Tiling Design Patterns —A Case Study Using the Interpreter Pattern

DAVID H. LORENZ

The Faculty of Computer Science,
Technion—Israel Institute of Technology,
Technion City, Haifa 32000, ISRAEL;
Email: david@CS.Technion.AC.IL

Abstract

This paper explains how patterns can be used to describe the implementation of other patterns. It is demonstrated how certain design patterns can describe their own design. This is a fundamental reflexive relationship in pattern relationships. The process of assembling patterns by other patterns is named pattern tiling. Tiling enables us to interweave simple understood concepts of patterns into their complex real-life implementation. Several pattern tilings for the Interpreter design pattern are illustrated.

1 Introduction

Tilings are sets of physical objects ("tiles"), typically made of stone or ceramic, which fit together in a recurring motif ("pattern") to cover surfaces without gaps or overlaps. The art of tiling and the relevance of its technology extends from early days uses to applications in modern engineering (e.g., VLSI design [36]) and in modern science (a study of their mathematical properties can be found in [18].) This paper presents tilings of non-physical objects: design patterns [14, 15] as declarative *entities* that fit together.

One of the nice features that Prolog [33] beginners struggle with at first is its declarative semantics. Declaring, for example, that conc(L_1, L_2, L_3) if L_3 is the concatenation of L_1 and L_2, allows the functor conc/3 to be used in more than one way. The obvious use is to concatenate two lists, L_1 and L_2, into L_3. But it may also be used to split a given list L_3. Likewise, the *intent* of a pattern and the manipulation of its *participants* can be taken declaratively.

The pattern *name* is perhaps its most profound declarative form. The names associated with design patterns enrich our vocabulary for reasoning about design at higher levels of abstraction. Their rapid growth threatens to become a language for design. One indication of a language's expressive power is its ability to describe itself. BNF (Backus-Naur-Form [29])–a formal notation to describing the context-free syntax of a given language, for example, can describe its own grammar. This begs a question: can design patterns describe their own design?

Design patterns improve considerably the manner in which object-oriented experience is communicated. But, no matter how well a design pattern is written, there will always remain a gap between comprehending a pattern and applying it. This gap is revealed in the complexity and cost of putting certain design patterns into practice. Implementing the *Interpreter* [15] pattern, for instance, is limited to *simple* grammars, since "for complex grammars, the class hierarchy for the grammar becomes large and unmanageable" [15, page 245]; changing the structure traversed by a *Visitor* [15] "requires redefining the interface to all visitors, which is potentially costly" [15, page 333].

This paper shows how patterns can be used to describe the implementation of other patterns. It is demonstrated how certain design patterns can describe their own design. This is a fundamental reflexive relationship in pattern relationships. The process of assembling patterns by other patterns is named *pattern tiling*. Tiling patterns reduces the comprehension-application gap. For example, we shall see how one Interpreter helps with implementing another Interpreter pattern, and that a Visitor to the first interpreter can generate a family of visitors to the second.

OOPSLA '97 10/97 GA, USA

Yet another gap is stretched between applying each pattern individually and integrating several patterns. Knowing the catalogue of patterns by heart is prudent perhaps but unfortunately not a sure recipe for successfully applying them. For example, one may fully understand both the Interpreter [15] and the *Reflection* [35, Sect. 17] patterns and yet not see how an implementation of an interpreter implicitly possesses a reflective architecture, as shall be shortly demonstrated.

The issue of pattern relationship is one of the difficult aspects of patterns. In [15] this is addressed by classifying patterns into groups according to their *purpose* and *scope*, specifying related patterns, and providing a *spaghetti-diagram* that shows graphically the relationships between patterns. For example, the Interpreter is a *behavioral* pattern. It applies primarily to *classes*, and can use the *Composite* [15] pattern in defining the grammar, the Visitor for adding operations, and the *Flyweight* [15] pattern for sharing terminal symbols. As informative as it is, this is far from satisfactory, albeit hard to improve, as indicated by attempts for improved classifications ([8, Sect. 18] is a good example) that followed. Noticeably, the spaghetti-diagram for showing related patterns contains no loops. The possible *reflexive* relationships between patterns seem to be missing.

An implementation of a pattern can simultaneously be (or share large portions with) an implementation of several other patterns. A special case of this is integrating several implementations of the *same* pattern. Integrating a pattern with itself is conceptually simpler than integrating two different patterns but just as powerful, (though somewhat more confusing). To this extent, *tiling* design patterns is one of many *reflexive* relationships between patterns.

Tiling is neither genericity nor inheritance. Inheritance is a form of abstraction that promotes reuse. Tiling is a form of reuse that promotes abstraction. Delegating requests along adjacent tiles brings inheritance and tiling to the same conceptual level. Tiling is different from templates in the same way grammars are different from regular expressions. A template can instantiate simple tiles. The tiles we shall be concerned with, however, are design patterns for which a template that generates them is something to be desired. Tilings compare to other techniques for pattern application, like automatic code generation [6]. Patterns can be classified by whether or not they can be tiled by other patterns.

The symmetry art works of Maurits Cornelis Escher [12], a Dutch graphic artist, (also the cover art on the design patterns book [15]), should serve as the motivation for this work. The main observation is that the Interpreter pattern and the Visitor pattern have a common interesting characteristic–they are building blocks capable of constructing themselves. Repeating Interpreter tilings result in a reflective Interpreter. A tessellation of Visitors molds a Visitor mosaic. In design patterns, as in Escher's work, what you see the first time is most certainly not all there is to see.

The outline of the paper is the following. The next section describes a systematic approach to implementing the Interpreter pattern. Section 3 demonstrates several tilings of the Interpreter pattern (reflective, mirror, in-out, and execution.) Section 4 presents tessellations of primitive visitors (null, exception, inheritance, and composition) that form more complex ones (e.g., known variants of the Visitor pattern.) Section 5 describes related work and offers conclusions.

2 Interpreter Pattern Making

"If a particular kind of problem occurs often enough, then it might be worthwhile to express instances of the problem as sentences in a simple language. Then you can build an interpreter that solves the problem by interpreting these sentences" [15, page 243].

At the heart of the Interpreter pattern is the class-based observation, shown in Figure 1, in which programs relate to grammars as objects relate to classes. An analogy is drawn between writing a program in a context-free language and instantiating an object from a class. A program is perceived as an instantiation of its grammar, just as an object is an instantiation of its class.

More precisely, let $InstanceTypeOf(c)$ (following the conventions in [1]) indicate the *type* of objects of class c and, analogically, $ProgramTypeOf(g)$ the

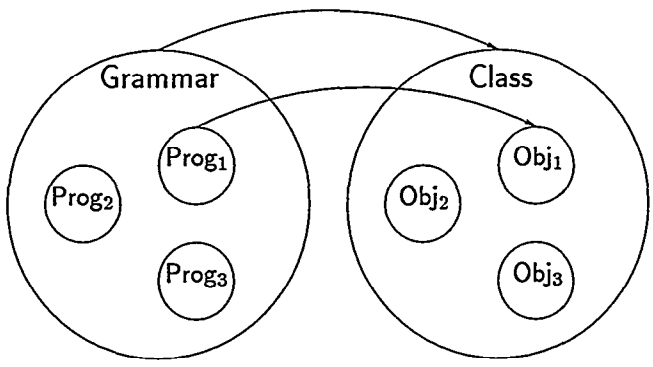

Figure 1: Class based approach to the Interpreter pattern

programs in the language generated by the context-free grammar g. We indicate by $\mathcal{L}$ the language in which the program is written, and by $\mathcal{G}_\mathcal{L}$ a context-free grammar for $\mathcal{L}$. The Interpreter pattern takes the approach of viewing a program in $\mathcal{L}$ as an instance of $\mathcal{G}_\mathcal{L}$, just as an object is an instance of a class, as follows. First, $\mathcal{G}_\mathcal{L}$ is transformed into a class definition $\mathcal{C}_\mathcal{L}$. Then, both $\mathcal{G}_\mathcal{L}$ and $\mathcal{C}_\mathcal{L}$ are used (with the help of a parser generator) to produce a parser capable of compiling any program $\mathcal{P}$: *ProgramTypeOf*($\mathcal{G}_\mathcal{L}$) into an object $\mathcal{O}_\mathcal{P}$: *InstanceTypeOf*($\mathcal{C}_\mathcal{L}$).

A Systematic Approach to Implementing an Interpreter

Figure 1 is, of course, an over simplification. The Interpreter pattern actually takes an *object-oriented* approach, in which the class definition $\mathcal{C}_\mathcal{L}$ is a hierarchy of classes, just like the grammar $\mathcal{G}_\mathcal{L}$ is a group of related rules. It's left to the pattern implementer to make the leap from the simple understood metaphor to its detailed implementation. Orderly done, though, the hierarchy $\mathcal{C}_\mathcal{L}$ can be systematically obtained in a relatively straightforward manner.

If $\mathcal{L}$ is a nonempty context-free language then it can be generated by a context-free grammar $\mathcal{G}_\mathcal{L}$ with the following properties. Every production of $\mathcal{G}_\mathcal{L}$ is of one of the forms: (1) A *choice* of having a number of given variants, $B \rightarrow A_1 | \cdots | A_n$, where A_i and B are variables; or (2) an *aggregate* made of a fixed number of parts, $A \rightarrow \alpha$, where α is a nonempty string of variables and terminals. Furthermore, any variable A appears at most once as the left-hand-side construct of a rule, and at most once at the right-hand-side of a choice rule. This restricts the format of productions without reducing the generative power of the grammar [38].

Once $\mathcal{G}_\mathcal{L}$ conforms to the above canonical form, it prescribes the hierarchy $\mathcal{C}_\mathcal{L}$: Each choice construct is made an abstract class, each aggregate–a concrete class. Choice and aggregation rules determine the inheritance and aggregation relationships among classes, respectively. With little additional effort this can be generalized to also handling higher level constructs like optional, datatypes, and repetition. For example, the two rules:

s ::= `IfStatement` | $\cdots$;

and

`IfStatement` ::= **if** b **then** s_1 [**else** s_2];

would determine that `IfStatement` is a concrete subclass of the abstract class s, and that it has three components: a condition, and two statements of which the second may be void. (See [16] for a complete example.)

3 Tiling a Reflective Interpreter

Whether systematic or not, interpreting $\mathcal{G}_\mathcal{L}$ and producing the class hierarchy $\mathcal{C}_\mathcal{L}$, especially for a long grammar like that of a programming language, is a chore. The description in [15] acknowledges this implicit complexity. The book even warns, as quoted above, that the Interpreter's application is therefore limited to simple grammars. In doing so the pattern has missed the point of its own design. Such an interpretation is the bread and butter of the Interpreter pattern itself.

Let $\mathcal{G}_\mathcal{B}$ be the context-free description of BNF itself. Then, the grammar $\mathcal{G}_\mathcal{L}$ is no more than a program in $\mathcal{B}$. By re-applying the Interpreter pattern to transform the BNF grammar $\mathcal{G}_\mathcal{B}$ into a class hierarchy $\mathcal{C}_\mathcal{B}$, we can compile our original grammar $\mathcal{G}_\mathcal{L}$: *ProgramTypeOf*($\mathcal{G}_\mathcal{B}$) into an instance $\mathcal{O}_\mathcal{L}$: *InstanceTypeOf*($\mathcal{C}_\mathcal{B}$) in the precise same way we compiled $\mathcal{P}$. The procedure can be repeated for compiling $\mathcal{G}_\mathcal{B}$ into $\mathcal{O}_\mathcal{B}$: *InstanceTypeOf*($\mathcal{C}_\mathcal{B}$). Since $\mathcal{G}_\mathcal{B}$: *ProgramTypeOf*($\mathcal{G}_\mathcal{B}$) is a "fixed point" of this process, it does not result in an infinite sequence. Rather, it stops at $\mathcal{O}_\mathcal{B}$, as illustrated in Figure 2.

3.1 A Universal Interpreter Generator

To better understand the dynamics of this process, suppose there were a universal machine for generating Interpreter patterns. We could configure this machine with Eiffel's grammar so that it would accept an Eiffel [28] program and generate an object encapsulating the program. This object would then serve as an *agent* for performing miscellaneous analyses on the program (e.g., pretty printing, type checking, testing, and computing software metrics).

Suppose further that the machine model we've got requires, for the process of configuration, an expert on Eiffel's grammar rather than a textual description. We could first configure our universal machine with BNF's grammar and use it to translate Eiffel's grammar into a knowledgeable object, an agent of Eiffel, which in turn we'll use for our original configuration purposes. But before we do that we use the machine once more to translate the textual description of BNF to a BNF agent, and we can do that, putting the issue of bootstrapping aside for a moment, because BNF uses BNF to describe itself.

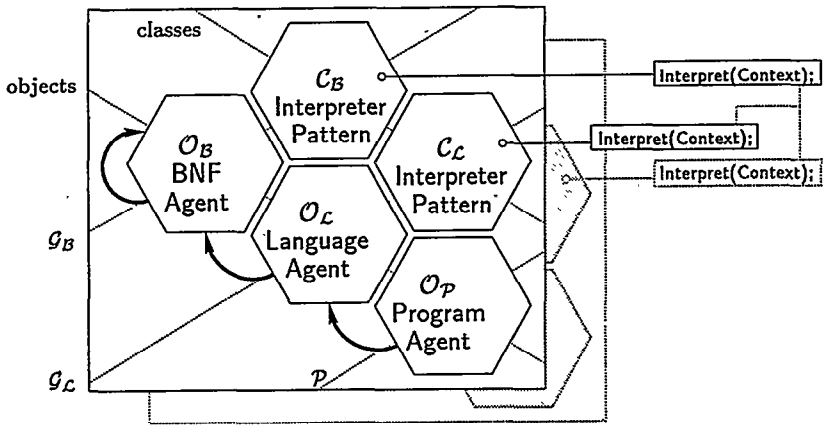

Figure 2: Tiling a Reflective Interpreter

We end up substituting our textual description $\mathcal{P}$ with three objects, tiled as depicted in Figure 2: an object $\mathcal{O}_\mathcal{P}$ which is an instance of the class $\mathcal{C}_\mathcal{L}$, its corresponding metaclass object $\mathcal{O}_\mathcal{L}$ which is an instance of $\mathcal{C}_\mathcal{B}$, and a meta-metaclass object $\mathcal{O}_\mathcal{B}$. In Figure 2, a class object which represents a class x is named an *x agent*, to distinguish it from an *x object* which is simply some object of type *InstanceTypeOf*(x). Our Eiffel agent is hence a BNF-instance, and an Eiffel-program agent would be an Eiffel-instance.

Along the dashed diagonal lines sketched in Figure 2, we can observe an information trinity $\langle text, instance, type \rangle$ for each kind of knowledge: The BNF language described by $\mathcal{G}_\mathcal{B}$ is also represented by both the agent $\mathcal{O}_\mathcal{B}$ and the Interpreter pattern class hierarchy $\mathcal{C}_\mathcal{B}$. Similarly, $\mathcal{G}_\mathcal{L}$, $\mathcal{O}_\mathcal{L}$, and $\mathcal{C}_\mathcal{L}$, each in its own way, represent the programming language. The letters: $\mathcal{G}$, $\mathcal{O}$, and $\mathcal{C}$, were chosen to emphasize the shape of the description. If you need a reminder of the contents, just look at the subscripts.

The objects $\mathcal{O}_\mathcal{P}$, $\mathcal{O}_\mathcal{L}$, and $\mathcal{O}_\mathcal{B}$ form together a reflective architecture. For example, generating $\mathcal{C}_\mathcal{L}$ could be a **once** routine, in Eiffel's terminology, for $\mathcal{O}_\mathcal{B}$. This reflective structure lends itself to various degree of reuse.

3.2 Tiling as Means of Reuse

In order for the program agent $\mathcal{O}_\mathcal{P}$ to compute something, we need to place appropriate interpret routines in $\mathcal{C}_\mathcal{L}$ to do the job. If several interpretations perform a common computation, or if similar ones are parameterized by only a syntactical property, even if these interpretations belong to entirely different interpreters, then this commonality can be placed as a computation in the meta-interpreter, as depicted in Figure 2. A resemblance between tiling and inheritance is apparent in the figure.

Computing Syntax-Based Metrics

A simple example showing how to compute basic software metrics ([7, 4, 23, 19]) will make this clearer. Syntax-based metrics are the most common software metrics. They measure how the source code is written, differing in what they measure as size. Lines-of-Code (LOC), for example, is an elementary syntax-based metric. Nonetheless, computing it properly requires a non-trivial analysis. It is a misnomer since it really refers to executable statements. In C, for instance, counting semicolons (";") would not provide the accurate measure, since it would count *lines* instead of *statements*. A `for` statement contains two semicolons but should count as one, whereas an `if` should count as two but contains only one semicolon.

By placing a counter in the interpret routine associated with the class `Statement` of an Interpreter pattern for the C language, we can accurately compute the LOC measurement. More generally, a meta-routine in the meta-Interpreter can count all constructs of a particular type. We can use such a general routine to count statements as well as other kinds of repeated constructs (the number of class declarations, the number of functions, etc.) Moreover, this general routine would be just as applicable in an entirely different Interpreter. Having an Interpreter pattern for Eiffel, for example, we'll be able to count the LOC of Eiffel programs.

Complex metrics can be handled in much the same way. Computing the deepest static nesting-level of class declarations, for instance, can be abstracted to a general metric routine in $\mathcal{C}_\mathcal{B}$ that computes the deepest level of a given kind of syntactical construct. As a consequence, getting, for example, the deepest static

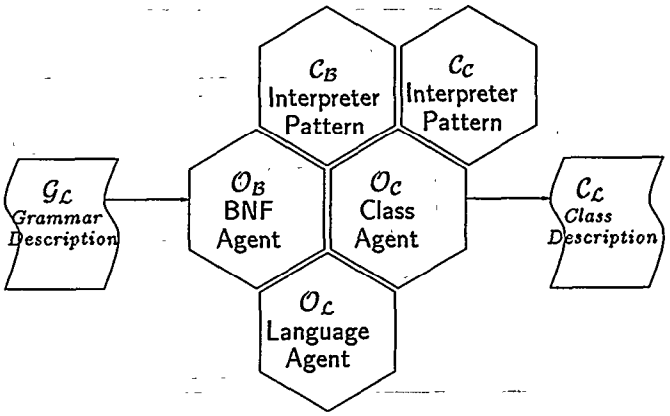

Figure 3: Tiling an In-Out Interpreter

level of conditional statements is obtained for free. More sophisticated analysis can be carried out by delegating requests back and forth between the base and meta-levels, resembling perhaps method invocations up and down the class hierarchy due to inheritance.

3.3 The Interpreter's Mirror

A tile can also be flipped over. A design pattern tells you how, given a right problem, to arrange your classes and objects in order to solve it. Conversely, an arrangement of classes and objects can remind you of a pattern you know, putting them in a totally new light.

The concept of a grammar, that ordinarily designates (formal) languages, is used by the Interpreter patterns to designate a class hierarchy. But the *grammar* metaphor works both ways. The Interpreter views a grammar $\mathcal{G}$ in terms of a class $C_\mathcal{G}$. The Interpreter's mirror would view any given class hierarchy C in terms of a grammar $\mathcal{G}_C$. Concepts taken from the theory of languages would then bring new meanings into the system's class structure.

For example, a grammar definition is associated with a special variable S called the start symbol. This symbol would correspond to the set of classes that can instantiate composition roots (i.e., *standalone* [17] objects.) In grammars, a symbol X is useful if there is a derivation in which it appears. Otherwise it is *useless* [20]. If X is useful then some terminal string must be derived from X and X must occur in some string derivable from S. We can compare usefulness of classes to other class traits provided by the programmer and supported by the programming language, such as abstract, concrete, interface, implementation, and so on.

Environmental acquisition [17] refers to the ability of objects to acquire properties in a *type safe* manner from the classes of objects in their environment. As such, it may be viewed as a generalization over class traits. Had environmental acquisition been an established design pattern, it would have been the Interpreter's mirror.

In the next section, we will make use of a flipped Interpreter tile in refining the tiling of the Reflective Interpreter.

3.4 Tiling an In-Out Interpreter

Given a language, the Interpreter usually [1] interprets sentences in the language. It can also generate sentences instead of interpreting them. The role of the grammar changes then from determining how the program is parsed to being a *synthesizing grammar* [16].

Returning to our Eiffel example, if the analyzed program is to be processed by a Java application on the Web, then we would like the class representation of the Eiffel grammar to be in Java. By implementing an Interpreter pattern for Java, we can decouple the process of translating Eiffel from the act of generating classes in Java. The Eiffel program will be compiled into an Eiffel object. The Eiffel object will create a Java object, and the Java object will generate the desired class description in Java.

Generally, two Interpreters can be tiled back-to-back such that one implementation of the pattern interprets the input, shares the interpretation with the other, which in turn outputs the desired translation. Figure 3 gives an example. Let $\mathcal{G}_C$ be a context-free grammar such that $C_\mathcal{L}$: *ProgramTypeOf*($\mathcal{G}_C$). ($\mathcal{G}_C$ is the mirror of $C_\mathcal{L}$.) Then, an Interpreter for $\mathcal{G}_B$ and an Interpreter for $\mathcal{G}_C$ can cooperate in translating $\mathcal{G}_\mathcal{L}$ into $C_\mathcal{L}$.

[1] A notable exception is a *Reversible Interpreter* [26] which is an interpreter capable of running programs backwards.

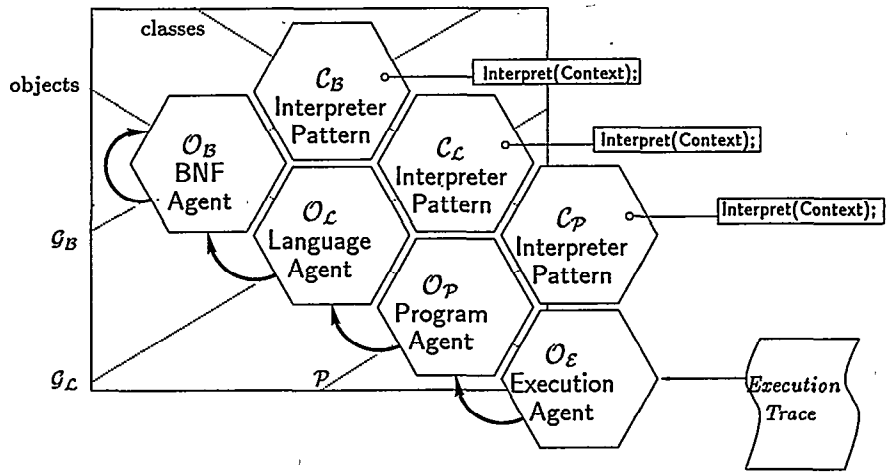

Figure 4: Tiling an Execution Interpreter

We see that two Interpreters can help bring to life a third one. Being themselves Interpreter patterns, they may be subjected to the same treatment. Of course, eventually some "fixed-point" Interpreter would need to be handled manually. Hopefully, its corresponding grammar, i.e., its mirror, would fit in the category of "simple grammars".

It is possible to merge Figures 2 and 3, applying the construction applied to $\mathcal{C}_{\mathcal{L}}$ in Figure 3 to $\mathcal{C}_{\mathcal{B}}$ and $\mathcal{C}_{\mathcal{C}}$ also, but presenting it requires a true graphic artist.

3.5 Tiling an Execution Interpreter

Recall that the Reflective Interpreter in Figure 2 established a duality between objects and types. $\mathcal{O}_{\mathcal{B}}$ is an agent representing the BNF Interpreter pattern. $\mathcal{O}_{\mathcal{L}}$ is an agent representing the language Interpreter pattern. A natural question that comes about is: does the program agent $\mathcal{O}_{\mathcal{P}}$ have a dual Interpreter pattern too (which we would denote $\mathcal{C}_{\mathcal{P}}$, as shown in Figure 5)? Tiling the Reflective Interpreter from

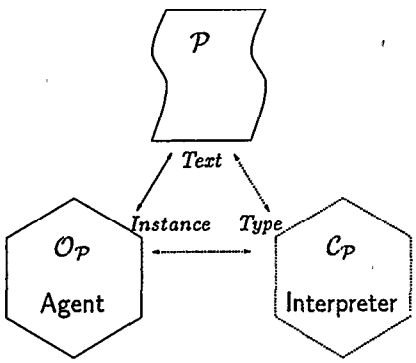

Figure 5: Program representations

the base up to the meta-level has reached its limit in $\mathcal{O}_{\mathcal{B}}$ being its own type. The concern raised here is whether we may add tiles at the other end of the Reflective Interpreter. Answering this question in the affirmative raises another one regarding the nature of a program's instantiations. Viewing a program as a type makes a program's execution (which we shall denote $\mathcal{O}_{\mathcal{E}}$) one of its values. Figure 4 illustrates how the Reflective Interpreter of Figure 2 can be extended to deal with program executions.

Generating $\mathcal{C}_{\mathcal{P}}$ is harder than it seems. For generating the language Interpreter pattern $\mathcal{C}_{\mathcal{L}}$ we needed knowledge about the language *syntax*. A separation of abstract and concrete syntax was required. For example, a syntax rule for a conditional `IfStatement` of the form

`IfStatement ::= if` b `then` s_1 [`else` s_2];

is regarded a triple $\langle b, s_1, s_2 \rangle$ (the keywords ignored but the order preserved), and implemented as a class with three data members.

```
struct IfStatement:
    public Statement {
 Expression condition_;
 Statement then_;
 Statement* else_;
 //..
};
```

Generating now the Interpreter pattern of a certain *program* requires knowledge of the *semantics* of the programming language. For example, a particular `IfStatement` instance in P, $\mathtt{If}_1$, would be mapped to a type `IfExecution` that designates the regular expression $b[s_1|s_2]$. Each time $\mathtt{If}_1$ is executed, an `IfExecution` object will be instantiated

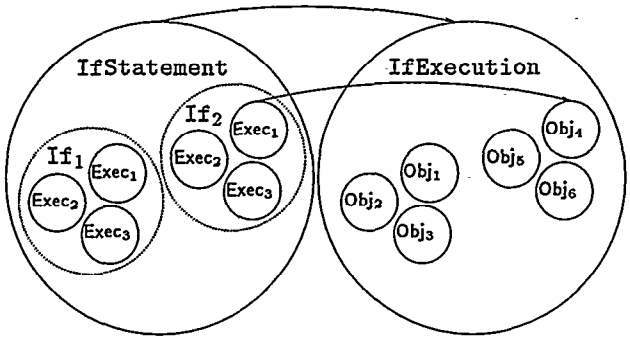

Figure 6: Class based approach to program execution

in $\mathcal{O}_\mathcal{E}$. The class `IfExecution` would comprise only two slots, as opposed to the three of its meta-class `IfStatement`: A slot of type expression that contains the execution of the condition b, and another of type statement, which would contain either the execution of s_1 or the execution of s_2.

```
struct IfExecution:
    public StatementExecution {
 ExpressionExecution condition_;
 union {
    StatementExecution then_;
    StatementExecution else_;
 } action_;
 //..
};
```

A point of interest regarding the Execution Interpreter is that although, in principle, an $\texttt{If}_2$ in P is a different type than $\texttt{If}_1$, it makes sense that both instantiate the same kind of objects. That is, `IfExecution` should be a type shared by all `IfStatement` executions, as shown in Figure 6. The meta-object representing this type could be implemented as a shared Flyweight whose meta-information link is extrinsic.

The Interpreter design pattern reflects primarily the *static* design of the program interpreted. It captures static relationships embodied in code. Applying the Interpreter pattern seamlessly to program execution captures also *dynamic* behavior of the program. Some useful implications of this are listed here.

Object-Oriented Visualization

The class definition $\mathcal{C}_\mathcal{P}$ may model only selective parts of the program execution. For example, if $\mathcal{C}_\mathcal{P}$ contains classes just for method invocation and nothing else, then an Annotalk [9, 10] trace of method calls in runtime will correspond to an object instantiation of this class definition.

Computing Execution-Based Metrics

Syntax-based metrics measure how the source code is written. Execution-based metrics measure runtime characteristics. Perceiving an execution as an instance of its program puts these two at the same conceptual level.

Executing an Interpreter Pattern

Tracing the execution of P yields an execution agent $\mathcal{O}_\mathcal{E}$. Tracing the execution of $\mathcal{O}_\mathcal{P}$, $\mathcal{O}_\mathcal{L}$ or $\mathcal{O}_\mathcal{B}$, yields traces with information from several source levels. Parts of the trace data would reflect the interpretation of P, while others would reflect the interpretation process itself and its meta processings, somewhat like what Smalltalk [13] traces look like.

4 Visitor Pattern Tessellations

The Interpreter and Visitor patterns play the role the syntax and the semantics play in a programming language, respectively. The Interpreter is concerned with the grammatical structure of programs. The Visitor is concerned with the meaning of grammatically correct programs. Unlike the nature of programming languages, the Visitor makes it possible to interpret a parsed program in new ways by simply changing the semantics of the language.

Given an interpret method invocation the Visitor pattern provides a lookup scheme for identifying the appropriate interpretation to be executed. Without the Visitor, the Interpreter's *interpret* routine is associated with a unique object. It is a mapping $m : C \rightarrow S$ from the set of element types C into the set of semantic actions S. The Visitor turns this mapping into a *curried* mapping $m : C \xrightarrow{\text{accept}} V \xrightarrow{\text{visit}} S$, where V is the set of visitors. In doing so, "Visitor lets you define a new operation without changing the classes of the elements on which it operates" [15, page 331].

It is simpler to view the Visitor as a matrix $m : C \times V \rightarrow S$ implementing a double-dispatch scheme in a single-dispatched language. One of the Visitor's variations, the *Extrinsic Visitor* [21] pattern, even implements a visitor by building this matrix explicitly, using runtime type information to access it. Indeed, in a language that supports multiple dispatch, like CLOS [11] or Dylan [3], you can do without the Visitor pattern.

The Visitor suffers from two main problems that can be put in terms of matrices: (1) Extending the matrix with a new row (class) affects all the columns

(visitors), and consequently all the other rows (making, for example, re-compilation necessary). (2) We are forced to write a method for each entry $m(c, v)$, $c \in C$, $v \in V$, regardless if we need it or not. These are especially problematic in large matrices, the ones we shall take interest in.

The *Acyclic Visitor* [27] variation provides, at the cost of multiple inheritance and dynamic casting across the visitor hierarchy, a partial solution to the cyclic dependency problem: The type of the visitor is temporarily forgotten and reclaimed using dynamic casting (see also [34]). We, on the other hand, shall focus on decreasing the complexity of implementing visitors whose matrix is *sparse* (i.e., only few classes really need special handling.) We shall see that some of the visitor variations (e.g., [21]) can be imitated by a mosaic of primitive visitors.

4.1 Primitive Visitors

The description of the Visitor pattern as it appears in [15] applies a single level of *subtyping*. An abstract `NodeVisitor` defines an interface to the `Node` hierarchy, and we can provide as many concrete visitor implementations as necessary as long as they conform with the `NodeVisitor` interface. A further degree of reuse in the Visitor pattern can be gained by allowing also *class inheritance*, that is, letting visitors refine and extend other visitors.

We can make it the Interpreter's responsibly to supply, on demand, a set of primitive visitors for that purpose. The following scheme is proposed using a Reflective Interpreter. A meta-visitor to the meta-interpreter systematically generates a group of simpler visitors to the base-interpreter. Then, these visitors serve as building blocks in constructing more complex visitors. A visitor's matrix is tessellated by inherited entries.

For example, a primitive *Null Visitor*, that does nothing at all (taking after the *Null Object* pattern [37]) is generated as following. The language agent $\mathcal{O}_{\mathcal{L}}$ is visited by a meta-visitor defined over $\mathcal{C}_{\mathcal{B}}$. This meta-visitor generates, for each encountered *rule* named `X`, an empty `NullVisitor::Visit_X` routine. [2]

For our `IfStatement` example we might get a definition like:

[2]It's a recurring pattern that whenever Visitor is discussed a footnote about overloading the visit routine appears. In [15, page 337] and [21, page 3] *function overloading* is regarded an option when implementing a Visitor in C++, only an unrecommended option. However, in C++, overriding an overloaded method hides all other overloaded definitions (unless explicitly re-using [22, Sect. 7.3.3] them.) Hence, if you wish visitors to be refined and you're not careful, overloading might change the semantics of not providing an implementation.

```
class NullVisitor {
public:
  NullVisitor() {}
  virtual ~NullVisitor() {}
  class VisitException {};

  virtual void
    Visit_IfStatement (const IfStatement& it) {}
  //...
};
```

By placing in each routine a `throw` statement, a different primitive visitor is generated, which we may call an *Exception Visitor*.

```
class ExceptionVisitor {
public:
  ExceptionVisitor() {}
  virtual ~ExceptionVisitor() {}
  class VisitException {};

  virtual void
    Visit_IfStatement (const IfStatement& it) {
      throw VisitException();
  }
  //...
};
```

The usefulness of these visitors will be made apparent immediately, but first let us introduce two additional building blocks, namely an *Inheritance Visitor* and a *Composition Visitor*.

For each superclass `Y` of `X`, an `InheritanceVisitor::Visit_X(X& it)` routine explicitly calls `Visit_Y(it)`, implementing one possible kind of refinement (element hierarchy). A second kind of refinement (visitor hierarchy) is achieved, for example, by overriding only `Visit_Y(it)` in a sub-visitor. Figure 7 illustrates this for `IfStatement` and `Statement`.

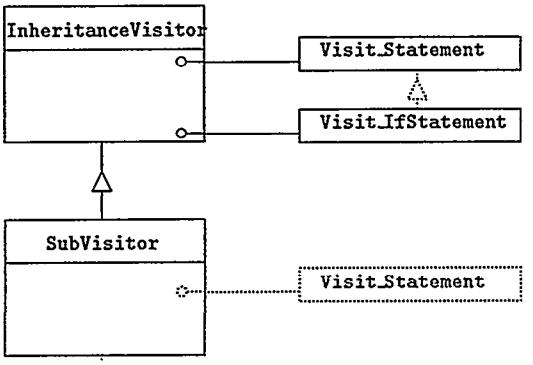

Figure 7: Inheritance Visitor

```
template <class SuperVisitor>
class InheritanceVisitor: public SuperVisitor {
public:
  InheritanceVisitor() {}
  virtual ~InheritanceVisitor() {}
  class VisitException:
    public SuperVisitor::VisitException {};

  virtual void
    Visit_IfStatement (const IfStatement& it) {
      SuperVisitor::Visit_IfStatement(it);
      //visit the sub-object
      Visit_Statement(it);
  }
  //...
};
```

A `CompositionVisitor::Visit_X(X& it)` routine explicitly calls `Visit_Z(it)` for each component `Z` of `X`.

```
template <class SuperVisitor>
class CompositionVisitor: public SuperVisitor {
public:
  CompositionVisitor() {}
  virtual ~CompositionVisitor() {}
  class VisitException:
    public SuperVisitor::VisitException {};

  virtual void
    Visit_IfStatement (const IfStatement& it) {
      SuperVisitor::Visit_IfStatement(it);
      it.condition_.Accept(*this);
      it.then_.Accept(*this);
      if (it.else_)
        it.else_->Accept(*this);
  }
  //...
};
```

All four visitors are essentially trivial and can be generated using meta-level information. Nevertheless, for the rest of this section, we'll assume that our set of tiles consists of these primitive visitors alone.

4.2 Composing Visitors

Using inheritance, a tile can refine and extend an inherited tile, as illustrated in the preceding code fragments. Genericity, as demonstrated in Figure 8, adds a *subject-oriented* [30] flavor in allowing tiles to be glued in any order we like. Suppose that v is made a subclass of another visitor v'. Then, in the absence of a handler for $m(c, v)$ the inherited entry $m(c, v')$ will be called. This rather standard mechanism of

```
template <class SuperTile>
class Tile: public SuperTile {
 //...
};
```

```
template <class SuperTile>
void Tile<SuperTile>::Visit_X(const X& it)
{
 SuperTile::Visit_X(it);
 //...
};
```

Figure 8: Gluing tiles using genericity and inheritance

inheritance can be exploited in several ways. If v' defines a default action for each c, then v is exempt from providing an implementation for element types it will never visit. Inheriting from `ExceptionVisitor` would provide runtime checking that v indeed does not visit elements it's not supposed to. If v inherits from `NullVisitor`, v need not check beforehand whether an element is visitable.

Another possibility is that v will rely on v' for delegating the interpret request to some other matrix entry. For example, if v wishes that an implementation for $m(c', v)$ is "inherited" by all entries $m(c, v)$ for which c is a subclass of c', then v should inherit from an instantiation of the `InheritanceVisitor` class template. Similarly, by inheriting from an instantiation of the `CompositionVisitor`, a call $m(c, v)$ recursively calls $m(c'', v)$ for each component c'' of c. This results in a complete scanning of the containment subtree.

Derive your visitor from `InheritanceVisitor<NullVisitor>` if you wish `YourVisitor::EvaluateY` to be called whenever you do not specify a handler for `X`, and `Y` is a superclass of `X`. If you do define `YourVisitor::EvaluateX` you may still force a call to `YourVisitor::EvaluateY` by calling `SuperVisitor::EvaluateX(it)`. Note that the Visitor `InheritanceVisitor<NullVisitor>` is precisely the *Default Visitor* proposed in [21].

Use `CompositionVisitor<InheritanceVisitor<NullVisitor> >` if you wish to perform a Depth-First Scan (DFS) of the tree (Figure 9). `YourVisitor`'s `EvaluateX` routine, if it exists, is called for nodes of type `X`, and should have in its body a call to the `SuperVisitor` routine `EvaluateX`, otherwise the DFS is not pursued.

The DFS Visitor mimics the operation of a class default constructor. This also emphasizes the power of tilings over templates. We can generate a *Copy*

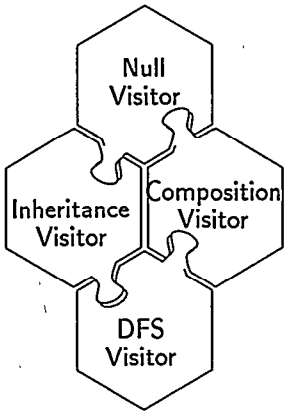

Figure 9: DFS Visitor tessellation

Visitor that will mimic the operation of the default copy constructors generated by the C++ compiler. We cannot, however, write a C++ template that would do the same.

Example

The following examples illustrates the relative simplicity of tiling visitors (although the details of application are secondary at best.) First, a visitor `V` that counts program statements is implemented in C++.

```
typedef CompositionVisitor<InheritanceVisitor<
  NullVisitor> > DFSVisitor;

// counting statements
class V: public DFSVisitor {
public:
  int counter;
  V(): counter(0) {}
  virtual void
    Visit_Statement(const Statement& it) {
      //invoked for all statements
      ++counter;
      DFSVisitor::Visit_Statement(it);
  }
};
```

That's the entire class. Sending an instance of `V` to visit a particular program p will yield the overall number of statements in that program:

```
V v;
p->Accept(v);
cout << v.counter << " statements." << endl;
```

Next, the visitor named `V_only_if` counts only if-statements.

```
// counting if-statements
class V_only_if:
  public CompositionVisitor<NullVisitor> {
public:
  int counter;
  V_only_if(): counter(0) {}
  virtual void
    Visit_Statement(const Statement& it) {
      //never invoked since Statement
      //is an abstract class
      assert(0);
  }
  virtual void
    Visit_IfStatement(const IfStatement& it) {
      ++counter;
      CompositionVisitor<NullVisitor>::
        Visit_IfStatement(it);
  }
};
```

Note that `Visit_Statement` is never called, because `InheritanceVisitor` is left out.

If we refine the first visitor `V` and add a handler also for `IfStatement`, we can count all statements except if-statements, as following:

```
// counting non-if statements
class V_sans_if: public V {
public:
  V_sans_if(): V() {}
  virtual void
    Visit_IfStatement(const IfStatement& it) {
      //invoked only for if-statements
      --counter; //undo ++counter
      V::Visit_IfStatement(it);
  }
};
```

5 Conclusions

The concept of tiling design patterns is a novel twist on pattern application. Tiling enables us to interweave simple understood concepts of patterns into their complex real-life implementation. With the help of tilings we've seen implementations of the Interpreter and Visitor patterns in a more concise and accurate manner.

Tessellations help us adapt more easily to changes (e.g., in the class hierarchy.) Similar considerations rooted the work on *propagation patterns*, later incorporated into *adaptive programming* [25, 2]. The *class dictionary graph* approach in the Demeter system, [24] which allows integrated definition of classes and syntactic forms, is of particular relevance. Tiling

match the benefits, that the *Adaptive* variants, proposed by Lieberherr and colleagues (e.g., [32]), have with respect to the traditional patterns, but without requiring language extensions.

Efforts of abstraction over design patterns have taken many directions. For example, attempts to discover patterns of patterns, or looking for recurring mini-patterns. The danger in that kind of abstractions is in becoming either obvious or too abstract. Tilings is a form of abstraction that restricts us to design pattern as wholes. As such, the building blocks at least are a solid foundation.

Some tiles fit nicely together (e.g., the Interpreter and the Visitor.) Some tiles may have the same shape but a different purpose (e.g., the *Adapter* [15] and the *Bridge* [15].) Some tiles may always need to work together (e.g., perhaps the Interpreter and the Visitor?) In that case, they may simply be parts of the same pattern. If a subset of the patterns can tile all the others, then that group would be a *basis* that can span the rest. If a set of pattern fractions can do the jobs, those would be mini-patterns. We have seen two related patterns that can tile themselves.

This work has focused on a particular design pattern, the Interpreter. Even in that limited scope, tiling has shown insight into the Interpreter's own design. Whether or not this phenomena is found in other patterns, is left for future work.

Acknowledgments I thank Harold Ossher, Yuval Sherman, Ricardo Szmit, and the anonymous referees for thoughtful comments on drafts of this paper.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer Verlag, 1996.

[2] Workshop on adaptable and adaptive software. In S. C. Bilow and P. S. Bilow, editors, *Addendum to the Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 149–154, Austin, Texas, USA, Oct. 15-19 1995. OOPS Messenger 6(4) Oct. 1995. Reported by K. Lieberherr.

[3] Apple Computer, Inc., Cupertino, CA. *Dylan: An object-oriented dynamic language*, 1992.

[4] G. M. Barnes and B. R. Swim. Inheriting software metrics. *Journal of Object-Oriented Programming*, pages 27–34, November-December 1993.

[5] J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors. *Proceedings of the 1st European Conference on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, Paris, France, June 15-17 1987. ECOOP'87, Springer Verlag.

[6] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM System Journal*, 35(2), 1996.

[7] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object-oriented design. In *Proceedings of the 6th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 197–211, Phoenix, Arizona, USA, Oct.6-11 1991. OOPSLA'91, Acm SIGPLAN Notices 26(11) Nov. 1991.

[8] J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[9] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 326–337, Portland, Oregon, USA, Oct. 23-27 1994. OOPSLA'94, Acm SIGPLAN Notices 29(10) Oct. 1994.

[10] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In M. Tokoro and R. Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming*, number 821 in Lecture Notes in Computer Science, pages 163–182, Bologna, Italy, July 4-8 1994. ECOOP'94, Springer Verlag.

[11] L. G. DeMichiel and R. P. Gabriel. The common lisp object system: An overview. In Bézivin et al. [5], pages 151–170.

[12] K. U. Erven. *The Graphic Work of M. C. Escher.* Macdonald & Co., London, new, revised and expanded edition, Oct. 1967.

[13] B. Foote and R. E. Jonson. Reflective facilities in smalltalk-80. In N. K. Meyrowitz, editor, *Proceedings of the 4th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 327–335, New Orleans, Louisiana, Oct. 1-6 1989. OOPSLA'89, Acm SIGPLAN Notices 24(10) Oct. 1989.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In O. M. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 406–431, Kaiserslautern, Germany, July 26-30 1993. ECOOP'93, Springer Verlag.

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

[16] J. Gil and D. H. Lorenz. SOOP – A synthesizer of an object-oriented parser. In *Proceedings of the 16th International Conference on Technology of Object-Oriented Languages and Systems*, pages 81–96, Versailles, France, Mar. 6-10 1995. TOOLS 16 Europe Conference, Prentice-Hall.

[17] J. Gil and D. H. Lorenz. Environmental Acquisition—A new "inheritance"-like abstraction mechanism. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, San Jose, California, Oct. 6-10 1996. OOPSLA'96, Acm SIGPLAN Notices 31(10) Oct. 1996.

[18] B. Grünbaum and G. Shephard. *Tilings and Patterns.* Mathematical Sciences. W. H. Freeman and Company, New York, 1987.

[19] B. Henderson-Sellers. *Object Oriented Metrics.* Object-Oriented Series. Prentice-Hall, 1996.

[20] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[21] M. E. N. III. Variations on the visitor pattern. In *PLoP '96* [31]. Group 5: Design Patterns.

[22] A. Koenig. *C++ Standard Draft Proposal X3J16/96-0225.* American National Standards Institute, Digital Equipment Corporation 1996.

[23] H. F. Li and W. K. Cheung. An empirical study of software metrics. *IEEE Transactions on Software Engineering*, 13(6):697–708, June 1987.

[24] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996.

[25] K. J. Lieberherr, I. Silva-Lepe, and C. Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, May 1994.

[26] H. Lieberman. Reversible object-oriented interpreters. In Bézivin et al. [5], pages 11–19.

[27] R. C. Martin. Acyclic visitor. In *PLoP '96* [31]. Group 5: Design Patterns.

[28] B. Meyer. *EIFFEL the Language.* Object-Oriented Series. Prentice-Hall, 1992.

[29] P. Naur. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):229–314, May 1960.

[30] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 235–250, Austin, Texas, USA, Oct. 15-19 1995. OOPSLA'95, Acm SIGPLAN Notices 30(10) Oct. 1995.

[31] PLoP '96. *Proceedings of the 3rd Annual Conference on the Pattern Languages of Programs*, Robert Allerton Park and Conference Center, University of Illinois at Urbana-Champaign, Monticello, Illinois, Sept. 3-6 1996. Washington University, Technical Report WUCS-97-07.

[32] L. M. Seiter, J. Palsberg, and K. J. Lieberherr. Evolution of object behavior using context relations. In D. Garlan, editor, *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 46–57, San Francisco, California, Oct. 16-18 1996. SIGSOFT'96, Acm Software Engineering Notes 21(6) Nov. 1996.

[33] E. Y. Shapiro and L. Sterling. *The Art of Prolog.* Logic Programming. MIT Press, Cambridge, Mass, second edition, 1994.

[34] J. Vlissides. Pattern hatchig: The trouble with observer. *C++ Report*, June 1996.

[35] J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors. *Pattern Languages of Program Design 2.* Addison-Wesley, 1996.

[36] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design - A System Perspective.* VLSI Systems. Addison-Wesley, 1988.

[37] B. Woolf. The null object pattern. In *PLoP '96* [31]. Group 5: Design Patterns.

[38] P.-C. Wu and F.-J. Wang. An object-oriented specification for compiler. *ACM SIGPLAN Notices*, 27(1):85–94, Jan. 1992.