A Case of Visitor versus Interpreter Pattern

Mark Hills[1,2], Paul Klint[1,2], Tijs van der Storm[1], and Jurgen Vinju[1,2]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] INRIA Lille Nord Europe, France

Abstract. We compare the Visitor pattern with the Interpreter pattern, investigating a single case in point for the Java language. We have produced and compared two versions of an interpreter for a programming language. The first version makes use of the Visitor pattern. The second version was obtained by using an automated refactoring to transform uses of the Visitor pattern to uses of the Interpreter pattern. We compare these two nearly equivalent versions on their maintenance characteristics and execution efficiency. Using a tailored experimental research method we can highlight differences and the causes thereof. The contributions of this paper are that it isolates the choice between Visitor and Interpreter in a realistic software project and makes the difference experimentally observable.

1 Introduction

Design patterns [7] provide reusable, named solutions for problems that arise when designing object-oriented systems. While in some cases it is clear which pattern should be used, in others multiple patterns could apply. When this happens, the designer has to carefully weigh the pros and cons ("consequences" [7]) of each option as applied both to the current design and to plans for future evolution of the system.

In this paper we describe one of these choices in the context of an interpreter for the Rascal[1] programming language [13], namely: the choice between structuring an abstract syntax tree-based language interpreter according to either the Visitor or the Interpreter pattern. While it seems clear (Section 3) that either pattern will do from a *functional* point of view, it is unclear what the *non-functional* quality of the interpreter will be in each case. In theory, the Interpreter pattern might have lower method call overhead because it does not involve double dispatch, it should allow easier extension with new language features, and it should be easier to add local state to AST nodes. In theory, the Visitor pattern should allow easier extension with new kinds of operations on AST nodes and should allow better encapsulation of state required by such operations. These and other considerations are exemplified in what has become known as the "expression problem" [18,4]. In this paper we investigate how the assumptions embedded in the expression problem manifest themselves in the context of a concrete case.

Our initial implementation of the Rascal interpreter was fully based on the Visitor design pattern. This choice was motivated mainly by a general argument for modularity, with each function (or algorithm) on the AST hierarchy separated into a single class. To be able to experiment with the decision of whether to use Visitor or Interpreter, we

[1] `http://www.rascal-mpl.org`

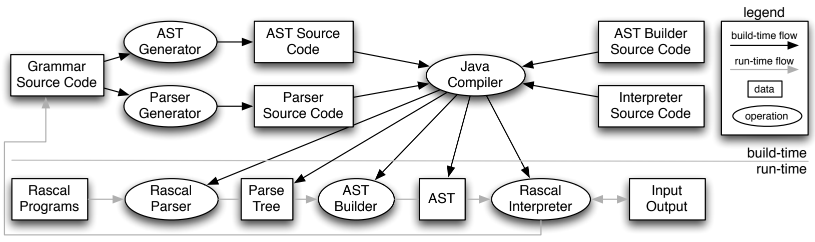

Fig. 1. Simplified build-time and run-time architecture of Rascal.

have used Rascal itself to automate an ad-hoc refactoring transforming the visitor-based design to an interpreter-based design (the details of this refactoring are outside the scope of the current paper, but we do explain the relevance of the *existence* of such an automatic refactoring for our approach). This then allows us to conduct a comparison between two implementations varying only in the choice of design pattern. In this comparison we focus on ease of maintenance and runtime performance. We show the differences between using the Visitor and Interpreter patterns in the Rascal interpreter by analysis of real maintenance scenarios and some initial performance measurements. While the results cannot be directly generalized to other software systems, we expect that other designers of tree-centric object-oriented software—compilers, interpreters, XML processors, etc.—will benefit.

Roadmap. Section 2 describes the Rascal interpreter, including the transformation from the Visitor to the Interpreter pattern, at a level of detail necessary to follow the remainder of the paper. Section 3 then explains the research methods we use to compare the maintainability and performance between the two different versions. Following this, Section 4 and Section 5 then apply these methods to analyze the differences in (respectively) maintainability and performance. Finally, we conclude in Section 6.

2 Design Patterns in the Rascal Interpreter

Rascal is a domain-specific language for meta-programming: to analyze, transform or generate other programs. While it has primitives for parsing, pattern matching, search, template instantiation, etc., it is designed to look like well-known languages such as C and Java. To facilitate integration into Eclipse[2], Rascal is implemented in Java and itself. Figure 1 depicts Rascal's build-time and run-time architecture. Because Rascal source code may contain both context-free grammars and concrete fragments of sentences for these grammars, the run-time and the build-time stages depend on each other.

The interpreter's core is based on classes representing abstract syntax trees (AST) of Rascal programs. These classes implement the Composite pattern (Figure 2) and a part of the Visitor pattern (Figure 3). Each syntactic category is represented by an abstract

[2] `http://www.eclipse.org`

class, such as `Expression` or `Statement`. These contain one or more nested classes that extend the surrounding class for a particular language construct, such as `If`, `While` (both contained in and extending `Statement`), and `Addition` (contained in and extending `Expression`). All AST classes also inherit, directly or indirectly, from `AbstractAST`. AST classes provide access to children by way of getter methods, e.g., `If` and `While` have a `getConditional()` method.

2.1 Creating and Processing Abstract Syntax Trees

Rascal has many AST classes (about 140 abstract classes and 400 concrete classes). To facilitate language evolution the code for these classes, along with the Rascal parser, is generated from the Rascal grammar. The AST code generator also creates a Visitor interface (`IASTVisitor`), containing methods for all the node types in the hierarchy, and a default visitor that returns null for every node type (`NullASTVisitor`). This class prevents us from having to implement a visit method for all AST node types, especially useful when certain algorithms focus on a small subset of nodes. Naturally, each AST node implements the `accept(IASTVisitor<T> visitor)` method by calling the appropriate visit method. For example, `Statement.If` contains:

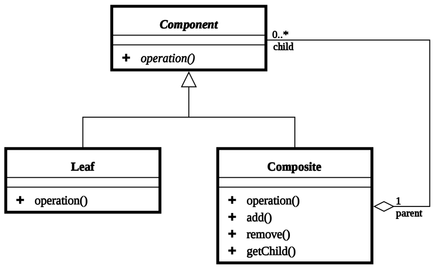

Fig. 2. The Composite Pattern[3]

```
public <T> accept(IASTVisitor<T> v) {
    return v.visitStatementIf(this);
}
```

The desire to generate this code played a significant role in initially deciding to use the Visitor pattern. We wanted to avoid having to manually edit generated code. Using the Visitor pattern, all functionality that operates on the AST nodes can be separated from the generated code. When the Rascal grammar changes, the AST hierarchy is regenerated. Many implementations of `IASTVisitor` will contain Java compiler errors and warnings because the signature of visit methods will have changed. This is very helpful for locating the code that needs to be changed due to a language change. Most of the visitor classes actually extend

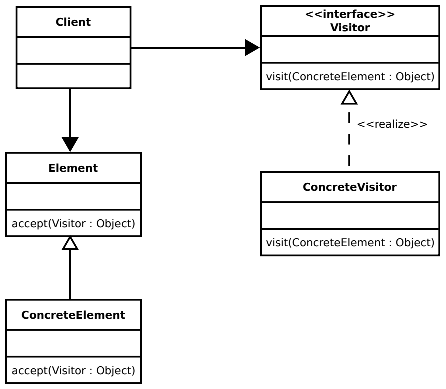

Fig. 3. The Visitor Pattern[4]

[3] Image from `http://en.wikipedia.org/wiki/Composite_pattern`
[4] Image from `http://en.wikipedia.org/wiki/Visitor_pattern`

NullASTVisitor though, which is why it is important that each method they override is tagged with the @Override tag[5]. Note that the class used to construct ASTs at runtime, ASTBuilder, uses reflection to map parse tree nodes into the appropriate AST classes. Hence, this code does not have to change when we change the grammar of the Rascal language.

2.2 A Comparison with the Interpreter Pattern

Considering that our design already employs the Composite pattern, the difference in design complexity between the Visitor and Interpreter patterns is striking (Figure 4). The Composite pattern contains all the elements for the Interpreter pattern (abstract classes that are instantiated by concrete ones)—only an interpret method needs to be added to all relevant classes. So rather than having to add new concepts, such as a Visitor interface, the accept method and NullASTVisitor, the Interpreter pattern builds on the existing infrastructure of Composite and reuses it. Also, by adding more interpret methods (varying either the name or the static type) it is possible to reuse the Interpreter design pattern again and again without having to add additional classes. However, as a consequence, understanding each algorithm as a whole is now complicated by the fact that the methods implementing it are scattered over different AST classes. Additionally, there is the risk that methods contributing to different algorithms get tangled because a single AST class may have to manage the combined state required for all implemented algorithms. The experiments discussed in Section 4 help make this tradeoff between separation of concerns and complexity more concrete.

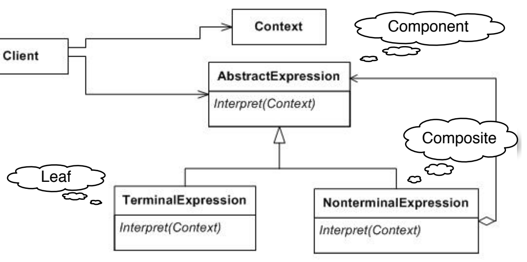

Fig. 4. The Interpreter Pattern with references to Composite (Figure 2).[7]

2.3 Refactoring from Visitor to Interpreter using Rascal

We constructed an automated refactoring tool for transforming Visitor classes to Interpreter methods. It is the key to our research method (see Figure 5). However, the details of constructing the refactoring are out of the scope of the current paper. They can instead be found online [11]. The benefits of an automated approach are:

Reproducible target code makes it easy to replay the refactoring during experimentation, while also allowing others to literally replicate the experiment;

[5] If a method is tagged with @Override the Java compiler will warn if it does not override any method anymore.

[7] Image from http://en.wikipedia.org/wiki/Interpreter_pattern, created by Jing Guo Yao and licensed under the Creative Commons Attribution-ShareAlike 3.0 License.

Automated analysis checks that semantics are preserved and the transformation is complete (i.e., no visitors are missed during the transformation);

One thing at a time automated refactoring does not suffer from the temptation during a large manual refactoring to make other changes as well, which would confound the analysis and hinder reproducibility.

The tool is implemented using a combination of Rascal and Java. The Java code is used to access features of the Eclipse JDT[8] used for fact extraction, source code cleanup, and refactoring. The Rascal code is used to analyze and aggregate this information, to call JDT refactorings with the right parameters and to generate the new code.

3 Comparing Design Patterns

The research strategy of this paper can be characterized as idiographic [1]: we seek to understand a single phenomenon (i.e. Visitor vs. Interpreter) in a particular context (the implementation of Rascal). The context for our study is further established by fixing the following variables: programming language (Java), application area (programming language interpreter), and the use of the Eclipse IDE. We assume that the AST classes used in the interpreter are implemented using the Composite pattern. Finally, we require all regression tests for the interpreter to run unchanged as we vary the system.

Within this context, the primary free variable is the choice between the patterns we are comparing: Visitor and Interpreter. The two dependent variables we wish to measure are differences in maintainability and runtime performance between two versions of the interpreter that use the two design patterns but are otherwise functionally equivalent. The dependent variables are measured in a number of maintenance scenarios categorized according to ISO 14764 [12]: *perfective* (speed optimization), *corrective* (bug fixes), and *adaptive* (new features).

3.1 Measuring Differences in Runtime Performance

In Section 5 we measure differences in speed between the two versions of the interpreter, as well as showing the improvement in both versions from one of the maintenance scenarios. We use a benchmark of running 4 different Rascal programs, designed as representative workloads. In our experiments runtime performance is measured in wall-clock time, averaged over multiple runs, with an initial run of each test to try to minimize differences from just-in-time compilation during later runs.

3.2 Measuring Differences in Maintainability

Differences in maintainability are less straight-forward to measure. A large number of metrics exist for measuring object-oriented systems [10], including metrics specifically aimed at maintenance. One such metric, "Maintenance Complexity"[9], is defined as an aggregate sum of weighted occurrences of uses of programming language constructs.

[8] Java Development Toolkit; `http://www.eclipse.org/jdt`

[9] By Mark Miller (unpublished).

While this may be used to get an indication of the complexity of maintaining a single method, it is not clear how it could be used to compare the complexity of two systems using different design patterns. In other efforts there have been attempts to quantify differences between systems using design patterns and those without, focusing either on understandability [2], maintenance effort [17], or modularity [8].

Metrics such as the maintainability index (MI) [16,3] and the SIG maintainability model (SMM) [9] also produce numerical results that help predict the long-term maintenance cost. The MI does not allow for cause analysis, while the SMM does. The difference lies in the (ir)reversibility of aggregation formulas. Both metrics produce a system-wide indicator of maintainability independent of the kind of changes that are applied to it. This level of abstraction is useful for managers who wish to track the evolution of a large software system, but is less useful for studying the effect of choosing design patterns. In reality, any object-oriented system is more amenable for certain kinds of changes than others.

Instead of the above metrics, we opt for a metric inspired by the concept of *Evolution Complexity* [5,15] (EC). EC was devised by Mens and Eden to provide a foundation for reasoning about the complexity of changes to software systems. EC is defined as the computational complexity of a meta program that transforms a system to obtain a new version. Each transformation is implied by a shift in requirements. As opposed to the aforementioned system-wide metrics, this provides a means to reason about maintainability, subject to specific evolution scenarios and specific parts of a system.

In the current paper we need a more precise measure that not only measures the effort to transform the system, but also the effort to analyze it before applying any transformations, the cost of which can govern the overall cost of maintenance [14]—one first needs to know where and what to change before actually making any changes. To account for this, we introduce the concept of a *maintenance scenarios*, which then allows us to determine the complexity of maintenance.

Definition 1. *A maintenance scenario S is a description of a required change to a program P that implies a set of changes in its source code. Implicitly, all previous requirements—unless contradicting the current change—need to be conserved.*

Definition 2. *The complexity of maintenance* COM *is the computational complexity of a meta program (M_S) that analyses and transforms the source code of program P to implement a specific maintenance scenario S:*

$$\textsc{Com}(P, M_S) = \textsc{ComputationalComplexity}(M_S(P)).$$

This definition implies a detailed subjective model of maintainability that depends on the design of the system, the maintenance scenario, the way the analysis and transformation is executed, and the definition of computational complexity. With so many subjective variables, it is impossible to use it to estimate maintainability of a specific system. Such an absolute complexity metric would be too sensitive to differences in interpretation. Instead, we use it as a *comparative* framework, specifically for comparing two systems that are equal in all but one aspect: the choice between two design patterns.

Figure 5 describes our framework to compare the maintainability of two versions n and m of a given system. Version m has been derived from version n by way of an automated refactoring, i.e. a meta-program that preserves the functional behavior of

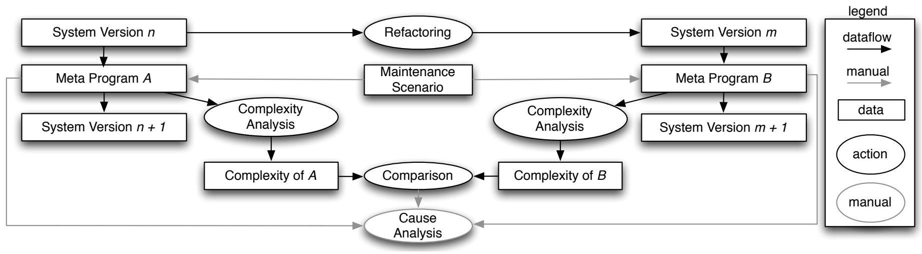

Fig. 5. Comparative framework for observing differences in maintainability.

version *n* but may change some non-functional characteristics. In our case study, version *n* is the Rascal interpreter based on the Visitor pattern and version *m* is the version of the Rascal interpreter based on the Interpreter pattern. The details of this automated refactoring are not relevant for the present analysis, but it is important to note that it is semantics preserving. The maintainability of both versions is now compared by designing a number of maintenance scenarios and applying them to both versions. For each maintenance scenario we do the following:

- Perform the maintenance scenario manually.
- Create an abstract description of this activity by expressing it as meta-program.
- Compare the computational complexity of the meta-programs needed to carry out the maintenance scenario for versions *n* and *m*.

This allows us to objectively calculate the complexity of the scenarios as applied to the two versions while at the same time pinpointing exact causes of the differences.

Results produced by this framework can be replicated by anybody given the source code of the two versions, a precise description of the meta programs and the scenarios, and a precise description of the complexity analysis. In Section 4.1 we define a "virtual machine for maintenance" that provides the foundation for our current comparison.

3.3 Alternative Methods to Measure Maintainability

Our framework tries to abstract from the human programmer that actually carries out the maintenance tasks. This makes it easier to replicate our results. Alternative ways of studying maintenance do focus on human beings, like programmer observation (e.g., [6]) and using models of cognition (e.g., [19]).

Statistical observation of the efficiency of a group of programmers while doing maintenance tasks can be done to summarize the effects of differences between design patterns. However, such an (expensive) study can not explain the causes of these effects, while our method can. The use of cognitive modeling can also shed light on the causes of complexity. With this method one explicitly constructs a representation of the knowledge that a human being is using while analyzing and modifying source code. Complexity measures for such representations exist as well and have been used to study understandability of programming in different kinds of languages [19]. We have not opted for this approach because such detailed cognitive models are difficult to construct

Cat	Action	Description	Motivation
(S)	*a*	Save Java file	Collect error messages by running the Java compiler.
(S)	*b*	Get type declaration	Look up a type by name and jump to it.
(S)	*c*	Get type hierarchy	Produces all classes and interfaces that implement or extend a given type.
(B)	*d*	Jump to error	Jump to the source code after having clicked on the error message.
(E)	*e*	Cut or copy a block	This is a basic action to perform removal and movement of consecutive blocks of code. A block is considered to be no longer than a single method.
(E)	*f*	Paste a block	The dual of *e*.
(E)	*g*	Type a block	We abstract from the difficulty of writing consecutive blocks inside method bodies. Typing several method bodies, or parts of method bodies, is counted as several steps, even if the methods are consecutive.
(S)	*h*	Get implementations	Produces all concrete methods that implement a certain abstract/interface method.
(B)	*i*	Jump to declaration	Jumps from a use site or other reference site to a declaration site of any named entity.
(S)	*j*	Find regexp	We abstract from the effort of creating a regular expression. The action produces a list of locations of entities that match the regexp.
(E)	*m*	Generate new class	Make a new class with the given name and superclass, including templates for methods to be implemented.
(E)	*n*	Delete a class	Remove a type and its source file.

Table 1. Atomic actions, categorized as (S)earch, (B)rowse or (E)dit actions.

well by somebody not well versed in cognitive science (there are many ways to do it), hard to reproduce and therefore hard to validate. Our current method, as inspired by [15], is lightweight and easy to construct by software engineers and easy to replicate.

4 Maintainability

This section instantiates the comparative framework discussed in Section 3.2 to compare the Visitor-based and Interpreted-based solutions. Section 4.1 defines how we construct and measure the meta-programs representing the scenarios. Section 4.2 then introduces the scenarios that will be measured, while Section 4.3 describes each scenario in detail.

4.1 A Virtual Machine for Maintenance Scenarios

Recall from Section 3.2 that each maintenance scenario is performed manually and then described by an abstract meta-program used to compute the complexity of the scenario. To precisely define these meta-programs we encode them as the language of a "virtual machine" for maintenance scenarios. This VM models the *actions* of a maintenance programmer as she interacts with Eclipse to analyze and transform source code.

The atomic actions (steps) taken by this virtual machine are defined in Table 1. We have Search (S) actions that produce lists of clickable items; Browse (B) actions that involve following links; and Edit (E) actions that change source texts in specific locations. From these atomic actions we may construct meta programs representing the various maintenance scenarios according to the following definition.

Definition 3. *All maintenance programs P have the following syntax*

$P ::= A \mid PP \mid P^I \mid (P)$,

where A is an atomic action from Table 1, juxtaposition denotes sequential composition, and a superscript (a non-zero positive integer) denotes iteration. We may use brackets to bind iteration to sequences of actions, otherwise iteration binds more strongly than sequence. Parts of a program may be represented by a variable (represented by uppercase letters in italics) and variables may optionally be indexed: A_i represents atomic actions, N_i and M_i represent values in $\mathbb{N}_1$, and P_i represents programs.

Definition 4. *The computational complexity of any maintenance program P is defined recursively as:*

$$\textsc{Com}(A) = 1, \qquad \textsc{Com}(P_0 P_1) = \textsc{Com}(P_0) + \textsc{Com}(P_1),$$
$$\textsc{Com}((P)) = \textsc{Com}(P), \qquad \textsc{Com}(P^N) = N \times \textsc{Com}(P).$$

With these definitions we can now explain each maintenance scenario in detail. The results are summarized in Table 2.

4.2 Maintenance scenarios

We have picked several maintenance scenarios to cover most categories of maintenance and to be fair to the theoretical (dis)advantages of either design pattern. We skip preventative maintenance, which will appear instead in the discussions below as refactorings that influence the comparison.

S1 (Adaptive) Add $n \geq 2$ new binary expression operators.

S2 (Perfective) Cache the lookup of (possibly) overloaded data-type constructors in expressions to improve efficiency. This can be generalized to caching *n* static language constructs.

S3 (Adaptive) Change the syntax and semantics of Rascal to allow arbitrary value patterns in function signatures. This new feature allows functions to be extended modularly, which is a big win for analyses and transformations that are constructed for languages that have a modular structure.

S4 (Adaptive) Add an outline feature to the Rascal IDE — a basic IDE feature already supported in IDEs for many different languages.

S5 (Corrective) Fix Bug #1020 — `NullPointerException`[10]

Note that at the time of writing, these are real maintenance scenarios. The interested reader can replay the meta programs below by checking out the Visitor[11] and Interpreter[12] versions of the Rascal interpreter that are used in this paper.

[10] `http://bugs.meta-environment.org/show_bug.cgi?id=1020`

[11] `http://svn.rascal-mpl.org/rascal/tags/pre-visitor-migration`

[12] `http://svn.rascal-mpl.org/rascal/tags/post-visitor-migration`

4.3 Results — Maintenance Scenarios

In this section we list all programs for all scenarios. We motivate the actions of each program, analyze the difference in complexity, and point to the possible causes. Table 2 summarizes all the acquired data points. Some scenarios require common preparation for both Visitor and Interpreter. This is discussed for completeness, but not included in the comparison and not represented in Table 2.

Scenario S1 — Add Two New Expression Operators

To prepare, we edit the Rascal grammar to add two new production rules to the definition of `Expression`. Then we generate and compile source code for the AST hierarchy.

For Visitor we find out that no new warnings or errors have arisen. This is due to the fact that all visitors extend `NullASTVisitor`, which is also generated from the grammar. We have to find all visitors now, and use the Show Type Hierarchy feature of Eclipse to find 11 of them (c). We look up the source code of each visitor to see if expressions are evaluated by it (i^{11}). This is true for just 2 of them, namely the main `Evaluator` and the `DebuggingDecorator`. Both visitors need two extra methods added ($(g^2a)^2$). We run the Java compiler (part of a, above) to ensure we did not make mistakes, obtaining the meta program: $ci^{11}(g^2a)^2$.

For Interpreter we also find out there are no new warnings after AST generation. We now add two concrete sub-classes to the generated sub-classes of `ast.Expression`(m^2). There appear to be four methods to implement, three of which we clone from `Expression.Add` (selected at random) because they seem to be default implementations ($b(ef^2)^3$). We then adapt the one method (`interpret`) in both classes that we must change ($(ga)^2$). The total meta-program is thus: $m^2b(ef^2)^3(ga)^2$.

A comparison of the complexity (18 vs. 16) shows a minimal difference in favor of Interpreter.

Scenario S1(N) — Add N New Expression Operators

To generalize to N new operators we can replace 2 by N in the two programs for S1 to obtain new programs $ci^{11}(g^Na)^2$ for Visitor and $m^Nb(ef^N)^3(ga)^N$ for Interpreter. Their complexity breaks even at $N = \frac{5}{2}$. This indicates that after adding 2 operators further additions will be easier in Visitor than in Interpreter. One cause may be the *cloning* of the 3 methods from `Expression.Add` (See **S1**). It is a seemingly unrelated design flaw. If these methods could be pulled up into `Expression`, the Interpreter program would have no need to clone the other three methods.

Scenario S1'(N) & S1'(N,M) — Pulling Up Methods and Another Generalization

Pulling up the method clones in Interpreter (see S1(N)) leads to a new program for adding N new expression operators, $m^N(ga)^N$. This program has complexity $3n$, which breaks even with Visitor at $N = 14$. Visitor wins in this case, but only after having added 14 operators. The cause is that only 2 out of 11 of Visitor classes actually need an extra pair of methods. If there would be more visitors to extend however, there would also be more methods to implement per class in the Interpreter version. Abstracting the number of operations on each operator to M (assuming the new ones all need extension, but 9 of the existing ones do not), we get $ci^{9+M}(g^Na)^M$ for Visitor and $m^N(ga)^{MN}$ for Interpreter. Break-even is when $N = \frac{2M+10}{M+1}$. The constant 10 increases with the number of irrelevant visitors and break even is harder to reach for Visitor while M increases.

In general we can conclude that for **S1** Visitor wins in the long run, although it wins more slowly in situations where there are a large number of visitor classes that do not need to be modified (but still need to be checked). Interpreter has a higher eventual maintenance cost because of the additional classes that need to be created.

Scenario S2 — Cache Constructor Lookup in Expressions

Constructors in Rascal can be defined at the top level of any Rascal module. When a constructor is used in a program, the current module, and all imported modules, are checked for definitions of the constructor. Since these definitions can only be changed when a constructor is (re)defined, it should be possible to improve performance by caching the lookup result, with the cache cleared at each redefinition.

For Visitor we first find the main `Evaluator` visitor to locate the visit method that represents function and constructor application (i^2). Reading the source code of `visitExpressionCall-OrTree` we learn that this visit method evaluates the name of the function or constructor to obtain an object of type `Result` that has a `call` method. We want to cache this object for future reference if it represents a constructor. In order to do this, a field must be added to the current visitor (we could instead add a field to the underlying AST node class, but since the AST classes are generated this would require changing the generator as well). This field will reference a hash table that maps the current AST node to the result of the name lookup. We need to add the field (g) and add the two locations in the code that cache and retrieve constructor values (gg). To clear the cache we need to find the method where constructors are declared. We use the outline feature to jump to `visitDeclarationData` (i) and add some code to clear the entire cache (g). The total program is i^2g^3iga.

For Interpreter we locate the AST class `Expression.CallOrTree` and its `interpret` method (i^2). We add a field to the AST class to store a cached constructor and we surround the lookup with the storage and retrieval code for this cached value (g^3). To clear this field when a module is reloaded, we choose to apply the Listener design pattern [7]. When a constructor is cached a new `IConstructorDeclared` listener will be registered with the current `Evaluator` (g), which is passed as a parameter to the `interpret` method. We now save the current class (a). The Listener design pattern needs to be completed by adding a container for the listeners, a `register` method and a `clear` method to `Evaluator`. For this we jump to the class and add the field and two methods (ig^3a). Then we find the `Declaration.Data` class to add the code to call the `clear` method when a constructor is (re)declared, yielding: $i^2g^3gaig^3aiga$.

In summary, interpreter is harder to maintain. An alternative design choice for Interpreter would be to use a global hash-table, like we did with Visitor. This removes the need for introducing the listener design pattern and thus gives the same complexity. Having a field instead of a hash-table is important for speed though (see Section 5). Alternatively, for Visitor we could have chosen not to use a hash-table but instead add a field to `AbstractAST`. However, this would break the encapsulation gained through Visitor and, as mentioned above, would require modifying the AST class generator as well.

The following change in requirements (S3) involves non-trivial and non-local changes in the syntax and semantics of the language. Again, we assume the maintainer has full understanding of the concepts and implications for the general architecture of the Rascal interpreter. She does, however, need to locate and check the details of implementing the necessary changes.

Scenario S3 — Allow Patterns in Function Signatures

To prepare, we need to edit the definition of formal parameters in the Rascal grammar. There we replace the use of `Formal` by `Pattern`. The AST hierarchy is regenerated and the Java checker and compiler are executed to produce error messages and warnings. We omit this common prefix in the following discussion.

For Visitor the compiler produces 14 error message, each about a reference to a missing class `Formal`. Uses of `Formal` need to be replaced with `Expression` and imports of `Formal` need to be deleted. This results in a cascade of changes up the call chain starting at these 14 error locations. Using the JDT we adapt each location one-by-one and save each file after each change to produce new error messages. Just the first error leads to dg^5eg. Then we find a nested visitor in `TypeEvaluator` that dispatches over the different kinds of type declarations. We decide to extend it with a type analysis of each pattern kind. There are 15 different kinds of patterns (known from reading the type hierarchy of `Expression`) (cg^{15}). Two more substitutions complete the changes to this file (g^2a).
These were the changes rooted at the first error. We now have 4 of 14 messages left. These happen to point to dead code that can be removed: $(eea)^4$.
Now we add a call to pattern matching. Given we are modifying function call logic, we first jump to `visitExpressionCall-OrTree` in the main `Evaluator` visitor (i^2). We find a call to the `call` method of an abstract class `Result`. All implementation of this method are suspect. We use action h to find all 9 of them. After inspection, 3 of these need additional functionality: `RascalFunction`, `JavaFunction` and `OverloadedFunction`. The others have names related to constructs that are not related to function declarations with formal parameters.
Pattern matching both returns true and binds variables if the match succeeds. We can replace the code that binds actual to formal parameters by pattern matching. We also need to add backtracking logic, and decide to do so with an exception mechanism. If the pattern match fails, the function was not to be called and we throw an (unchecked) exception that can be caught at a choice point in `OverloadedFunction`. The three `call` methods are adjusted to do just that ($(ga)^3$). The total program for Visitor is $dg^5egcg^{15}g^2a\ (eea)^4i^2h(ga)^3$.

For interpreter the generation of the AST hierarchy produces 17 error messages. The first is located in `DynamicSemantics-ASTFactory` which refers to a constructor that does not exist anymore (d). The constructor for `Formals.Default` still uses the old form of parameter lists. We fix this first (ig).
The next error message is in the `interpret` method of `Formals.Default` that evaluates ASTs of type literals. We jump to it and find a need to substitute `Formal` (iga). This recursive method maps ASTs of type literals to internal type objects. This method will also have to deal with all kinds of patterns now. We add an implementation of it to every kind of pattern. We look up the type hierarchy for `Expression` to identify the 15 classes and add a method to each of them ($(iga)^{15}$).
Jumping to the location of the next error, we end up in `JavaBridge`. A number of similar substitutions are needed and an import is removed: $(ig)^3ga$. Then we trace a broken method call to the class `TypeEvaluator` (i). There we find some substitutions (ig^2a). The last 3 errors point to dead code that can be removed, a dead class and a dead import in a class ($(igg)^2aniga$).
Now we may add pattern matching, which is done similarly to the Visitor implementation. We jump to the `Expression.CallOrTree` class to find the semantics of function calling; and use the same strategy we used for Visitor ($ih(ga)^3$).
The total program for Interpreter is $d(ig)^2a(iga)^{15}(ig)^3$ $gai(ig^2)a(igg)^2anigaih(ga)^3$.

Scenario S3' — S3, but Saving Incrementally for Visitor

The cause of the significant difference in complexity in **S3** (43 vs. 83) between the Visitor and Interpreter patterns is clearly the spread of code over the different classes. In Visitor there is much less browsing between classes and saving of classes, leading to almost twice the maintenance complexity for Interpreter. Note that browsing to a different class that needs editing always costs Interpreter a Browse and a Search action if something needs to be edited (for saving and compiling the file after editing), while Visitor may delay the saving of a file until all is done. It is questionable whether in reality one would delay saving the file after so many edits in a big visitor class. If we add save actions to the Visitor program after every edit, we get $d(ga)^5egac(ga)^{15}(ga)^2(eea)^4i^2h(ga)^3$, with complexity 70. Visitor still wins, but now it is only 16% cheaper instead of the previous 48%.

Scenario S4 — Add Outline

To prepare, both versions need similar code to register an outline computation with Eclipse.

For Visitor we simply add a new visitor class. This class needs methods for all AST nodes that need to be traversed to find the entries that appear in the outline view. There are 11 different nodes, yielding $mg^{11}a$.

For Interpreter we add a new virtual method to `AbstractAST` called `outline`. It will be overridden by 11 classes. The method needs a parameter to a `TreeModelBuilder` interface to construct the outline object that Eclipse will use. So this ties AbstractAST to an Eclipse interface. The meta program reads $bga(bga)^{11}$.

Visitor clearly wins in this case because of the improved encapsulation of the solution.

The description of Bug #1020 in our Bugzilla database contains the claim that the following Rascal statement produces a NullPointerException due to some issue in a regular expression: `switch ("aabb") {case /aa*b$/:println("HIT");}`

Scenario S5 — Fix Bug #1020 — `NullPointerException`

This issue indeed produces stack traces for both versions, and surprisingly the traces are the same. The reason is that a null reference to a result is passed all the way to the top command-line shell. We trace the flow of this reference down the call chain.

For *Visitor*. The outermost expression is a `switch`, so we jump to the evaluation of the switch in the method `Evaluator.visitStatementSwitch` (*bi*). The last statement of this method returns 'null' which needs to be replaced by a 'void' result (*ga*).

The *Interpreter* case has one fewer browse action (*bga*).

4.4 Discussion

On the one hand, even in scenarios where theoretically Interpreter would have better encapsulation (e.g. **S1** and **S2**), Visitor still has a lower cost of maintenance. This is surprising. On the other hand, the scenarios that theoretically suit Visitor better indeed show that it is superior. No counter indicators were found in the context of this realistic case. At least in the context of the Rascal interpreter, our research method consistently produces “Visitor is better”.

S	**Visitor**	(COM)	**Interpreter**	(COM)	**Vis.>Int.**
S1	$ci^{11}(g^2a)^2)$	(18)	$m^2b(ef^2)^3(ga)^2$	(16)	yes
S1(N)	$ci^{11}(g^Na)^2)$	$(14+2N)$	$m^Nb(ef^N)^3(ga)^N$	$(4+6N)$	if $N \leq 2$
S1'(N,2)	$ci^{11}(g^Na)^2)$	$(14+2N)$	$m^N(ga)^N$	$(3N)$	if $N \leq 14$
S1'(N,M)	$ci^{9+M}(g^Na)^M$	$(10+NM+2M)$	$m^N(ga)^{MN}$	$(N+2MN)$	if $N \leq \frac{2M+10}{M+1}$
S2	i^2g^3iga	(8)	$i^2g^3gaig^3aiga$	(14)	no
S3	$dg^5egcg^{15}g^2a(eea)^4i^2h(ga)^3$	(43)	$d(ig)^2a(iga)^{15}(ig)^3gai$ $(ig^2)a(igg)^2anigaih(ga)^3$	(83)	no
S3'	$d(ga)^5egac(ga)^{15}(ga)^2$ $(eea)^4i^2h(ga)^3$	(70)	$d(ig)^2a(iga)^{15}(ig)^3gai$ $(ig^2)a(igg)^2anigaih(ga)^3$	(83)	no
S4	$mg^{11}a$	(13)	$bga(bga)^{11}$	(36)	no
S5	$biga$	(4)	bga	(3)	yes

Table 2. A comparison of all maintenance programs (see Table 1).

In terms of *construct validity* one may argue that the COM framework may not measure all relevant aspects of maintenance. The first aspect that is missing is the general understanding that a programmer needs of the particular program, before she can decide what to look for and what to change. We argue that this knowledge is equally needed for Visitor and Interpreter. We do not use COM for predicting maintenance effort, but for comparison. The second aspect is that we did not distinguish whether or not method bodies are hard to understand. Fortunately, in the case of Visitor vs. Interpreter the method bodies are practically equivalent in complexity on both sides.

We do not claim much about *external validity*. The current study is highly focused on the Rascal case. We do expect that if the current study were replicated on different AST processing software written in Java, with different maintenance scenarios, the results would be comparable. This expectation is motivated by the fact that the scenarios above do not refer to any intrinsic details of the syntax and semantics of Rascal.

We have assumed ample use of browsing, searching and editing features of Eclipse. It is unknown what the effect of not having these tools would be on the case of the Rascal interpreter.

Finally, if other quality attributes enter the scene, or other refactorings are applied, our conclusions about maintainability and runtime performance may be invalidated. The dimension of (parallel) collaborative development—as enabled by a modular architecture—might have an unpredictable impact on our results.

In terms of internal validity, we hope to have provided enough detail for the reader to be able to replicate the scenarios and their measurement. If shorter but otherwise plausible meta programs are defined, this might invalidate our analysis. Naturally, our interpretation of the causes of differences is also open to discussion.

5 Efficiency

We now focus on the effect on run-time efficiency of moving from Visitor to Interpreter. The impact is measured using four programs, designed both to highlight different aspects of performance and to represent typical Rascal usage scenarios:

Add finds the sum of the first 1000000 integers using a loop. It isolates the dispatch overhead of the interpreter because the computation is so basic (i.e., does not involve Rascal function calls or complex primitives like transitive closure computation).

Gen consists of running the parser generator (implemented in Rascal) on Rascal's grammar.

Resolve is the name resolution phase of the Rascal type checker, applied to one of the parser generator modules. It exercises a wider range of AST classes then **Gen**.

Lambda is a parser and interpreter for the lambda calculus. The test involves parser generation, parsing and execution of lambda reductions over arithmetic expressions (Church numerals). It highlights the result of caching constructor names.

Each program is run using both the Interpreter and the Visitor versions, before and after applying scenario S2 (cache constructor names).

The results are shown in Table 3. In the **Add** example the Interpreter code is slightly slower, while in the others it is faster by 1.3% (**Gen**), 2.5% (**Resolve**), and 5.8% (**Lambda**). Except perhaps for **Lambda**, this means that the performance difference is not substantial in any of the cases that do not include caching. We found this surprising, since one of our assumptions was that we would see a performance improvement based on a reduction in method call overhead. Also, the improvements from an optimization like name lookup caching are far more significant than the improvements from changing from Visitor to Interpreter. While this means that these types of optimizations may be a more fruitful target to pursue, this also means that slow parts in the interpreter may be impacting performance enough that differences between the two patterns are harder to see. Additional performance testing, with a broader suite of test programs, should help to get a clearer idea of the performance differences, especially as additional optimizations are added to the Rascal runtime.

	Visitor		Interpreter	
	No Caching	Caching	No Caching	Caching
Add	7.55	7.70	7.71	7.52
Gen	275.50	273.65	271.88	243.24
Resolve	35.21	35.67	34.32	32.44
Lambda	610.81	655.19	575.61	567.80

Table 3. Interpreter performance figures (4 versions, all times in seconds; tests run on Intel Core2 6420, 2.13 GHz, 2 GB RAM, Fedora linux 2.6.32.21-168.fc12.x86_64).

6 Conclusion

We have used quantitative methods to observe the consequences of choosing between the Interpreter design pattern and the Visitor design pattern. The study focused on an AST based interpreter for the Rascal programming language. Surprisingly, for the five realistic maintenance scenarios we have studied, it appears that a solution using the Visitor pattern is more maintainable than a solution using the Interpreter pattern. Only in trivial scenarios is an Interpreter-based solution easier to maintain. Since this contradicts common wisdom regarding the expression problem, it underlines the importance of studying the consequences of choosing design patterns in realistic experiments.

With respect to performance, we have observed no significant differences between unoptimized solutions using the two patterns. Any differences between the two solutions may be easier to see as the Rascal interpreter is further optimized, leaving the call overhead in the Visitor implementation as a larger part of the total execution time. It may also be possible to see more differences as additional performance tests are selected beyond the four given in this paper.

References

1. I. Benbasat, D. K. Goldstein, and M. Mead. The case research strategy in studies of information systems. *MIS Q.*, 11:369–386, September 1987.
2. A. Chatzigeorgiou, N. Tsantalis, and I. S. Deligiannis. An empirical study on students' ability to comprehend design patterns. *Computers & Education*, 51(3):1007–1016, 2008.
3. D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27:44–49, August 1994.
4. W. R. Cook. On understanding data abstraction, revisited. In *Proceedings of OOPSLA'09*, pages 557–572. ACM, 2009.
5. A. Eden and T. Mens. Measuring Software Flexibility. *IEE Proceedings—Software*, 153(3):113–125, 2006.
6. S. D. Fleming, E. Kraemer, R. E. K. Stirewalt, L. K. Dillon, and S. Xie. Refining Existing Theories of Program Comprehension During Maintenance for Concurrent Software. In *Proceedings of ICPC'08*, pages 23–32. IEEE, 2008.
7. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
8. J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of OOPSLA'02*, pages 161–173. ACM, 2002.
9. I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *Proceedings of QUATIC'07*, pages 30–39. IEEE, 2007.
10. B. Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, 1996.
11. M. Hills. Rascal Visitor to Interpreter (V2I) Transformation. `http://www.cwi.nl/~hills/rascal/V2I.html`.
12. ISO. International Standard - ISO/IEC 14764 IEEE Std 14764-2006. *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998)*, pages 1–46, 2006.
13. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *Post-proceedings of GTTSE'09*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.
14. D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In *Proceedings of the First Workshop on Empirical Studies of Programmers*, pages 80–98. Ablex Publishing Corp., 1986.
15. T. Mens and A. H. Eden. On the Evolution Complexity of Design Patterns. In *Proceedings of SETra 2004*, volume 127 of *ENTCS*, pages 147–163, 2005.
16. P. Oman and J. Hagemeister. Construction and testing of polynomials predicting software maintainability. *J. Syst. Softw.*, 24:251–266, March 1994.
17. L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on Software Engineering*, 27(12):1134–1144, 2001.
18. P. Wadler. The expression problem. `http://www.daimi.au.dk/~madst/tool/papers/expression.txt` (accessed January 2011), November 1998.
19. K. F. Wender, F. Schmalhofer, and H.-D. Böcker, editors. *Cognition and computer programming*. Ablex Publishing Corp., 1995.