

# Implementing Design Patterns as Declarative Code Generators

Kris De Volder

`kdvolder@cs.ubc.ca`

The Department of Computer Science  
UNIVERSITY OF BRITISH COLUMBIA  
309-2366 Main Mall  
Vancouver, B.C.  
V6T 1Z4

**Abstract.** The implementation of a design pattern typically cannot be reused. Consequently implementing design patterns is a tedious, repetitive and error-prone activity with ample opportunities to make the same mistakes over and over again.

We address this issue by automating design-pattern implementation by means of code generators which are defined as declarative logic meta programs generating Java code and are therefore called *declarative code generators*.

While a reusable code-generator is typically somewhat harder to implement than a single instance of a design pattern, it is worth the effort because: 1) it avoids tedious and error prone copy-paste-edit implementation. 2) the extra effort is not prohibitive because declarative code generation allows the code-generators to be implemented relatively easily. 3) declarative code generators combine naturally and thus allow for generation of combinations of design patterns.

## 1 Introduction

### 1.1 Design Pattern Implementation

Design patterns [GHJV95] are ‘solutions to recurring design problems’ which are used over and over again. One of the reasons why they are called design ‘patterns’ is that they are not simple ‘atomic’ or ‘black-box’ abstractions which can be neatly packaged and simply reused. Rather, they are common ways of doing things and as such do have common structure, but despite of this, are typically reimplemented over and over again on a case by case basis. This seems to be an inherent property of design patterns for if it were possible to implement a design pattern simply as a reusable class or a procedure, it would not be called a design pattern.

As a result, design pattern implementations can be tedious, repetitive, complex and error prone. We address this issue, automating the tedious implementation process of complex, structure-rich design patterns by implementing a declarative code generator. Contrary to manually implemented design patterns, a code

generator can be made more reusable because it can be parameterized and driven by context dependent information which is separately provided.

### 1.2 Why Declarative Logic Programming?

Fundamentally, the core idea of this paper, using code-generators to automate and reuse design pattern-implementation, does not imply an obligation to use any particular programming language or system. Any language can be used, as long as it allows somehow manipulating representations of code in the target language of interest. However, obviously some programming languages or systems will be more suitable than others. We propose using a particular style of logic meta programming which we call *declarative code generators*. We believe this to be a good choice because as will be shown in the paper, fairly sophisticated code generators can be implemented with a relatively small effort and also because declarative code generators can be easily combined which facilitates the automatic combination of design pattern implementations.

## 2 Logic Meta Programming and Declarative Code Generators

Before we can go on about declarative code generators, we first have to explain a little bit more about the conceptual basis of logic meta programming in general. We will also talk a little bit about the experimental system, TyRuBa, that we use in this paper. TyRuBa is a logic meta programming system, supporting declarative code generators for Java. Consequently we will make a lot of references to Java. However, all underlying ideas are general in nature and can easily be applied to any other ‘target’ programming language.

### 2.1 A Representational Mapping

A meta-programming system, like any other software system, needs a way to represent the objects it operates on and reasons about. Since we want a system for generating Java code, we will need an explicit representation of programs in the target language, Java.

A defining concept in a logic meta-programming system is the notion of a *representational mapping*. This mapping defines how to represent Java programs in terms of logic facts. Evidently there are many ways to define such a representation and some will be better than others, depending on what you want to do. Rather than expound in great detail about this, we will simply explain the specific mapping which is used in this paper for representing Java programs. Consider the Java program in figure 1. A representation of this program is shown in figure 2. The representation is simply a set of facts, describing in detail, all important properties of the syntactic structure of the program.

A few technical details about the logic language should be clarified now. Note that we are not using standard Prolog, but a similar yet slightly different

```

class PrintVisitor
extends GlyphVisitor
{
    java.io.PrintStream output = System.out;

    PrintVisitor(java.io.PrintStream output) {
        this.output=output;
    }
    void visitCharNode(CharGlyph visited) {
        output.print(visited.c);
    }

    ...
}

```

**Fig. 1.** An example Java-code fragment: the PrintVisitor class

```

class(PrintVisitor).
extends(PrintVisitor,GlyphVisitor).

var(PrintVisitor, java.io.PrintStream, output, {System.out}).

constructor(PrintVisitor, [java.io.PrintStream], [output],
             {this.output=output;}).

method(PrintVisitor, void, visitCharNode, [CharGlyph], [visited],
        {output.print(visited.c);}).

...

```

**Fig. 2.** Facts representing the Java code from figure 1

programming language. This language is called TyRuBa [DV98,DV99] and has a few modifications to ease Java-code manipulation.

One difference with Prolog is that TyRuBa adopts different lexical conventions for denoting constants (and variables). So, in fact, all of the identifiers in the example here, including those starting with capitals such as **Array** and **Enumerable**, are denoting constants. The reason for adopting these non-standard lexical conventions is that we want to have a direct correspondence between logic constants and Java identifiers, which happens to be very convenient if what we want is to write programs manipulating and reasoning about Java code.

Another specialized TyRuBa feature used in this example is rudimentary support for representing Java code as terms. Notice in the example some pieces of Java code surrounded by curly braces. These are special TyRuBa terms, which can hold a piece of Java code. The easiest way to think about these is that they are just a fancy kind of strings.

Now returning to the example. Most facts in the representation speak for themselves. The most complex of facts are those representing method declarations. Their structure is as follows:

```
method(<ClassName>, <ReturnType>, <MethodName>,
      <ArgumentTypeList>, <ArgumentNamesList>,
      <MethodBodyQuotedJavaCode>).
```

The representation of a Java program is nothing but a description of its parse tree using facts. Each fact says something about the tree. All the facts together describe the program as a whole.

## 2.2 Logic Meta Programming: A Repository of Java Code

Once we have defined a representation we can parse Java programs, transform them into facts and store the facts in a logic repository. A query engine is the most intuitive interface for this repository. It allows us to formulate all kinds of interesting queries about the stored Java program. For example to find all implementations of **visit** methods we can formulate a query:

```
:- method(?WhichClass, ?ReturnType, visit, ?ArgTypes, ?ArgNames, ?Body).
```

Note that variables are starting with a ‘?’ in TyRuBa syntax.

More complicated queries can be formulated and we can use real logic programs to deduce interesting information. For a simple example, we can find all subclasses of **TreeNode** by first writing the following simple program:

```
/* Define a program with to compute subclasses */
subclass(?sub, ?sub) :- class(?sub).
subclass(?sub, ?super) :- extends(?sub, ?mid), subclass(?mid, ?super).
```

Then we can use the newly defined **subclass** predicate and launch the query:

```
:- subclass(?x,TreeNode).
```

To make sophisticated queries more easy to write, we provide a library of standard rules defining predicates such as `subclass` and many others.

Clearly a queryable repository is much more interesting than a set of Java source files. Even if we need explicit Java sources, for example to compile and run them, we can still extract them from the repository. We can build an ‘unparser’ which queries the repository and (re)assembles the sources. In a more integrated setup the compiler (and browsers and editors) can directly query the repository and there is no more need for explicit source files.

### 2.3 Code Generation: A ‘Computed Repository’

The notion of a source repository is also a central concept in declarative code generators. More specifically, the notion of a *computed source repository* is the key idea to declarative code generators. The main idea is that facts can be computed and need not be explicitly stored. The unparser program does not care how or if the facts are stored. It only needs a query interface to get at them. Since a logic program has a query interface and is nothing more than a declarative program to compute facts we can think of a logic program as a *computed repository*. A logic program consists out of some basic facts together with some rules. The rules can compute new facts from known facts. Together all of these facts, computed and known, form *the computed repository*.

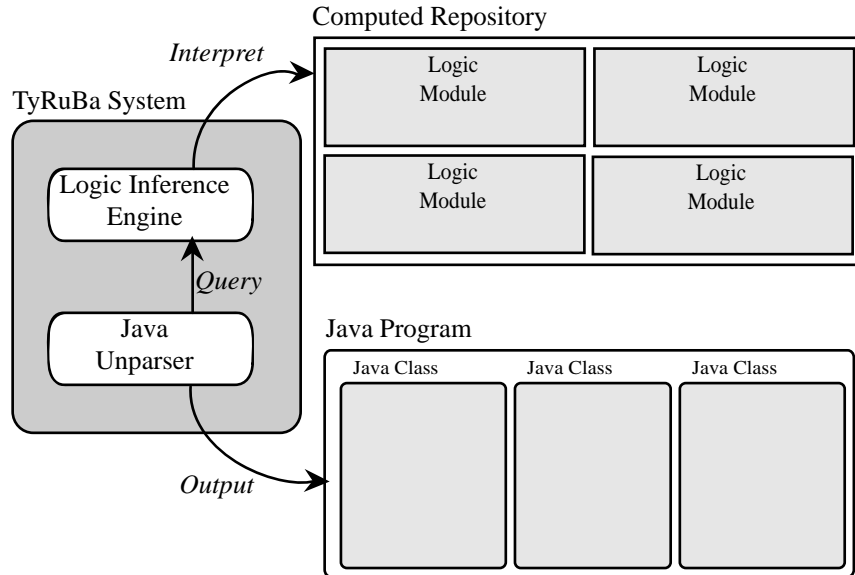
The general layout of the TyRuBa system, which exploits this idea, is depicted in figure 3. This is what it amounts to: Code generation happens as a combination of two things. An unparser queries a computed repository of facts. These facts are actually computed by a logic program. The unparser interprets these facts as representing Java code and unparses them as such. In this setting, the unparser is the one actually generating code, but the logic program is actually in control of what the code will be.

The style of declarative code generation supported by TyRuBa can be somewhat hard to grasp at first. Especially when we are used to thinking in more imperative terms. In declarative code generators we do not as much think in terms of ‘generating’ code anymore. We rather think of ‘declaring code’. The denomination “declarative code generators” is a very accurate one in this sense. A declarative code generator is about declaring classes, declaring variables, declaring methods, declaring interfaces and declaring specialization relationships, etc. Once something is declared it is as good as generated because the unparser takes care of the rest.

For example a declarative code generator for ‘generating’ a class called `AbstractTreeNode` simply looks like this:

```
class(AbstractTreeNode).
```

Of course, since this declaration provides no information about the class except that it exists, the generated class declaration will be empty. To fill the class with variables and methods other declarations may be provided elsewhere.



**Fig. 3.** How the TyRuBa system works

To give a more interesting example, rules can also be used to declare things. In fact, a single rule can simultaneously declare many things (e.g. methods) in many different classes at once. For example:

```
method(?Visited,void,accept,[AbstractTreeVisitor],[v],
      {v.visit<?Visited>(this);})
:- subclass(?Visited,AbstractTreeNode),
   NOT(abstract(?Visited)).
```

This is a rule which might be used to help implement a visitor pattern. The above rule should be read as an inverse implication “conclusion is implied by condition” (**conclusion :- condition**). The conclusion of this rule is a method declaration so it can be thought of as a conditional method declaration. The condition of the rule specifies under what *condition* the method should be declared. In plain English it roughly says: “declare an **accept** method in a class **?Visited** when **?Visited** is a non-abstract subclass of **AbstractTreeNode**”. Note that **?Visited** starts with a ‘?’ which means it is a logic variable. Therefore it should be interpreted as ranging over all **?Visited** which meet the condition of the rule. Consequently several **accept** methods are declared in different visited classes. The bodies of the declared **accept** methods are slightly different in each case as they are parameterized by **?Visited**. The generated code will look somewhat like below:

```
class LeafNode extends AbstractTreeNode {
```

```

    ...

    void accept(AbstractTreeVisitor v) {
        v.visitLeafNode(this);
    }

    ...
}

class InternalNode extends AbstractTreeNode {
    ...

    void accept(AbstractTreeVisitor v) {
        v.visitInternalNode(this);
    }

    ...
}

```

A technical note: TyRuBa's Java-code terms (in curly braces), can contain logic variables and also 'name patterns' like `visit<?Visited>`. Both of these are useful to express parametric code templates. This parametrization mechanism is simply implemented as string concatenation. In this example, we used the mechanism to vary the name of the `visit` method.

### 3 Running Example: Implementing Glyphs

We will introduce our running example: an implementation of 'glyph' classes for representing text organized in rows and columns.

This example was modeled closely after the introductory case-study example in the design pattern book [GHJV95]. To ease our experiments and the discussion in this paper, we simplified it somewhat, by assuming a pure ascii editor. This reduces the complexity but retains most interesting aspects of the original example. We want to be able to compose and manipulate text into arbitrarily deeply nested rows and columns. We want to perform several different operations over the text, for example printing it on a file, formatting the rows and columns on a two dimensional display (which is essentially the reason for representing them as glyphs), counting characters, counting words, etc.

All of the classes which represent text will be subclasses of the abstract class `Glyph`. The basic implementation of the classes is shown below:

```

abstract class Glyph {
}

class CharGlyph extends Glyph {
    char content;

    CharGlyph(char c) {

```

```

        content = c;
    }
}

class StringGlyph extends Glyph {
    String content;

    StringGlyph(String c) {
        content =c;
    }
}

class BinaryGlyph extends Glyph {
    Glyph left;
    Glyph right;

    BinaryGlyph(Glyph aLeft,Glyph aRight) {
        left = aLeft; right = aRight;
    }
}

```

For now we keep the hierarchy simple. Basically, the text is represented as a kind of binary tree. The leaves contain the data and internal nodes represent composition. We have two kinds of textual data: an individual character in a `CharGlyph` and a string in `StringGlyph`. A binary glyph represents the concatenation of two glyphs.

This way of composing glyphs deviates from the example in the book and admittedly is somewhat awkward. It is for expository purposes only. Later in the paper, `BinaryGlyph` will become obsolete when we introduce more interesting ways of composing glyphs into rows and columns, using the composite design pattern.

## 4 Visitor Design Pattern, Manual Implementation

Our next step is to implement some operations on glyphs. We want to make it easy to implement new operations, so we will use the visitor design pattern. In this section we simply present a ‘manual’ implementation. In the next section, we will implement a declarative code generator to do the work for us.

The idea behind the visitor pattern is to encapsulate operations over the data-structure into a so called visitor object. All the elements of the data-structure must then implement an `accept` method which allows the visitor to be accepted and do its work. A tedious part of visitor implementation is that before we can actually implement concrete visitors, we must first implement some supporting infrastructure: an abstract `GlyphVisitor` class and `accept` methods in the visited classes.

The implementation of this supporting infrastructure is outlined below. Note that we need to make a few changes to the glyph classes, namely adding declarations for the `accept` methods.

```

abstract class GlyphVisitor {
    abstract void visitBinaryGlyph(BinaryGlyph visited);
    abstract void visitCharGlyph(CharGlyph visited);
    abstract void visitStringGlyph(StringGlyph visited);
}

abstract class Glyph {
    abstract void accept(GlyphVisitor v);
}

class CharGlyph extends Glyph {
    ...

    void accept(GlyphVisitor v) {
        v.visitCharGlyph(this);
    }
}

class StringGlyph extends Glyph {
    ...

    void accept(GlyphVisitor v) {
        v.visitStringGlyph(this);
    }
}

class BinaryGlyph extends Glyph {
    ...

    void accept(GlyphVisitor v) {
        v.visitBinaryGlyph(this);
    }
}

```

We adopt a naming convention and chose to name each of the visit methods according to the the pattern `visit<?VisitedClass>`. Strictly speaking this is not necessary as we can simply name it `visit` and rely on Java's overloading to distinguish based on the argument's static type (this would work because the static type of `this` is different in every accept method). Rather than relying on this obscure 'Java hack' we chose to make the distinction more obvious by making it explicit in the invoked method's name.

After implementing the infrastructure we can implement concrete visitors. For example a concrete `PrintVisitor` which traverses a `Glyph` and prints out all the characters and strings in its leafs.

```

class PrintVisitor extends GlyphVisitor
{
    java.io.PrintStream output;
}

```

```

PrintVisitor(java.io.PrintStream output) {
    this.output=output;
}

void visitBinaryGlyph(BinaryGlyph visited) {
    visited.left.accept(this);
    visited.right.accept(this);
}

void visitStringGlyph(StringGlyph visited) {
    output.print(visited.content);
}

void visitCharGlyph(CharGlyph visited) {
    output.print(visited.content);
}
}

```

One more useful thing to do is to provide an easy way to invoke the operation implemented by the visitor. To this end we implement a `print` method in the `Glyph` class:

```

abstract class Glyph {
    ...

    void print(java.io.PrintStream output) {
        PrintVisitor v = new PrintVisitor(output);
        this.accept(v);
    }
}

```

Note that in this ‘manual’ implementation of the visitor there are some obvious parts which look like they would be easily generated. The implementations of the `accept` methods for example are particularly repetitive. Also the declaration of the abstract `visit<?VisitedClass>` methods follows a very strict pattern.

## 5 A Domain Specific Language for Visitor Declaration

Our goal is to make a reusable code generator to generate visitor implementations for us. We want to make this code generator as much as possible independent from the fact that we are implementing it for Glyphs. Only then will we be able to reuse it for other data structures as well. This is achieved by making a clear separation between the code generator itself and any context or glyph specific information that it relies on. Therefore we have to meticulously refrain from letting the code generator itself refer directly to anything which is specific for the glyph classes. Instead, glyph dependent information must be declared explicitly as separate facts. The code generator will of course need to use this information, but may only do so by querying the facts.

The end result of our efforts to extract and explicitly declare all glyph dependent information is shown in figure 4. These declarations actually provide a complete specification for two visitors on Glyphs. This concise description is all we need to know to generate their complete implementation.

```

/** Abstract GlyphVisitor class name */
abstractVisitor(GlyphVisitor).

/** A GlyphVisitor can visit any subclass of Glyph */
visitorVisits(GlyphVisitor,?AnyGlyph) :- subclass(?AnyGlyph,Glyph).

/** PrintVisitor: a concrete GlyphVisitor */
concreteVisitor(PrintVisitor).
visitorExtends(PrintVisitor,GlyphVisitor).
visitorOperation(PrintVisitor,Glyph,
                 void,print,[java.io.PrintStream],[output]).

visitorBody(PrintVisitor,CharGlyph,{
            output.print(visited.content);}).

visitorBody(PrintVisitor,StringGlyph,{
            output.print(visited.content);}).

visitorBody(PrintVisitor,BinaryGlyph,{
            visited.left.accept(this);
            visited.right.accept(this); }).

/** CharacterCountVisitor: a concrete GlyphVisitor */
concreteVisitor(CharacterCountVisitor).
visitorExtends(CharacterCountVisitor,GlyphVisitor).
visitorOperation(CharacterCountVisitor,Glyph,int,countCharacters,[],[]).
visitorInitializeResult(CharacterCountVisitor,0).

visitorBody(CharacterCountVisitor,CharGlyph,{
            result = result+1;}).

visitorBody(CharacterCountVisitor,StringGlyph,{
            result = result+visited.content.length();}).

visitorBody(CharacterCountVisitor,BinaryGlyph,{
            visited.left.accept(this);
            visited.right.accept(this); }).

```

**Fig. 4.** Declaring Glyph-specific information for Visitors

One way to look at what we are doing by carefully sifting out all context and glyph dependent information into separate declarations is that we are effectively

defining a declarative domain specific language for specifying visitor patterns. In this sense the declarations in figure 4 constitute a description of two glyph visitors, expressed using a notation specifically designed for specifying visitors. Likewise, the code generator that will be presented next, constitutes a concrete implementation for this domain specific language.

## 6 A Declarative Code Generator for Visitors

We will now present the implementation of the declarative code generator for visitor. This code generator consists out of a set of declarative code generation rules and auxiliary rules.

Besides showing that it is possible to do so, we also want to convey an impression of how difficult (or easy) it is to actually define these rules. Therefore we will be fairly complete in covering the implementation of this first example.

The manual implementation of visitors is a fairly complicated affair and consequently so are the code generation rules. All in all however, despite the complexity of the visitor pattern, the complexity of the rules is certainly not prohibitive. The rules are an almost direct expression of how we manually implemented the visitor before. Thus, we will also present them roughly in the same order as we discussed the manual implementation.

To understand the rules more easily, it is helpful to keep a concrete implementation context in mind. So you may occasionally want to refer back to the declarations in figure 4. The code generator rules are interpreted relative to these declarations and every one of the declarations will be important at some point in the code generator to shaping the resulting code.

### 6.1 Generating Visitor-Support Infrastructure

We start by presenting the rules that take care of generating the visitor's supporting structure: the abstract visitor class and an accept method in each of the visited classes.

The following two rules declare the abstract visitor class. Note that they do not refer to the name of the class explicitly. Instead they get the name through a query which matches one of the glyph specific facts from figure 4.

```
class(?AbstractVisitor) :- abstractVisitor(?AbstractVisitor).
abstract(?AbstractVisitor) :- abstractVisitor(?AbstractVisitor).
```

Our next rule declares the (abstract) visit methods for the abstract visitor class. One such method will be declared for each *concrete* visited class, following the naming convention `visit<?Visited>`. Again, none of the actual visited classes are mentioned explicitly. They are obtained by means of queries.

```
abstractmethod(?AbstractVisitor,void,visit<?Visited>,[?Visited],[visited]) :-
    visitorVisits(?AbstractVisitor,?Visited),
    abstractVisitor(?AbstractVisitor),
    concrete(?Visited).
```

The next two rules declare an accept method in each of the visited classes. There are two rules because we need to cover two different cases, depending on whether the visited class is abstract or concrete.

```
/* Defining an abstract accept method for an abstract visited class */
abstractmethod(?Visited,void,accept,[?AbstractVisitor],[v])
    :- abstractVisitor(?AbstractVisitor),
       visitorVisits(?AbstractVisitor,?Visited),
       abstract(?Visited).

/* Defining a concrete accept method for a concrete visited class */
method(?Visited,void,accept,[?AbstractVisitor],[v],{
    v.visit<?Visited>(this);})
    :- abstractVisitor(?AbstractVisitor),
       visitorVisits(?AbstractVisitor,?Visited),
       concrete(?Visited).
```

The rules presented up to now declare all of the supporting infrastructure. Next, we move on to rules for the concrete visitors.

**Generating the Concrete Visitors** These rules declare the concrete visitor classes and their extends clauses.

```
class(?ConcreteVisitor) :- concreteVisitor(?ConcreteVisitor).
extends(?CV,?AV) :- visitorExtends(?CV,?AV).
```

The method bodies for concrete visitors are hard to generate automatically. The reason is that they are strongly dependent on what we actually want the visitor to do and what structures they are visiting, etc. Therefore the actual implementations of the concrete visit methods, their method bodies, are provided on a case by case basis as part of the context dependent declarations (figure 4). Consequently the concrete visitor's methods are declared by a rule which puts each one of these method bodies into a method carrying the correct signature:

```
method(?Visitor,void,visit<?Visited>,[?Visited],[visited],?body) :-
    visitorBody(?Visitor,?Visited,?body).
```

Despite the fact that concrete visitor implementations are highly context dependent, it is still true that in some (but not all) cases they do follow almost algorithmically defined structure. Therefore it could be possible in some cases at least to generate the concrete method bodies. Although we did not bother with this right now, we will have more to say about this issue later when we look into how we can specialize the visitor generator for composites.

**Generating the operation method** In this section we discuss how to generate the operation method which instantiates and then starts a visitor. The operations implemented by a different visitors may vary both in their arguments and in their

return type. For this reason, an `visitorOperation` declaration was provides the operation signature for each concrete visitor in figure 4.

The facts of interest are:

```
visitorOperation(CharacterCountVisitor,Glyph,
                 int,countCharacters,[],[]).
visitorOperation(PrintVisitor,Glyph,
                 void,print,[java.io.PrintStream],[output]).
```

These two declarations state that the `CharacterCountVisitor` and the `PrintVisitor` both implement an operation on the `Glyph` class. The methods have names `countCharacters` and `print` respectively. The signature of the operation is declared in a format similar to method declarations.

This information is not only used to define the operation methods but is also used to generate extra details of the concrete visitor classes. The names and types of the provided parameters are used to automatically declare some instance variables for the visitor. These are needed, so that visitor methods can access the arguments of the operation through them. For example, `PrintVisitor` will get an instance variable `output` as per the following rule:

```
var(?Visitor,?Type,?name) :-
    visitorOperation(?Visitor,?,?Ret,?oper,?Args,?args),
    JavaFormals(?Args,?args,?formals),
    JavaFormalsElement({?Type ?name},?formals).
```

Since the thus declared instance variable(s) will need to be initialized, a constructor implementation is automatically declared as well, by the following rule.

```
constructor(?Visitor,?Args,?args,?initFromArgs) :-
    visitorOperation(?Visitor,?,?Ret,?oper,?Args,?args),
    JavaInitFromArgs(?args,?initFromArgs).
```

Note, the `JavaInitFromArgs` predicate is an auxiliary predicate which produces initialization code of the form:

```
this.?a1 = ?a1;
this.?a2 = ?a2;
...
```

Since this kind of code is actually fairly common when generating constructors TyRuBa provides the `JavaInitFromArgs` predicate as part of a library of useful auxiliary predicates.

Operations may also need to return a result. For example the `charCountVisitor` returns an `int`. Similar to the parameters, the result is mapped to an instance variable of the visitor, which may be updated during the traversal process and finally returned. This instance variable will be called `result` and is declared by the following rule:

```

var(?Visitor,?Ret,result,?init) :-
    visitorOperation(?Visitor,?,?Ret,?oper,?Args,?args),
    NOT(equal(?Ret,void)),
    visitorInitializeResult(?Visitor,?init).

```

The rule verifies that the return type is not `void` because then there is no need for a `result` variable.

The operations themselves can now be easily defined. Again there will be two distinct cases, depending on whether or not the the return type of the operation is `void` or not. The following rule handles the case when there is actually a real return value. There is another rule for handling the case where the return type is `void` but we will not show it because it is very similar.

```

method(?Class,?Ret,?oper,?Args,?args,{
    ?Visitor v = new ?Visitor(?Jargs);
    this.accept(v);
    return v.result;
})
:- visitorOperation(?Visitor,?Class,?Ret,?oper,?Args,?args),
    NOT(equal(?Ret,void)),
    JavaCommaList(?args,?Jargs).

```

What this rule does, is declare the method `?oper` according to the signature provided by the `visitorOperation` declaration. The implementation of the method instantiates a visitor of the corresponding type and passes its formal arguments on to the constructor of the visitor. There is a slight technical complication with this, because we have to convert the formal argument names `?args`, which is a logic list, into a Java argument list, `?Jargs`. Since these kinds of conversions are fairly common when code generation becomes a bit more sophisticated, TyRuBa actually provides a library of predefined predicates to do these kinds of conversions. Here we call on one of these predicates, `JavaCommaList`, to convert a logic list into a Java comma separated list.

This concludes the discussion of the visitor code generator. From the discussion here, it may seem that the code generator is rather complicated. However, all in all, it is not really all that bad. Consider that we effectively discussed nearly the entire code generator here. We only a skipped a few simpler and similar rules, while treating the most complicated rules explicitly in the text.

A few numbers about the visitor implementation may help a lot to put its relative complexity into perspective. The code generator, consists out of

- 15 logic rules
- 91 lines of source code (comments and blanks included)
- 50 actual (i.e. non-blank, non-comment) lines of code.

We don't want to misleadingly understate the amount of effort involved in implementing these rules however. The real complexity of the task of implementing the code generator is actually not the implementation itself. The quoted numbers speak for themselves. The code generator really is not all that hard to implement.

What is harder however is understanding how to disentangle the code generator from the specifics of the Glyphs example in nice way. In other words, to come up with a clean DSL notation which captures precisely and concisely all the context sensitive information needed to generate the entire visitor implementation.

Despite the fact that the code generator only amounts to 15 logic rules it did take us nearly two days to implement these from scratch. This example was the first design pattern generator we implemented. Having no prior experience with this, we were actually somewhat over ambitious and spent a lot of time trying to also generate the code for concrete visitor bodies. This seemed a good idea, because they do often seem to have a lot of common structure. Since it was one of our prerequisites to generate complete implementations which require no editing of generated code at all, this implies either a very complex DSL notation or only covering a small number of cases. Finally, the idea was abandoned and we settled for a more modest approach, allowing the DSL to provide the concrete method bodies by means of `visitorBody` declarations. Once this decision was made, all the rest followed fairly easily. This turned out to be a good choice because it is an open solution which is reusable in many context. Also, it can be easily specialized upon to handle special cases more effectively. We will see an example of this later, to implement visitors for composites in section 9.

A valuable lesson we learned from this first experiment is that it is probably better not to be too ambitious in trying to over automate. It is more reasonable to focus on the concrete examples of immediate interest and see to coping with these in a satisfying way. This leads much more quickly to an implementation of a simpler code generator. What's more, when the need arises later, we can usually specialize or refine the code generator and its DSL easily. This 'short-sighted strategy' will be applied almost like a religion in the remainder of the paper.

## 7 Making the DSL More Explicit

As explained in section 5, the code generator defines an implicit domain specific language. The contextual information is specified as facts have to conform to the format that the code generator expects. This, in effect, is our DSL's syntax. Unfortunately, the syntax is currently only implicitly defined. To make things worse, the implementation of the code generator is actually not very robust with respect to accepting incorrect syntax. It makes several implicit assumptions but there is no mechanism to enforce them. In other words, it is very easy to accidentally misuse the code generator and specify incorrect DSL programs.

To make it easier to (re)use the code generator. We should clearly document the DSL design and make its assumptions as explicit as possible. One way is to carefully document the DSL using elaborate comments in natural language. This kind of documentation is certainly useful and very important. Additionally, a more active and complementary verification mechanism desirable. As it turns out, a logic notation is very good for expressing many of the assumptions naturally and clearly. For example, the following rule declares the assumption that

all concrete visitors must have a declared `visitorBody` for all concrete visited classes.

```

mustHold(visitorBodyDeclared<?Visitor,?Visited>) :-
    concreteVisitor(?Visitor),
    visitorVisits(?Visitor,?Visited),
    concrete(?Visited).

visitorBodyDeclared(?Visitor,?Visited) :- /* Directly declared */
    visitorBody(?Visitor,?Visited,?body).
visitorBodyDeclared(?Visitor,?Visited) :- /* Inherited from ... */
    visitorExtends(?Visitor,?AbstractVisitor),
    visitorBody(?AbstractVisitor,?Visited,?body).

```

The TyRuBa system supports declaring assumptions by means of `mustHold` declarations such as this one and it will verify that all declared assumptions are satisfied, warning us when they are not.

## 8 Implementing Composites

Our next example is the implementation of two composite glyphs, `RowGlyph` and `ColumnGlyph`, for representing horizontally and vertically aligned compositions of glyphs. For this we will use the composite design pattern. This time we will not bother with a manual implementation first, but move on directly to implementing a declarative code generator.

To make things a little easier we will not think about visitors now. But keep in mind that sooner or later however, we will have to face up to combining visitor and composite since we already defined some visitors on glyphs and we obviously want to keep these working.

Exactly as was the case with visitors, we will use a DSL-facts notation to separate all glyph dependent information from the code generator. The DSL declarations for composite glyphs are presented in figure 5. These declarations simply state that we want to have two composite Glyphs, called `RowGlyph` and `ColumnGlyph`.

```

composite(RowGlyph,Glyph).
composite(ColumnGlyph,Glyph).

```

**Fig. 5.** Declaring Composite Glyphs

We will not present the actual code generation rules this time, they are not all that interesting. Suffice it to say that they are similar in style to the rules for generating visitor, albeit significantly less complex (17 logic rules, 37 lines of actual code).

The code generator will generate implementations for two concrete composite classes `RowGlyph` and `ColumnGlyph`. The code generated for `RowGlyph` looks roughly as shown below. The generated code for `ColumnGlyph` is nearly identical.

```
class RowGlyph extends Glyph {

    java.util.Vector children = new java.util.Vector();

    RowGlyph() {}

    void addChild(Glyph child) {
        children.addElement(child);
    }

    Glyph getChild(int i) {
        return (Glyph)children.elementAt(i);
    }

    int numberOfChildren() {
        return children.size();
    }

    void accept(GlyphVisitor v) {
        v.visitRowGlyph(this);
    }

    ...
}
```

The reader may be surprised to find that, even though we did not pay any attention to it yet, some of the visitor implementation already automatically show up in the generated `RowGlyph` class. Just remember this as an interesting fact for now. We will explain more about this in the next section.

The code generator will also declare implementations of the child access methods (`getChild`, `addChild`, `numberOfChildren`) in the component class (`Glyph`). These are also part of the design pattern's implementation and provide for a uniform interface across leafs and composites. Only the `numberOfChildren` method has a really meaningful implementation for non-composite classes (it returns 0). The others methods just throw 'illegal operation' exceptions.

It is worth pointing out that there are many different ways for implementing composites. There are many possible variations on how to store the children in a composite. Also variations in its child access interface are possible. From the many possible variations, we only picked a single one. The one we picked is a useful, fairly general, and happens to be sufficient for our examples. If we want to, we could also design a more elaborate DSL and code-generator to cover a broader variety of different implementations. However, having learned the lesson from the visitor generator implementation, we were satisfied with a more 'short sighted' approach rather than complicating things with variations we have no immediate use for.

This being said, it remains true that the simple code generator is already very useful. Despite its straightforward simplicity it can effectively be re-used to generate (similarly implemented) composites for other structures besides glyphs.

## 9 Design-Pattern Combination: Visitor + Composite

In this section we will look at how declarative code generators make it relatively easy to combine different patterns. Our goal now is to make sure that `Visitors`, `CharCountVisitor` and `PrintVisitor` continue to work after we introduce the composites. Note that `RowGlyph` and `ColumnGlyph` are automatically declared as being visited by `GlyphVisitors` because we already declared (Fig. 4) the following rule:

```
visitorVisits(GlyphVisitor,?AnyGlyph) :- subclass(?AnyGlyph,Glyph).
```

This means that if we simply take the basic glyph classes, all code-generation rules and all glyph-specific declarations that we have so far, and load all of these into the TyRuBa system, it will already generate some of the visitor code for us correctly, without us having to even lift a finger. This why we, if you still remember, already found an `accept` method in `RowGlyph` in the previous section.

What the system will also do however, is warn us that it suspects a problem with the generated code, because it cannot find `visitorBody` declarations for the concrete visited classes `RowGlyph` and `ColumnGlyph`. These warnings come to us because of the declared `mustHold` assumption from section 7. Declarations for concrete `visitorBody` implementation for visiting the two composite glyphs `RowGlyph` and `ColumnGlyph` are actually the only missing pieces to make the combination complete. So all that remains to be done is provide concrete implementations for `CharacterCountVisitor` and `PrintVisitor` to visit `RowGlyph` and `ColumnGlyph`. This amounts to 4 (i.e. 2 times 2) additional declarations we have to make.

However, what is really interesting is that we can do better than that! We can actually take advantage of the combination. The reason for this is: knowing that a visited class is a composite immediately suggest a default implementation for visiting it. Consider the following rule which makes this clear:

```
/* Defining a visitorBody for composites */
visitorBody(?Visitor,?Composite,{
    for (int i=0; i<visited.numberOfChildren(); i++) {
        visited.getChild(i).accept(this);
    }
}) :- visitorVisits(?Visitor,?Composite),
    composite(?Visited).
```

This rule will automatically declare a `visitorBody` implementation for any visited class which is also a composite. In many cases, this will provide exactly the implementation that we want. For example, for the `characterCountVisitor` it gives us a correct implementation. Unfortunately, for `PrintVisitor` we are

not as lucky. Most likely we want the printed output to somehow reflect the nesting of rows and columns and, unfortunately, it will not do that.

It is tempting to let the default be generated anyway and then simply edit it because it is not all that far off from what we'd really want. However, we want to make it a point *never ever to edit generated code*. The generated code *must* be complete and exactly right. So we will go the extra mile and refine this single-ruled code generator, in the process also defining the third DSL in this paper: a notation for specifying composite visitor bodies. By now, you know the drill... the specification for visiting `RowGlyphs` and `ColumnGlyphs` is shown in figure 6. Note that we only declare information about `PrintVisitor`. We don't need to specify any information for `CharacterCountVisitor` because we assume that the plain default is adequate. We will build the code generator so that it uses the simple default, unless told otherwise. Thus, for the `PrintVisitor` additional `preVisitActions` and `postVisitActions` are provided for extending the default into what we really want.

```
preVisitAction(PrintVisitor,RowGlyph,{output.print("**ROW>");}).
postVisitAction(PrintVisitor,RowGlyph,{output.print("<ROW**");}).

preVisitAction(PrintVisitor,ColumnGlyph,{output.print("**COL>");}).
postVisitAction(PrintVisitor,ColumnGlyph,{output.print("<COL**");}).
```

**Fig. 6.** Declaring how to implement `PrintVisitor` for composite glyphs

The code generator is not that hard to implement. First we declare a rule to define the simple default as follows:

```
simpleCompositeVisitorBody(?Visitor,?Composite,{
  for (int i=0; i<visited.numberOfChildren(); i++) {
    visited.getChild(i).accept(this);
  }
}) :- visitorVisits(?Visitor,?Composite),
      composite(?Composite).
```

Then we define a rule which selects this one as the actual `visitorBody` when there are no declarations that specify pre- or post-visit actions.

```
visitorBody(?Visitor,?Composite,?simple) :-
  simpleCompositeVisitorBody(?Visitor,?Composite,?simple),
  NOT(preVisitAction(?Visitor,?Composite,?)),
  NOT(postVisitAction(?Visitor,?Composite,?)).
```

There are a few other rules, that take care of pre and post actions. The rule below handles the case where there is both a pre- and a post-visit action.

```
visitorBody(?Visitor,?Composite,{
```

```

    ?pre
    ?simple
    ?post
}) :- simpleCompositeVisitorBody(?Visitor,?Composite,?simple),
    preVisitAction(?Visitor,?Composite,?pre),
    postVisitAction(?Visitor,?Composite,?post).

```

Two more rules, we did not show, handle the cases where there is only a pre- or only a post-visit action.

## 10 Unanticipated Evolution of a Declarative Code Generator

Given the nature of design patterns it is very hard, if not impossible, to capture them in their full generality. A more reasonable approach is to focus on some concrete interesting example and a particular style of implementation. It is fairly easy to then provide a code-generator and DSL notation which work well for these. This was basically the kind of ‘short-sighted strategy’ we followed in the previous sections for composite and visitor.

Because of this approach, what will frequently happen is that it becomes necessary to refine the code-generator and its DSL notation to adapt it to new situations in which we want to use it. We are bound to encounter new and interesting variations we want to express, but which are not covered by the current DSL and code generator.

The point we want to illustrate now is that this need not be a dramatic problem. The code generators themselves are typically not all that complicated and therefore relatively easy to change in order to fit the new context. Even more important, is that we can arrange to do this in such a way that it will not disrupt the existing users of the code-generator. We can extend its DSL notation in a ‘conservative’ way so that existing user’s get exactly what they are used to as a ‘default’ implementation.

In a way, the refinement of the simplest composite visitor rule was already an example of this. We will show one more example. In this example we will need to extend the composite-visitor DSL to be able to define yet another glyph visitor. The `DisplayVisitor` is a visitor to display a `Glyph` onto a two-dimensional textual display, aligning its sub-glyphs in rows and columns.

The `DisplayVisitor` will keep track of its current screen position. It displays glyphs relative to this position, moving it around to align them as desired. Unfortunately just declaring `preActions` and `postActions` is not enough to express what we want to do. We also need to be able to make this visitor perform some actions associated with visiting individual children. After visiting a child, row and column glyphs need to update the current screen position in order to align the next printed child correctly. So we need to be able to say something like

```
childVisitAction(DisplayGlyphVisitor,RowGlyph,{...blah...}).
```

The problem is that we did not foresee this in the initial implementation of the composite-visitor generator. Fortunately, it is not too difficult to extend the generator to support this new declaration. Only one new rule and a small change to an existing rule suffice. We add the following rule:

```
simpleCompositeVisitorBody(?Visitor,?Composite,{
    ?Component visitedChild;
    for (int i=0; i<visited.numberOfChildren(); i++) {
        visitedChild = visited.getChild(i);
        visitedChild.accept(this);
        ?statements
    }
}) :- visitorVisits(?Visitor,?Composite),
    childVisitAction(?Visitor,?Composite,?statements),
    composite(?Composite,?Component).
```

Now, to ensure that everything keeps working exactly as before for the visitors we already have, we slightly edit and replace the old `simpleCompositeVisitorBody` rule:

```
simpleCompositeVisitorBody(?Visitor,?Composite,{
    for (int i=0; i<visited.numberOfChildren(); i++) {
        visited.getChild(i).accept(this);
    }
}) :- visitorVisits(?Visitor,?Composite),
    NOT(childVisitAction(?Visitor,?Composite,?statements)), /*!*/
    composite(?Composite).
```

So now we have two rules declaring a `simpleCompositeVisitorBody`, the conditions of both rules are mutually exclusive however. The last rule (edited old rule) is used only in cases where there are no child actions, in this case everything remains as before. The other one correctly handles the newly added child action declarations. So now, we can define the `DisplayVisitor` nicely. Its full definition is shown in figure 7.

## 11 Other Design Patterns

The examples we have shown have one thing in common. We picked relatively complicated, structure-rich design patterns. This is not a coincidence, because it are exactly these design patterns which will benefit most from implementing them as code generators. Other structure rich design patterns (e.g decorator) would probably work equally work well.

In essence, any conceivable pattern is implementable as a declarative code generator. But for very simple patterns, for example factory method, this is less likely to benefit them. Basically, because there is not enough useful structure to generate. For such patterns, the DSL notation would not be significantly less complex than the actual implementation. However, there may still be other

```

concreteVisitor(DisplayGlyphVisitor).
visitorExtends(DisplayGlyphVisitor,GlyphVisitor).

visitorOperation(DisplayGlyphVisitor,Glyph,
                 void,displayOn,[TextDisplay],[display]).

/* Variables to maintain the current position on the display. */
var(DisplayGlyphVisitor,int,row,0).
var(DisplayGlyphVisitor,int,col,0).

visitorBody(DisplayGlyphVisitor,CharGlyph,{
    display.printAt(row,col,visited.content);
    row = row+1;
    col = col+1;
}).

visitorBody(DisplayGlyphVisitor,StringGlyph,{
    display.printAt(row,col,visited.content);
    row = row+1;
    col = col + visited.content.length();
}).

preVisitAction(DisplayGlyphVisitor,RowGlyph,{
    int startRow = row;
    int maxRow = row; }).
childVisitAction(DisplayGlyphVisitor,RowGlyph,{
    if (row>maxRow) maxRow=row;
    row = startRow;}).
postVisitAction(DisplayGlyphVisitor,RowGlyph,{
    row = maxRow;}).

preVisitAction(DisplayGlyphVisitor,ColumnGlyph,{
    int startCol = col;
    int maxCol = col; }).
childVisitAction(DisplayGlyphVisitor,ColumnGlyph,{
    if (col>maxCol) maxCol=col;
    col = startCol;}).
postVisitAction(DisplayGlyphVisitor,ColumnGlyph,{
    col = maxCol;}).

```

**Fig. 7.** Declaring how to implement DisplayVisitor for Glyphs

reasons, besides code generation, for which it might be useful to make contextual information explicit in a DSL. Such information may also be useful, simply as a form of documentation and also to automatically verify conformance and correct usage. Some related work in this area, also using logic meta programming, is discussed below.

## 12 Related Work

### 12.1 Logic Meta Programming and Design Patterns

The idea of logic meta programming itself, i.e. using a logic language as an expressive and powerful means to reason about programs is not new. Some general surveys of the field can be found in [HG98,Bar95]. Logic programming languages are known to be good for implementing various kinds of meta programs, such as compilers, interpreters, type checkers, type inferencers etc. It's powerful unification and backtracking mechanism make it especially suitable for implementing these kinds of programs. Also, many features have been added to logic languages to facilitate meta programming. Prolog [DEDC96,CM81,SS94] for example has features to support meta programming. It offers definite clause grammars for example, a feature that facilitates the implementation of parsers. The programming language Gödel [HL94] is a declarative higher-order logic language designed for meta programming. Lambda Prolog [FGH<sup>+</sup>90] is an extension of Prolog with unification of lambda terms. Lambda terms are an extension specifically intended to facilitate the manipulation of formulas and programs [MN87]. It is especially useful in manipulating functional programs.

A logic programming approach has also been proposed for expressing sophisticated pattern matching and verification of programs [Wuy98,Cre97,BGV90,CMR92,Min96,KP96]. The power of logic is exploited in two major ways in these approaches: as a verification/enforcing tool (e.g. Law Governed Architectures [Min96,MWD99,Men00]) or as an information gathering tool [Cre97,BGV90,CMR92,KP96] or a combination of both [Wuy98]. Both kinds of uses can be interesting in the context of managing design pattern implementation: for finding design-patterns in code [KP96,Wuy98] or for verifying design-pattern related implementation constraints [Wuy98].

### 12.2 Automatic Code Generating for Design Patterns

To our knowledge an approach using declarative logic meta programming for *generating* design-pattern implementation has never before been investigated in the literature. However, there is some prior work using other formalisms [BFVY96,FMVW97,OYM99]. Our work is distinguished from these prior efforts mostly because we are able to generate complete implementations for concrete design pattern instantiations, requiring no further editing of generated code whatsoever. The aforementioned approaches all rely on editing generated implementations to some extent in order to concretize them.

The system developed by Budinsky et. al. [BFVY96] covers a much wider range of patterns and a much wider range of possible variations than the few examples we have experimented with. They use a combination of Perl scripting and a specially developed macro expansion language to generate code. One clear disadvantage of their system, is that generated code has to be integrated into the software and concretized further by copy, paste and edit. Not only does this make the integration and concretization irreversible, it also obviously complicates combination of design patterns which are separately generated. The system developed by Ohtsuki and Yushida [OYM99] uses an SGML based representation of template code, but exhibits the same problems they also rely on editing generated code. In our approach however, code generators combine very naturally as we have shown.

Florijn's system [FMVW97] uses a representation of design patterns and the systems in which they occur, as a code repository, which is represented as a graph of program fragments. In many ways this looks similar to our repository of code-representing facts. Generating code for a design pattern in this system, is like cloning a 'prototype implementation graph'. The system also relies on editing the cloned/generated pattern to concretize it. We must point out however, that this seems to be much better supported than in the other systems, because the graph-repository maintains a close relationship between the evolving implementation and the design-pattern definition.

### 12.3 Generative Programming

It is worth noting that our use DSL notation and terminology in this paper was inspired by the usage of DSL's in generative programming. Generative programming is a methodology for developing software product lines. In this, DSL notations and code-generation are used to automate the configuration and assembly of families of software-systems from reusable components. We share in common with this approach the idea of using DSL's and code generation, and also the way a DSL can be extended in conservative, non disruptive ways. Generative programming uses many different techniques for generating code, most thoroughly explored is C++ template meta programming. Declarative code generators or logic meta-programming in general have never been used for generative programming, but this is an interesting idea and it would probably work well.

## 13 Future Work

The examples shown in this paper show a lot of promise. However still a lot of work lies ahead. The TyRuBa system is very experimental and leaves a lot to be desired. In its current form it is not useful in practical programming. Its module system for example is a simple include file mechanism. While good enough for limited and moderate sized experiments. It is doubtful it will scale up to handling larger systems with many design patterns to be generated. A crucial point to ensure salability is adding a good module system to the TyRuBa model.

Modules and encapsulation mechanisms for logic have been studied extensively in the literature [McC92,Mos94,Dav93], but it is not yet clear what kind of module system would best fit our purpose.

The development of a programming environment around the idea of declarative code generators and computed repositories is also an interesting topic. Rather than working with sources in the traditional sense, the sources would be represented as a logic data base of facts and rules. Thus it provides an integrated view on hand-written code (represented directly by facts) mixed with generated code (represented by rule based computation of facts). This environment should provide ways to browse and edit the rules, edit hand-written code and view generated code. It is important that the ‘code browser’ offers good support in tracking down where a particular portion of generated code comes from: to determine which are the rules and facts that have impacted this portion of generated code.

Another point of research is that the current model is very much syntax oriented. Therefore the DSL’s have rather ad-hoc semantics. A semantic-based rather than syntax-based notion of ‘generation’ would yield DSL’s with cleaner and clearer semantics and make code generators more easy to implement and maintain. Compare this distinction to the difference between macros and procedures. Macros and procedures are much alike in many ways. However, macros come with all kinds of technical idiosyncrasies related to the fact that they are based on syntactic transformation. TyRuBa code generators are very much like macros in this sense. A point of research is therefore to find a ‘procedure-like’ equivalent, with much the same characteristics as the current system, but cleaner semantic properties. An immediate step in this direction is investigating ways to define representational mappings which are based on semantics and not based on parse-tree representation as is currently the case.

## 14 Summary and Conclusions

### 14.1 The Examples

This paper was run by examples. We quickly recall them here. We presented a case study implementation of glyphs, using design patterns. Our goal was to show how design pattern implementation can be automated by means of declarative code generators. A declarative code generator is implemented relative to a separately provided set of declarations specifying the needed contextual information to correctly implement the design pattern. The notation used to express this information amounts to a domain-specific language specifically designed to express a particular design pattern. In the course of this paper we have seen three such DSL’s and corresponding code generators:

1. A Visitor DSL: to specify how to generate visitor patterns.
2. A Composite DSL: to specify how to generate composite patterns.
3. A Composite Visitor DSL: A refinement of the visitor DSL which provides an automatic default for visiting composites. It allows to ‘tweak’ the default, by adding pre, post and child actions.

## 14.2 Discussion of Pros and Cons of Generating Design Patterns

In this section we summarize some of the pro's and contra's of the code generation approach to design-pattern implementation.

The initial motivation for starting this work was the observation that implementing design patterns is often tedious and that the implementation cannot be reused in a different context.

The most obvious advantages of design patterns as declarative code generators, is that the code generators *are* reusable and that they automate the tedious and error prone parts of the implementation. All of this is achieved by separating out context dependent information and expressing it in a light-weight declarative DSL notation using facts.

Our approach has significant advantages over others. It is possible to easily combine code generators for different design patterns. Additionally, we can offer a specialized code generator extension to take advantage of knowledge which results from combination to automate the combined implementation even further.

We strive for a code generation approach, which is accurate to the point that no 'post-editing' of generated code is ever necessary. Instead, we advocate editing DSL specifications, or if necessary, refining the code generator and the DSL. In our examples this was always possible to do relatively easily.

A criticism of our approach might be that a reusable code generator is often harder to implement than a single instance of the design pattern. We believe that we have shown in the examples that the implementation is not all that hard. Even to the extent that it is feasible to implement code-generators 'on demand' and also refine them 'on demand' the very moment you actually need them. The extra effort is reasonably small compared to hand-implementation of a design pattern, but offers some very important advantages which are worth the effort.

- A 'DSL programmer' focuses on the interesting aspects of the implementation: aspects which cannot be automated.
- Tedious, repetitive parts of the implementation are generated. Consequently the resulting code is less error prone.
- The DSL specification is more concise and reveals more interesting information than its actual implementation.
- The code generator can be reused, a manual implementation cannot.

Another criticism may be that a DSL and code generator typically only captures a limited view on a particular design pattern. Design patterns are very open, very fuzzy concepts and can vary enormously from one context to the next. The code generator implementation is typically much more concrete and makes a number implicit and explicit choices in its implementation. Not all of these choices are typically offered through its DSL. So, typically it will capture only a limited set of fairly similar implementations. This was very clear in our examples. However, note that just this is already a significant improvement over manual implementation, which even if we do need a similar implementation, can only be reused by error prone copy, paste and edit. The code generators are also relatively simple which accommodates unanticipated future variations in a backward compatible way.

## References

- [Bar95] J. Barklund. Metaprogramming in logic. *Encyclopedia of Computer Science and Technology*, 33:205–227, 1995. Also available as UPMAIL Technical Report No. 80.
- [BFVY96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, Vol. 35, No. 2, 1996.
- [BGV90] R. Ballance, S. Graham, and M. VanDeVanter. The Pan Language-Based Editing System for Integrated Development Systems. In *Proc. 4th ACM SIGSOFT Symp. on Software Development Environments*, volume 15:6 of *ACM SIGSOFT Software Engineering Notes*, pages 77–93, 1990.
- [CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.
- [CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.
- [CMR92] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and Querying Software Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, May 1992.
- [Cre97] R.F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.
- [Dav93] A. Davison. A Survey of Logic Programming-based Object Oriented Languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Trends in Object-Based Concurrent Computing*, pages 42–106. MIT Press, Cambridge, MA, 1993.
- [DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, New York, 1996.
- [DV98] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.
- [DV99] Kris De Volder. Aspect-oriented logic meta programming. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer Verlag, 1999.
- [FGH<sup>+</sup>90] Amy Felty, Elsa Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov.  $\lambda$  Prolog: An extended logic programming language. In M. Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction (Kaiserslautern, West Germany)*, volume 449 of *lncs*, pages 754–755, Berlin, 1990. sv.
- [FMVW97] G. Florijn, M. Meijers, and P. Van Winsen. Tool support for object-oriented patterns. *ECOOP'97 - Object-Oriented Programming 11th European Conference. Proceedings*, pages 472–95, 1997.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
- [HG98] P. M. Hill and J. Gallagher. Meta-programming in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming*, 5:421–498, January 1998.
- [HL94] P. Hill and J. Lloyd. *The Gödel Programming Language*. MIT Press, Cambridge, MA, 1994.

- [KP96] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. *Proceedings of WCRE '96: 4rd Working Conference on Reverse Engineering*, pages 208–15, 1996.
- [McC92] Francis G. McCabe. *Logic & Objects*. International Series in Computer Science. Prentice-Hall, 1992.
- [Men00] Kim Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, October 2000.
- [Min96] Naftaly H. Minsky. Law-governed regularities in object systems, part 1: An abstract model. *Theory and Practice of Object Systems*, 2(4):283–301, 1996.
- [MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.
- [Mos94] Chris Moss. *Prolog++: The Power of Object-Oriented and Logic Programming*. Addison-Wesley, New York, N.Y., 1994.
- [MWD99] Kim Mens, Roel Wuyts, and Theo D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, June 1999.
- [OYM99] M. Ohtsuki, N. Yoshida, and A. Makinouchi. A source code generation support system using design pattern documents based on sgml. *Proceedings Sixth Asia Pacific Software Engineering Conference (APSEC'99)*, pages 292–9, 1999.
- [SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- [Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS USA '98*, 1998.