JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 5, No. 8, November-December 2006

Matching Objects Without Language Extension

Joost Visser, Departamento de Informática, Universidade do Minho, Portugal

Pattern matching is a powerful programming concept which has proven its merits in declarative programming. The absence of pattern-matching in object-oriented programming languages is felt especially when tackling source code processing problems. But existing proposals for pattern matching in such languages rely on language extension, which makes their adoption overly intrusive.

We propose an approach to support pattern matching in mainstream object-oriented languages without language extension. In this approach, a pattern is a first-class entity, which can be created, be passed as argument, and receive method invocations, just like any other object.

We demonstrate how our approach can be used in conjunction with existing parser generators to perform pattern matching on various kinds of abstract syntax representation. We elaborate our approach to include concrete syntax patterns, and mixing of patterns and visitors for the construction of sophisticated syntax tree traversals.

Keywords: Pattern matching, object-oriented programming, syntax, parsing, tree traversal, term rewriting, strategic programming.

1 INTRODUCTION

Pattern matching is a programming concept that plays a central role in declarative paradigms, such as term rewriting and functional programming. Given a *term* and a *pattern*, i.e. a term in which variables may occur, the pattern matching problem consists in finding a *substitution* for the variables in the pattern such that the pattern and the term become equal. For instance, given the following pattern and term:

$$\begin{array}{ll} \text{pattern} & F(x, G(x, y)) \\ \text{term} & F(H(A), G(H(A), B)) \end{array}$$

where x and y are variables, pattern matching yields the following substitutions:

$$x := H(A) \qquad y := B$$

Note that variable x occurs twice. When variables are allowed to occur more than once in a pattern, the pattern-matching problem is called *non-linear*.

Mainstream object-oriented programming languages, such as Java, C#, and C++, do not count pattern matching among their features. This makes these general purpose languages less suitable for the specific purpose of building source code proces-

Cite this article as follows: Joost Visser,: *Matching Objects Without Language Extension*, in Journal of Object Technology, vol. 5, no. 8, November-December 2006, pages 81–100, http://www.jot.fm/issues/issue_2006_11/article2

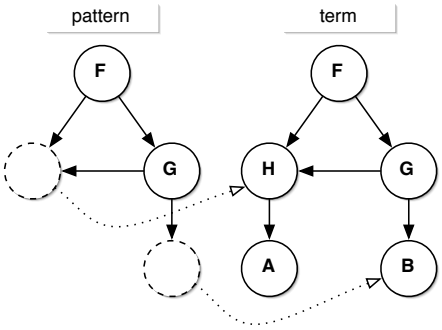


Figure 1: The basic idea of matching object graphs. The pattern is an object graph that contains variable objects, indicated by empty dashed circles. After matching, these variables are bound to corresponding objects in the term, indicated by the dotted arrows.

sors. Switching to a declarative language is often not desirable for compelling practical reasons, such as the availability of programmers, tools, and libraries. Several proposals for *extending* object-oriented languages with pattern matching features exist, but adoption of such language extensions is likewise intrusive, since they interfere with the use of existing tools, development environments, libraries, modeling methods, refactoring aids, etc.

As we will see below, pattern-matching can be realized within the boundaries of existing mainstream object-oriented programming languages. Rather than providing pattern matching as a language construct, we will model patterns as objects that implement a particular behavioral interface. Thus, in this approach, pattern matching is not a syntactic construct that the compiler transforms into a matching engine. Rather, patterns are first-class entities that are created, can be passed as arguments, and respond to method invocations, just like any other object.

In Section 2 we provide an abstract specification of object matching in the form of a pattern interface and associated implementation obligations. Subsequently, in Section 3, we provide several concrete implementations of that interface for various kinds of compound objects. In Section 4, we elaborate the implementations for objects that represent abstract syntax trees by showing how patterns can be created via concrete syntax expressions. In Section 5 we show how object matching can be combined with object graph traversal based on visitor combinators. Section 6 briefly relates lessons learned from commercial application of our approach. Section 7 discusses related work, and Section 8 concludes.

2 OBJECT MATCHING, ABSTRACTLY

The basic idea of matching object graphs is illustrated in Figure 1. A pattern is an object graph that contains designated *variable* objects (indicated with dashed circles). These variables have no name and are not syntactically distinguished; they

```
public interface Pattern {
  public void addVariable(Object variable);
  public boolean match(Object term, Map bindings);
  public Object substitute(Map bindings);
}
```

```
public abstract class AbstractPattern implements Pattern {
  protected List variables = new ArrayList();
  public void addVariable(Object variable) { variables.add(variable); }
  protected boolean matchVariable(Object p, Object t, Map bindings) {
    if (variables.contains(p)) {
      if (bindings.containsKey(p)) { return bindings.get(p).equals(t); }
      else { bindings.put(p,t); return true; }
    } else { return false; }
} }
```

Figure 2: The Pattern interface and an abstract class that implements its `addVariable` method. The reusable `matchVariable` method creates variable bindings. In case of non-linear patterns, the value of a previously bound variable is retrieved from the bindings for comparison.

are identified by reference, as are all objects. The pattern is matched against an object graph that does not contain variable objects. Matching involves a comparison between the structure of both object graphs as well as between the types and states of the objects that appear in them. During matching, bindings are created from variables to corresponding objects (dotted arrows in the figure). To prevent pollution of class hierarchies with artificial variable objects, and to enable reuse of the same pattern for several matches, the variables and their bindings can be modeled with external collections (lists and maps, respectively).

The pattern interface

Given this basic idea, we can give an abstract specification of object matching in terms of a Java class interface, called `Pattern` in Figure 2, and its implementation obligations. The interface consist of three methods, one of which can be implemented once and for all, as shown by the abstract class `AbstractPattern`. Concrete implementations of the Pattern interface should have a constructor that initializes a field referencing the root of the pattern object graph. After pattern creation, the `addVariable` method is used for pattern initialization. It can be implemented by adding the variables to a list of variables in the pattern's state, as demonstrated in `AbstractPattern`. Typically, several invocations of the `addVariable` method will follow the construction of the pattern.

The `match` method is intended to supply the actual pattern matching behavior. It takes the root of the term object graph, as well as a map into which to store the resulting variable bindings. The boolean result indicates whether the match was successful or not. This method should be implemented by simultaneously traversing the two object graphs. At each step, the pattern node should be checked for being

a variable, as exemplified by the method `matchVariable`. If the pattern node is a variable that has been bound before, its value should be compared for equivalence with the corresponding node in the term graph. If the pattern node is a free variable, a new binding to the corresponding term node is created. Traversal should not descend into variables. In case of pattern nodes that are not variables, the corresponding nodes in the term graph should be checked for type equivalence and state equivalence; when equivalent, traversal proceeds deeper; when different, the traversal can be broken off and `false` should be returned.

The `substitute` method takes a map holding variable bindings as argument, and substitutes variables in the pattern by their corresponding values. As a result, an object graph of the type of the root of the pattern is returned. If the map does not contain mappings for all variables in the pattern, the substitution is unsuccessful, and returns `null`.

Note that the method implementations in `AbstractPattern` implicitly rely on the behaviour of the `equals` methods of objects that act as variables and their values. For example, the `contains` and `containsKey` tests will result in equality tests on the elements of the variable list and the keys of the map of bindings. In addition, there is an explicit call to the `equals` method involving the value of a previously bound variable. In cases where it is not appropriate to rely on the `equals` methods of the objects in question, one may provide alternative implementations of the `addVariable` and `matchVariable` methods, or resort to wrapper objects with alternative implementations of `equals`.

Using the interface

Returning to the example of Figure 1, a typical usage scenario starts with creating a pattern, as follows:

```
H x = ... ; B y = ... ; \\ Any objects of type H and B can serve as variables.
Pattern p = new FPattern(new F(x,new G(x,y)));
p.addVariable(x); p.addVariable(y);
```

Here we assume that `FPattern` is a concrete implementation of the pattern interface for objects of type `F` (specific implementations follow below). Subsequently, the created pattern can be used to perform a match:

```
Map bindings = new HashMap() ;
H h = new H(new A());
F f = new F(h,new G(h,new B()));
boolean success = pattern.match(f,bindings);
```

The match being successful, the bindings will hold entries for `x` and `y`, and the value of `success` will be `true`. The bindings can be queried for the values of these variables, or, optionally, to perform a substitution on a second pattern:

```
public class ListPattern extends AbstractPattern {
  private List pattern;
  public ListPattern(List pattern) { this.pattern = pattern; }
  public boolean match(Object term, Map bindings){
    return (term instanceof List) && matchList((List) term,bindings);
  }
  private boolean matchList(List term, Map bs) {
    if (pattern.size()==term.size()) {
      Iterator patIter = pattern.iterator(); Iterator trmIter = term.iterator();
      boolean success = true;
      while (success && patIter.hasNext()) {
        Object p = patIter.next(); Object t = trmIter.next();
        success = matchVariable(p,t,bs) || p.equals(t);
      }
      return success;
    } else { return false; }
  }
  public Object substitute(Map bindings) {
    if (bindings.keySet().containsAll(variables)) {
      List result = new ArrayList(pattern.size());
      for (int i = 0; i < pattern.size(); i++) {
        if (variables.contains(pattern.get(i))) {
          result.add(bindings.get(pattern.get(i)));
        } else { result.add(pattern.get(i)); }
      }
      return result;
    } else { return null; }
} }
```

Figure 3: Pattern implementation for Lists. The worker method `matchList` performs simultaneous iteration over the pattern list and the term list. The `substitute` method creates a new list while iterating over the pattern list.

```
Pattern p2 = new GPattern(new G(y,x));
p2.addVariable(x); p2.addVariable(y);
G g = p2.substitute(bindings);
```

In this particular case, the resulting object assigned to `g` would be equal to `new G(new B(),new H(new A()))`.

3 IMPLEMENTATIONS

The abstract specification of matching behaviour given above can be implemented differently for different types of objects, or even differently for the same type. We will discuss implementations for specific collection types and abstract syntax tree types. Finally, we will show a reflection-based implementation that handles a variety of different types by dispatching to type-specific implementations.

Containers

Starting simple, we will explain how a pattern implementation can be provided for any ordered container, such as an array or a list. A code example, provided in Figure 3, shows the implementation for the `List` class. The state of `ListPattern` holds a root object of type `List`, which is initialized with the constructor. The `match` method first checks the type of the term object, and then calls a worker method `matchList`. If the two lists are of the same size, this worker performs simultaneous iteration over both. During iteration, the `matchVariable` method is called to deal with elements of the first list that are registered as variables. Other elements are compared with the `equals` method[1].

The `substitute` method creates a new list while iterating over the pattern list. At each step, either an element of the pattern list is added to the new list, or, in case of a variable element, the value element to which it is bound.

For other ordered containers, such as arrays, implementations are similar. Matching on containers without order is also imaginable, but would be more involved and computationally more demanding due to a larger search space of possible bindings.

Trees

Tree representations come in different flavors. For instance, the Antlr parser generator [14] comes with an abstract syntax tree interface, called `AST`, where tree nodes hold references to their first child and to their first sibling. Figure 4 shows this interface, together with a corresponding implementation of `Pattern`. To avoid repetition, we show only fragments of the worker method `matchAST`; the remainder of the code is similar to the code for `ListPattern`, modulo the type of the pattern root[2]. As can be gleaned from the code, the matching algorithm performs a simultaneous pre-order traversal of two ASTs. At each step, pattern nodes that are free variables give rise to new bindings. Pattern nodes that are previously bound variables, lead to equality checks. Pattern nodes that are constants lead to equality checks on the states of the nodes, and to further traversal into the respective first children and their siblings. The `substituteAST` method (not shown) likewise performs a pre-order traversal, where each encountered variable node is replaced by its value, and all other nodes are cloned.

For other tree representations where all nodes share a common node interface, similar implementations of the `Pattern` interface can be given. For instance, the JJTraveler visitor combinator framework [19, 4] offers a common tree interface, called `Visitable`, where subtrees are accessed by index. The `ATerm` interface of the Annotated Term Library [1] provides similar access to a maximally shared tree repre-

[1] At the end of this section, this equivalence check becomes a recursive invocation of our more generic, reflection-based implementation.

[2] For simplicity, we ignore that Antlr `AST`s do not necessarily implement *deep* equality (traversing into children) in their `equals` method. The actual implementation uses a wrapper class to fix that.

```
public interface AST {
  public int getType();            public String getText();
  public AST getFirstChild();      public void setFirstChild(AST ast);
  public AST getNextSibling();     public void setNextSibling(AST ast);
}
```

```
public class ASTPattern extends AbstractPattern {
  ...
  private boolean matchAST(AST p, AST t, Map bs) {
    if (matchVariable(p,t,bs) { return true; }
    else if (p.getType()==t.getType() && p.getText().equals(t.getText())) {
      AST pNxt = p.getFirstChild(); AST tNxt = t.getFirstChild();
      boolean success = true;
      while (success && pNxt!=null) {
        if (tNxt==null) { success = false; }
        else {
          success = match(pNxt,tNxt,bs);
          pNxt = pNxt.getNextSibling(); tNxt = tNxt.getNextSibling();
      } }
      return success;
    } else { return false; }
  }
  ...
}
```

Figure 4: Pattern implementation for ASTs. The pattern AST and term AST are traversed simultaneously.

sentation. The `Node` interface of the Document Object Model (DOM) [21] provides access to child nodes both by index, and by child and sibling references.

In cases where the tree (or graph) is represented with objects with disjunct node interfaces, one must provide equally many implementations of the pattern interface which call each other on pairs of sub-nodes with matching types. In the event that some references between nodes are not typed (for instance when they are held in an untyped collection like `List`), one can resort to reflection to select an appropriate pattern implementation when available. In the next section, we show how such reflection-based matching can be harnessed in a generic implementation of the pattern interface.

Generic Object Matching

Each of the implementations of the pattern interface discussed so far provides matching functionality for objects of a single type. In each case, the matching process starts with checking (using `instanceof`) whether the term object type is equal to the designated type of the pattern graph. The `GenericPattern` implementation in Figure 5 takes a different approach: the pattern graph can be of any type, and a choice is made between different matching algorithms, based on the types of both pattern and term. As can be seen in the figure, the choice is modeled with the logical

```
public class GenericPattern extends AbstractPattern {
  protected Object pattern;
  ...
  public boolean match(Object term, Map bindings) {
    return matchVariable(pattern,term,bindings)
        || matchList(pattern,term,bindings)
        || matchArray(pattern,term,bindings)
        || matchAST(pattern,term,bindings)
        || matchVisitable(pattern,term,bindings)
        || matchDomNode(pattern,term,bindings)
        || pattern.equals(term);
  }
  private boolean matchList(Object term, Map bs) {
    return (pattern instanceof List) && (term instanceof List)
       && matchList((List)pattern,(List)term,bs);
} }
```

Figure 5: Generic pattern matching. Various matching policies are tried in turn. Run-time type inspection and type casts guard the dispatch to appropriate methods.

or operator (*cf.* `||`) where each disjunct represents a different matching attempt.

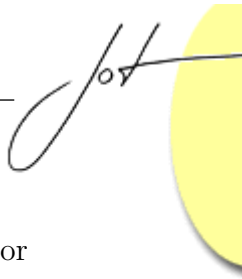
The first attempt takes care of the case where the pattern graph is a variable, using again the `matchVariable` method. Subsequently, a series of attempts is made using the match policies for various types of objects, exactly as discussed in the previous implementations except for an additional check on the type of the pattern type. For instance, the `matchList` method tests both pattern and term for being of type `List` before calling a list matching worker method. To ensure that this worker method descends into container elements, the equivalence check on elements (*cf.* Figure 3) is replaced by a recursive call to the generic matching algorithm. Finally, as default policy when all other matching attempts fail, an equality test is performed.

4 CONCRETE SYNTAX PATTERNS

The first-class status of patterns in our approach opens opportunities. For instance, pattern creation can be done not only by direct calls to constructor methods, but also by computation from concrete syntax expressions. We explain how this is done with two different parser generators: JJForester and Antlr.

Parsing patterns with **JJForester**

The input to JJForester are grammars in the purely declarative grammar formalism SDF [8]. From such a grammar, a Java class hierarchy is generated to represent parse trees: non-terminals become abstract classes, and production rules become their concrete subclasses. The benefit of such a many-typed representation of parse trees is that Java's type system can be leveraged to guard well-formedness of tree



construction and access with respect to the abstract syntax.

Each generated abstract class has a static parse method that allows files or strings to be parsed by invocation of an external parser using the corresponding non-terminal as start symbol. The parsing technology employed is Generalized LR parsing [15, 2]. All generated classes implement the aforementioned `Visitable` interface of the JJTraveler framework, which allows subtree access by index. We supplied an SDF grammar of the Java language to JJForester and obtained a class hierarchy of 104 abstract classes and 276 concrete classes.

Now, patterns can be created from concrete syntax expressions, for instance as follows:

```
Pattern p = new VisitablePattern(Statement.parse("x = e;"));
p.addVariable(Identifier.parse("x"));
p.addVariable(Expression.parse("e"));
```

Thus, invocations of three different parse methods result in the construction of the pattern itself and its two variables. This style of pattern creation from concrete syntax expressions is not only more convenient than constructor invocations, it also provides some isolation against grammar refactorings by abstracting over the underlying abstract syntax tree shapes.

Of course, these parser invocations are performed at run-time, rather than during compilation, leading to performance loss and potential run-time parse errors. These risks can be attenuated by taking care that each pattern is constructed only once, and by proper (unit) testing of all pattern construction code.

Note that the availability of concrete pattern matching reduces the utility of the generated class hierarchies. Without concrete patterns, the generated classes are used to guard well-formedness. With concrete syntax patterns, the parser performs this task, albeit at run-time only. As a consequence, one may opt to dispense with JJForester's conversion of generic ASTs to strongly typed ASTs, processing the former directly. We have followed this approach in a case study involving Transact-SQL, on which we comment in Section 6.

Parsing patterns with **Antlr**

When using Antlr, the situation is slightly more complex. This parser generator consumes Yacc-like grammars, containing semantic actions. Depending on the decision of the programmer, these semantic actions may build abstract syntax trees of the `AST` type. The structure of the syntax trees does not necessarily strictly follow the structure of the grammar. The tool generates a parser class that provides one method per non-terminal. These parse methods take no arguments; the string or file to be parsed is passed to the parser via its constructor method. We have supplied a C# grammar to Antlr leading to a parser class with 219 parse methods and AST building capability.

To abstract over details of parser creation, we have defined a class `CSharpParserFactory` with a method `parse` that takes as argument a string to be parsed and the name of a non-terminal to use as start symbol. This class allows us to conveniently create patterns from concrete syntax expressions. For instance:

```
CSharpParserFactory f = new CSharpParserFactory();
Pattern p = new ASTPattern(f.parse("statement","if(!b) s1; else s2;"));
p.addVariable(f.parse("expression","b"));
p.addVariable(f.parse("statement","s1;"));
p.addVariable(f.parse("statement","s2;"));
```

This pattern can subsequently be used to match abstract syntax trees of conditional statements with negative conditions.

Note that this approach works only with Antlr grammars that are in some sense well-behaved with respect to tree construction. If the grammar contains non-terminals that do not return trees, or that return collections of subtrees that are joined only at higher levels of the grammar, then such non-terminals can not appear as pattern roots or in variable positions.

5 MATCHING AND TRAVERSAL

To take further advantage of the first-class status of our patterns, this section investigates some possibilities of combining generic pattern matching with generic visitor combinators. Generic visitor combinators are small, reusable classes that capture basic functionality, and can be combined in different constellations to construct more complex behavior [19].

We will generalize our pattern matching approach by combining patterns with visitors in two complementary ways:

patterns in visitors What if we want to match a pattern, not at the root of an object graph, but at a deeper node? If we capture the pattern inside a visitor combinator, it can be combined with a visitor combinator that performs traversal to realize pattern matching *at arbitrary depths*.

visitors in patterns What if we want to apply a visitor to a subgraph that matches with a variable in our pattern? We can eliminate the intermediate steps of binding to the variable and retrieving its value to be visited, simply by associating visitors directly to the pattern's variables. Since visitors encapsulate behavior, this makes our patterns *active*.

Before explaining these two generalizations in detail, we provide a brief exposition of the concept of visitor combinators. For a more elaborate account, the reader should look elsewhere [19, 8, 4, 20].

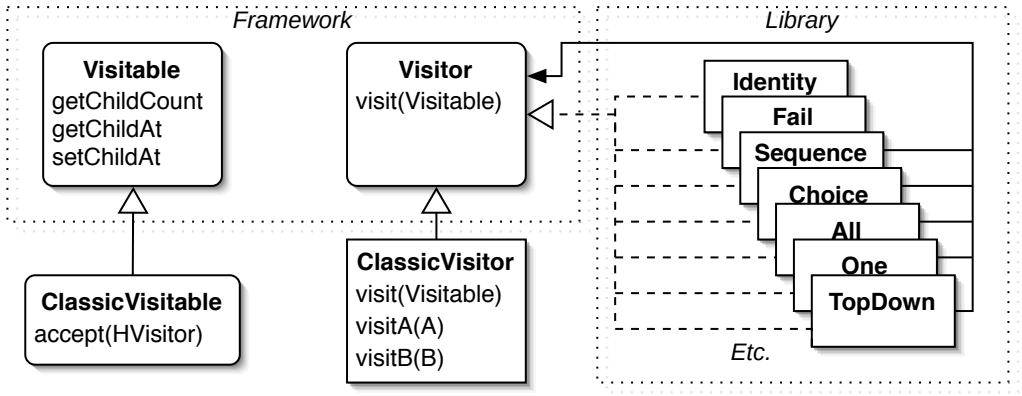


Figure 6: Overview of JJTraveler, which offers a simple framework and a library of reusable visitor combinators.

Generic visitor combinators

Visitor combinator programming was introduced in [19], and can be understood as an improvement over the classic visitor design pattern [5] where three well-known short-comings are removed. Classic visitors resist combination, allow little traversal control, and are tied to a given class hierarchy. In contrast, visitor combinators capture pieces of functionality that can easily be combined, they allow the construction of sophisticated traversal strategies, and they are reusable across class hierarchies.

Visitor combinator programming in Java is supported by JJTraveler, which offers a simple framework and a library of reusable visitor combinators, as depicted schematically in Figure 6.

The JJTraveler framework offers two class-hierarchy independent interfaces, `Visitor` and `Visitable`. The latter, as mentioned before, provides a minimal interface for nodes that can be visited. The visitor interface provides a single `visit` method that takes any visitable node as argument. Each visit can succeed or fail, which can be used to control traversal behavior. Failure is indicated by a `VisitFailure` exception[3].

The library of JJTraveler contains a number of predefined visitor combinators. These rely only on the generic visitor and visitable interfaces, not on any specific underlying class hierarchy. An excerpt from the library is shown in Table 1. There are basic combinators for sequencing, biased choice, and one-step traversal. From these, defined combinators can be composed. An example of a recursively defined visitor is the traversal strategy combinator *TopDown(v)*, specified as follows:

$$TopDown(v) = Sequence(v,\ All(TopDown(v)))$$

Thus, *TopDown* first applies v to the current node, and then recursively applies

[3]A variation of this design uses boolean return values to indicate success and failure, at the expense of supporting node replacement via `Visitable` return values.

Name	Description
Identity	Do nothing.
Fail	Raise *VisitFailure* exception.
$Not(v)$	Fail if v succeeds, and *vice versa*.
$Sequence(v_1, v_2)$	Do v_1, then v_2.
$Choice(v_1, v_2)$	Do v_1, if it fails, do v_2.
$All(v)$	Apply v sequentially to all immediate children until it fails.
$One(v)$	Apply v sequentially to all immediate children until it succeeds.
$IfThenElse(c, t, f)$	If c succeeds, do t, otherwise do f
$Try(v)$	*Choice(v,Identity)*
$TopDown(v)$	*Sequence(v,All(TopDown(v)))*
$TopDownWhile(v)$	*Try(Sequence(v,All(TopDownWhile(v))))*
$Innermost(v)$	*Sequence(All(Innermost(v)),Try(Sequence(v,Innermost(v))))*

Table 1: JJTraveler's library (excerpt). Basic combinators are shown in the upper part. Defined combinators appear in the lower part together with their specifications in terms of the basic ones.

the top down strategy to each of the children of the current node, yielding a pre-order traversal of a tree. Similar specifications are provided in Figure 1 for other defined combinators. In fact, visitor combinators can be used to program all kinds of sophisticated generic traversal strategies. Details about the actual encodings of such defined combinators can be found elsewhere [19, 4].

To use JJTraveler on the class hierarchy of an application, one needs to instantiate the framework appropriately. This is a simple job for tree representations with a single node interface, such as those of Antlr, DOM, or the ATerm Library (*cf.* Section 3). For more elaborate hierarchies, generative support may be used. For example, JJForester generates classes that implement JJTraveler's visitor and visitable interfaces. Whether manually coded or generated, an important element in instantiations of the JJTraveler framework is a classical hierarchy-specific visitor which additionally implements the generic visitor interface by dispatching when possible to classical hierarchy-specific visit methods (*cf.* `ClassicVisitor` in Figure 6). By virtue of this additional *implements* relation, hierarchy-specific visitors can be passed as arguments to generic visitor combinators, and their hierarchy-specific behaviour will be triggered whenever appropriate.

Patterns inside visitors

The encapsulation of a pattern inside a visitor is shown in Figure 7. The class `Match` holds a reference to a pattern of type `VisitablePattern` that is initialized with its constructor. It implements the `Visitor` interface of JJTraveler. When visiting a visitable term, it tries to match it with the pattern. The success or failure of the visitor is determined by the success or failure of the match.

An example of using the `Match` combinator is to find all targets of `null` assign-

```
public class Match implements Visitor {
  private VisitablePattern pattern;
  private List maps = new ArrayList();
  public List getMaps() { return maps; }
  public Match(Visitable visitable) {
    this.pattern = new VisitablePattern(visitable);
  }
  public addVariable(Object variable) { pattern.addVariable(variable); }
  public Visitable visit(Visitable term) throws VisitFailure {
    Map bindings = new HashMap();
    if (pattern.match(term,bindings)) {
      maps.add(bindings);
      return term;
    } else {
      throw new VisitFailure("No match");
} } }
```

Figure 7: Patterns in visitors. The visitor combinator `Match` implements the `visit` method by matching the pattern to which it holds a reference against the visitable that it visits. Each successful visit leads to a new entry in a list of maps.

ments inside a Java file:

```
Match m = new Match(Statement.parse("x = null;"));
m.addVariable(Identifier.parse("x"));
Visitor v = new TopDown(new Try(m));
Visitable cu = CompilationUnit.parse(file);
v.visit(cu);
```

After the traversal, the `getMaps` method applied to `m` will return a list of maps that each represents a singleton set of bindings. Each singleton contains a binding of an `Identifier` to the pattern's variable `x`.

Patterns can also be used inside visitors to implement *rewriting*. Consider the following rewrite rule:

$$\begin{array}{lcl} \texttt{if not}(x) & & \texttt{if } x \\ \texttt{then } t & \rightarrow & \texttt{then } e \\ \texttt{else } e & & \texttt{else } t \end{array}$$

This law of programming [6] states that an `if` statement with a negative condition can be replaced by an `if` statement with a positive condition with swapped branches.

This rewrite rule can be implemented with the visitor combinator defined in Figure 8. The `Rewrite` visitor holds two patterns that represent the left-hand and right-hand sides of a rewrite rule. It implements the `visit` method by attempting to match the left-hand side pattern against the term being visited. If this match succeeds, the resulting bindings are substituted into the right-hand side pattern to obtain the rewrite result. Match failure makes the visitor fail.

Negative condition elimination can now be implemented as follows:

```
public class Rewrite implements Visitor {
  private VisitablePattern lhs, rhs;
  public Rewrite(Visitable lhs, Visitable rhs) {
    this.lhs = new VisitablePattern(lhs); this.rhs = new VisitablePattern(rhs);
  }
  public addVariable(Object variable) {
    lhs.addVariable(variable); rhs.addVariable(variable);
  }
  public Visitable visit(Visitable term) throws VisitFailure {
    Map bs = new HashMap();
    if (lhs.match(term,bs)) { return (Visitable) rhs.substitute(bs); }
    else { throw new VisitFailure("No rewrite"); }
} }
```

Figure 8: Rewriting. The visitor `Rewrite` combines two patterns into a rewrite rule. When the first pattern matches on a visitable, the resulting bindings are used to create a new visitable from the second pattern.

```
Rewrite r = new Rewrite(
  Expression.parse("if (!c) then s1; else s2;"),
  Expression.parse("if (c)  then s2; else s1;"));
r.addVariable(Expression.parse("c"));
r.addVariable(Statement.parse("s1;"));
r.addVariable(Statement.parse("s2;"));
Visitor v = new Innermost(r);
CompilatonUnit cu = CompilationUnit.parse(file);
v.visit(cu);
```

In this case, we selected the `Innermost` combinator, which captures the leftmost innermost rewriting strategy. This combinator is part of the JJTraveler library.

Visitors inside patterns

Active patterns, or patterns with visitors inside, can be implemented by refining a concrete pattern class, as shown in Figure 9. The class `ActiveATermPattern` extends the pattern interface with a `putVisitor` method for associating visitors with pattern graph nodes. These associations are stored in a map. The method `visit` applies a visitor associated to a pattern node, if such an association exists, to a corresponding term node. Success of the visit determines success of the match. `ActiveATermPattern` overrides the `match` method to attempt such a visitor-based match before the regular match attempt defined in `ATermPattern` (*cf.* `super`).

An example of using active patterns is to find `if` statements that contain `null` tests in their condition. First we create a visitor that searches for such tests, using the `Match` visitor:

```
VisitablePattern vp = Expression.parse("x==null;");
vp.addVariable(Expression.parse("x"));
Match match = new Match(vp);
```

```
public class ActiveATermPattern extends ATermPattern {
  private Map visitors = new HashMap();
  ...
  public void putVisitor(ATerm variable, Visitor visitor) {
    addVariable(variable);
    visitors.put(variable,visitor);
  }
  protected boolean matchATerm(ATerm pattern, ATerm term, Map bindings) {
    return visit(pattern,term,bindings)
        || super.matchATerm(pattern,term,bindings);
  }
  private boolean visit(ATerm pattern, ATerm term, Map bindings) {
    if (visitors.containsKey(pattern)) {
      try {
        ((Visitor) visitors.get(root)).visit(term);
        bindings.put(pattern,term);
        return true;
      } catch (VisitFailure e) { return false; }
    } else { return false; }
} }
```

Figure 9: Visitors in patterns. The specialization `ActiveATermPattern` of `ATermPattern` extends the pattern interface with `putVisitor` for associating visitors with pattern graph nodes. Also it extends the `matchATerm` method with an additional match attempt, which calls `visit`.

Then we embed this visitor in an active pattern, this one for `Visitable`s:

```
ActiveVisitablePattern ap
  = new ActiveVisitablePattern(Expression.parse("if (c) then s1; else s2;"));
ap.putVisitor(Expression.parse("c"),match);
ap.addVariable(Statement.parse("s1;"));
ap.addVariable(Statement.parse("s2;"));
```

Finally, we embed the active pattern in another `Match` visitor, and apply it to a file via a traversal combinator:

```
Visitor v = new TopDown(new Try(new Match(ap)));
Visitable cu = CompilationUnit.parse(file);
v.visit(cu);
```

After traversal, the bindings stored in the state of `match` hold expressions tested for nullity in conditions of `if` statements. As we will explain in the related work section, this style of mixing traversal strategies with first-class matching patterns was inspired by the capabilities of the strategic term rewriting language Stratego.

6 APPLICATION AND ITS LESSONS

Our techniques have been applied in the construction of source code analysis components for the Software Analysis Toolkit (SAT) of the Software Improvement

Group [16]. The SAT is used for software risk assessment [3] and software portfolio monitoring [9]. The SAT has an open architecture including frameworks for source code analysis and for storing, processing, and visualizing analysis results. The main implementation language is Java. Both Antlr and JJForester are used in its source code analysis components.

Our pattern-matching techniques have been applied for processing Java, PL/SQL, Transact-SQL, and C#. JJForester was used for the first three languages and Antlr for the last. In case of Java, generated AST classes were employed while generic ASTs were used for the SQL dialects and for C#. For each language, patterns have been developed for the construction of several metrics collectors and dependency extractors. In all cases, concrete patterns were employed in combination with implementations of the `Pattern` interface for common node interfaces, such as `antlr.AST`, `aterm.ATerm`, and `jjtraveler.Visitable`.

Based on lessons learned from using our techniques in a commercial software construction context, some preliminary practical guidelines can be formulated:

- To prevent repeated parsing of concrete patterns and their variables, caching techniques such as the singleton design pattern can be employed.
- At least one unit test should be developed for each concrete syntax pattern, to check the syntactic well-formedness of the pattern.
- Concrete syntax patterns can be decoupled from particular grammars to make them reusable across dialects or families of languages. For examples, in spite of many deviations, the PL/SQL and Transact-SQL dialects are sufficiently similar to allow many patterns to be developed once and used for both.

The application experiences have also led to the development of additional library functionality, for instance for combining multiple patterns, for more convenient variable insertion, and for recurring test tactics.

7 RELATED WORK

Java language extensions The Pizza language [13] is an extension of the Java language with three concepts taken from functional programming: higher-order functions, parametric polymorphism (type parameters), and algebraic data types with pattern-matching. JMatch [10] is also an extension of the Java language with pattern-matching. In this case, matching is not based on algebraic datatypes, but through a generalization of constructor methods, called *pattern constructors*. Tom [11] also provides support for algebraic data types with pattern-matching. Tom is not a full fledged language extension, but rather a pre-compiler.

In comparison to our approach, which stays completely within Java, language extensions provide several advantages. They allow optimizing compilation of matching expressions. They allow static checking beyond the capabilities of the Java type

system. They may provide more concise syntax. On the flip side, the introduction of a new language is more intrusive on the existing programming practices of potential users. Also, modeling pattern matching within Java provides more flexibility, allowing, as we have demonstrated, to introduce variations or to combine the approach with other techniques.

Scala The novel Scala programming language [12] provides a fusion of functional and object-oriented programming and counts pattern matching among its features. This is realized by allowing the user to tag classes with a `case` modifier, with two effects regarding its primary[4] constructor method. Firstly, the constructor can be called without using the `new` keyword. Secondly, the constructor can be used in the pattern positions of case branches in switch-like pattern-match expressions.

Though Scala programs resemble Java programs and are interoperable with them, Scala is a distinct language with distinct semantics, documentation, libraries, tool support, and user community. Our pattern-matching approach is viable within the confines of any main-stream object-oriented language such as Java, C#, or C++.

Nevertheless, a comparison between Scala and our approach is interesting. Patterns in Scala are syntactically nicer than ours, due to the possibility of dropping the `new` keyword. Scala's patterns can be nested, just as ours, but non-linear matches are not supported. Also, there is no notion of active patterns. Scala's patterns are not first class, but the blocks of case branches in which they occur are anonymous functions, which are first class.

Strategic pattern matching The Stratego language [18, 17] is a term rewriting language that does not offer a single fixed rewriting strategy, but allows the programmer to compose different rewriting strategies from basic strategy combinators. The notion of visitor combinator can be seen as an object-oriented counterpart of Stratego's strategies. Stratego's pattern matching features include *congruences*, which are term patterns that appear in strategy positions, and that can contain strategies in subterm positions. The semantics of a congruence is to match against an incoming term, and apply the strategies in subterm positions to the subterms of the incoming term. Thus, congruences allow mixing of traversal and pattern matching, in a similar vein as our `Match` and `ActivePattern` combinators.

8 CONCLUDING REMARKS

We have demonstrated that sophisticated pattern matching support can be obtained within mainstream object-oriented languages without resorting to a language extension. Fundamental to the approach is the observation that object graphs themselves

[4] In Scala, a class definition can have parameters, which at once define instance variables and a so-called *primary* constructor with those parameters, that initializes these instance variables.

can serve as patterns and their variables. We have provided an abstract specification of object graph matching in terms of a behavioral interface and its implementation obligations. We have shown that these obligations can be met for specific types of objects as well as in a generic, but customizable manner. Taking advantage of the first-class status of patterns, we have elaborated support for concrete syntax patterns, match and rewrite visitors, and active patterns.

Since we do not rely on extending an existing language, the adoption of our approach does not interfere with the use of existing tools, development environments, libraries, modeling methods, refactoring aids, etc.

Still, our pattern matching approach is not intended to disavow the merits of language extensions. In fact, a number of caveats of our approach are worth pointing out. For example, since pattern matches are not compiled, there is no opportunity for static optimizations. Also, the concrete syntax expressions used to create patterns are only syntactically checked at run-time and require testing to dispel errors.

A final trade-off between using a language extension or the programmatic matching technique presented in this paper can only be made in the context of particular development projects and processes.

Future work

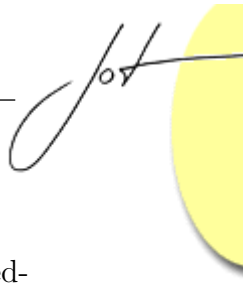
The work presented in this paper can be generalized in several ways. We list some.

Unification When the pattern is not matched against a closed term, but against another pattern, a more general problem, called *unification*, must be solved. Unification plays a central role in logic programming, type inference in functional programming, and in some artificial intelligence approaches. A comprehensive survey of unification can be found in [7]. We would like to investigate a generalization of object matching to object unification.

Further host languages We implemented our ideas in Java, but other host languages are equally viable. A C# implementation is high on the wish list.

Availability

The code presented in this paper is part of a library distribution, named MatchO, which is available from the author's web pages.



Acknowledgments

Thanks to Rob van der Leek of the Software Improvement Group for valuable feedback regarding this paper and the MatchO library. The author is recipient of a research grant from the Fundação para a Ciência e a Tecnologia, under grant number SFRH/BPD/11609/2002.

REFERENCES

[1] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[2] M.G.J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *LNCS*, pages 143–158. Springer, 2002.

[3] A. van Deursen and T. Kuipers. Source-based software risk assessment. In *Proceedings 21 International Conference on Software Maintenance (ICSM'03)*, pages 385–388. IEEE Computer Society, 2003.

[4] A. van Deursen and J. Visser. Source model analysis using the JJTraveler visitor combinator framework. *Software: Practice and Experience*, 34(14):1345–1379, 2004.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] C.A.R. Hoare. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[7] K. Knight. Unification: a multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, 1989.

[8] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. *Science of Computer Programming*, 47(1):59–87, November 2002.

[9] T. Kuipers and J. Visser. A tool-based methodology for software portfolio monitoring. In Mario Piattini and Manuel Serrano, editors, *Proceedings of the 1st International Workshop on Software Audits and Metrics*, pages 118–128. INSTICC Press, 2004.

[10] J. Liu and A.C. Myers. JMatch: Iterable abstract pattern matching for Java. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, January 2003.

[11] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In G. Hedin, editor, *Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 61–76. Springer, 2003.

[12] M. Odersky et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, 2004.

[13] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 146–159. ACM Press, 1997.

[14] T. Parr. The Antlr web page. www.antlr.org.

[15] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[16] Software Improvement Group home page. www.sig.nl.

[17] E. Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *LNCS*, pages 30–44. Springer, 1999.

[18] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, 1999. Proceedings of the International Conference on Functional Programming, ICFP'98.

[19] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, 2001. Proceedings of the 2001 ACM Conference on Object-Oriented Systems, Languages and Applications, OOPSLA 2001.

[20] J. Visser. *Generic Traversal over Typed Source Code Representations*. PhD thesis, University of Amsterdam, 2003.

[21] World Wide Web Consortium. Document object model. www.w3.org/DOM/.

ABOUT THE AUTHORS

Joost Visser Currently, Joost Visser is Post-Doctoral fellow at the Departamento de Informática of the Universidade do Minho in Braga, Portugal. Before, he was a researcher at the Centre for Mathematics and Computer Science (CWI) in Amsterdam, The Netherlands, and he worked as senior consultant at the Software Improvement Group (SIG) in Amsterdam, The Netherlands. He can be reached at joost.visser@di.uminho.pt. See also http://www.di.uminho.pt/~joost.visser.