Journal of Universal Computer Science, vol. 10, no. 7 (2004), 872-891
submitted: 16/2/04, accepted: 21/6/04, appeared: 28/7/04 © *J.UCS*

MetaJ: An Extensible Environment for Metaprogramming in Java

Ademir Alvarenga de Oliveira
(Federal University of Minas Gerais, Brazil
ademirao@dcc.ufmg.br)

Thiago Henrique Braga
(Federal University of Ouro Preto, Brazil
thiagohb@iceb.ufop.br)

Marcelo de Almeida Maia
(Federal University of Ouro Preto, Brazil
marcmaia@iceb.ufop.br)

Roberto da Silva Bigonha
(Federal University of Minas Gerais, Brazil
bigonha@dcc.ufmg.br)

Abstract: MetaJ is a programming environment that supports metaprogramming in the Java language. The environment is designed to allow extensions via plug-ins which permit the user to manipulate programs written in different languages. This facilities concern only syntactic aspects. Semantics aspects are language-dependent and are not addressed here, but could be tackled with other tools, which could even be layered on the top of MetaJ. Accessing patterns by example inside ordinary Java programs is a major feature of MetaJ programming. This paper presents a conceptual description of the environment, implementation details and three applications on analysis, restructuring and generation of programs.

Key Words: metaprogramming, metaprogramming tools, refactoring, program transformation, generative programming, Java, object-oriented frameworks
Category: D.1.5, D.3.3, D.2.6

1 Introduction

Computer programs define internal data structures to abstract from the entities of a problem domain. Since it is potentially possible to abstract from almost every existing concrete entity, one may write programs to reason about almost everything, even other programs, excluding, of course, the class of undecidable problems. Metaprograms are special programs (meta level programs) whose problem domain are another programs (base level programs). Some applications of metaprogramming are program translation from one language to another, program transformation, program refactoring, program comprehension, program optimization, partial evaluation, program verification, type inference/checking, design patterns detection/application, program slicing [Partsch and Steinbrüggen, 1983] [Oppen, 1980] [Mens et al., 2001] [Rugaber, 1995] [Sheard, 2001] [Tip, 1995].

In principle, a metaprogram can be written in any general-purpose programming language. Generally, input programs are represented internally as (abstract) syntax trees, which are either records in procedural languages, or objects in object-oriented languages, or terms in functional languages or rewriting systems [Cameron and Ito, 1984]. Using such representation is not a comfortable task because: (i) there is a large conceptual gap between concrete programs and the operations used to compose and decompose such structures; (ii) metaprograms are not meant to be written only by programmers with expertise in compilation techniques [Visser, 2002].

Metaprogramming is hard because programs are complex. Programmers may have to resort to specialized many features to manage this complexity. These features are often built into programming languages and include: type-systems (to catch syntactically correct, yet semantically meaningless programs), scoping mechanisms (to localize the names one needs think about), and abstraction mechanisms (like functions, class hierarchies, and module systems to hide implementation details). These features add considerably to the complexity of the languages they are embedded in, but they generally worth the cost. Writing programs to manipulate programs means dealing with this complexity twice [Sheard, 2001].

A possible solution to alleviate the inherent difficulty of metaprogramming is the use of metaprogramming-specific languages. One approach for constructing these meta-level languages is by extending a general-purpose language, but this generally presents, two major drawbacks: it is required from the metaprogrammer a great effort for expressing queries on the source code; and the developed metaprograms, usually, lack extensibility and are hard to mantain [Klint, 2003]. Another approach is the definition of a new metalanguage, which provides built-in functionality for expressing high-level operations on the source code, thus yielding simpler metaprograms. On the other hand, a new language is more difficult to learn than a new library for a known language. This difficulty is even more dramatic if the language paradigm is not widely popular. It is also more difficult building a new language from the scratch rather than building a new library.

Cordy and Shukla [Cordy and Shukla, 1992] highlight two difficulties concerning the metaprogramming process: the lack of a general approach for developing metaprogramming languages, and the difficulty in learning metalanguages. Klint [Klint, 2003] argues that despite the similarities between compilation, restructuring and comprehension of programs, the last two processes do not apply the same well-known techniques. Some differences can be pointed out: (i) compilation generally deals with only one source language, while restructuring and comprehension may involve several languages; (ii) a compiler abstracts the source code generating a suitable internal representation. On the other hand, in restructuring, it is necessary to keep a link between the internal representation and the source code because the latter may have to be rewritten. Program comprehension requires user interaction while compilation is usually batch processed. Program comprehension and restructuring are used for reverse engineering, while compilers are used in forward engineering.

Since providing a desirable metaprogramming tool is still a challenge, some requirements to which such a system should satisfy are: expressiveness, readability, analysis, generation, manipulation, transformation, easiness of learning, program patterns *by example*, guarantee of syntactic consistency and multiple base languages.

The sections that follow present MetaJ, an extensible environment for developing metaprograms in Java. Section 2 shows a general picture of the environment, its design decisions, and how it can be used. In Section 3, implementation details are discussed. Section 4 shows some applications that are being carried out with MetaJ. In Section 5, MetaJ is compared with another metaprogramming tools. Finally, conclusions and future work are presented.

2 The MetaJ Environment

MetaJ embodies a set of concepts that are independent of the base language: syntax trees, code references, code iterators and code templates. It defines a framework which supports this independence by isolating the features common to most languages, defining generic operations for them and allows plugging components that are language dependent. Metaprograms are written in Java, and access the generic concepts of MetaJ, dealing with the syntax of specific base languages.

2.1 Internal Representation of Base Programs

Base programs are represented as syntax trees. The type of the tree nodes are derived from the nonterminal symbols of the base language grammar, which may have to be extended to include important node types as nonterminals, i.e., the node types that the metaprogram may have access.

All tree nodes are accessed by means of the interface `Reference`. Thus, references are used to manipulate fragments of base language code, hide tree nodes and provide only operations that preserves the syntax consistence of the resulting tree. The most important operations of the `Reference` interface are:

- `String toString()`: returns the corresponding source code of the sub-tree;
- `boolean match(Reference)`: performs structural comparison of sub-trees;

– `void set(Reference)`: updates the value of a reference;

– `Iterator getIterator()`: returns an iterator to traverse the node sub-tree.

The iterator [Gamma et al., 1995] object returned by `getIterator` method allows traversing the tree node encapsulated by the reference. During the traversal, reference operations are possibly used to test or modify fragments of base code. Despite of usefulness and flexibility, iterators are low-level components and demand expertise in the base language grammar. Next section will describe another way of exploring programs.

2.2 Templates and Variables

A first requirement for a metaprogramming language is providing high-level abstractions to hide the internal representation of base programs. MetaJ allows plugging small domain-specific languages for writing program patterns by example, called templates[1]. Each template language is specific for a base language and is generated from it. In this sense, a language of templates is a superset of the base language. Templates are abstractions that encapsulate a program pattern written by example.

A template has a name, a type and a body. An example of a template is shown in Listing 1 (a). It has name `MyTempl`, and type `CompilationUnit`. In the template body (bounded by braces) is defined a program pattern. The syntax used to describe a program pattern was borrowed from the JaTS [Castor and Borba, 2001] syntax. The type of a template must be one of the node types defined by the base language grammar. The body of a template must include a sentential form, which must match any sentence that can be derived from the nonterminal that defines the respective type of that template. A sentential form can be written with appropriate terminals, metavariables and delimiters of optional sentential sub-forms (#`[ ... ]`#).

Metavariables store references to code fragments. They have types which must be one of the node types, just as templates. The types of metavariables are classified as *single-valued* or *multivalued*, the latter ended with suffix `List`. For example, `ImportDeclaration` denotes a type of a reference to one import declaration. ImportDeclarationList denotes a type of reference to multiple import declarations, and has additional operations for inserting and removing elements.

A sentential subform only can be delimited by optional delimiters (#`[` and `]`#) if the original grammar of the base language allows it to be optional in a program (like package declarations are optional in a Java program).

As an example, it can be noticed that the pattern described in the body of the template MyTempl has an optional `PackageDeclaration` which can be

[1] In Section 5 templates are usually referred as patterns. Templates were preferred to avoid confusion with homonymous concept in software engineering *design patterns*

```
package myTemplates;
language = Java // plug-in name
template #CompilationUnit MyTempl{
   #[#PackageDeclaration:pck]#
   #[#ImportDeclarationList:imps]#
   class MyTempl { ... }
   #TypeDeclaration:td
}
```

(a) The template `MyTempl`.

```
package myTemplates;
import metaj.framework.AbstractTemplate;
public class MyTempl extends AbstractTemplate{
   public final Reference imps, td, pck;
   ... // Implementation of superclass abstract methods
}
```

(b) The corresponding Java code.

Listing 1: An example of template

bound to any reference with type `PackageDeclaration` and has a class whose signature exactly equals the fragment with tokens `class` followed by `MyTempl`.

A template is translated into a Java class, which has the same name of the template. Listing 1 (b) exhibits the translation into a Java class of the template of the Listing 1 (a). Each template can be arbitrarily instantiated. Each instance of a template has its own environment, i.e., its own binding from metavariables to references. The metavariables occurring in templates are available in the corresponding template instances. The API available to manipulate the template has the following methods:

- `String toString()`: returns the corresponding source code of the template. Requires all variables to be bound;
- `boolean match(Reference)`: pattern-matching with a syntax tree. Binds all metavariables to `Reference` objects on the tree. If a metavariable is already bound, it verifies if the corresponding references match;
- `void setXXX(Reference)`: these methods are available for each metavariable in the template, where `XXX` is the name of the metavariable with its first letter capitalized.
- `void addInXXX(Reference, int)`: idem as previous, but adds a child reference to a multivalued metavariable named `XXX`.

The intensive use of templates to specify program restructuring is quite enticing. However, the more complex is a template, the more specific is the program structure it can match, thus reducing its reusability and generality. It is the metaprogrammer's responsability to determine the adequate balance of templates and iterators to achieve a better result. In Section 4, examples of how templates may be used is presented .

3 Implementation Details

MetaJ environment has three major components: the MetaJ framework, base language plug-ins, and the template compiler. In order to introduce a new plug-in into the environment, a context-free grammar processor is available. Figure 1 shows the MetaJ elements and their respective interdependence.

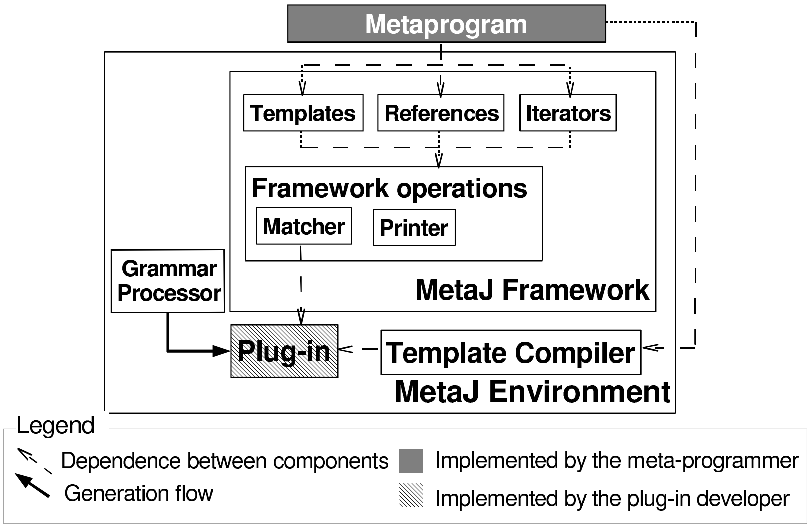

Figure 1: The relationship between MetaJ elements

The grammar-processing tool generates a template language grammar from a specific base language grammar. A template language is a superset of the base language because it can accept any base program, i.e., a template containing no metavariables. The generated grammar is the source code to a parser generator, which produces the template language parser. The MetaJ environment uses this parser to build both the template body and the base program syntax trees. The current implementation is a façade for the Cup parser generator [Hudson, 1999].

The grammar generated is an extension of the base language grammar. It has new productions that allow meta-constructions to be combined with base language constructions. There are two kinds of new productions: metavariables productions, which allow declaration of metavariables, and productions that define the usage of optional marks. In the former case, the user specifies information about which types of metavariables are allowed. In latter case, the decision on which base language structures can be delimited by optional marks is made through the detection of nullable symbols, i. e., symbols that can produce an empty string (λ).

Some base language grammar adjustments may be necessary in order to let the user achieve the desired metalanguage grammar. The usual kinds of adjustments are: addition and deletion of nonterminal symbols and the restructuring of some productions of the base language's grammar. These adjustments simplify template definition by allowing new useful types of metavariables or by removing useless metavariables types. In the case of Java, these adjustments have proved to be very simple. Basically, it was necessary to add the new production (`identifier` $\rightarrow$ `IDENTIFIER`), and to replace all ocurrences of the terminal `IDENTIFIER` by `identifier`. This adjustment allows the declaration of metavariables of type `identifier` to occur in some Java structures like class signatures (see Listing 4), declarations of fields and methods.

The MetaJ framework encapsulates all MetaJ concepts that are not base language specific. Concepts which depend on the base language should be provided by the *plug-ins*. The framework provides abstract and concrete base elements used to develop plug-ins and metaprograms. The main generic operations carried out by the framework are matching and printing.

The matcher is the component which verifies if a base program is in accordance with a pattern encapsulated by a template. The result of this operation is a table that defines a value for each metavariable of the pattern. Metavariables values can be accessed by means of references.

The matching operation acts on two parameters: the pattern p (a template body) and the input base language code c. The matcher starts aligning the beginnings of p and c, and follows comparing each reached structure from p with the corresponding structure of c. The pattern p guides the matching process. The matcher determines its next action based on the kind of the current structure of p, so that:

1. If the current structure of p is a base language fragment of code and this fragment of code occurs on the current position of c, then the matching process continues on the next structure of p and c; otherwise a matching error occurs;

2. If the current structure of p is delimited by optional marks then the matching process proceeds without this structure. If matching the following segment results in error, then matching is tried again using the delimited structure;

3. If the current structure of p is a simple variable, then the compatibility

between the type of the p current structure and the type of the c current structure is verified. If they are compatible, then the matching follows verifying next structures of p and c;

4. If the current structure of p is a list variable, then the smallest number of structures of c, which are compatible with the type of current structure of p, and succeeding the matching continuation, are matched with the list. If no structure is matched, then a matching error occurs.

The printer is capable of rebuilding the program source code from a syntax tree. The print operation expects only one parameter: the pattern or base language code syntax tree to be printed. If the tree represents a base language code, then the printer just prints its corresponding code. When the tree is a pattern, then all occurrences of metavariables are replaced by their respective value. It goes throughout the syntax tree and takes its decisions about what to do based on the kind of reached structure:

1. If the current structure is a base language fragment of code, it is converted to its textual representation. The printer outputs each token stored in the subtree of the structure.

2. If the current structure is a metavariable (simple or list), then its value is printed.

3. If the current structure is delimited by optional marks, then it is printed, if and only if, all metavariables used in this structure have a defined value. Optional marks are not printed.

If any metavariable without a defined value is reached, and if it is not delimited by optional marks, then a printing error is issued.

Plug-ins implement the language specific interfaces and abstract methods of the framework. The framework components related to the plug-in construction are: template language parser interface and syntax tree nodes. The framework does not define a specific parser. It just provides an interface which should be implemented by the template language parser. The most important plug-in component is the template language parser. This parser should build syntax trees for templates bodies and base language programs using the syntax tree nodes provided by the framework. The template language parser uses these elements to build syntax trees. There are three kinds of nodes: (1) simple nodes, which are used to represent base code fragments; (2) variable nodes, which represent metavariables occurrences, and (3) optional marks, which are nodes that mark optional fragments of code.

A MetaJ program is a Java program that uses MetaJ components. Templates are translated into Java classes, so they could be accessed in the metaprogram.

Although the template compiler depends on the template language parser, the plug-in developer does not implement it. The template compiler is a language-independent compiler that uses the template language parser interface to access the parser implemented by the plug-in developer aided by the grammar processor tool. The template compiler is an automatic way to implement the abstract methods of the class `AbstractTemplate` provided by the framework. The concrete methods of this class act as a front-end of the framework. They hide details about how to use the framework operations. In addition to implementing the abstract methods of the class `AbstractTemplate`, the compiler generates methods to access a reference to the value of each template metavariable (see Section 2).

4 MetaJ Validations

MetaJ has been applied in the development of a few applications. Here are shortly presented some examples that range over analysis, transformation and generation of programs.

4.1 Metrics Collector

Any software tool that requires source code analysis usually produces, internally, a suitable representation for the code. Figure 2 shows a class diagram as an example of how one could model a Java system. This representation usually

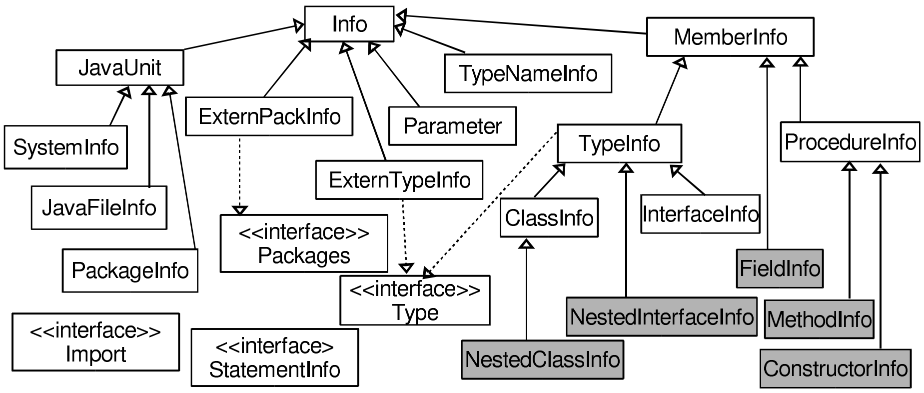

Figure 2: A possible (partial) metamodel for Java systems

requires more expressiveness than that already available in a syntax tree. For instance, for program slicing tools, it is necessary to have a control flow graph

[Tip, 1995]. Another example could be an expert tool for detecting specific refactorings, which represent opportunities for enhancing the internal quality of a system. A required infrastructure to develop this kind of tool is a collector of system metrics that enables the definition of a quality function for the system code. Most of the implementation of such a collector can be driven by the syntactic representation of the code. In this case, MetaJ appears to be a useful environment to extract the skeleton of the code. This skeleton could be represented by the metamodel shown in Figure 2. In Listing 2, it is presented a method that given any possible class member identifies member's type and creates the respective concrete object for that member (one of the possible concrete classes that extend the `MemberInfo` class, colored in Figure 2). Matching the member with an appropriate template for each possibility identifies the member kind (nested class or interface, field, method or constructor). The class `FieldInfoCollector` creates an object `FieldInfo` and updates it with the information extracted with other appropriate templates.

```
public class MemberInfoCollector {
  private String member;
  private MemberInfo memberInfo;
  public void execute() {
    VerifyFieldDecl tvfd = new VerifyFieldDecl();
    VerifyMethodDecl tvmd = new VerifyMethodDecl();
    VerifyConstructorDecl tvcd = new VerifyConstructorDecl();
    VerifyNestedClassDecl tvncd = new VerifyNestedClassDecl();
    VerifyNestedInterfaceDecl tvnid =
                                 new VerifyNestedInterfaceDecl();
    // FieldDeclaration
    if (tvfd.match(member)) {
      FieldInfoCollector fic = new FieldInfoCollector(member);
      fic.execute();
      memberInfo = fic.getFieldInfo();
    }else
        // MethodDeclaration
        if (tvmd.match(member)) {
          ... //idem
  } ...
}
```

Listing 2: A method that collects information of a class member

Constructing metrics collectors from the scratch is not a simple task. The developer should build the parser (using a parser generator), to represent internally the program (using (abstract) syntax trees) and to extract code informations by exploring the internal representation (using visitors [Gamma et al., 1995]). MetaJ encapsulates this task by providing a standard way to represent and explore programs. Allowing code pattern matching makes easier to write and understand the process of collecting source code information. Patterns written by example are more readable and easier to maintain than other common approaches used to extract code information (such as visitors or semantics actions added to the parser). In addition, MetaJ abstractions hide the program's internal representation and provide operations to traverse (using code iterators), decompose and check (using pattern-matching) the source code.

4.2 Programmable Refactoring

Refactorings are semantics-preserving transformations on the source code [Opdyke, 1992] [Fowler, 1999]. They are used to enhance the internal quality of the system. It is commonly accepted that refactoring tools are extremely important when it is necessary to proceed with a considerable system reengineering. There are already some tools that provide built-in refactoring capabilities [Mens et al., 2003]. In MetaJ approach, refactorings are performed as ordinary Java classes, and thus can be composed into more elaborated transformations. They can also be reused in any part of a system and integrated into development environments that provide open APIs, such as JBuilder and Eclipse. Listing 3 shows an adapted version of the *Move Field* refactoring [Fowler, 1999]. A field will be moved from a source class to a target class, and the dependences of this movement will be properly arranged. The instance variables are configuration parameters of the refactoring. There are certain preconditions that must be satisfied, for example, the field to be moved must exist in the source class.

This precondition is verified by matching the `ClassWithField` template shown in Listing 4 with the input file containing the source class. Then, the field is encapsulated in the source class. This implementation potentially writes the modified source class to a different file, depending on the values of the respective instance variables. For this task it is called another refactoring, the `EncapsulateField`.

The template for (re)writing a file used in the `EncapsulateField` refactoring is shown in Listing 5. After encapsulating the field in the source class, the bodies of the newly created methods are modified to access the field using an instance of the target class. The transformation `RedirectMeth` is responsible for this action. Note that a new instance variable of the target class is created, if no one is encountered. Next, the field is removed from the source class, but not the get and set methods. After that, the field and corresponding get and set methods are inserted in the target class using the class `AddEncapsulatedField`. Finally, it is checked if there is any instance variable in the target class with the

```
public class MoveField {
  String isf, osf; // Input/Output file with source class
  String itf, otf; // Input/Output file with target class
  String sc, tc;   // Source/Target class
  String fn;       // Field name
  public void execute() throws ActionException {
    ClassWithField vf = new ClassWithField();
    vf.setClassName(sc); vf.setFieldName(fn);
    if (vf.match(...isf...)){
      // If the source class has the field
      EncapsulateField ef = new EncapsulateField(isf,osf,sc,fn);
      ef.execute();
      RedirectMeth rm = new RedirectMeth(osf,osf,sc,
                                         ef.getSetMeth(),tc);
      rm.execute();  // Update Set method body
      rm.setMethodName(ef.getGetMeth() );
      rm.execute();  // Update Get method body
      RemoveField rf=new RemoveField(osf,osf,sc,fn);
      rf.execute();
      AddEncapsulatedField aef = new AddEncapsulatedField(...);
      aef.execute();
      vf.unBindAll();
      vf.setClassName(getIdTC());
      vf.setType(getType(sc));
      if (vf.match(...otf...)) {
         //If there is field of source type in the target
         new RedirectClass(otf,otf,tc,sc,...).execute();
      }
    }else
       throw new ActionException("Field not found in " + sc);
   } ...
}
```

Listing 3: Java class for the Move Field refactoring

type of the source class. If so, all accesses to the recently moved field qualified by that variable must be updated by means of the proper method call. The transformation `RedirectClass` is responsible for this action.

Implementing the proposed refactoring *MoveField* is not a very simple task even using MetaJ features. But, using code iterators, pattern matching and MetaJ abstractions, analysis and code generation become easier and safer (no invalid syntactic construction is generated) than doing that by using tree visitors or an open abstract syntax tree library, and directly accessing the internal representation of the program. MetaJ components allow the manipulation of source code without explicitly handling the internal representation.

```
template #CompilationUnit ClassWithField {
   #[#PackageDeclaration:pck ]#
   #[#ImportDeclarationList:ids]#
   #[#TypeDeclarationList:tds]#
   #[#ClassModifiers:cm]# class #Identifier:className
                  #[#Extends:ce]# #[#Implements:impc]#{
       #[#ClassBodyDeclarationList:cbds]#
       #[#FieldModifiers:fm]# #Type:type #Identifier:fieldName
                                  #[#VariableInitializer:vi]#;
       #[#ClassBodyDeclarationList:cbds2]#
   }
   #[#TypeDeclarationList:tds2]#
}
```

Listing 4: Template representing a class with a field

```
template #CompilationUnit ClassWithFieldAndGetAndSet {
   #[#PackageDeclaration:pck]#
   #[#ImportDeclarationList:ids]#
   #[#TypeDeclarationList:tds]#
   #ClassSignature:cs {
       #[#ClassBodyDeclarationList:cbds]#
       #[#FieldModifiers:fm2]# #Type:type #fieldName
                                #[#VariableInitializer:vi]# ;
       #[#ClassBodyDeclarationList:cbds2]#
       public void #setMethod (#Type:type arg){
             #fieldName = arg;
       }
       public #Type:type #getMethod (){ return #fieldName; }
    }
    #[#TypeDeclarationList:tds2]#
}
```

Listing 5: Template used in the Encapsulate Field Refactoring

4.3 Generative Programming

This case study presents the implementation of a generative domain model [Czarnecki and Eisenecker, 2000] that constructs data-driven applications from simplified entity-relationship specifications. A framework instantiation is generated from configuration parameters written with a domain-specific language. The generation process is implemented using MetaJ templates.

4.3.1 The Application Framework

Firstly, it will be presented a framework for four-tier database applications. Since a generated application is a specialization of this framework, the architectures of both the application and the framework are the same. The framework defines abstract classes that are to be extended by concrete generated classes, which implement application specific behaviors for template methods declared in framework abstract classes. The applications generated follow a four-tier architecture: presentation (user interface), logic, communication, and data access. The generation process implemented in the generative domain model is divided in two parts: logic and data access generation process and presentation generation process.

In this case study, some constraints are imposed by the generative domain model. The model only generates desktop applications with predefined user interface. New applications are generated from a simplified entity-relationship specification and are constructed in a four-tier architecture: presentation, logic, communication and data access tiers. Generated applications accesses a relational database.

Design patterns [Gamma et al., 1995] are used in all layers of both application and framework, so the generated system is expected to have satisfactory internal quality, and thus can be manually customized.

The configuration knowledge is specified in a domain-specific high-level configuration language. From the above restrictions, it was introduced just a data configuration language. Below, it is shown an example of specification. It is supposed that a corresponding database with three tables has already been created. Figure 3 shows the overall architecture of framework specialization generators.

```
Entities:
    Client(code:Integer; name:String;
                              address:String; age:Integer)
    Account(number: Integer; balance: Real)
Relationships:
    AcountClient: One to many (1..*) from Client to Account
    Fields: code:Integer; number:Integer;
```

Algorithmic generation methods receive, externally, parameters that represent the configuration knowledge. These methods define values for the metavariables of associated rules and use each `print` template operation to generate code.

The requirement specification is translated into a method which calls the algorithmic generation methods, shown in Listing 6. Since there is no way to specify user interface layout, some standard behaviors were defined. For each entity, a form is generated. Text fields representing each entity field compose this form. There is a navigation panel to browse the entities. Relationships result on

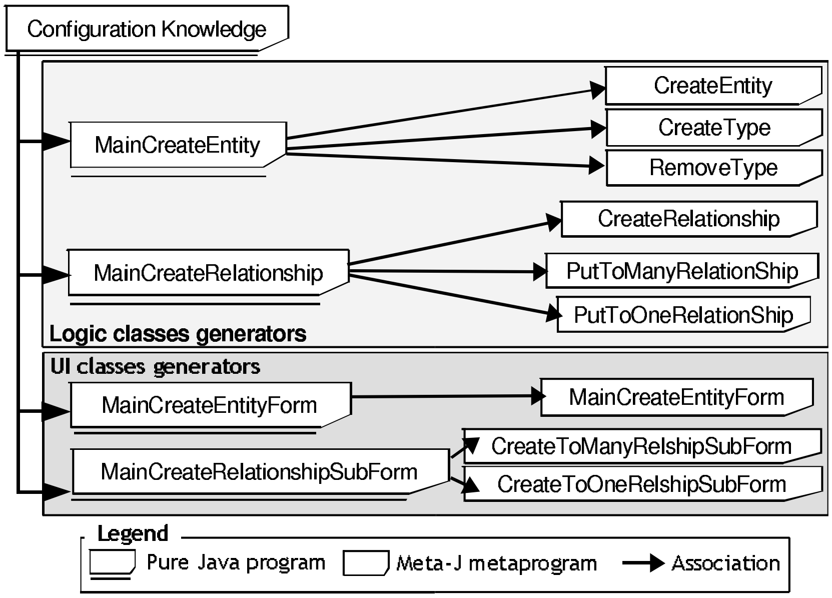

Figure 3: The architecture of the generators

insertion of a child entity form into the parent entity form. In Listing 7, it is shown classes which configure and call the generators. The class `EntityBuilder` is the façade that, in fact, is the implementation of a high-level specification language. This class encapsulates the calls to the generators. A generator is composed of a class that instantiates and assigns values to all manipulation variables (e.g. `ComposedCreateEntity` in Listing 7), and a rule that can be seen as a template of the generated source file (e.g. `CreateEntity` in Listing 8). This harder and complex problem shows the scalability of MetaJ expressive power.

5 Related Work

Traditional compiler generation tools such as Cup [Hudson, 1999] could be seen as a starting point for MetaJ. These tools free programmers from the burden of having to worry about the intricacies of parsing algorithms. Nonetheless, they are still limited to help the verification of syntax rules and the syntax tree construction. MetaJ provides higher-level abstractions for manipulating source code. Indeed, MetaJ concepts are not entirely new. There have been developed several systems that provide facilities for metaprogramming. Some of them are self-contained metaprogramming systems based on the rewriting paradigm. TXL [Cordy et al., 1988], Refine [Kotik and Markosian, 1989], and ASF+SDF

```
public static void main(String[] args)throws ActionException{
  EntityBuilder client = new EntityBuilder ("Client");
  client.setBdUrl ("jdbc:cloudscape:rmi:DataBank");
  client.addField("Integer","code",20,true);
  client.addField("Integer","age",20,false);
  client.addField("String","name",20,false);
  client.addField("String","address",20,false);
  client.createEntity();
  EntityBuilder account = new EntityBuilder ("Account");
  account.setBdUrl ("jdbc:cloudscape:rmi:DataBank");
  account.addField("Integer","number",20,true);
  account.addField("Integer","balance",20,false);
  ...
  account.createEntity();
  //Relationships and forms
  client.createEntityToManyRelationship(account);
  account.createEntityToOneRelationship(client);
  client.createEntityForm(); account.createEntityForm();
  client.createEntityToManyRelationshipSubForm(account);
}
```

Listing 6: Code Generator for High-Level Specifications

```
class EntityBuilder{
   public void createEntity ()throws ActionException {
     ComposedCreateEntity cce = new ComposedCreateEntity ();
     cce.setEntityName (entityName);
     cce.setEntityDAOName(getEntityDAOName());
     cce.setEntityDataInterface(getEntityDataInterfaceName());
     cce.setEntityTableName(getEntityTableName());  ...
     cce.setFields (fields);
     cce.execute();
   }...
}
class ComposedCreateEntity {
   public void execute ()throws ActionException {
     AbstractTemplate create = new CreateEntity ();
     if(entityName!= null && !entityName.equals(""))
       create.setEntityName(entityName);
     else throw new ActionException("Name non specified.");
     // set manipulation variables of the CreateEntity Template
     ... create.print(); // generate the file
   }...}
```

Listing 7: Some methods for calling generators and the CreateEntity façade

```
template #CompilationUnit CreateEntity {
  package #Name:entitySchema;
  #ImportDeclarationList:entityDependencies
  import generation.*;
  import java.sql.*;
  public class #entityName #ClassExtends:entExt
                                   implements Entity{
    private #entityDAOName #entityDAOId;
    private #entityName(#eDataName _entidade,
                                     #entityDAOName _edn){
      #setEntity (_entidade);
      #entityDAOId  = _edn;
    }
    #ClassBodyDeclarationList:cbdsEntityData
    public static #entityName create (#eDataName data)
                                             throws...{
      #entityDAOName dao = new #eImpDAOName ();
      dao.create (data);
      return new #entityName (data , dao);
    }
    public static #entityName[] findAll ()throws ...{
      #entityDAOName dao = new #eImpDAOName ();
      DataInterface[] data = dao.findAll ();
      if (data != null){
        #entityName[] entities =
                             new #entityName [data.length];
        for (int i = 0; i < data.length; i++)
          entities[i] = new #entityName((#eDataName)data[i],
                                       new #eImpDAOName ());
        return entities;
      } else return null;
    }
    ...
  }
}
```

Listing 8: MetaJ template for specifying an entity generator

[van den Brand et al., 2001] are examples of such systems. Even if these systems can define patterns based on predefined context-free languages and use them to define transformation rules, writing by-example patterns is not directly possible. Even so, the generality of such tools enables this possibility defining additional modules [Cordy and Shukla, 1992] [Sellink and Verhoef, 1998]. The main disadvantage of these tools is necessity of learning a new paradigm, which may be undesirable in some industrial environments.

SCRUPLE [Paul and Prakash, 1994] is a framework that uses a pattern language to define queries on the source code. Pattern languages can be derived extending the base language with pattern-matching symbols, such as wildcards, set variables, sequence variables. The pattern language design seems to be carried in an ad-hoc basis since it includes some semantic dependent features, which may cause difficulties when extending the framework for different languages. Set variables, named wildcards and matching equivalent statements are examples of such semantic dependent features. Another example of framework for source code analysis is Genoa [Devanbu, 1999]. Both SCRUPLE and Genoa only provide mechanisms for querying the code, allowing neither transformation nor generation, differently from MetaJ.

A* [Ladd and Ramming, 1995] and TAWK [Griswold et al., 1996] are pattern-action languages that extend the lexical pattern syntax of AWK, saving the programmer the effort of emulating parsing with regular expressions. A* has mechanisms for specifying the order (preorder, postorder) of the implicit loop for traversing trees. Also, matches can be interleaved with actions. Wildcards and variables are missing in A*. TAWK is built on top of the Ponder toolset [Griswold and Atkinson, 1995], which provides facilities for manipulating AST's. Extending TAWK to a new base language requires retargeting Ponder. Both A* and TAWK take the consequences of being embedded in a untyped language such as AWK, advantageous for rapid prototyping, but unsafe for large projects.

JPearl is a pattern-action language for Java [Maia and Oliveira, 2002]. It defines two specialized languages for describing restructurings of Java programs. A primitive transformation is described in a rule language, which has four optional sections: input pattern, input actions, output pattern, and output actions. A composed transformation is described in a language, which is a mixture of Java code and mechanisms to instantiate and execute primitive transformations and to access its pattern variables. Furthermore, JPearl is too verbose and cannot be extended for other base languages.

6 Conclusions

MetaJ, at first sight, may appear to be a simple metaprogramming utility. Indeed, we expect this feature to be a positive one in the sense that it confirms its easiness of use. But also, the applications developed in MetaJ environment show that reasonably large applications can be developed with less effort than using traditional compiler generation tools, and thus confirming its expressiveness. However, MetaJ is still far from being a definitive metaprogramming tool. Its simple and elegant design is charged on the absence of highly expressive features present in fully featured metaprogramming systems such as TXL, Refine and ASF+SDF. In MetaJ, much of the metaprogramming work is done in Java. This can be an advantage if we consider that it can be applied all successful knowledge acquired developing reusable, robust object-oriented system, but also can be seen as a disadvantage if we consider that the metaprograms are mostly written in a imperative style, to the detriment of the declarative style of templates. We expect to introduce such declarative features to MetaJ.

References

[Cameron and Ito, 1984] Cameron, R. and Ito, M. (1984). Grammar-based definition of metaprogramming systems. *ACM Transactions on Programming Languages and Systems*, 6(1):20–54.

[Castor and Borba, 2001] Castor, F. and Borba, P. (2001). A language for specifying java transformations. In *VI Brazilian Symposium on Programming Languages*, pages 236–251, Curitiba, Brazil.

[Cordy et al., 1988] Cordy, J., Halpern, C., and Promislow, E. (1988). TXL: A rapid prototyping system for programming language dialects. In *Proc. of Int'l Conf. of Computer Languages*, pages 9–13.

[Cordy and Shukla, 1992] Cordy, J. and Shukla, M. (1992). Practical metaprogramming. In *Proc. of the IBM Centre for Advanced Studies Conference*, pages 215–224.

[Czarnecki and Eisenecker, 2000] Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming*. Addison-Wesley.

[Devanbu, 1999] Devanbu, P. (1999). GENOA - a customizable, front-end-retargetable source code analysis framework. *ACM TOSEM*, 8(2):177–212.

[Fowler, 1999] Fowler, M. (1999). *Refactoring-Improving the Design of Existing Code*. Addison-Wesley.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object Oriented Software*. Addison Wesley.

[Griswold and Atkinson, 1995] Griswold, W. and Atkinson, D. (1995). Managing design trade-offs for a program understanding and transformation tool. *Journal of Systems and Software*, 30:99–116.

[Griswold et al., 1996] Griswold, W., Atkinson, D., and McCurdy, C. (1996). Fast, flexible syntactic pattern matching and processing. In *WPC '96: Proceedings of the IEEE Fourth Workshop on Program Comprehension*, pages 144–153. IEEE Computer Society Press.

[Hudson, 1999] Hudson, S. E. (1999). Cup LALR parser generator for Java – User's Manual. *Available at http://www.cs.princeton.edu/ appel/modern/java/CUP/manual.html.*

[Klint, 2003] Klint, P. (2003). How understanding and restructuring differ from compiling: a rewriting perspective. In *Proc. of the 11th Int'l Workshop on Program Comprehension (IWPC03)*, pages 2–12.

[Kotik and Markosian, 1989] Kotik, G. and Markosian, L. (1989). Automating software analysis and testing using a program transformation system. *ACM SIGSOFT Software Engineering Notes*, 4(8).

[Ladd and Ramming, 1995] Ladd, D. and Ramming, C. (1995). A*: A language for implementing language processors. *IEEE Transaction on Software Engineering*, 21(11):894–901.

[Maia and Oliveira, 2002] Maia, M. and Oliveira, A. (2002). JPearl - a language for defining Java program restructurings (in portuguese). In *VI SBLP*, pages 166–179, Rio de Janeiro, Brazil.

[Mens et al., 2001] Mens, K., Michiels, I., and Wuyts, R. (2001). Supporting software development through declaratively codified programming patterns. *Journal on Expert Systems with Applications*, 234(4):405–413.

[Mens et al., 2003] Mens, T., Demeyer, S., Bois, B. D., Stenten, H., and Gorp, P. V. (2003). Refactoring:current research and future trends. In Bryant, B. and Saraiva, J., editors, *Electronic Notes in Theoretical Computer Science*, volume 82. Elsevier.

[Opdyke, 1992] Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.

[Oppen, 1980] Oppen, D. (1980). Prettyprinting. *ACM TOPLAS*, 2(4):465–483.

[Partsch and Steinbrüggen, 1983] Partsch, H. and Steinbrüggen, R. (1983). Program transformation systems. *ACM Computing Surveys*, 15(3):199–236.

[Paul and Prakash, 1994] Paul, S. and Prakash, A. (1994). A framework for source code analysis using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475.

[Rugaber, 1995] Rugaber, S. (1995). Program comprehension. In Dekker, M., editor, *Encyclopedia of Computer Science and Technology*, pages 341–368.

[Sellink and Verhoef, 1998] Sellink, M. P. A. and Verhoef, C. (1998). Native patterns. In *Proc. 5th Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Society Press.

[Sheard, 2001] Sheard, T. (2001). Accomplishments and research challenges in meta-programming. In *Proc. of 2nd Intl. Workshop on Semantics, Applications, and Implementation of Program Generation, LNCS 2196*, pages 2–44, Italy.

[Tip, 1995] Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):11–189.

[van den Brand et al., 2001] van den Brand, M., van Deursen, A., Heering, J., de Jong, H. A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P. A., Scheerder, J., Vinju, J. J., Visser, E., and Visser, J. (2001). The ASF+SDF meta-environment:a component-based language development environment. In *Computational Complexity*, pages 365–370.

[Visser, 2002] Visser, E. (2002). Meta-programming with concrete object syntax. In *Generative Programming and Component Engineering (GPCE'02), LNCS 2487*, pages 299–315.