discussion to the design of such a compiler system to address its extensibility.

This pattern language is by no means complete. As long as experience is accumulated in the object-oriented parser development, this language can be enriched when more and more patterns will be added.

The next chapter will present the design of a framework for the reflective class-based object model.

Chapter 5

Reflective Class-Based Object Model

This chapter presents a framework for reflective class-based object model. The framework design focuses on how to define the relationships among objects, classes, and the metaclass. This object model can be used to implement a reflective object-oriented programming language that has a single inheritance system with the support of only one metaclass.

Section 5.1 is the general principles of the reflective class-based object model. Section 5.2 presents its class hierarchy. Section 5.3.1 to Section 5.3.9 present the design of its participants in details.

5.1 General Principles

Forman's book [33] presents the theory of systematically constructing a reflective class-based model. The reflective class-based object model presented in this chapter is a simplified object model based on a single inheritance system with the support of only one metaclass. The followings are the general design principles:

- Every object has a uniquely associated entity called a class.
- Every class is an object and it has an associated metaclass `Class`.
- A metaclass is an object whose instances are classes.

- In single inheritance, the number of parents of a class is no more than one. Multiple inheritance is not allowed.
- The inheritance hierarchy has a single root class named `Object`.
- To solve the metaclass incompatibility problem [33], the object model defines a single metaclass `Class`. Subclassing the metaclass `Class` is not allowed.
- A field defined in a class is an object.
- A method defined in a class is an object.
- All fields have types that are classes of `Field`.
- All methods have types that are classes `Method`.

5.2 Class Hierarchy of the Object Model

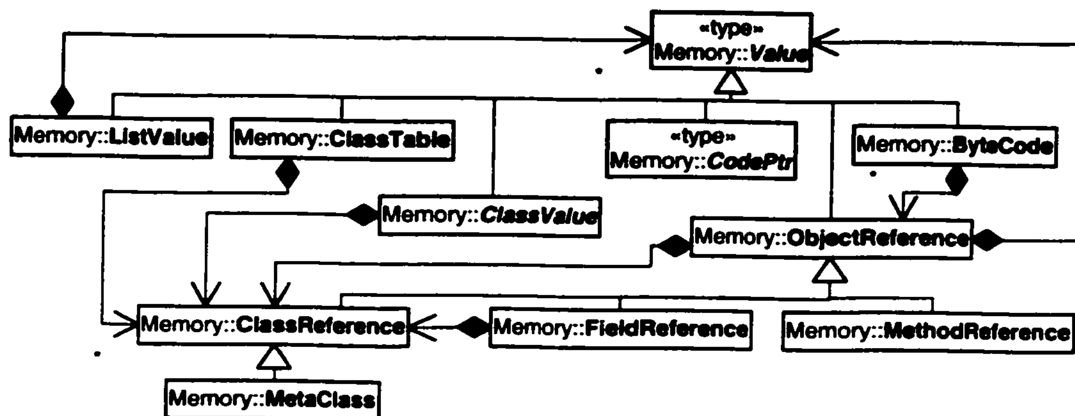

Figure 21: Class Hierarchy of the Object Model

The object model defines the relationship among objects, classes, and the metaclass. This relationship can be organized into a class hierarchy using COMPOSITE design pattern [37]. Figure 21 shows the class hierarchy of the object model. All classes defined in the object model share a unified interface `Value`. Class `ObjectReference` represents the type of an object reference. An object reference is primitive so class

ObjectReference is an immediate subclass of the interface Value. Since each class is an object, class ClassReference inherits from class ObjectReference. Each object reference has a class type ClassReference. As the class of a class (the type of a class), a metaclass is a class too. The metaclass is represented by class Metaclass, which is a subclass of class ClassReference. In this simplified object model, Metaclass has only one instance named "Class". Class ClassTable maintains all class (class reference) declarations defined in an object-oriented program.

Abstract class ClassValue defines a sub-hierarchy to encapsulate the properties of a class reference. Some of these properties may change due to the differences between different programming languages. The concrete properties of a specified programming language can be defined by subclassing the ClassValue. A field declared in a class can be considered as a special object and is represented by class FieldReference, which is derived from the class ObjectReference. A method declared in a class is also an object reference and can be represented by class MethodReference.

Each build-in class has some predefined methods. The implementation of a predefined method can be represented by a code pointer, which is a concrete subclass of CodePtr. In contrast, a user-defined method is represented in a byte code format encapsulated in the class ByteCode.

To treat composite objects and individual objects uniformly, class ListValue that contains a linked list of instances of Value is designed as a subclass of class Value.

5.3 Elements of the Object Model

5.3.1 Class Table

Class ClassTable maintains a collection of class references. It plays the role of the symbol table for classes in an object-oriented language. Figure 22 shows its class diagram.

All declared classes in a program are stored in a hash table (attribute *classes* in the class ClassTable) as pairs where the key is the class name and the slot is a reference to the corresponding instance of class ClassReference. Public method *topSort* performs a topological sort to check whether there exists a cycle in the inheritance hierarchy of the program.

The method *addClassRef* is executed during the compile time to add a newly

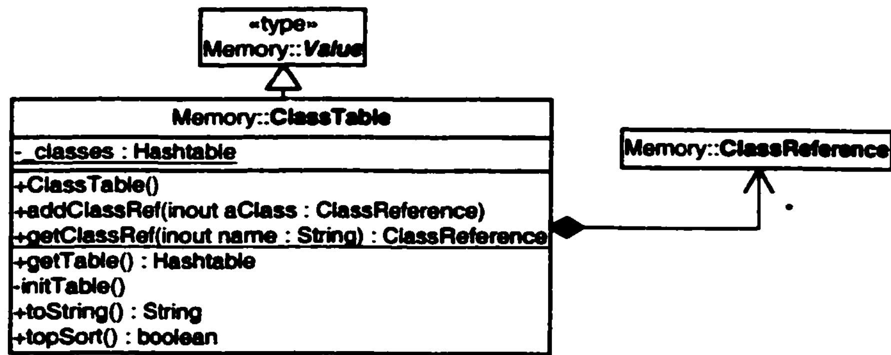

Figure 22: Class Diagram for ClassTable

recognized class into the hash table. The build-in classes are also pushed into the hash table through this method.

5.3.2 Object Reference

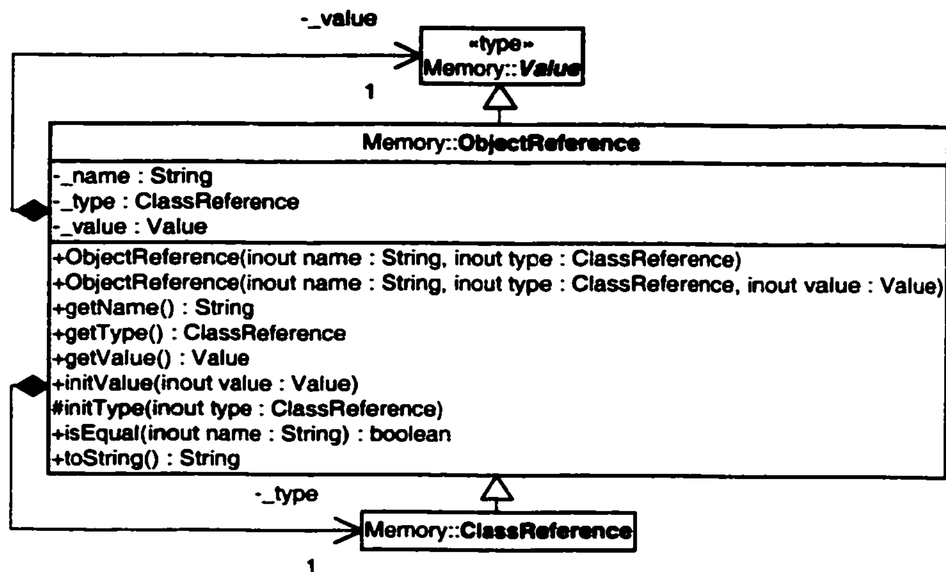

Figure 23: Class Diagram for ObjectReference

An object reference is used to identify an object. Each object has a class as its type, an object name, and an object value which specifies the properties of the

object. The structure of an object reference can be represented in a dictionary format as follows:

```
{ "type"  : ClassReference
  "name"  : String
  "value" : Value
}
```

The value of an ordinary object (i.e. the object that has no instances) is determined by the field values of its corresponding class. These fields are defined and initialized either in the current class or in its ancestor classes. The value of an object reference can be illustrated as:

```
{ "value" : ListValue = { FieldReference,
                          ...
                        }
}
```

Figure 23 shows the class diagram for class `ObjectReference`. The method *getType* returns a class reference, which is the class of this object. Using the name "getType" rather than "getClass" is to avoid conflict with the method *getClass* of class `Object` that is defined in Java. Method *isEqual* compares two object references and returns true if they refer to the same objects. The method *initValue* initializes the object with a linked list of `FieldReference`, which has the interface `Value`.

5.3.3 Class Reference

A class reference is an object reference that represents a class. a class reference can be constructed as an object reference as follows:

```
{ "type"  : ClassReference<Class>
  "name"  : String
  "value" : Value
}
```

Figure 24 shows the class diagram for class `ClassReference`. Since a class is an object, the class `ClassReference` inherits from the class `ObjectReference`. As an

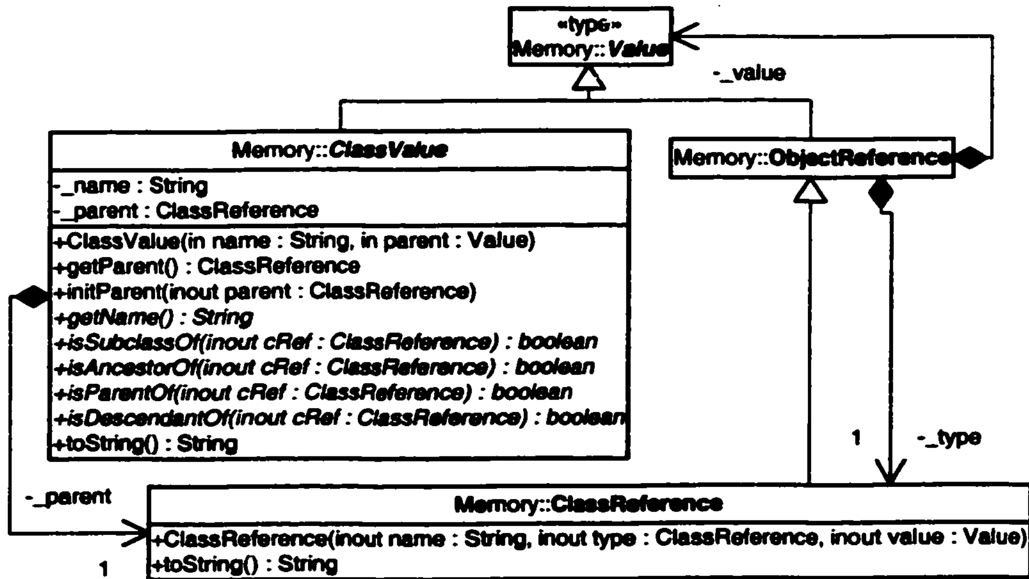

Figure 24: Class Diagram for Class Reference

object, a class must have a type known as the metaclass "Class", which is defined as `MetaClass` in this object model. The *value* of a class reference represents the properties of a class.

All these properties can be encapsulated in a sub-hierarchy with an abstract class `ClassValue` as the root. The inheritance relationship property is maintained by class `ClassValue`. Other properties such as fields and methods can be customized for a concrete programming language by subclassing the class `ClassValue`.

The inheritance relationship between two classes is defined as a reference from a class to its superclass. This reference is encapsulated in an instance of `ClassValue` with format:

```
{ "parent" : ClassReference }
```

Basically, there are five operations related to the inheritance properties of a class:

- Get Superclass
  The method *getParent* return a class reference referring to the superclass of the current class.

- Is A Subclass Of

  Given a class reference, method *isSuperclassOf* returns a boolean value that is true if this class reference refers to the superclass of the current class. Otherwise, it returns false.

- Is A Descendant Of

  Given a class reference, method *isDescendantOf* returns a boolean value that is true if the current class is one of the ancestors of what is referred by this class reference. Otherwise, it returns false.

- Is A Parent Of

  Given a class reference, method *isParentOf* returns a boolean value that is true if the current class is the superclass of what is referred by this class reference. Otherwise, it returns false.

- Is An Ancestor Of

  Given a class reference, method *isAncetorOf* returns a boolean value that is true if this class reference refers to one of the ancestors of the current class. Otherwise, it returns false.

5.3.4 MetaClass

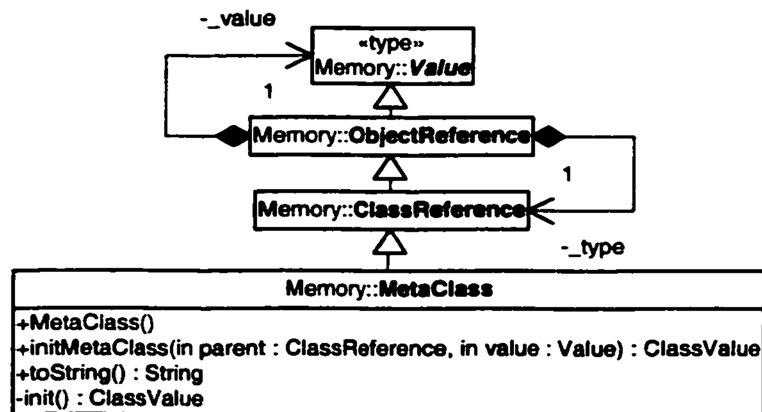

Figure 25: Class Diagram for MetaClass

`MetaClass` is a class reference that represents the unique metaclass `Class` in the object model. It is the type of all classes. The type of the `MetaClass` is itself. Figure 25 shows the class diagram for the metaclass `Class`.

The metaclass can be represented as follows:

```
{ "type"  : ClassReference<Class>
  "name"  : String<"Class">
  "value" : Value
}
```

The *type* of the metaclass is a class reference referring to itself. The *name* of the metaclass is "Class". The value of the `MetaClass` is a reference of class `ClassValue`, which will be customized for defining the metaobject protocol of the object model.

The method *initMetaClass* initializes the metaclass with a class reference to its superclass `Object` and a customized `ClassValue`. It returns a class reference referring to itself.

5.3.5 Field Reference

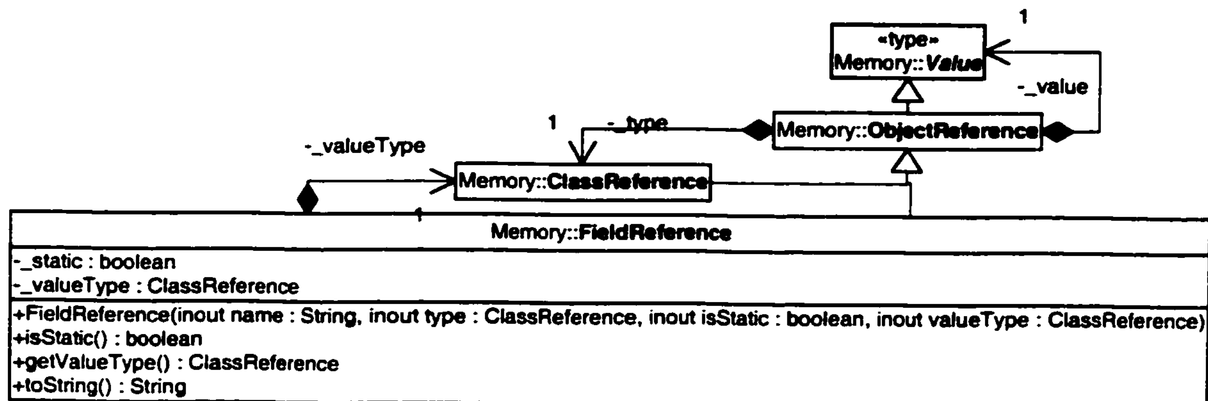

Figure 26: Class Diagram for FieldReference

A field reference is a representative of a field defined in a class. Each field is an object. Its type is a class reference to a build-in class, which is the same as the class `Field` in Java. Each field has a value that will be set up at run time. This value has a

declared type *fieldType*, which is a class reference. A field may be static or non-static. A field reference can be structured as follows:

```
{ "type"      : ClassReference<Field>
  "name"      : String
  "value"     : Value
  "fieldType" : ClassReference
  "static"    : boolean
}
```

Figure 26 shows the class diagram of FieldReference. The method *getValueType* returns the class reference of the declared type of the field's value. The method *isStatic* is a boolean function that returns true if this field is static. Otherwise, it returns false.

5.3.6 Method Reference

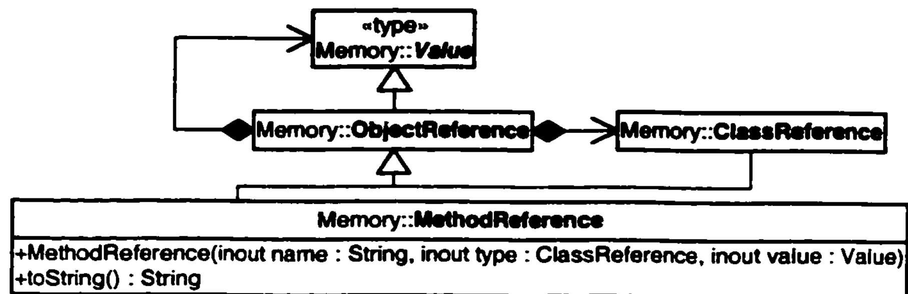

Figure 27: Class Diagram for MethodReference

A method reference is a representative of a method defined in a class. Each method is an object. A method can be represented as follows :

```
{ "type"      : ClassReference<Method>
  "name"      : String
  "value"     : Value
}
```

Figure 27 shows the class diagram of MethodReference. The class **MethodReference** is a subclass of class **ObjectReference**. The type of a method is a class reference to a build-in class, which is the same as the class **Method** in Java.

The *value* of a **MethodReference** represents the properties of a method such as return type, parameters, local variables, and method body. These properties can be encapsulated in a separate class can be customized for a concrete programming language.

5.3.7 Code Pointer

The interface **CodePtr** defines an interface for all build-in methods. These methods are generally defined in the build-in classes to provide default functionality for these classes. For each build-in method, a concrete class is defined by subclassing the class **CodePtr**. This concrete class provides the execution support for the corresponding method. Figure 28 shows the class diagram of **CodePtr**. Method *execute* is defined across the code pointer hierarchy to perform an execution of the concrete method.

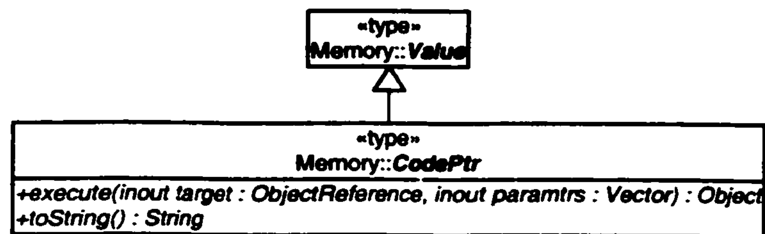

Figure 28: Class Diagram for Code Pointer

5.3.8 Byte Code

The execution information of a user-defined method are generated during the compile-time. For each user-defined method, there are two kinds of execution information:

- Run-time Literal References

  A list of literal references indicates the usages of various literals during the

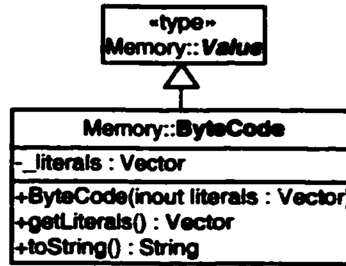

Figure 29: Class Diagram for Byte Code

method execution. These literals, such as parameters, local variables, non-static or static method calls through object reference or class reference, etc, should be valid to the method.

- Byte Code
  A list of byte code for a method are composed by opcodes and integer values. The opcodes are defined in the instruction set on the virtual machine platform.

These two execution information will be used together when a method is interpreted and executed. In Figure 29, the class `ByteCode` encapsulates the execution information (method body) of a user-defined method. The literal references and byte code list are linked together and stored in the attribute *literals*.

5.3.9 ListValue

To treat a list of `Value` and an individual `Value` uniformly, class `ListValue` is defined as a subclass of `Value` to maintain a linked list of `Values`. Figure 30 shows the class diagram for class `ListValue`.

5.4 Summary

This chapter presents the design of a framework for reflective class-based object model. This object model can be integrated in a virtual machine platform which supports the systems with a single inheritance and a unique metaclass.

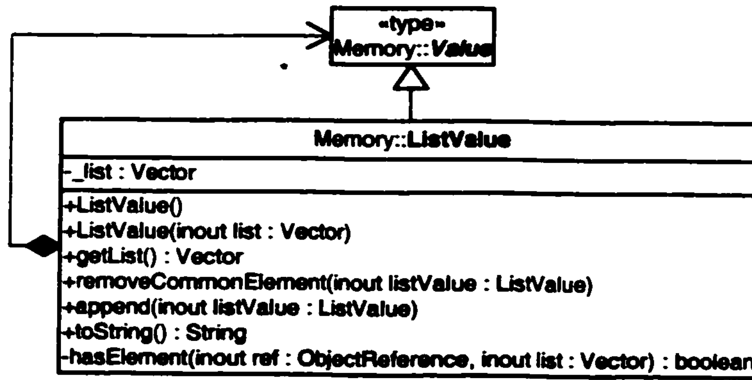

Figure 30: Class Diagram for ListValue

The next chapter will present the Decaf compiler, a customization example of the frameworks presented in this thesis.

Chapter 6

Decaf Compiler: An Application Example

This chapter illustrates how to customize our frameworks towards developing an extensible and reusable compiler system (front-end) for the Decaf programming language [26]. The Decaf programming language is an extensible tiny object-oriented programming language. We are going to focus on addressing the basic design issues in developing this compiler system. First, we will illustrate the architecture view of the compiler design and briefly introduce the component design for the lexical analyzer. Then we will show the customization of a syntactical analyzer with the parser framework introduced in Section 4.1. At the end we will illustrate how to build a code generation for the Decaf language with the REFLECTIVE VISITOR and how to implement the virtual machine based on the reflective class-based object model introduced in Chapter 5.

Section 6.1 is the Decaf language specification. Section 6.2 presents the architectural design for the Decaf compiler. Section 6.3 to Section 6.5 are the detailed design for the lexical analyzer, parser, and code generation.

6.1 Decaf Language Specification

The Decaf programming language [26] is a tiny reflective object-oriented programming language for embedded system programming. It is based on C-- [20], Java [9], Bob [10], Synapse [21, 22, 23], and React [24, 25]. The major features of the Decaf

language are uniform type system, meta-information, and classes as basic software building blocks.

6.1.1 Decaf Grammar

6.1.1.1 Conventions for Syntax

The syntax of the Decaf language [26] is described by a set of syntax rules called productions. Each production described a valid sequence of tokens called lexical elements.

Non-terminals represent a production and begin with an upper-case letter. Identifiers are named as "ID". Terminals are either keywords (with all upper-case letters) or quoted operators. Each production is terminated by a period. We use the notation for defining the syntax as shown in Table 6.1.1.1.

Table 4: Conventions of Decaf Syntax

Notation	Meaning
A*	Repetition — 0 or more occurrences of A's
A+	Repetition — 1 or more occurrences of A's
A?	Option — 0 or 1 occurrence of A's
A B	Sequence — A followed by B
A | B	Alternation — A or B
(A B)	Grouping — of a sequence A B
"0" .. "9"	Alternation — one character betwwen 0 and 9 inclusive
"string"	A string enclosed in a pair of double quotes is a reserved word

6.1.1.2 Class Declaration Productions

The following shows the productions for the class declaration in Decaf:

```
Compilation =  TypeDecl* EOF.
TypeDecl    =  "class" ID ( "extend" ID )? ClassBody.
ClassBody   =  "{" MemberDecl* "}".
MemberDecl  =  Visibility? ("static")? ID ID (";" | ("(" MethodDecl ) ).
MethodDecl  =  Signature MethodBody.
Signature   =  ParamtrList? ( ";" LocalVariableList )? ")".
```

```
MethodBody  =  "{" BlockStmt.
ParamtrList =  ID ID ( "," ID ID )*.
LocalVariableList =  ID ID ( "," ID ID )*.
```

6.1.1.3 Statement Productions

The following shows the productions for the statement definition in Decaf:

```
Statement  =  Expr ";"
           |  "if" IfStmt
           |  "while" WhileStmt
           |  "return" ReturnStmt
           |  "{" BlockStmt.
IfStmt     =  "(" Expr ")" "{" BlockStmt ( "else" "{" BlockStmt )?.
WhileStmt  =  "(" Expr ")" "{" BlockStmt.
ReturnStmt =  Expr? ";".
BlockStmt  =  Statement* "}".
```

6.1.1.4 Expression Productions

The following shows the productions for the expression definition in Decaf:

```
Expr       =  AssignExpr.
AssignExpr =  OrExpr   ( "="   AssignExpr )?.
OrExpr     =  AndExpr ( OrOp  AndExpr )*.
AndExpr    =  EquExpr ( AndOp EquExpr )*.
EquExpr    =  RelExpr ( EquOp RelExpr )*.
RelExpr    =  AddExpr ( RelOp AddExpr )*.
AddExpr    =  MulExpr ( AddOp MulExpr )*.
MulExpr    =  UnaExpr ( MulOp UnaExpr )*.
UnaExpr    =  PrimaryExpr
           |  SignExpr
           |  NotExpr
           |  "new" ID "(" ArgumentList.
SignExpr   =  AddOp UnaExpr.
NotExpr    =  "!" UnaExpr.
```

```
PrimaryExpr    =  ID ( ( "::" StaticCall )
                     | ( "."  ReferenceCall )
                     | ( "("  ArgumentList ) )?
               | "(" Expr ")"
               | IntLiteral.
StaticCall     =  ID ( "(" ArgumentList )?.
ReferenceCall  =  ID ( "(" ArgumentList )?.
ArgumentList   =  ( Expr ( "," Expr )* )? ")".
OrOp    =  "||".
AndOp   =  "&&".
EquOp   =  "=="  | "!=".
RelOp   =  "<"   | "<=" | ">" | ">=".
AddOp   =  "+"   | "-".
MulOp   =  "*"   | "/"   | "%".
Visibility   =  "+" | "-" | "#".
IntLiteral   =  ( "0" .. "9" ) | ( "1" .. "9" ) ( "0" .. "9" )+
```

6.1.1.5 Predefined Type Production

The following shows the productions for the predefined types in Decaf:

```
Type =  Object
      | Class
      | Method
      | Field
      | Integer.
```

Figure 31 illustrates the class hierarchy of the build-in classes defined in the Decaf language. The class `Object` is the root of the class hierarchy and all classes are its subclasses. The class `Class` is the metaclass and all classes (including class `Class` itself) are instances of it.

6.1.2 Virtual Machine

The virtual machine is an abstract machine that can be executed by a program such as an interpreter or assembler. It represents a machine-independent platform

Table 5: Instruction Set

Opcode	Meaning
OP_BRT	branch on true
OP_BRF	branch on false
OP_BR	branch unconditionally
OP_PUSH	push nil onto stack
OP_NOT	logical negate top of stack
OP_NEG	negate top of stack
OP_ADD	add top two stack entries
OP_SUB	subtract top two stack entries
OP_MUL	multiply top two stack entries
OP_DIV	divide top two stack entries
OP_BAND	bitwise and of top two stack entries
OP_BOR	bitwise or of top two stack entries
OP_LT	less than
OP_LE	less than or equal to
OP_EQ	equal to
OP_NE	not equal to
OP_GE	greater than or equal to
OP_GT	greater than
OP_LIT	load literal
OP_RETURN	return from interpreter
OP_CALL	call a function
OP_REF	load a variable value
OP_SET	set the value of a variable
OP_MREF	load a member variable value
OP_MSET	set a member variable
OP_AREF	load an argument value
OP_ASET	set an argument value
OP_TREF	load a temporary variable value
OP_TSET	set a temporary variable
OP_TSPACE	allocate temporary variable space
OP_SEND	send a message to an object
OP_NEW	create a new class object

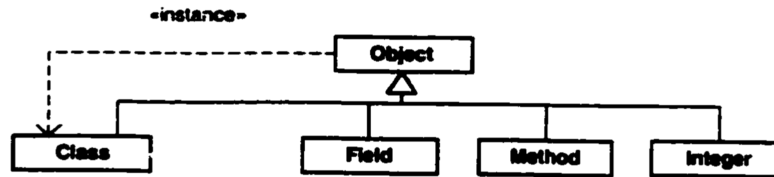

Figure 31: Class Hierarchy for Decaf Language

encapsulating all run-time information that are necessary for execution.

The Decaf virtual machine is based on the virtual machine of the Bob programming language [10]. It defines a platform that is constructed by a set of class objects, which correspond to the class declarations in the Decaf program.

An instruction set is defined in the Decaf virtual machine in order to support the compile of the methods in Decaf. This instruction set is a subset of the instruction set of the Bob virtual machine [10]. Table 5 defines the instruction set of the Decaf virtual machine.

6.2 Architectural Design

As the most critical part in developing an object-oriented system, the architectural design determines the top-level system structure of the Decaf compiler system. A good system architecture will let us keep an easy way to design a system.

Structuring a system into subsystems helps to reduce complexity. Our design model divides the compiler system based on the phases of a compiler. In general, the basic phases of a compiler system include lexical analysis, syntactic analysis, and code generation. Based on this, our design model divides the compiler system into five subsystems:

1. the runner subsystem;
2. the lexical analyzer subsystem;
3. the syntactic analyzer subsystem;
4. the code generation subsystem;

5. the memory subsystem.

6.2.1 Overview of the Subsystems

The runner subsystem is the controller whose role is to control the execution flow of the whole system. It also takes a role of the facade of the system. It is visible to the client of the compiler. It takes the responsibility to response the clients' requests and to offer the compilation services.

The task of the lexical analyzer subsystem is to break the source code into meaningful units, the tokens. It executes the tokenizing process and frequently forwards the recognized token to the syntactic analyzer upon request.

The syntactic analyzer subsystem is the core of the front end of the compiler. It analyzes the structure of the source program and checks for syntactic errors.

The code generation subsystem takes the action to produce the compiling results on a pre-defined virtual machine. It also performs partial semantic checking on the program.

The memory subsystem handles the run-time memory management of the compiler system. It encapsulates data structures generated during the compilation process. Typically, the memory subsystem represents the object model of the virtual machine structure resulting from the code generation phase.

We will further discuss each subsystems in the following sections.

6.2.2 Dependency Relationships among the Subsystems

In the architectural design, the idea of building the design model for the potential solution is to capture the relationships between the subsystems and define the top-level classes for each subsystem.

The relationships between the subsystems reflect the flexibility and extensibility features of an extensible compiler system. We aim for reducing the coupling among the subsystems. Figure 32 illustrates the package diagram for the extensible Decaf compiler system.

To reduce the coupling, the major components of the compiler system (lexical analyzer, syntactic analyzer, and code generation) will not have mutual knowledge of each other. The runner subsystem coordinates the execution of the system and

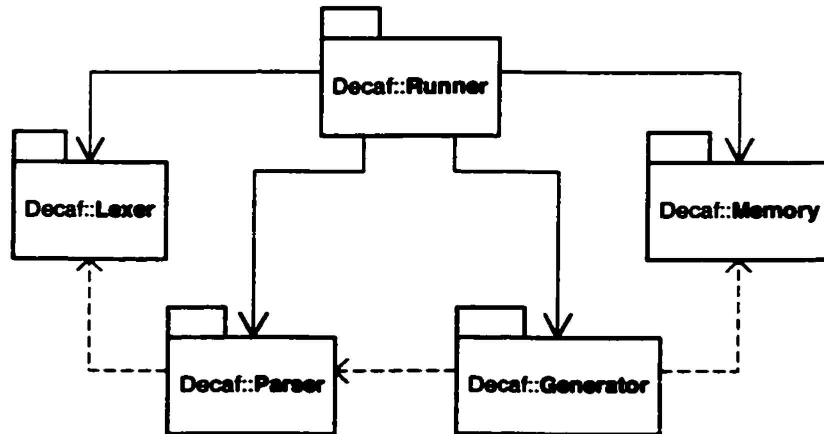

Figure 32: Package Diagram for Decaf Compiler

the message passing between these subsystems. It also plays a role of the creator and the owner of the instances of the top-level classes in these subsystems. Thus, the relationships among the lexical analyzer, the syntactic analyzer, and the code generation subsystem can be designed as the association relationships between each of the subsystems and the runner subsystem via the callback implementations.

6.3 Lexical Analyzer

An extensible lexical analyzer is important when developing an extensible and reusable compiler system. We need to minimize the impact from any changes of the language specification, especially from the token type and the language reserved word set. Hence, the lexical analyzer subsystem can be designed by dividing it into three components as showed in Figure 33.

6.3.1 The Lexer Component: Facade of the Lexical Analyzer

The Lexer component defines the `Lexer` interface and its implementation subclass, the `DecafLexer` in Decaf. The Lexer component plays two roles in the lexical analyzer

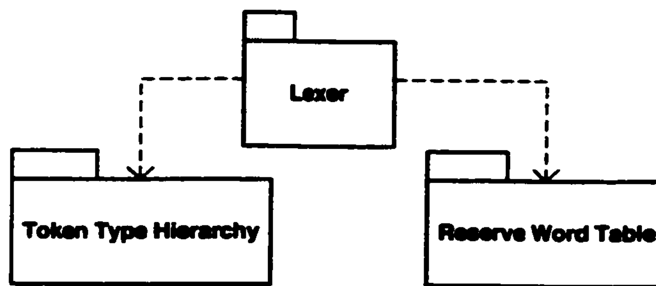

Figure 33: Separating the Lexical Analyzer Subsystem into Three Components

subsystem: one is the facade of the subsystem; the other is the controller of the tokenizing process.

Method *getToken* is a public method declared in the **Lexer** interface. It forwards the recognized token to the syntactic analyzer upon request. The subclass **DecafLexer** implements this method by scanning the source code stream and recognizing the valid token. Figure 34 shows the class diagram for the Lexer component.

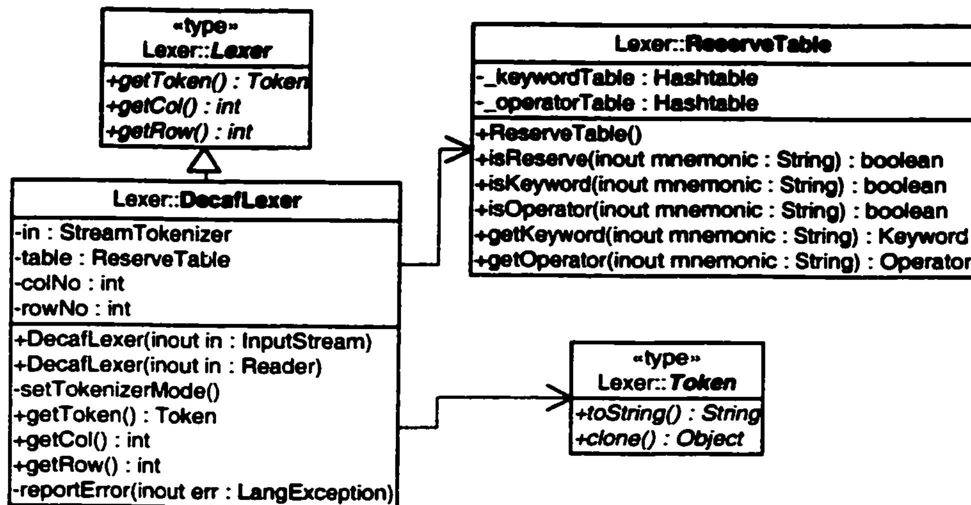

Figure 34: Class Diagram for the Lexer Component

6.3.1.1 The Facade of the Lexical Analyzer Subsystem

The responsibility of a lexical analyzer is to response the requests from the syntactic analyzer and to provide the recognized tokens. In order to reduce the coupling between the lexical analyzer and the other parts of the compiler system and hide the

concrete implementation information, we introduce the FACADE pattern [37] in the design of the lexical analyzer subsystem.

The public **Lexer** interface as well as its implementation subclass (**DecafLexer**) construct the facade of the lexical analyzer. They are declared as public and are exposed to the outside of the subsystem. The syntactic analyzer can communicate with the Lexer component to get the token.

6.3.1.2 The Controller of the Lexical Analyzer

Class **DecafLexer** plays a role of the control handler of the tokenizing process. It responses the request from the syntactic analyzer, searches in the reserved word table and determines whether or not the current mnemonic (i.e. lexeme) matches any system reserved symbol. According to the search result and the token type hierarchy, it creates and returns the corresponding token object. The token information includes the content of the token, the type of the token, and the position of the token.

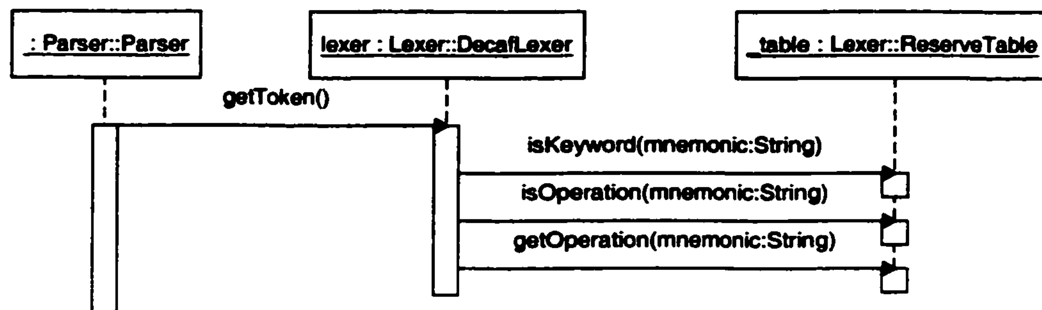

Figure 35: Sequence Diagram for a Tokenizing Process

Figure 35 shows a scenario of getting an operation token. When the `Parser` calls the `DecafLexer` requesting a token, The `DecafLexer` will ask the `ReserveTable` to justify the token. If the token is a keyword, method *getKeyword* is called on the `ReserveTable` to return a keyword token to the `Parser`. In this scenario, the token is not a reserved word. Then method *isOperation* is called on the `ReserveTable`. Since the token is an operation, it returns true and method *getOperation* is called on the `ReserveTable` to return an pre-defined operation token to the `Parser`.

6.3.2 The Token Type Hierarchy

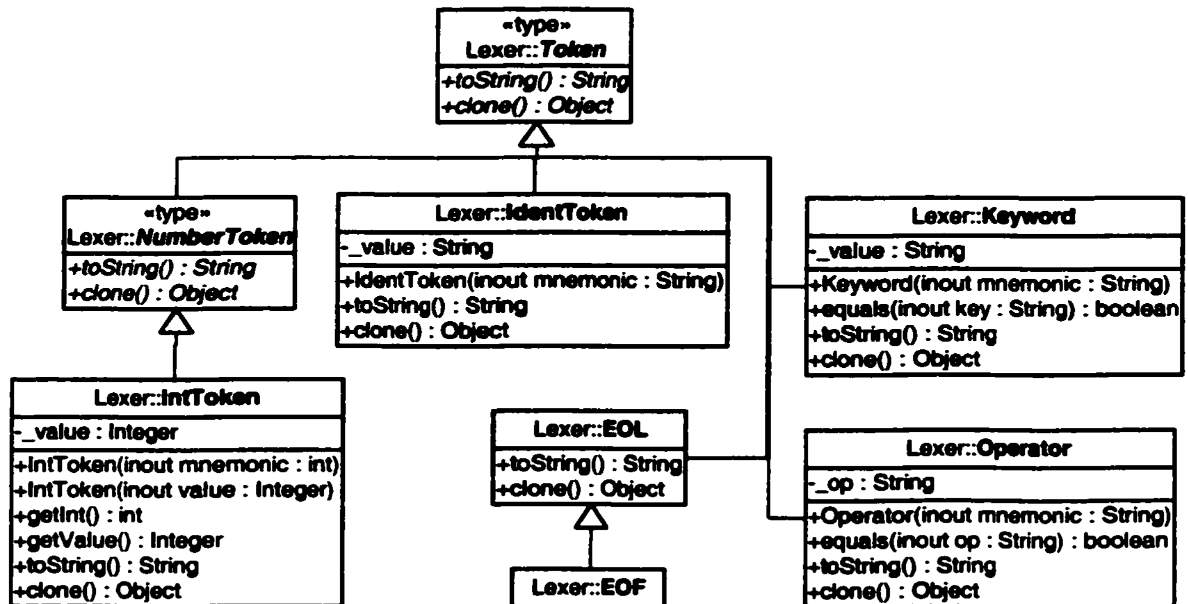

Figure 36: The Token Type Hierarchy

To easily manage and extend the lexical analyzer, we separate the token type information from the tokenizing process. The key point is to abstract the token types and organize them into a class hierarchy. This abstraction gives us more flexibility for a modest increase in complexity.

Figure 36 shows the token type hierarchy. There are six token types defined in Decaf: Integer, Identifier, Keyword, Operator, EOL (end of line), and EOF (end of file). The root class is the `Token`. Each token type is defined as a concrete subclass. A sub-hierarchy is defined for all numbers with interface `NumberToken` as the root. Class IntToken is a subclass that implements the `NumberToken` interface. New types of numbers can be easily added by implementing the `NumberToken` interface. `EOF` is treated as a special case of `EOL`, so it is defined as a subclass of `EOL`.

6.3.3 The Reserved Word Table and the Flyweight Pattern

Different program languages may define different system reserved symbol set, which includes reserved words (i.e. keywords) and operation symbols (e.g. "+","-", etc.). To reduce the impact on the compiler system due to any changes of the reserved symbol set, we separate this set from other parts of the lexical analyzer subsystem. Our design is to keep the set elements in a hash table. The controller of the lexical analyzer can search the hash table to determine whether the current lexeme is a system reserved symbol or not.

If the source code is very large, a large number of the token objects will consume lots of the memory and may incur unacceptable run-time overhead. To handle this problem, the FLYWEIGHT pattern [37] can be applied to reduce the number of the token objects. The FLYWEIGHT pattern uses sharing to support large number of objects efficiently.

6.4 Customize the Parser Framework

The syntactic analyzer performs the syntax checking for the Decaf program. In Section 4.1, we have introduced a pattern language for the design of a parser framework. This pattern language includes four patterns. The PARSERBUILDER pattern and METAPARSER pattern are alternative solutions to address the dynamic aspects of the parsing process. We choose to apply the METAPARSER pattern in the Decaf compiler design since it is more flexible and extensible. Therefore, three patterns in the pattern language are applied to the syntactic analyzer design of the Decaf compiler:

- PARSER STRUCTURE pattern
  This pattern reflects the architectural design for the parser. By applying the PARSER STRUCTURE pattern, the Decaf parser is composed by three components: parser handler, language structure, and grammar rules.

- LANGUAGE STRUCTURE pattern
  This pattern is used to define the language structure for the Decaf compiler. It addresses the static relationships among the elements that make up the Decaf language. This pattern organizes the language components as a composite

hierarchical structure. The parsing result, the abstract syntax tree, is a composite object created based on the language structure. To construct the language structure for Decaf language, we implement it in three steps:

1. Construct expression hierarchy.
2. Construct statement hierarchy.
3. Construct class declaration hierarchy.

- METAPARSER pattern
  This pattern addresses the parsing process and constructs the abstract syntax tree during the parsing. It divides the system into two levels, meta-level and base-level. The base-level contains a set of RULE objects defined based on the language grammar. The objects of class **`MetaRule`** in the meta-level take a role as metaobjects in the REFLECTION pattern. They control the execution sequence of the parsing process. Thus, the **`Rule`** objects in the base-level are isolated each other and have no knowledge about other rules. To customize the METAPARSER, we need to define all the concrete **`Rule`** classes based on the Decaf language grammar.

6.4.1 Define the Language Structure

A programming language is specified by its grammar. The language structure is an abstraction of the grammar. It represents the static view of the grammar. The LANGUAGE STRUCTURE pattern defines the language structure as a composite hierarchy that captures the static relationships among the various language components.

In order to explain how this language structure can be easily extended, we implement the language structure for Decaf grammar in the following three steps:

1. Construct the expression hierarchy. This hierarchy defines the most basic operations in most programming languages.
2. Construct the statement hierarchy. This hierarchy supports different statements such as conditional statement and loop statement. The expression hierarchy is integrated into statement hierarchy as a whole.

3. Construct the class declaration hierarchy. This hierarchy represents the whole Decaf object-oriented language.

The language structure shares a unified interface named **Language**. As the result of parsing, the abstract syntax tree is a composite object created based on the language structure. On the other hand, code generation process can be viewed as a visiting process on the abstract syntax tree. Therefore, interface `Visitable` is defined as the root of the language structure hierarchy to identify the elements that can be visited. That is, the **Language** then becomes a interface subclass of the `Visitable`.

6.4.1.1 Construct Expression Hierarchy

Expressions defined in the Decaf grammar can be grouped into three major categories and some special kinds of expressions. These three categories are: expressions with two operands (binary expressions), expressions with one operand (unary expressions), and primary expressions such as constants, variables, and method calls.

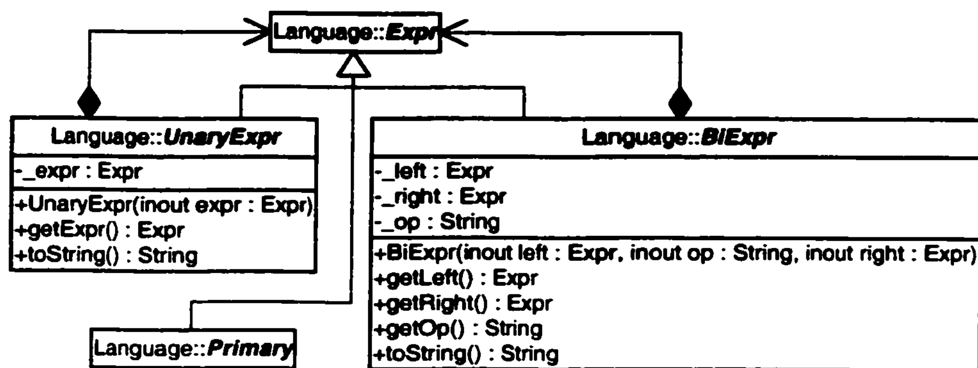

Figure 37: Top Level Classes in the Expression Hierarchy

Figure 37 shows the top level classes in the expression hierarchy. Each category is organized into a hierarchy. **UnaryExpr** is the root class for all unary expressions, **BiExpr** is the root class for all binary expressions, and **Primary** is the root class for all primitive constructs such as constants, variables, and method calls. All expressions derive from the same global root interface **Expr**.

Expressions with Two Operands All expressions with two operands are grouped into a class hierarchy called binary expression hierarchy, which has a root abstract class `BiExpr`. According to the Decaf grammar, this binary expression hierarchy can be further specialized into three categories: arithmetic expressions, relational expressions, and logical expressions. Hence, we define three sub-hierarchies: the arithmetic expression hierarchy with an abstract class `ArithmeticExpr` as the root, the relational expression with an abstract class `RelaExpr` as the root, and logical expression with an abstract class `LogicalExpr` as the root. Figure 38 shows the class diagram of the binary expression hierarchy.

Expressions with One Operand All expressions with one operand are grouped into a class hierarchy called unary expression hierarchy, which has a root abstract class `UnaryExpr`. As shown in Figure 39, the expressions with one operand (i.e. unary expressions) include:

- `ParenExpr`, which represents the parentheses expression that is an expression within a pair of parentheses.
- `LogicNOT`, which represents an expression that is performed by a "logic not" (prefixed by a symbol "!") operation.
- `SignExpr`, which represents an expression that is performed by a negative or positive operation (prefixed by a symbol "-" or "+").

Classes `LogicNOT` and `SignExpr` can be generalized into a sub-hierarchy with abstract class `PrefixExpr` as the root.

Primary Expressions All primary expressions are grouped into a separate hierarchy with abstract class `Primary` as the root. These primary expressions can be further divided into three categories: constants, variable accesses and method calls. Each of these categories is defined into a sub-hierarchy. The constant hierarchy defines the class `Constant` as the root and includes a subclass `IntLiteral` that represents the integer constants in the Decaf program. The variable access hierarchy defines the class `Variable` as the root. There are two kinds of variable accesses defined in the Decaf language: one is the static field access that the field is accessed through a class name, and the other is the reference field access that the field is accessed through an object

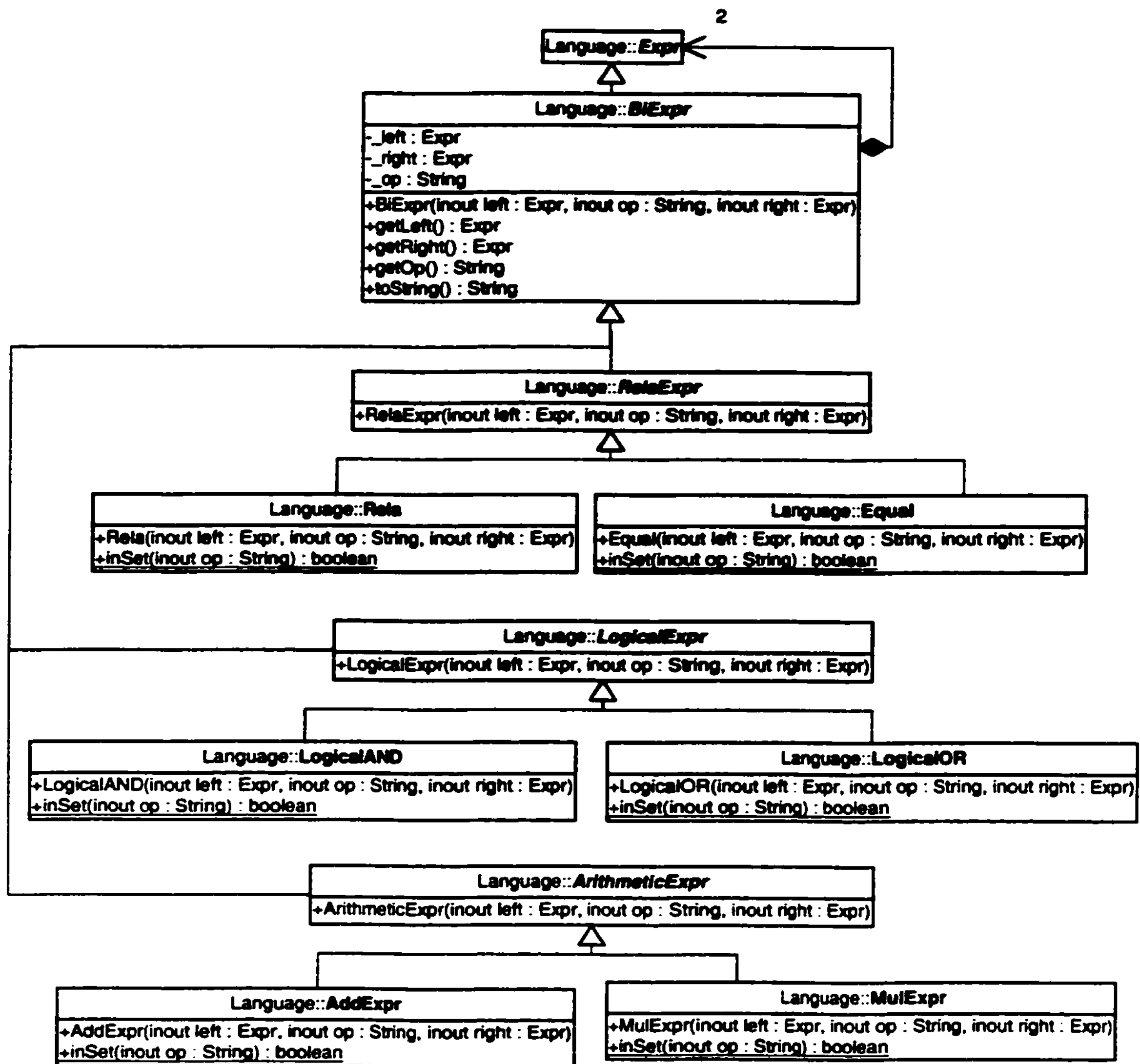

Figure 38: Class Diagram for Expressions with Two Operands

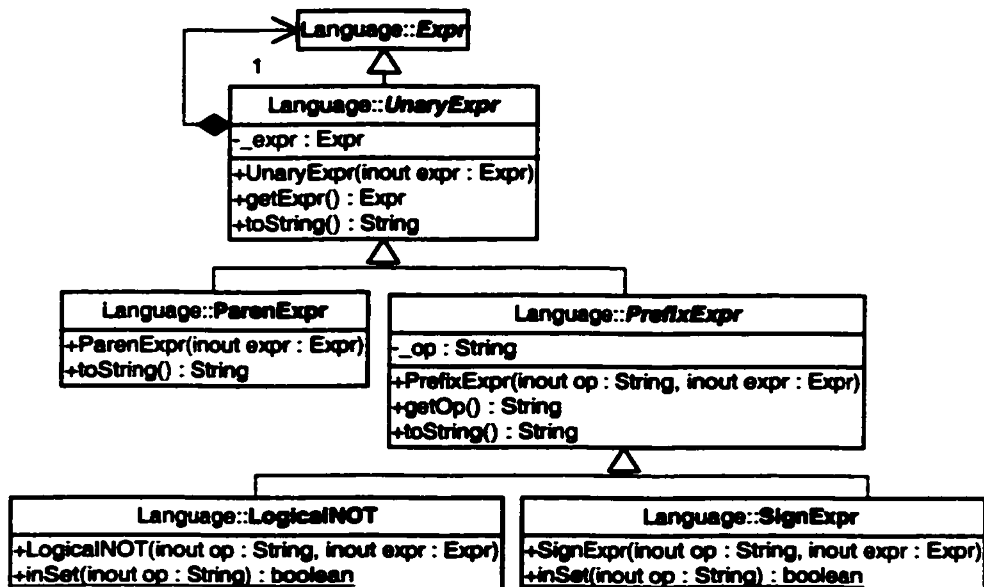

Figure 39: Class Diagram for Expressions with One Operand

reference. These two categories are represented by two classes **StaticFieldCall** and **ReferenceFieldCall** as shown in Figure 40. Similarly, there are two kinds of method classes in the Decaf language: one is the static method call where the method is called through a class name, and the other is the reference method call where the method is called through an object reference.

Special Expressions The expressions that can not be grouped into any of the above expression category are considered as special cases of expressions. The special expressions in the Decaf language include assignment expression, "new" expression, and variable declaration. These three special expressions are represented by class AssignExpr, class NewExpr, and class VariableDecl in the Expr hierarchy as shown in Figure 41.

6.4.1.2 Construct Statement Hierarchy

The Decaf language grammar defines a statement as:

Statement = Expr ";"
| "if" IfStmt

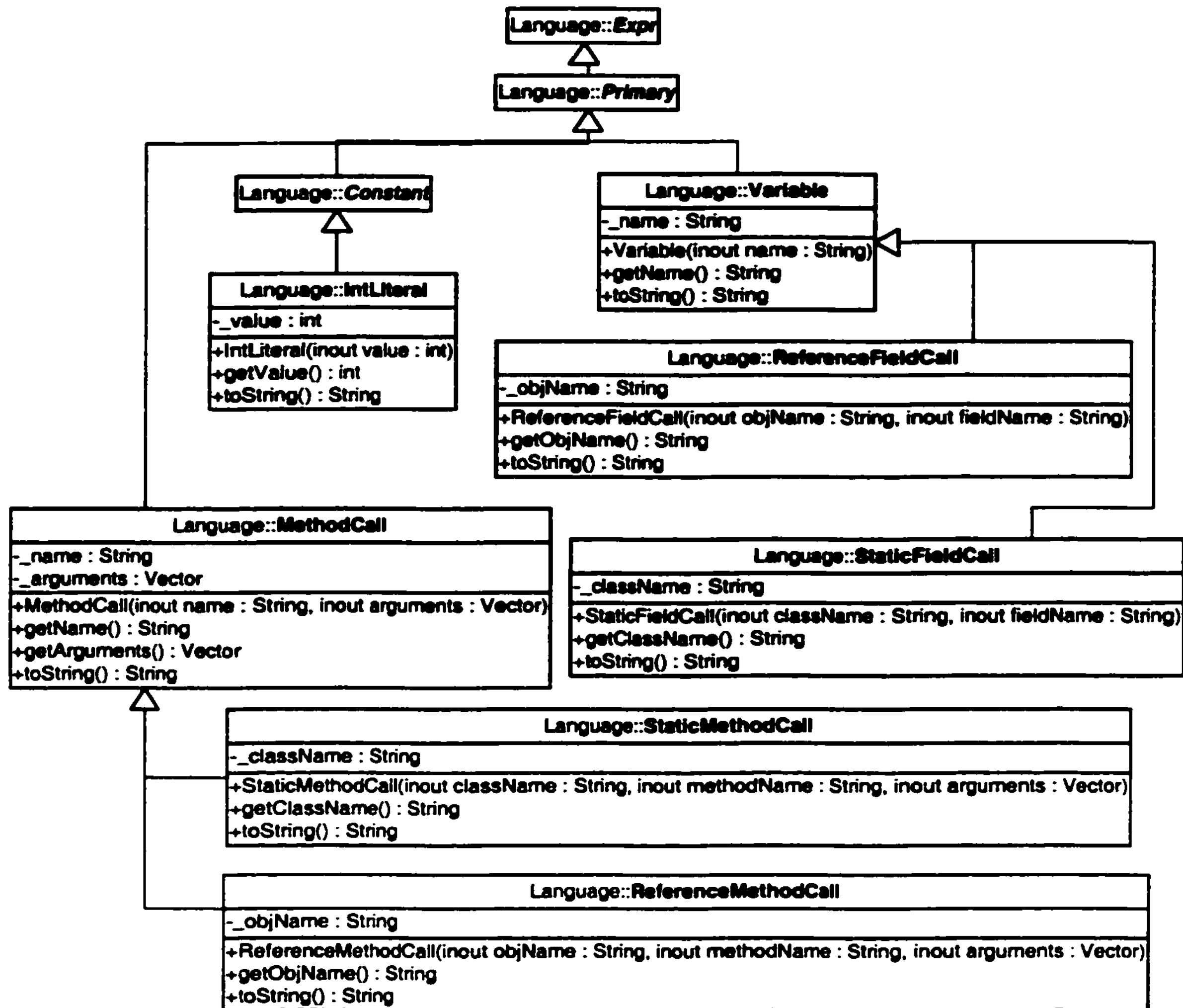

Figure 40: Class Diagram for Primary Expressions

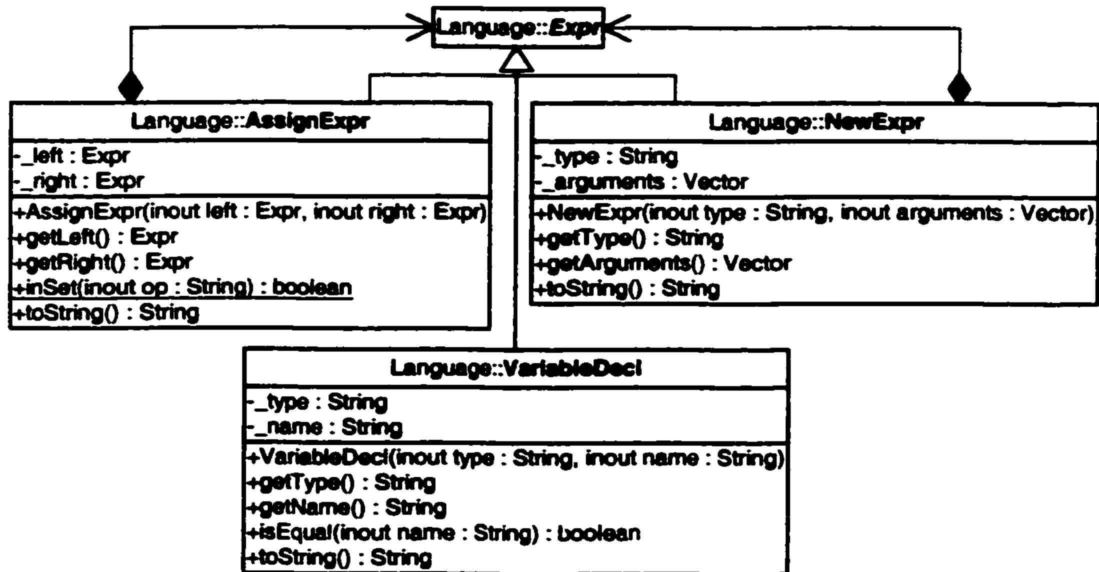

Figure 41: Class Diagram for Special Expressions

| "while" WhileStmt
| "return" ReturnStmt
| "{" BlockStmt

These grammar rules show that there are five kinds of statements defined in the Decaf language:

- expressions.
- "if" statement represented by class `IfStmt`.
- "while" statement represented by class `WhileStmt`.
- "return" statement represented by class `ReturnStmt`.
- block statement represented by class `BlockStmt`.

We organize these statements into a statement hierarchy that defines the interface `Statement` as the root. Figure 42 shows the class diagram of this statement hierarchy.

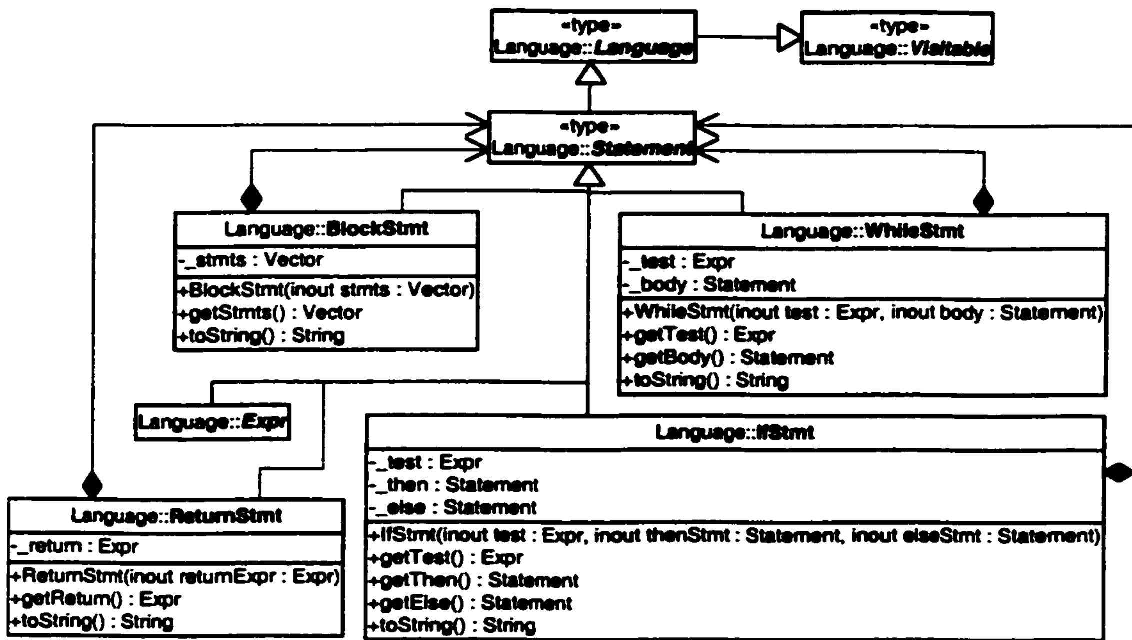

Figure 42: Class Diagram for Statement Hierarchy

6.4.1.3 Construct Class Declaration Hierarchy

Section 6.1.1.4 presents the productions of class declaration in the Decaf language. These productions can be separated into two categories: class declaration and class member declaration.

Class Declaration The design for the class declaration hierarchy are based on the following forces:

- A Decaf language (`DecafLanguage`) Is A kind of language (`Language`).
- A Decaf language is composed by a series of class declarations (`ClassDecl`).
- class declarations are distinguished by their class names, and each class declaration has a super class name associated with it. The contents of a class are encapsulated in a class body (`ClassBody`).
- A class body is composed by a series of member declarations (`MemberDecl`).

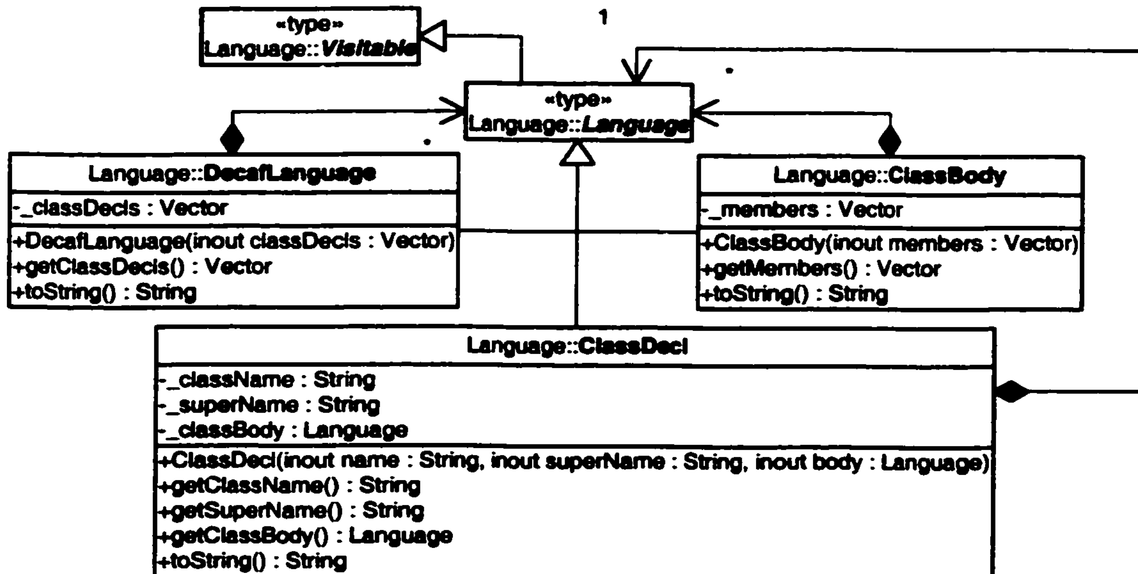

Figure 43: Class Diagram for Class Declarations

Figure 43 shows the class diagram for class declaration. The participants in this class diagram are organized with a COMPOSITE pattern.

Member Declaration In the Decaf language, class members are divided into two categories: attributes and behaviors. Figure 44 shows the class diagram for class member declaration in Decaf. The declarations for these two categories are designed as class `FieldDecl` and class `MethodDecl`.

6.4.2 Define the Grammar Rules

The METAPARSER pattern introduced in Section 4.1 separates the parser design into two levels: the meta-level and the base-level. The meta-level handles the relationship between the rules. It determines the parsing execution sequence. The base-level defines a rule library that includes a set of `Rule` classes. Each of these rules is associated with a `MetaRule` object defined in the meta-level, which determines this rule's execution successors during the parsing.

To apply this pattern to the parser design for the Decaf program, the rule library needs to be constructed based on the Decaf grammar and the relationship between

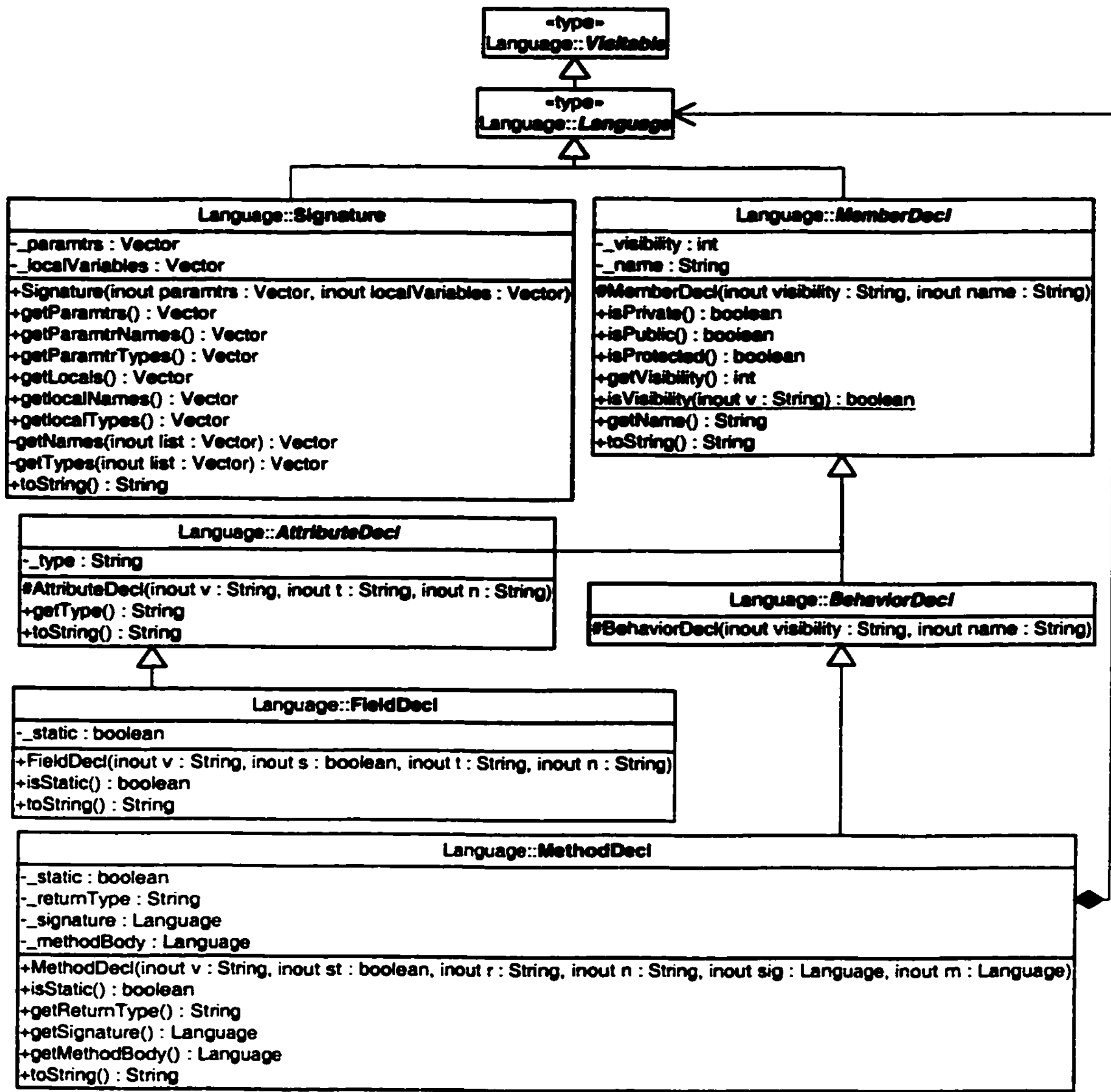

Figure 44: Class Diagram for Member Declarations

the rules must be specified and initialized in the meta-level class `Grammar`.

6.4.2.1 Construct Rule Library for Decaf Language

Figure 45 shows the class diagram of the rule library defined for the Decaf language. The abstract class `Rule` is the root class of all the rules in the rule library. The method *parse* is defined throughout the rule hierarchy to perform the parsing operation on each concrete rule. The method *init* gets references to rule's successors from the meta-level (a corresponding `MetaRule` object). The `Rules` in the rule library are designed based on the productions listed in the Decaf grammar (Section 6.1.1).

6.4.2.2 Initialize the Rules' Relationships in the Meta-level

Figure 46 shows the class diagram for participants who initialize the rule's relationships. The class `DecafGrammar` is derived from the class `Grammar`. It implements the hook method *initRuleSet* to initialize the grammar relationships for the Decaf language. The class `ParsingEnvironment` maintains and supplies information for parsing at the meta-level.

The relationship between the rules must be specified and initialized in the meta-level. It is abstracted directly from the grammar rule productions. This relationship is defined as a sequence of rule names, in which the first rule name in the sequence represents the rule that is applied and the subsequence represents the successors after this rule is applied.

The following programming list shows the implementation of the class `DecafGrammar`, which initializes the relationships between the rules defined in the rule library. The relationships are maintained by a hash table defined in the super class `Grammar`, in which the key represents the current rule name that is applied and the value refers to the rule names of its successors.

```
public class DecafGrammar extends Grammar {
    protected void initRuleSet(){
        push( "START",             new String[] {"Compilation_Rule"} );
        push( "Compilation_Rule",  new String[] {"TypeDecl_Rule"} );
        push( "TypeDecl_Rule",     new String[] {"ClassBody_Rule"} );
        push( "ClassBody_Rule",    new String[] {"MemberDecl_Rule"} );
```

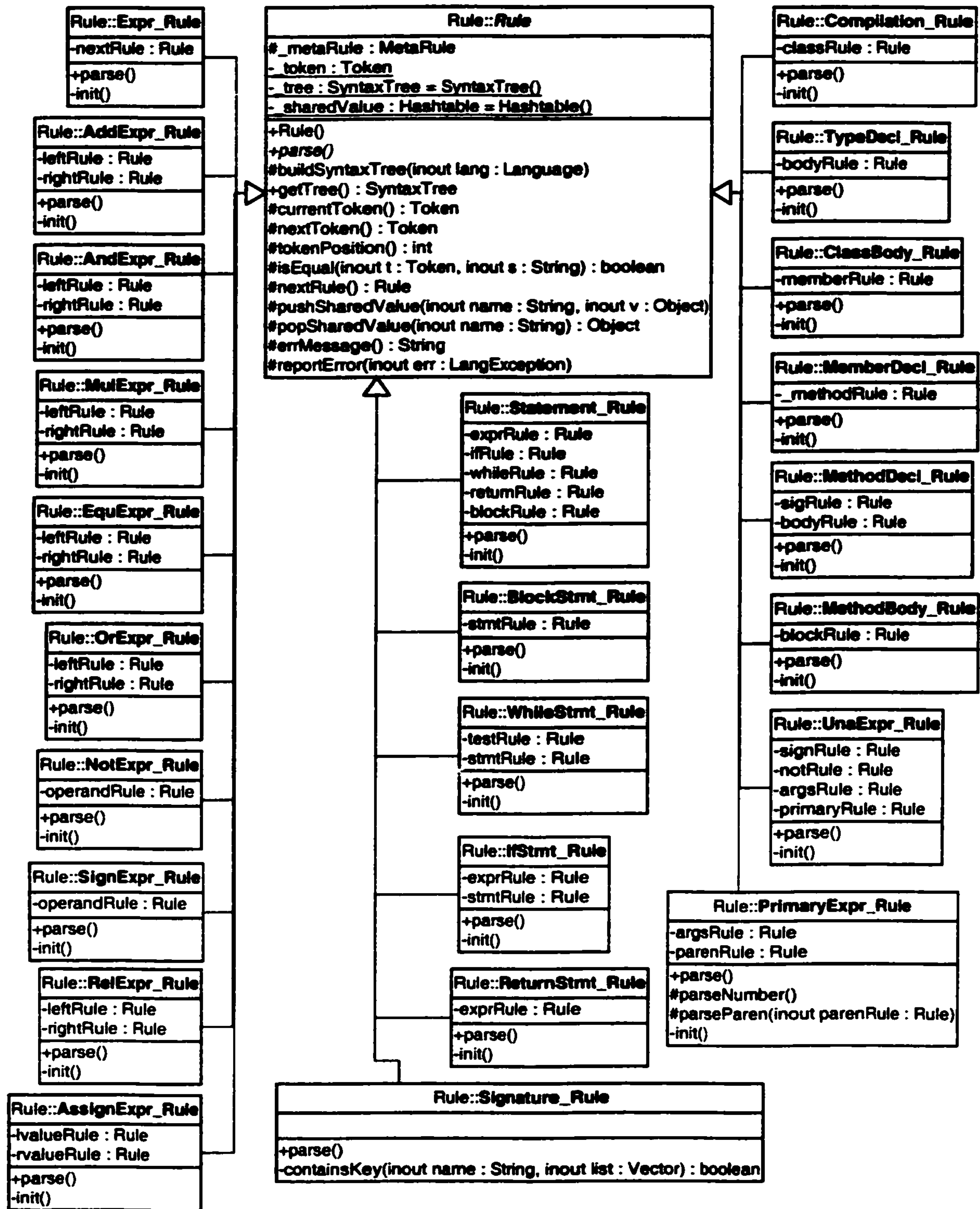

Figure 45: Class Diagram for Rule Library

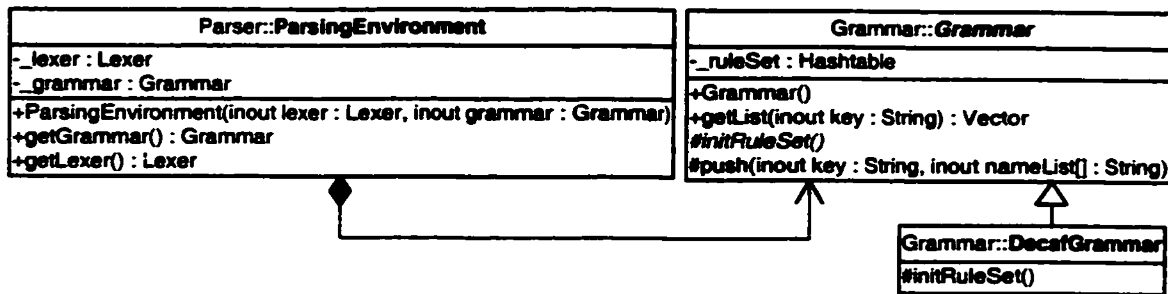

Figure 46: Class Diagram for Class DecafGrammar

```
push( "MemberDecl_Rule",   new String[] {"MethodDecl_Rule"} );
push( "MethodDecl_Rule",   new String[] {"Signature_Rule",
                                         "MethodBody_Rule"} );
push( "MethodBody_Rule",   new String[] {"BlockStmt_Rule"} );
push( "BlockStmt_Rule",    new String[] {"Statement_Rule"} );
push( "Statement_Rule",    new String[] {"Expr_Rule",
                                         "IfStmt_Rule",
                                         "WhileStmt_Rule",
                                         "ReturnStmt_Rule",
                                         "BlockStmt_Rule"} );
push( "IfStmt_Rule",       new String[] {"Expr_Rule",
                                         "Statement_Rule"} );
push( "WhileStmt_Rule",    new String[] {"Expr_Rule",
                                         "Statement_Rule"} );
push( "ReturnStmt_Rule",   new String[] {"Expr_Rule"} );

push( "Expr_Rule",         new String[] {"AssignExpr_Rule"} );
push( "AssignExpr_Rule",   new String[] {"OrExpr_Rule",
                                         "AssignExpr_Rule"} );
push( "OrExpr_Rule",
              new String[] {"AndExpr_Rule", "AndExpr_Rule"} );
push( "AndExpr_Rule",
```

```
                  new String[] {"EquExpr_Rule", "EquExpr_Rule"} );
        push( "EquExpr_Rule",
                  new String[] {"RelExpr_Rule", "RelExpr_Rule"} );
        push( "RelExpr_Rule",
                  new String[] {"AddExpr_Rule", "AddExpr_Rule"} );
        push( "AddExpr_Rule",
                  new String[] {"MulExpr_Rule", "MulExpr_Rule"} );
        push( "MulExpr_Rule",
                  new String[] {"UnaExpr_Rule", "UnaExpr_Rule"} );
        push( "UnaExpr_Rule",     new String[] {"SignExpr_Rule",
                                                "NotExpr_Rule",
                                                "Expr_Rule",
                                                "PrimaryExpr_Rule"} );
        push( "SignExpr_Rule",    new String[] {"UnaExpr_Rule"} );
        push( "NotExpr_Rule",     new String[] {"UnaExpr_Rule"} );
        push( "PrimaryExpr_Rule",
                 new String[] {"Expr_Rule", "Expr_Rule"} );
    }
}
```

6.5 Code Generation

The code generation process generates the compiling result on the virtual machine by visiting the abstract syntax tree, which is created as a result of the syntactic analysis. Considering the extension on both the Decaf grammar and the virtual machine, we choose to apply the REFLECTIVE VISITOR pattern to the code generation design. The application of the REFLECTIVE VISITOR pattern can reduce the impact to the existing system due to any changes in the language specification. On the other hand, the reflective class-based object model introduced in Chapter 5 provides an extensible solution to implement the virtual machine.

6.5.1 Extend the Reflective Class-based Object Model for Decaf

Chapter 5 presents a framework for reflective class-based object model. This object model framework defines the relationships among object, class, and metaclass. It also provides some utility classes such as class `ListValue`, class `ByteCode`, and interface `CodePtr`. To implement a reflective object-oriented programming language, more classes should be defined based on the specified language grammar. These classes includes:

- Class `DecafClassValue` is derived from class `ClassValue` to encapsulates the properties of a class reference in Decaf. Attribute *value* defined in class `ClassReference` refers to an instance of class `DecafClassValue`.
- Class `MethodValue` encapsulates the properties of a method. Attribute *value* defined in class `MethodRefrence` refers to an instance of class `MethodValue` in Decaf.
- Subclasses of the interface `CodePtr` implement the build-in methods that defined for the build-in classes.
- Build-in classes. According to the Decaf grammar, some basic build-in classes are necessary for the system. These classes include: `Object`, `Method`, `Field`, and `Integer`. More build-in classes can be added by subclassing class `ClassReference`.

6.5.1.1 Package Memory

The Decaf compiler defines a package `Memory`, which encapsulates the compiling result of the compiler. The package `Memory` also represents the run-time information of the virtual machine.

The package `Memory` includes the classes defined in the reflective object model and the class `Mem`, the top level class of the package. The class `Mem` manages an instance of class `ClassTable`, which maintains a set of class references. Each of these class references represents the definition of a single class. Figure 47 shows the top level structure of the package `Memory`.

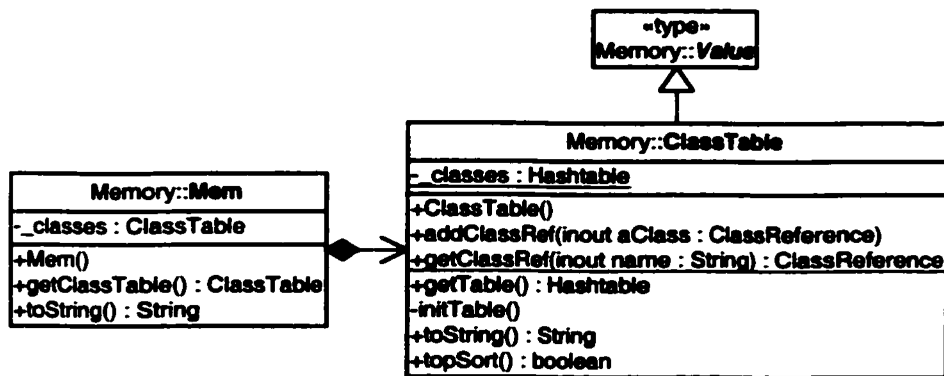

Figure 47: Structure of the Memory Package

6.5.1.2 ClassValue and DecafClassValue

The *value* of a class reference represents the properties of a class. These properties can be encapsulated in a separate class hierarchy that is composed by class `ClassValue` and its subclass `DecafClassValue`. Figure 48 shows the class diagram for this class hierarchy.

According to the Decaf language grammar, there are five kinds of properties in a class definition:

- class name.
- type of the class (i.e. the class of the current class).
- a reference to its superclass.
- class members defined in current class.
- class members inherited from its ancestors (do not include those overridden class members).

The class name and the type of a class are defined in class `ClassReference`. Class `ClassValue` maintains the reference to its superclass. The member properties in the class definition are encapsulated by class `DecafClassValue`.

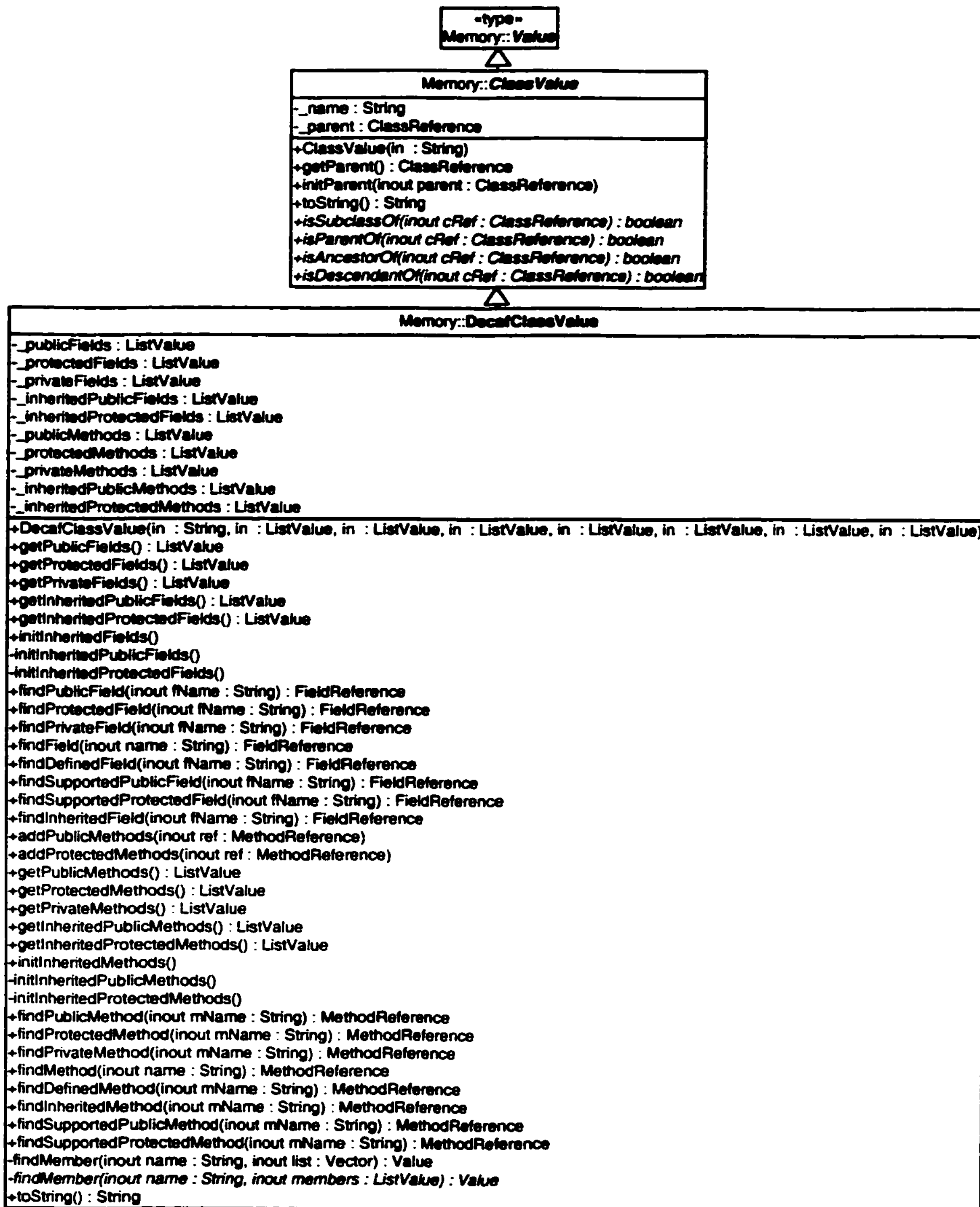

Figure 48: Class Diagram for DecafClassValue

The member information defined in the current class, also known as "defined" members (in contrast to the "inherited" members which refer to those members inherited from ancestors) , includes:

- defined public method list.
- defined protected method list.
- defined private method list.
- defined public field list.
- defined protected field list.
- defined private field list.

Encapsulating the inherited member information in the class `DecafClassValue` can improve the efficiency feature in the interpreter/execution phase. These inherited members include:

- inherited public method list.
- inherited protected method list.
- inherited public field list.
- inherited protected field list.

As a summary, class properties included in class `DecafClassValue` can be constructed in a dictionary format as follows. This dictionary constructs the attributes of class `DecafClassValue` as shown in Figure 48.

```
{ "publicMethods"     : ListValue = { publicMethodRef, ... }
  "protectedMethods"  : ListValue = { protectedMethodRef, ... }
  "privateMethods"    : ListValue = { privateMethodRef, ... }
  "publicFields"      : ListValue = { publicFieldRef, ... }
  "protectedFields"   : ListValue = { protectedFieldRef, ... }
  "privateFields"     : ListValue = { privateFieldRef, ... }
  "inheritedPublicMethods"
                      : ListValue = { inheritedPublicMethodRef, ... }
```

```
  "inheritedProtectedMethods"
                        : ListValue = { inheritedProtectedMethodRef, ... }
  "inheritedPublicMethods"
                        : ListValue = { inheritedPublicFieldRef, ... }
  "inheritedProtectedMethods"
                        : ListValue = { inheritedProtectedFieldRef, ... }
}
```

Basically, there are three use cases in searching for the members (fields or methods) of a class.

- Find Defined Members
  Methods *findDefinedField* and *findDefinedMethod* find a field and a method in the defined member set with the supplied name, respectively. The defined members includes public, protected and private members introduced in the current class.

$$\begin{aligned} DefinedMembers &= PublicMembers \\ &\cup\ ProtectedMembers \\ &\cup\ PrivateMembers \end{aligned}$$

- Find Supported Members
  Methods *findSupportedField* and *findSupportedMethod* find a field and a method in the supported member set with the supplied name. The supported members are visible to other classes or its descendent classes. They include public and protected members in the current class and the ancestors of the current class.

$$\begin{aligned} SupportedPublicMembers &= DefinedPublicMembers \\ &\cup\ InheritedProtectedMembers \end{aligned}$$

$$\begin{aligned} SupportedProtectedMembers &= DefinedProtectedMembers \\ &\cup\ InheritedProtectedMembers \end{aligned}$$

- Find Members
  Methods *findField* and *findMethod* find a field and a method in all defined members and inherited members with the supplied name.

6.5.1.3 MethodValue

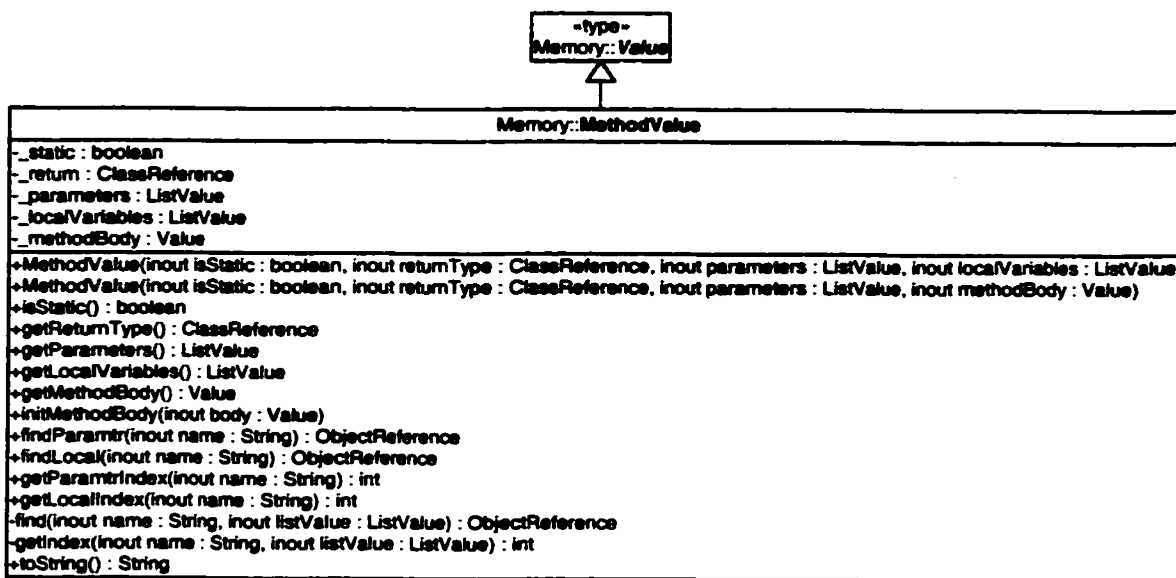

Figure 49: Class Diagram for MethodValue

Method reference is a representative of a method defined in a class. The *value* of a method reference represents the properties of a method. These properties can be encapsulated in a separate class named `MethodValue`. Figure 49 shows the class diagram for the class `MethodValue`.

According the Decaf grammar, the *value* of a method reference can be constructed as follows:

```
{ "static"            : boolean
  "returnType"        : ClassReference
  "parameterList      : ListValue = { ObjectReference, ... }
  "localVariableList" : ListValue = { ObjectReference, ... }
  "methodBody"        : Value
}
```

In Figure 49, class `MethodValue` defines the above structure as its attributes. The attribute *methodBody* can be either a code pointer `CodePtr` or a byte code format

ByteCode. For a build-in method, the *methodBody* is a code pointer CodePtr. While for a user-defined method, the *methodBody* refers to an instance of ByteCode that includes the byte code list and the literal information of the method body.

6.5.1.4 Code Point Classes

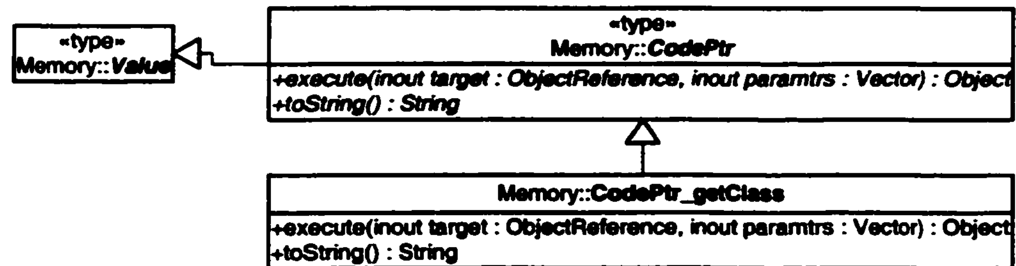

Figure 50: Class Diagram for Code Pointer

The code pointer interface CodePtr defines an interface for all build-in methods. The implementation of a concrete build-in method is defined as a subclass of the interface CodePtr. The method *execute* is defined across the code pointer hierarchy to perform the execution on the concrete build-in method.

Figure 50 shows that the build-in method *getClass* is defined as one of the concrete subclass of CodePtr. This method is defined in the class Object to return the type of the class reference.

6.5.1.5 Build-in Classes

There are a set of build-in classes defined in the Decaf language. They include:

- Object — The root class of the Decaf class hierarchy. It is represented by the class ObjectClass in the Decaf object model.

- Field — The class for all fields defined in a class. It is represented by the class FieldClass in the Decaf object model.

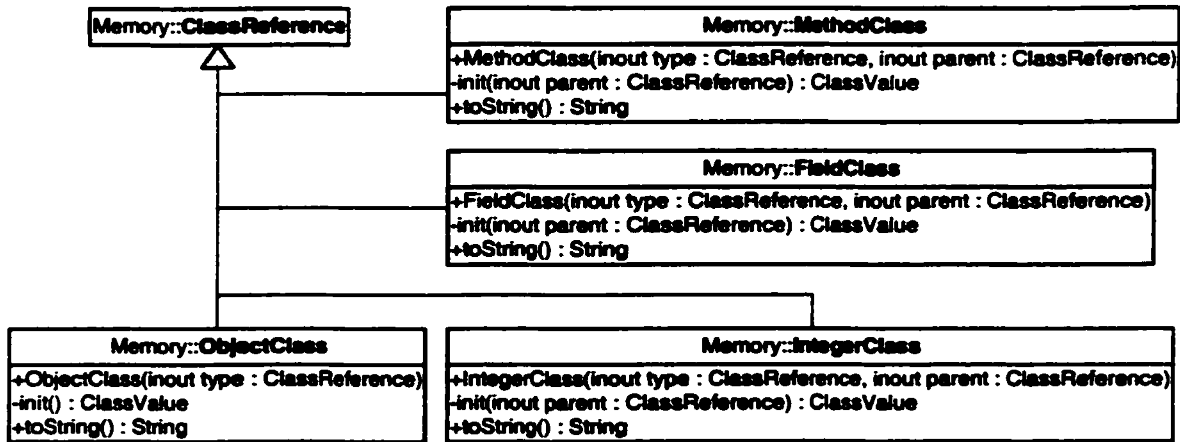

Figure 51: Build-in Classes

- `Method` — The class for all methods defined in a class. It is represented by the class `MethodClass` in the Decaf object model.
- `Integer` — A system-defined build-in class. It is represented by the class `IntegerClass` in the Decaf object model.

Figure 51 shows the class diagram for these build-in classes. More system-defined classes can be added by subclassing the class `ClassReference`.

6.5.2 Implement the Visitor Hierarchy

Section 3.2 introduces the REFLECTIVE VISITOR pattern, one of the variations of VISITOR pattern. The REFLECTIVE VISITOR pattern can be used as a framework to perform operations over an extensible and reusable object structure. Unlike most of the VISITORs that perform dispatch actions through DOUBLE DISPATCH, the REFLECTIVE VISITOR pattern defines a unique dispatch method *visit* in the class `Visitor`. This method handles the dispatch via reflection. Hence, the client can call this method to visit an object structure and this object structure has no knowledge about the visitor. Furthermore, any operation defined in the visitor hierarchy can call this dispatch method to perform a re-dispatch action.

By applying REFLECTIVE VISITOR pattern, the code generation can generate compiling result on the target virtual machine by visiting the parsing result, an abstract syntax tree that is a composite object. This design is compatible to any potential changes on either the virtual machine specification or the language grammar.

6.5.2.1 Structure Overview

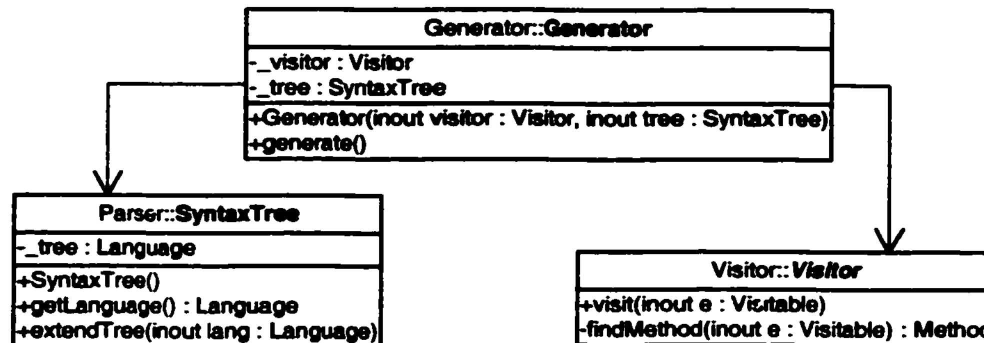

Figure 52: Structure of Code Generation

Figure 52 shows the design of the visitor hierarchy for the code generation. For extension purpose, this visitor hierarchy are designed in five levels:

- Level One includes the class `Visitor`, which defines the method *visit* to perform the dynamic dispatch. This class is the root of the visitor hierarchy. Section 3.2 presents the definition of this class.
- Level Two includes the class `CodeGeneration`, which encapsulates the basic properties for code generation. All concrete code generation processes are derived from this class and they use the sources provides by this class.
- Level Three includes the class `ExpressionGenerator`, which implements the code generation for expressions defined in the Decaf language.
- Level Four includes the class `StatementGenerator`, which implements the code generation for statements defined in the Decaf language.
- Level Five includes the class `ClassGenerator`, which implements the code generation for class declarations defined in the Decaf language.

6.5.2.2 CodeGeneration Class

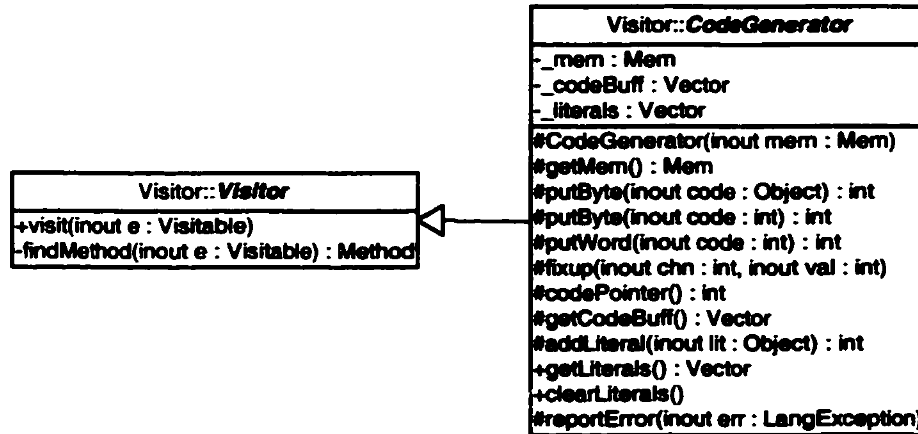

Figure 53: Class Diagram for Class CodeGeneration

Figure 53 shows the class diagram for `CodeGenerator`. The class `CodeGeneration` is derived from class `Visitor`. It encapsulates the basic properties to support the execution of the code generation. All concrete code generation classes that handle the code generation processes are derived from this class so that they can use and share the sources provided by this class. Generally, there are three properties defined in the class `CodeGeneration`:

- *Mem*, which references the location that stores the compiling result.
- *codeBuff*, which temporarily stores the byte code sequence for each method. The byte code is generated during the code generation.
- *literals*, which temporarily stores the literal information for each method.

6.5.2.3 Code Generation for Expressions and Statements

Figure 54 shows the class diagram for `ExpressionGenerator` and `StatementGenerator`. The set of methods *evaluate* defined in these two classes recursively generates byte code for expressions and statements.

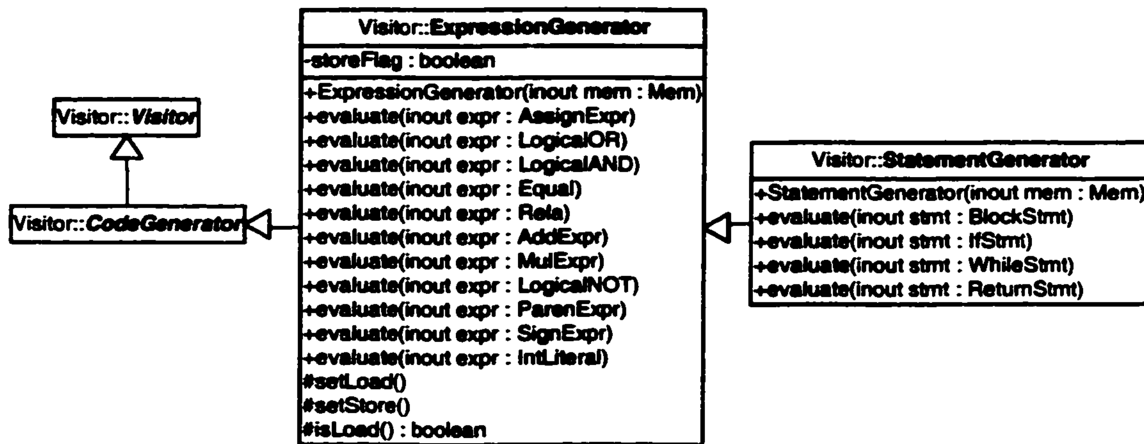

Figure 54: Class Diagram for Classes ExpressionGenerator and StatementGenerator

Expression Generator Class `ExpressionGenerator` is directly derived from class `CodeGeneration`. It implements the code generation for variant expressions. These expressions are defined in the `Expr` hierarchy in the language structure (Section 6.4.1). The *evaluate* methods in this class receive expression objects as parameters and recursively visit these expressions to generate byte code sequence according the instruction set defined in the virtual machine specification. The generated byte code is stored in the attribute *codeBuff* defined in the superclass `CodeGeneration`.

Statement Generator The statement is the basic component of a class method in the Decaf program. Class `StatementGenerator` inherits from `ExpressionGenerator` so that changes in the statement-level specification will not affect the code generation part for expressions. As a part of the code generation for a method, the *evaluate* methods in this class generate byte code sequence and store them in the attribute *codeBuff* defined in the superclass `CodeGeneration`.

6.5.2.4 Code Generation for Class Declaration

As the leaf of the visitor hierarchy, the class `ClassGenerator` implements the code generation for class definitions, which compose of the target Decaf program. It derives from the class `StatementGenerator`. Changes in the specification of the class definition level will not affect the code generation for statements. The *evaluate* methods

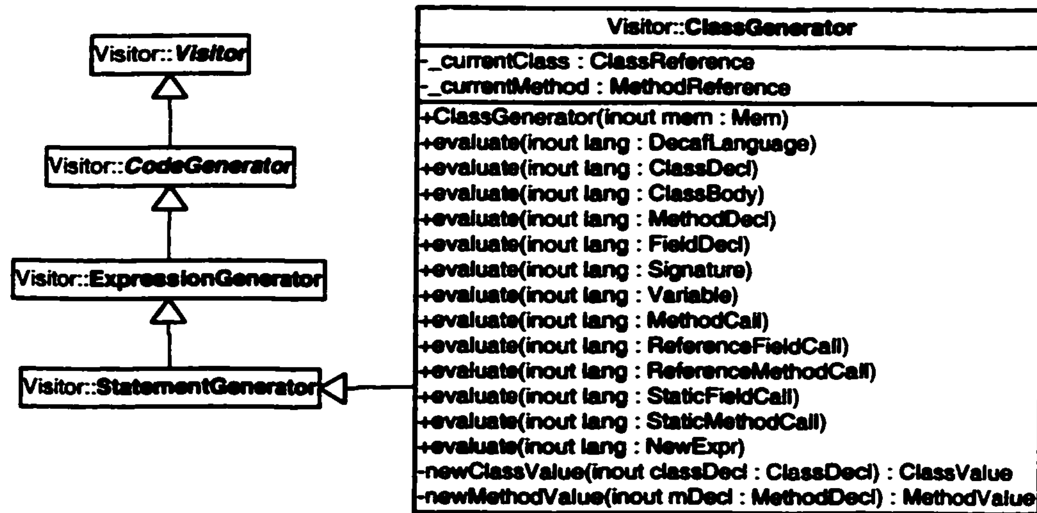

Figure 55: Class Diagram for Class ClassGenerator

that are defined in this class and inherited from its ancestors are executed recursively to complete the code generation task for the whole Decaf program. Those literals that will be used during the execution of current method such as user-defined variables, method parameters, attributes and some constant literals are temporarily stored in the attribute *literals* that inherited from class `CodeGeneration`. After the completion of code generation for the current method, this *literals* is stored in the `ByteCode` object as the *value* of *currentMethod*. The byte code sequence generated for the current method is temporarily stored in the attribute *codeBuff* inherited from the superclass `CodeGeneration`. The content of the *codeBuff* will be stored in the `ByteCode` object as the value of *currentMethod* after the completion of code generation for the current method. The compiling result for the current class (*currentClass*) is pushed into the hash table (*classes*) defined in the class `ClassTable`, which is maintained by class `Mem` in the package `Memory`.

6.6 Summary

This chapter presents the design for developing an extensible and reusable compiler system (front-end) for the Decaf programming language, which is an extensible tiny object-oriented programming language. This compiler system customizes the parser

framework by defining rules for the Decaf language, constructs virtual machine platform by extending the reflective class-based object model, and applies REFLECTIVE VISITOR for implementing the code generation. The Decaf language itself can also be defined as a language framework because a more complex language can be developed based on Decaf without changing the existing system.

The next chapter is the conclusion of the thesis. We will also discuss the future research work.

Chapter 7

Conclusion and Future Work

7.1 Summary and Conclusions

Design for change is a major concern in software design since new requirement and new technology emerge over time. Designing a system that meets a wide range of different requirements is a difficult task. A better solution is to specify an architecture that is open to modification and extension. The resulting system should be able to adapt to changing requirements on the fly.

Reflection provides a means for a software system to dynamically change its structure and behavior. It supports the modification of some fundamental aspects like type structures, function call mechanisms, etc. Since its first introduction by Smith [67] as a framework for language extension, reflection technology has been an interesting research topic and has been successfully applied in many domains. Specifically, in the recent years, reflection has been integrated into programming languages and has become an important feature in most object-oriented programming languages such as Java [45, 32], Smalltalk [38], Oberon [77, 70], Scheme [28], etc.

The thesis work performs a series of experiments on the application of the reflection technique to better improve the software design in some circumstances.

First, a REFLECTIVE VISITOR pattern [53] was captured to improve the structure of the VISITOR pattern design so that both the element hierarchy and visitor hierarchy are extensible. In the REFLECTIVE VISITOR pattern, a visitor can perform the run-time dispatch action on itself by using the reflection technique (e.g. Java Reflection). The cyclic dependencies between the visitor hierarchy and the element hierarchy are

therefore removed. The REFLECTIVE VISITOR is thus more flexible and reusable.

Second, due to the high demand of the VISITOR pattern in the software design, a pattern language for Visitors [52] is presented to classify and organize the VISITOR variants. This pattern language will assist the application developer to choose the right VISITOR pattern that best suites the intended purpose by enumerating all important forces and consequences for each variant.

Third, a parser framework [51] was developed. A pattern language is presented for developing a framework for parsing in object-oriented compiler design based on the principle of the predictive recursive-descent parsing approach. It describes four patterns that address three design aspects in developing an object-oriented parser. Two alternative patterns are presented to provide alternative solutions to solve the recursion problem in the object-oriented software design. A two-level structure is defined for implementing the parsing process control based on the REFLECTION pattern [13, 34]. The base-level contains a set of classes, where each represents a grammar rule. The meta-level handles the complex relationships of the rules that are maintained in a hash table. Reflection is used to discover rules at run-time and determine the parsing order. The base-level delegates the dynamic dispatch to a meta-level object.

Fourth, a dynamic object model is defined for a virtual machine that can support reflection. We demonstrate Forman's theory [33] by developing a simplified object model based on a single inheritance system with the support of only one metaclass.

Finally, we designed and implemented an extensible and reusable compiler system (front-end) for the Decaf programming language, which is an extensible tiny object-oriented programming language. This compiler system customizes the parser framework by defining concrete grammar rules for the Decaf language, constructs virtual machine platform by extending the reflective class-based object model, and applies REFLECTIVE VISITOR for implementing the code generation.

The Decaf language itself can be further refined as a framework for experimental object-oriented languages so that a more complex language can be developed based on it without changing the existing system.

7.2 Future Work

1. The target of our reflective object-oriented programming language is for embedded system programming. Further study should be made to investigate the issues arising in the embedded systems, and in the future, the Decaf programming language will be tailored for embedded system programming.

2. The dynamic object model and the front-end compiler system (Decaf compiler) presented in this thesis can be used to develop an extensible virtual machine framework for embedded systems. A reflective interpreter/assembler system [27] will be developed and integrated into this virtual machine to support program execution on the target machines.

3. Garbage collection is the automatic recovery of resources [79, 44]. As an important feature in memory management for modern object-oriented languages, the garbage collection should be supported in our virtual machine.

4. Error recovery has not been covered in this thesis. The error recovery can be defined as a separate package and be integrated into the compiler framework. The design of the error recovery in the compiler system should support handling of lexical errors, syntactic errors, and semantic errors. It will also handle three basic error categories: warning, skippable errors, and fatal errors.

5. Applying reflection leads to low efficiency. Jens Palsberg and C. Barry Jay [60] made some discussions on the efficiency issues in using reflection. We need to do the performance study in our target systems and compare them to other related systems.

Bibliography

[1] C. Alexander. *Notes on the Synthesis of Form.* Harvard University Press, 1964.

[2] C. Alexander. *The Oregon Experiment.* Oxford University Press, 1975.

[3] C. Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[4] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language.* Oxford University Press, 1977.

[5] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns: Smalltalk Companion.* Addison-Wesley, 1998.

[6] Noriki Amano and Takuo Watanabe. Lead++: An Object-Oriented Reflective Language for Dynamically Adaptable Software Model. *IEICE TRANS. Fundamentals, Vol. E82-A, No.6*, June 1999.

[7] Andrew W. Appel. *Modern compiler implementation in Java.* Cambridge University Press, 1998.

[8] Brad Appleton. Patterns and Software: Essential Concepts and Terminology. *http://www.enteract.com/~bradapp/*, November 1997.

[9] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1997.

[10] D. Betz. Your Own Tiny Object-Oriented Language. *Dr. Dobb's Journal*, pages 26–33, September 1991.

[11] Jeremy Blosser. Reflect on the Visitor Design Pattern. *http://www.javaworld.com/javatips/jw-javatip98.html*, January 2001.

[12] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[13] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[14] Gregory Butler and Pierre Dénommée. *Documenting Frameworks to Assist Application Developers, Chapter of Building Application Frameworks: Object-Oriented Foundations of Framework Design, edited by Mohamed E. Fayad, Douglas C. Schmidt, Ralph E. Johnson.* John Wiley & Sons. Inc., 1999.

[15] Walter Cazzola, Robert J. Stroud, and Francisco Tisato. *Reflection and Software Engineering (Lecture Notes in Computer Science, 1826).* Springer, Heidelberg, Germany, 2000.

[16] James W. Cooper. *The Design Patterns: Java Companion.* Addison-Wesley, 1998.

[17] James O. Coplien. *Software Patterns.* SIGS Books & Multimedia, 2000.

[18] James O. Coplien and Douglas C. Schmidt. *Pattern languages of program design.* Addison-Wesley, 1995.

[19] Hans de Bruin. Software Architecture: State of the Art . *http://www.cs.vu.nl/~hansdb/state/node34.html*, December 1998.

[20] Michel de Champlain. Report on C- Programming Lanugage. *Technical Report, Computer Science Department, Concordia University, Montréal, QC, Canada,* April 1987.

[21] Michel de Champlain. Synapse: A Real-Time Programming Language. Master's thesis, Computer Science Department, Concordia University, Montréal, QC, Canada, September 1989.

[22] Michel de Champlain. Synapse: A Small and Expressive Object-Based Real-Time Programming Language. *SIGPLAN Notices, ACM Press, vol. 25(5)*, pages 124–134, May 1990.

[23] Michel de Champlain. Synapse: An Object-Based Real-Time Programming Language. *Structured Programming, Springer-Verlag, vol. 12(3)*, pages 145–155, December 1991.

[24] Michel de Champlain. Manuel de Référence du Langage Réact. *Technical Report, Collège Militaire Royal de Saint-Jean, Computer Science and Engineering Department, Richelain, QC, Canada*, 1994.

[25] Michel de Champlain. *Modèle Réactif et Réflexif pour Méta Machines à états Finis.* PhD thesis, Départment de génie électrique et de génie informatique, Ecole Polytechnique de Montréal, Université de Montréal, Montréal, QC, Canada, 1995.

[26] Michel de Champlain. The Decaf Language Specification. *Technical Report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada*, May 2001.

[27] Michel de Champlain and Cheng-Yu Pai. A Reflective Architecture for Cross-Assemblers. *Engineering Solutions for the Next Millennium. 1999 IEEE Canadian Conference on Electrical and Computer Engineering*, May 1999.

[28] R. Kent Dybvig and Kent Dybbig. *Scheme Programming Language, The: ANSI Scheme.* Prentice-Hall ECS Professional, 1996.

[29] Etienne Gagnon. Sablecc, An Object-Oriented Compiler Framework. Master's thesis, McGill University, 1998.

[30] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* John Wiley & Sons. Inc., 1999.

[31] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Implementing Application Frameworks: Object-Oriented Framework at Work.* John Wiley & Sons. Inc., 1999.

[32] David Flanagan. *Java In a Nutshell: A Desktop Quick Reference.* O'Reilly, 1999.

[33] Ira R. Forman and Scott H. Danforth. *Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming.* Addison-Wesley, 1998.

[34] Martin Fowler. *Analysis Pattern: Reusable Object Models.* Addison-Wesley, 1997.

[35] Garry Froehlich, H. James Hoover, Ling Liu, and Paul Sorenson. Designing Object-Oriented Frameworks. *Handbook of Object Technology*, pages 25–1 – 25–22, December 1998.

[36] Richard P. Gabriel. *Patterns of Software: Tales From the Software Community.* Oxford University Press, 1998.

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[38] A. Goldberg and D. Robson. *Smalltalk-80, the Language and Its Implementation.* Addison-Wesley, 1983.

[39] CERN Programming Techniques Group. A conceptual framework for system fault tolerance. *http://hissa.nist.gov/chissa/SEI_Framework/framework_11.html*, March 1995.

[40] Neil Harrison, Brian Foote, and Hans Rohnert. *Pattern Languages of Program Design, 4.* Addison-Wesley, 2000.

[41] Juha Hautamaki. A Survey of Frameworks. *Report A-1997-3, Department of Computer Science, University of Tampere*, March 1997.

[42] Ralph E. Johnson. Components, Frameworks, Patterns. *Communications of the ACM, Volume 40 , Issue 10*, October 1997.

[43] R.E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming, 1*, pages 22–35, 1988.

[44] Richard Jones and Rafael Lines. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.

[45] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification, Second Edition (The Java Series).* Addison-Wesley, 2000.

[46] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[47] Doug Lea. Christopher Alexander: An Introduction for Object-Oriented Designers. *ACM Software Engineering Notes*, January 1994.

[48] Pattie Maes. Concepts and Experiments in Computational Reflection. *Proceeding of OOPSLA '87, volumn 22, SIGPLAN Notices*, pages 147 – 155, October 1987.

[49] Yun Mai and Michel de Champlain. An Extensible One-Pass Assembler Framework. *Technical Report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada*, June 2000.

[50] Yun Mai and Michel de Champlain. Design A Compiler Framework in Java. *Technical Report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada*, November 2000.

[51] Yun Mai and Michel de Champlain. A Pattern Language for Parsing. *In preparation to be submitted to the 3th Conference TOOLS Eastern Europe '2001, Varna, Bulgaria*, September 2001.

[52] Yun Mai and Michel de Champlain. A Pattern Language to Visitors. *Submitted and accepted for the 8th Conference PLoP '2001, Monticello, Illinois, USA*, September 2001.

[53] Yun Mai and Michel de Champlain. Reflective Visitor Pattern. *Submitted and accepted for EuroPLoP '2001 Writer's workshop, Irsee, Germany*, July 2001.

[54] Robert C. Martin. Acyclic Visitor. *PLoP '96*, September 1996.

[55] Robert C. Martin, Dirk Riehle, Frank Buschmann, and John Vlissides. *Pattern Languages of Program Design, 3.* Addison-Wesley, 1998.

[56] Gerard Meszaros and Jim Doble. A Pattern Language for Pattern Writing. *http://www.hillside.net/patterns/Writing/patterns.html*, 1996.

[57] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, 1997.

[58] Martin E. Nordberg III. The Variations on the Visitor Pattern. *PLoP '96 Writer's Workshop*, September 1996.

[59] Object Management Group, Inc. *Meta Object Facility (MOF) Specification.* http://www.omg.org/technology/documents/, 1999.

[60] Jens Palsberg and C. Barry Jay. The Essence of the Visitor Pattern. *Technical Report 05, University of Technology, Sydney*, 1997.

[61] Thomas W. Parsons. *Introduction To Compiler Construction.* Freeman, 1992.

[62] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1994.

[63] Dirk Riehle and Heinz Zullighoven. Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems 2, 1, Page 3-13*, 1996.

[64] Don Roberts and Ralph Johnson. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. *plOp '96*, 1996.

[65] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[66] Jeanne Sebring. Reflecting on the Visitor Design Pattern. *Java Report*, March 2001.

[67] B.C. Smith. Reflection and Semantics in a Procedural Language. *MIT-LCS-TR-272, Mass. Inst. of Tech., Cambridge, MA*, January 1982.

[68] B.C. Smith. Reflection and Semantics in Lisp. *Conf. Rec. 11th ACM Symp. on Principles of Programming Language*, pages 23 – 35, 1984.

[69] Jonathan M. Sobel and Daniel P. Friedman. An Introduction to Reflection-Oriented Programming. *Reflection '96, San Francisco*, April 1996.

[70] Christoph Steindl. Reflection in Oberon. *6th ECOOP Workshop for PhD Students in Object-Oriented Programming, Linz, Austria*, July 1996.

[71] Sun Microsystems, Inc. *Java Core Reflection: API and Specification.* JavaSoft, 1997.

[72] Taligent Inc. Building Object-Oriented Frameworks. *A Taligent White Paper*, 1994.

[73] Joost Visser. Visitor Combination and Traversal Control. *http://www.jforester.org*, 2001.

[74] John Vlissides. *Pattern Hatching: Design Patterns Applied.* Addison-Wesley, 1998.

[75] John Vlissides. Visitor in Frameworks. *C++ Report*, November 1999.

[76] John Vlissides, James O. Coplien, and Norman L. Kerth. *Pattern Languages of Program Design, 2.* Addison-Wesley, 1996.

[77] Niklaus Wirth and Martin Reiser. *Programming in Oberon.* Addison-Wesley, 1992.

[78] A. Wollrath. Java's New Reflection. *UNIX Review's Performance Computing Vol: 16 Issue: 11*, pages 25–9, October 1998.

[79] Stuart A. Yeates and Michel de Champlain. Design Patterns in Garbage Collection. *Proceeding of the 4th Conference PLoP '1997, Monticello, Illinois, USA*, September 1997.

[80] Akinori Yonezawa and Takuoo Watanabe. An Introduction to Object-based Reflective Concurrent Computations. *Proceeding of 1988 ACM SIGPLAN Workshop on Object-based Current Programming, Volume 24, SIGPLAN Notices*, pages 50 – 54, April 1989.

[81] Saba Zamir. *Handbook of Object Technology.* CRC Press, 1998.

[82] Chris Zimmerman. *Advances in Object-Oriented Metalevel Architecture.* CRC Press, 1996.