MetaJ: An Extensible Environment for Metaprogramming in Java

Ademir Alvarenga de Oliveira[1] **, Thiago Henrique Braga**[2] **,**
Marcelo de Almeida Maia[2] **, Roberto da Silva Bigonha**[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
Campus da Pampulha – 31270-010 Belo Horizonte, MG

[2]Departamento de Computação – Universidade Federal de Ouro Preto
Campus Morro do Cruzeiro – 35400-000 Ouro Preto, MG

{ademirao,bigonha}@dcc.ufmg.br, {thiagohb,marcmaia}@iceb.ufop.br

Abstract. *MetaJ is an environment that facilitates metaprogramming using the Java language. The environment is designed to be extended with plug-ins that can manipulate programs written in different languages. This requirement have influenced MetaJ design since the facilities of the environment concern only syntactical aspects. Semantics aspects are language-dependent and are not treated here, but could be tackled with other tools, which could even be layered in 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.*

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 every concrete entity one can imagine, it is possible to write programs to reason about everything, even another programs. 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, refactoring, program comprehension, program optimization, partial evaluation, program verification, type inference/checking, design patterns detection/application, program slicing [Sheard, 2001] [Partsch and Steinbrüggen, 1983] [Oppen, 1980] [Mens et al., 2001] [Tip, 1995] [Rugaber, 1995].

In principle, a metaprogram can be written in any general-purpose programming language. Generally, the programs being manipulated are represented internally as (abstract) syntax trees, which are either registers 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 utilize 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, object hierarchies, and module systems to hide irrelevant details). These features add considerably to the complexity of the languages they are embedded in, but are generally considered 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 metalevel languages is extending a general-purpose language, which has, generally, 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, it is more difficult to learn a new language rather 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.

In [Cordy and Shukla, 1992], it is highlighted two difficulties concerning the metaprogramming process: the lack of a general approach for developing metaprogramming languages, and the difficulty in learning metalanguages. In [Klint, 2003], it is argued that despite the resemblance between compilation, restructuring and comprehension of programs, the last two processes should not be approached with the well-known techniques developed for the first one. Some differences can be pointed out. Compilation deals, generally, with only one source language, while restructuring and comprehension may involve several languages. A compiler can abstract from the source code as soon as it becomes available. On the other hand, in restructuring it is necessary to keep a link between the abstraction and the source code because the latter should 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, below it is pointed out some requirements, which such a system should satisfy, and that has guided MetaJ design:

- Expressiveness and readability. The main goal of a metaprogramming tool is to facilitate a hard task of metaprogramming. So, the tool should provide abstractions that hide the internal representation of the source program.
- Flexibility. the tool should support analysis, generation, manipulation and transformation.
- Easiness of learning. according to Cordy [Cordy and Shukla, 1992], one of the reasons why metaprogramming is not largely used is that most tools are hard to learn.
- Definition of program patterns *by example*. In this case, the tool should support the matching of source program fragments.
- Consistency. When generating, manipulating, or transforming a program, it is necessary to guarantee that the target program satisfies at least the syntactic properties of the base language.
- Multiple base languages. Software systems artifacts are usually written with more than one language. So, the tool should be able to accept new base languages on demand.

In the rest of the paper, it is presented MetaJ, an extensible framework for developing metaprograms in Java. In Section 2, it is shown a general picture of the environment,

its design decisions, and how it can be used. In Section 3 it is discussed the implementation details. In Section 4, it is shown 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 is an object-oriented framework that defines a set of concepts that are independent of the base language: syntax trees, code references, and code templates. However, each instance of each concept is dependent on a specific base language. The framework supports this independence by isolating the features common to any language and defining generic operations for them and besides, it 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 Prograams

Base programs are represented as syntax trees. All node types are derived from the non-terminals of the base language grammar. An important consideration is that this grammar may need a reformulation to include all important node types as non-terminals, i.e., the node types that the metaprogram may have access. In the case of JavaCC grammar of Java, these changes were few and trivial.

All nodes are accessed with an interface `Reference`. Thus, references are used to manipulate fragments of base language code. References hide the tree nodes and provide only operations that preserves the syntax consistence of the resulting tree. Tree traversal is possible using iterator objects. They implement the classical design pattern. During the traversal it is possible to use reference operations to test or modify a fragment of base code.

The most important methods of the `Reference` interface are:

- `String print();` Returns the corresponding source code of the node sub-tree
- `boolean match(Reference);` Structural comparison of sub-trees
- `Iterator getIterator();` Returns an iterator to traverse the node sub-tree
- `void set(Reference);` Updates the value of a reference
- `String toString();` Returns the corresponding code fragment of a reference

2.2. Variables and Templates

A first requirement for a metaprogramming language is providing high-level abstractions that hide the internal representation of base programs. MetaJ enables 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 template language 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. The type of a template must be one of the node types defined from 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 non-terminal that defines the respective type of that template. A sentential form can be written with appropriate terminals, code variables and delimiters of optional sentential sub-forms.

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

A code variable stores a reference for a code fragment. It has a type which must be one of the node types, just as templates. The types of code variables are classified as a *single-valued* type or a *multivalued* type, the latter ended with suffix `List`. For example, a `ImportDeclaration` denotes a type that references one import declaration. An `ImportDeclarationList` denotes a type that references multiple import declarations, and has additional operations for inserting and removing elements.

An optional sentential sub-form is acceptable only if the original grammar of the base language allows it to be optional.

A template is translated into a Java class, which has the same name of the template. In Listings 1 and 2, it is shown an example of a template translation. The template MyTempl has type `CompilationUnit`; has an optional `PackageDeclaration` which can be bound to any reference with type `PackageDeclaration`; has a class with signature exactly equals the fragment with tokens `class` followed by `MyTempl`.

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

Listing 1: An example of template

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

Listing 2: Java class generated from template of Listing 1

Each template can be arbitrarily instantiated. Each instance of a template has its own environment, i.e., its own binding from code variables to references. The code variables occurring in templates are available in the corresponding template instances – besides, an API is available to manipulate the template. The most important methods are:

- `String print();` 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 code variables to `Reference` objects on the tree. If a code variable is already bound, it verifies if the corresponding references match.
- `Iterator getIterator();` Returns an iterator to traverse the template tree
- `void setXXX(Reference);` These methods are available for each code variable in the template, where `XXX` is the name of the code variable with its first letter capitalized.
- `void addInXXX(Reference, int);` Idem as previous, but adds a child reference in a multivalued code variable named `XXX`.

In Section 4, it will be presented examples of how templates may be used.

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 major components and their respective interdependence.

Therefore, there are two kinds of MetaJ users: the metaprogrammer and the plug-in developer. The metaprogrammer is the final user of the environment. A MetaJ metaprogram developed by this user is a Java program that uses framework abstractions to manipulate the base code.

Designing and building a plug-in is not a trivial task, mostly because of the meta/base language implementation. Even using a parser generator does not simplify the task. These difficulties create the necessity of a tool to help the *plug-ins* developer. QUAIS DIFICULDADES??? ACHO MELHOR TIRAR ESTE PARAGRAFO OU COLOCA-LOS COMO UM TIPO DE CONCLUSAO.

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 will be the source code to a parser generator, which will produce the template language parser. The MetaJ environment will use this parser to build the both the template body and the base program syntax trees. The current implementation uses the Cup parser generator.

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

It is important to highlight that some base language grammar adjustments may be necessary in order to the user to achieve the desired metalanguage grammar. However, in the case of Java, these adjustments have proved to be very simple.

The MetaJ framework encapsulates all MetaJ concepts containing no base language specific information, which 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:

1. 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 using references.
   The match operation expects two parameters: the pattern p (a template body) and the input base language code c. The matcher starts aligning the beginning of p and c, and follows comparing each reached structure of p with the reached structure of c. The pattern p guides the matching process. The matcher defines its next action based on the kind of the current structure of p, so that:
   (a) 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

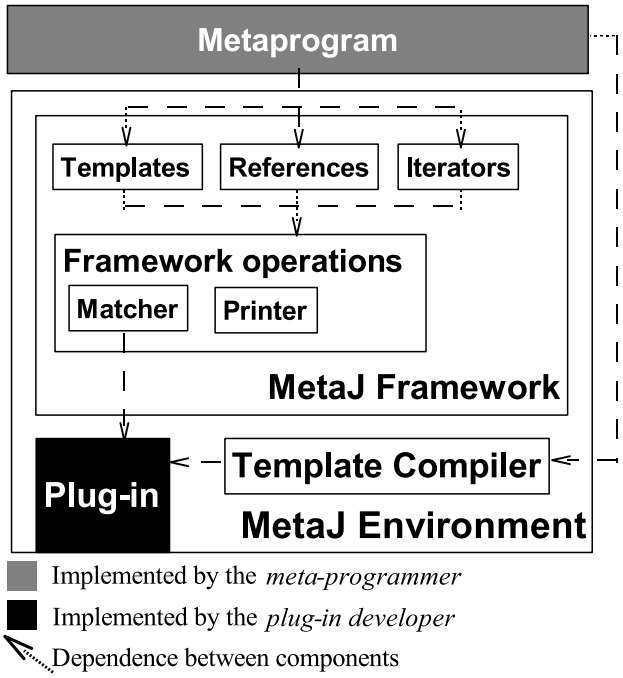

Figure 1: The relationship between MetaJ components

process continues on the next structure of p and c; otherwise a matching error occurs;

(b) If the current structure of p is delimited by optional marks then the matching process tries to continue without this structure. If matching the following segment results in error, then matching is tried again using the delimited structure;

(c) 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;

(d) 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.

2. 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 makes its decisions about what to do based on the kind of reached structure:

   (a) If the current structure is a base language fragment of code, it is converted to its textual representation. The printer prints each token stored in the subtree of the structure to perform this operation.

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

   (c) If the current structure is delimited by optional marks, then it is printed if, and only if, all meta-variables used in this structure have a defined value. Optional marks are not printed.

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

Plug-ins will implement the language specific interfaces and abstract methods of the framework. The framework components related to the plug-in construction are listed below:

1. Template language parser interface. 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.
2. Syntax tree nodes. The template language parser will use 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 represents metavariables occurrences, and (3) optional marks, which are nodes that mark optional fragments of code.

A MetaJ program is a Java program that use MetaJ components. The decision to implement this link was compiling templates are to 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 and set the value of each template metavariable, as it has been shown in Section 2.

4. Applications

MetaJ has been applied in the development of several 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 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 3, it is presented a method that given any possible class member, it identifies member's type and creates a respective concrete object for that member (one of

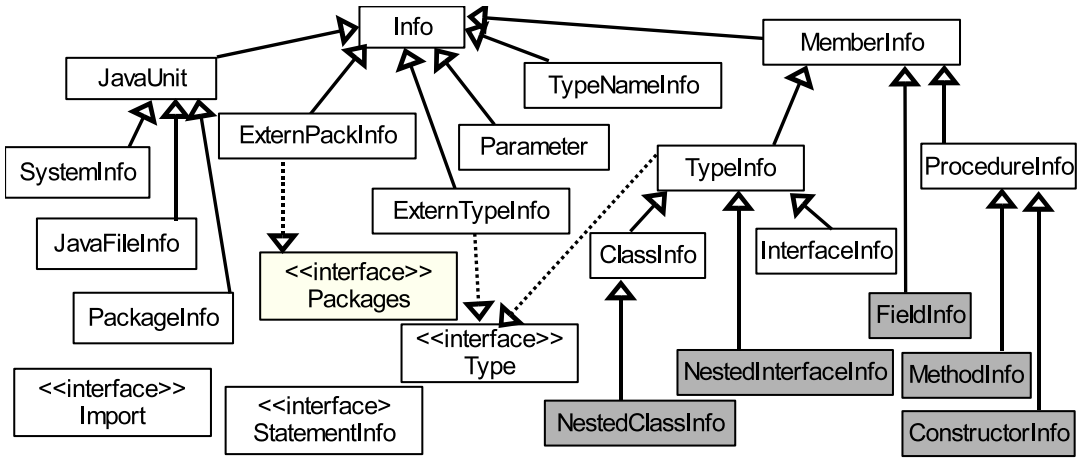

Figure 2: A possible (partial) metamodel for Java systems

the possible concrete classes that extend the `MembroInfo` class, colored in Figure 2). Trying to match the member with an appropriate template for each possibility carries out the identification. The template for simply verifying if a member is a field declaration is shown in Listing 4. The class `FieldInfoCollector` creates an object `FieldInfo` and updates it with the information extracted with other appropriate templates. The whole collector iterates the all the system using appropriate collectors.

```
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();
    if (tvfd.match(member)) { // FieldDeclaration
      FieldInfoCollector fic = new FieldInfoCollector(typeMember);
      fic.execute();
      memberInfo = fic.getFieldInfo();
    }
    else if (tvmd.match(member)) { // MethodDeclaration
      ... //idem
  } ...
}
```

Listing 3: A method that collects information of a class member

```
template #ClassBodyDeclaration VerifyFieldDecl {
  #FieldDeclaration:fd
}
```

Listing 4: A template for verifying a field declaration

4.2. Programmable Refactoring

Refactorings are semantics-preserving transformations on the source code[Opdyke, 1992][Fowler, 1999]. It aims to enhance the internal quality of the system. It is commonly agreeded 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 and et.al., 2003]. In MetaJ approach, refactorings are proposed as common 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 5 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 6 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`. For the lack of space, it is not presented all transformations. The template for (re)writing a file used in the `EncapsulateField` refactoring is shown in Listing 7. 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 getter and setter methods. After that, the field and corresponding getter and setter 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 type of the source class. If so, all accesses to the recently moved field qualified by that variable must be updated with the proper method call. The transformation `RedirectClass` is responsible for this action.

```
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...) { // Verify if the source class has the field
   EncapsulateField ef = new EncapsulateField(isf,osf,sc,fn);
   ef.execute();
   RedirectMeth rm;
   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(itf, otf, tc,
                                    vf.fm, vf.type, vf.fieldname, vf.vi);
   aef.execute();
   vf.unBindAll();
   vf.setClassName(getIdTC()); vf.setType(getType(sc));
   if (vf.match(...otf...)) {//there is field of Source type in the Target
    RedirectClass rc = new RedirectClass(otf, otf,tc,sc,
                              ef.getSetMeth(), ef.getGetMeth());
    rc.execute();
   }
  }
  else throw new ActionException("Field declaration not found in " + sc);
 }
 ...
}
```

Listing 5: Java class for the Move Field refactoring

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

Listing 6: Template for 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 7: Template used in the Encapsulate Field Refactoring

4.3. Generative Programming

This case study implements a generative domain model that constructs data-driven applications from simplified entity-relationship specifications. A framework instantiation is generated from configuration parameters written with a domain-specific language.

4.3.1. The Application Framework

Firstly, it will be presented a framework for four-tier database applications. Since a generated application will be a specialization of this framework, the architectures of both the application and the framework are the same. This architecture is shown in Figure 3. The framework defines abstract classes that will be extended by concrete generated classes. The latter implements application specific behaviors for template methods declared in framework abstract classes. Generated applications will be constructed following a four-tier architecture: presentation (user interface), logic, communication, and data access. The generation process implemented in the generative domain model will be 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. Generated applications 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.

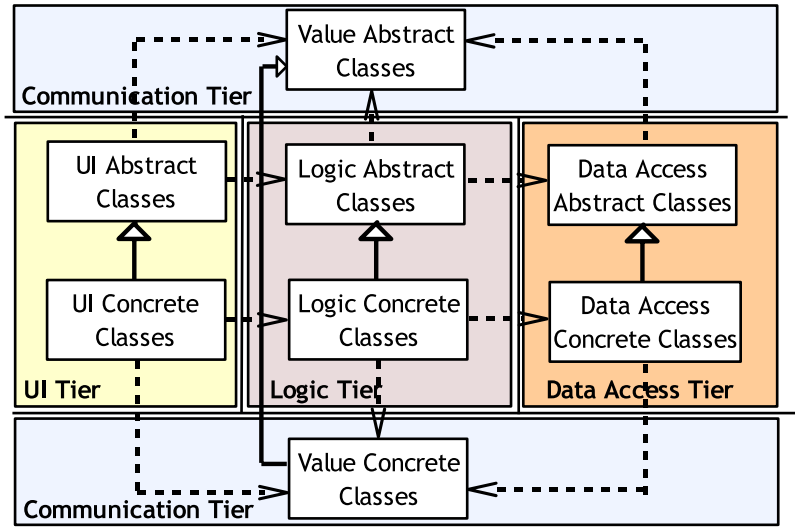

Figure 3: The architecture of both the generated application and the application framework

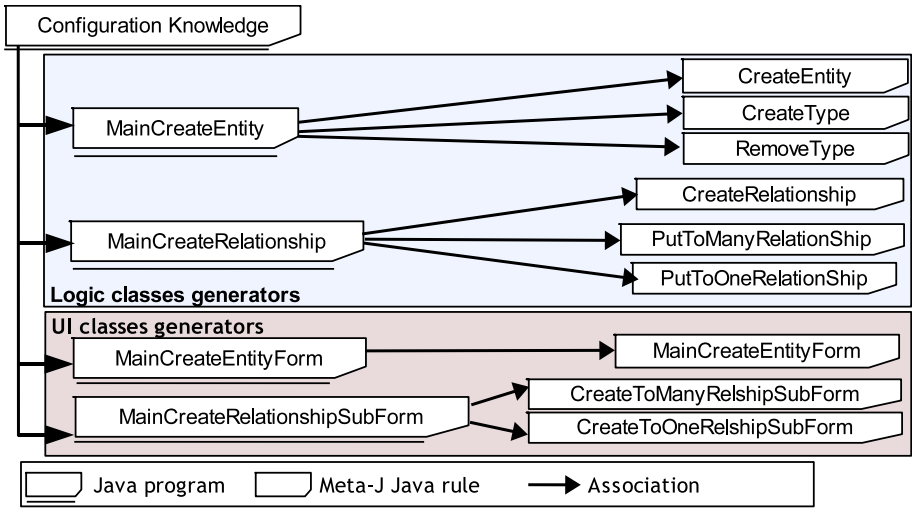

Figure 4: The architecture of the generators

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.

```
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;
```

4.3.2. The Generators

Figure 4 shows the overall architecture of framework specialization generators. Algorithmic generation methods receive, externally, parameters that represent the configuration knowledge. These methods define values for the code variables of associated rules and use each `print` template operation to generate code.

The requirement specification is translated in a method which calls the algorithmic generation methods, shown in Listing 8. 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 insertion of a child entity form into the parent entity form.

In Listing 9, it is shown the configurators of the generators. The class `EntityBuilder` is the façade that, in fact, is the realization 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 9), and a rule that can be seen as a template of the generated source file (e.g. `CreateEntity` in Listing 10).

```
public static void main(String[] args)throws ActionException{
   //-- Client Entity
   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();
   //-- Account Entity
   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
   client.createEntityToManyRelationship(account);
   account.createEntityToOneRelationship(client);
   //-- Forms
   client.createEntityForm();
   account.createEntityForm();
   client.createEntityToManyRelationshipSubForm(account);
}
```

Listing 8: 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 ("Entity name not specified.");
      ... // set other manipulation variables of the CreateEntity rule
      create.print(); // generate the file
  ...
```

Listing 9: Some methods for calling generators and the CreateEntity façade

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

Listing 10: Meta-J template for specifying an entity generator

5. Related Work

Traditional compiler generation tools such as Yacc [Johnson, 1975] (or Cup, a similar version for Java) 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 [van den Brand and 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 prohibitive 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 had been carried in an ad-hoc basis since it is included 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. Its design decisions can be considered verbose and awkward compared to MetaJ. Furthermore, JPearl cannot be extended for other base languages.

6. Conclusions

MetaJ, in a first seen, 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 with MetaJ show the reasonably large applications can be developed with less effort than using traditional compiler generation tools, and thus confirming its expressiveness. However, MetaJ is far from being an definitive metaprogramming tool. Its simple and elegant design is bought with 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 an 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 within a layered architecture, without interfering with its current design.

References

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

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

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

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

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

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, 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.

Johnson, S. (1975). Yacc: Yet another compiler compiler. Technical report, Bell Telephone Laboratories.

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

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

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, 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, T. and et.al. (2003). Refactoring: Current research and future trends. *Electronic Notes in Theoretical Computer Science*, 82(3).

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

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

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

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

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

van den Brand, M. and et.al. (2001). The ASF+SDF meta-environment: A component-based language development environment. In *Computational Complexity*, pages 365–370.

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