Iterators, Templates and Queries in Program Analysis and Manipulation

Anonymous[1]

[1]Anonymous Institute

{anonymous}@anonymous

Abstract. *Maintenance and software evolution tasks can benefit from reverse engineering and program restructuring. Tools and meta-tools for source code analysis and manipulation play an important role on these activities. However, these tools are hard to develop due to their intrinsic complexity. Better meta-tools could alleviate the challenges in developing such tools. Several meta-tools have been already proposed, but challenges still remain. This work proposes the integrated use of iterators and relational queries for supporting the construction of modules for program analysis and manipulation. These mechanisms operate on the notion of typed source code references and pattern-matching on syntax trees. The construction of a static semantics graph of Java programs based on declaration, scope and use of symbols is presented to demonstrate the effectiveness of the approach. Also, two refactorings for Java programs were implemented using both iterative and declarative styles in order to assess the performance of each style. The meta-tools shown in this paper have presented advantages over related tools and thus can be considered to help developers involved in maintenance activities.*

1. Introduction

Maintainance and software evolution have posed challenges for software developers. Software systems are still becoming increasingly complex, and during the maintainance process, systematic code restructuring is being widely adopted, aided by integrated development environments enhanced with refactoring facilities[Mens and Tourwé, 2004]. Indeed, the context can be much wider, for instance, restructuring processes may also consider evolving object-oriented systems into aspect-oriented systems.

In this scenario, tools that help developers manipulating source code systematically are welcome. Such tools are often not adequate because they do not match entirely the requirements of development or maintenance teams. Unfortunely, the construction of such customized tools for specific needs of software teams are costly and difficult. General meta-tools that help tackling the task of constructing those tools have already been extensively studied in the literature[Johnson, 1975, Cordy et al., 2002, van den Brand and et.al., 2001]. These meta-tools provides mechanisms for constructing tools for program analysis and manipulation. However, they still are not widely adopted by software engineering teams, maybe because some of them are too *low-level*, such as Yacc, and some others require learning a paradigm based on rewriting systems.

The goal of this paper is to investigate the integration of some well-known concepts of general meta-tools for program analysis and manipulation into an object-oriented language. These concepts are iterators and relational queries applied on typed source code references, and templates for pattern-matching over syntax trees.

This paper uses the MetaJ environment to provide iterators and templates, and the SCQL language to provide SQL-like queries[Oliveira, 2004, de A. Oliveira et al., 2004]. Thes proposed mechanisms can be divided in two classes: (i) declarative, such as, templates and queries, and (ii) imperative, such as visitors and iterators.

This paper presents issues related to use of imperative and declarative metaprogramming resources together, and argues that this approach has some advantages over others. Applications built using MetaJ templates and iterators and using SCQL queries are presented. MetaJ and SCQL were chosen because they are compatible with each other and can be used in the same program. The programmer decides which is more appropriate.

Next section presents the MetaJ Environment. section 3 presents the SCQL language. In section 4 related work is presented. section 5 presents a case study that brings MetaJ and SCQL together to implement a Java program model suitable for static semantic analysis. section 6 presents results of implementing Java refactoring with an iterator-based approach and a query-based approach. In section 7, this work is discussed. And finally, concluding remarks are presented.

2. The MetaJ Environment

MetaJ is an object-oriented meta-programming environment. It is implemented as an extension of the Java programming language. It has four basic concepts for meta-programming: program references (p-references), program templates, program iterators and language-dependent plug-ins. MetaJ meta-programs are Java programs that use implementations of these concepts. This design decision enables the meta-programmer to use all the knowledge of object-oriented software development to produce meta-programs. See an example of MetaJ program below.

```
// importing p-reference API and a user defined template
import metaj.framework.PReference;
import myTemplates.SelectPackageName;
public class Main {
   public static void main (...) throws ...{
      // Creating a typed p-reference
      PReference r = MetaJSystem.createPReference("java","#compilation_unit");
      // Setting a p-reference value
      r.setFile("/samples/HelloWorld.java");
      // Using a template defined by the user
      SelectPackageName spn = new SelectPackageName ();
      // Verifying if the code matches with the pattern defined by the template spn
      if(spn.match(r)){
         // Get an iterator to explore the package name
         Iterator it = spn.getPackageName().getIterator();
         // Iterating to access each identifier of the package name
         while(it.hasNext()){
            it.next();
            ...
         }
      }else{ ... // error }
  }
}
```

The code above shows a simple MetaJ program, which manipulates the Java program specified in the file /samples/HelloWorld.java. First, a p-reference to manipulate Java programs is created and its value is defined as the content of the file. After that, an instance of a user defined template (see it in section 2.3) is used to select the package name declared in the program. Finally, an iterator is created from the selected package name and used to explore each identifier which occurs in the name. It is important to highlight the implicit use of language dependent plug-ins. This MetaJ component holds specific information about the source language. When a p-reference or template is created, the name

of the plug-in must be provided. This name defines implicitly the source language manipulated by the new p-reference or template. In the above example, when the p-reference `r` is created, the name of the plug-in (`"java"`) is passed as the first parameter of the method `MetaJSystem.createPReference`. More information about plug-ins can be found in [Oliveira, 2004, de A. Oliveira et al., 2004]. Other MetaJ features are presented in the next sections.

2.1. Program References

Program references (p-references) are abstractions that store pieces of source code. They hide the internal representation of the code, and only allow operations that guarantee syntactic consistency of this code. Program references are typed with the corresponding type of its syntax tree root node. Nodes of the syntax tree are typed with corresponding types extracted from the nonterminals of the language grammar. Plug-ins are responsible for generating this functionality from a specific grammar.

The available functionality for p-references is shown in Table 1.

Method	Functionality
`set, setFile, setDeref,` `add, remove, replace`	modifies the value of a p-reference
`match, equals,contains,` `hasType, isComposedBy,`	compares and verifies p-reference value
`duplicate, toString,` `get, getSize`	duplicates , converts to string or retrieves values
`getIterator`	returns an iterator to the stored code

Table 1: Methods for the p-reference abstraction

P-reference API is not detailed in this paper, but more information about these methods can be found in [Oliveira, 2004, de A. Oliveira et al., 2004]. The last method shown in Table 1 returns an iterator, which is described in the next section.

2.2. Iterators

Iterators are objects used to traverse the source code stored in a p-reference. Such objects encapsulate iterative top-down traversal, starting from the root node of the program tree. The *Iterator* interface provides methods that allow full control of the traversal. The main methods of the interface are presented in Table 2. More information about these methods can be found in [Oliveira, 2004, de A. Oliveira et al., 2004].

Method	Functionality
`boolean hasNext()`	verifies if there is some piece of code to be reached in the next step
`void nextIn()`	reaches the next nested piece of code, in a top-down order.
`PReference getPReference`	returns a new p-reference with the piece of code reached in the last call to the `nextIn` method.

Table 2: Iterator interface main methods

2.3. Templates

A template is an abstraction which encapsulates a program pattern used to decompose and compose pieces of programs. The pattern is a sentence of the source language (language

of the program being manipulated) where a meta-annotation can appear in place of a specific syntactic construction. The pattern is composed by two types of meta-annotations: meta-variables and markers for optional pieces of sentence. Every meta-annotation has an associated type, which corresponds to a syntactic structure of the source language. Templates provide two basic operations: *match*, which verifies if a piece of code is in accordance with the pattern, and *print*, which builds a piece of program based on the pattern structure. The syntactic structures available for the meta-programmer are selected from the non-terminals of the source language grammar. They coincide with the types for program references. Every syntactic type symbol must be prefixed with the anti-quotation symbol (#). Following, a template declaration is shown, supposing the source language is Java.

```
// package declaration
package  metaj.examples.basicTemplates;
language java;  //  plug-in name (language for template)
template #compilation_unit  SelectPackageName #{
    package #name pack;
    #import_declaration_opt[ #import_declarations imps ]#
    #type_declarations tds
}#
```

This template matches with any Java program (compilation unit) which has a package declaration and any type declaration. Notice the language declaration, which specifies the plug-in for the source language (Java in this case), and meta-variable declarations, such as `#name pack`, `#type_declarations tds`, and the optional sentence `#import_declaration_opt[...]#`. This last construction makes the occurrence of import declarations in the Java program optional .

To make all features of a template (meta-variables, match and print operations) available to the meta-program, a template declaration must be compiled into a Java class with the MetaJ template compiler. After this, a class with the same name and package of the template is created. It provides the methods presented in Table 3.

Method	Functionality

`boolean match(PReference),`
`boolean match(String),`
`boolean matchFile(String)`
	verifies if the code received by parameter matches the template pattern. This operation assigns values to template meta-variables.

`setXXX(String),` `setXXX(PReference)`	defines the value of the meta-variable XXX.
`PReference getXXX()`	returns a p-reference to the code assigned to the meta-variable XXX.

`String toString(),`
`toFile(String),`
`PReference getPReference()`
	builds a program from the template and returns it as a file, a String, or a p-reference

Table 3: The templates API

These are the basic methods to access templates' features. More information about templates can be found in [Oliveira, 2004, de A. Oliveira et al., 2004].

3. The SCQL language

SCQL is a declarative source code query language defined with underlying MetaJ concepts, such as p-references and non terminal syntactic structures. The language allows writing SQL-like queries to select occurrences of syntactic structures in the source code, e.g., identifiers, variable declarations, class declarations, etc..

The SCQL design was carried out based on the concepts of the relational database language SQL. The program source code (a syntax tree) is visualized as a table, thus making relational queries possible. The operator VIEW TREE is responsible for this visualization and it can be compared to the CREATE TABLE SQL operation. The operator SELECT performs projections, selections and joins. The operators INSERT, UPDATE and DELETE modify the content of the source code referenced in a result set.

A shell that executes queries on program files is provided. Also, an API, which allows MetaJ programs to execute queries as easily as Java programs execute SQL queries on databases by using JDBC API, is provided. The language syntax and semantics is presented here by means of examples that assume that Java is the source language.

A VIEW TREE query returns visualization of a program piece in the format of a table. The query presented below visualize the program $prog$ as a table with only one column, lvd.

```
VIEW TREE prog
    AS TABLE #local_variable_declaration lvd
```

The column lvd is filled with all syntactical constructions of type `#local_variable_declaration`, i.e., with all local variable declarations.

The query below visualize the program $prog$ as a table of two columns, lvd and t.

```
VIEW TREE prog
    AS TABLE #local_variable_declaration lvd, #type t
    WHERE lvd.isComposedBy(t)
```

Each line of this table is filled with a pair (lvd, t), where lvd is a local variable declaration, t is the usage of a type and t composes syntactically lvd. The condition "t composes lvd" is specified by the predicate $lvd.isComposedBy(t)$. SCQL calculates the values to be filled in the table by combining (cartesian product) all constructions of type `#type` with constructions of type `#local_variable_declaration`. This combination generates a lot of pairs (lvd, t), but just those pairs which satisfy the `WHERE` condition are inserted in the table.

This is a simple and yet important query, since it is the initial base for the implementation of complex queries to extract simultaneous variables declarations.

The VIEW TREE command also provides the operators FILTERED BY, FOCUSED ON and EXCLUDING which allow the optimization of the query by filtering values to be inserted in a column, focusing the query in a specific piece of the program tree and avoiding some undesirable pieces of program, respectively.

The SELECT operation allows the combination of queries to build a new table. This table is filled with tuples generated by the combination of the composed queries results. A predicate can be specified to select just the desired tuples. The example presented below joins local variable declarations that occur inside `prog1` with type declarations `td` that occur inside `prog2`.

```
SELECT lvd, t, td, id FROM
    VIEW TREE prog1
    AS TABLE #local_variable_declaration lvd, #type t
    WHERE lvd.isComposedBy(t),

    VIEW TREE prog2
    AS TABLE #type_declaration td, #identifier id
    WHERE outer.verifier.isTypeName(td,id)
WHERE t.match(id)
```

The `isTypeName(PReference, PReference)` method of object `outer.verifier` verifies if the identifier `id` is the same as the type identifier of `td`. This method can be easily implemented with templates and p-references.

As an example, consider `prog1` and `prog2` to be the following pieces of Java code.

prog1:
```
public class Test {
  int x;
  Test (int x) { this.x = x;}
  void calc() {
    Util u = new Util();
    Other z1;
    System.out.println(u.fat(x));
  }
  public static void main(...) {
        Other z2;
        new Test(10).calc();
  }
}
```

prog2:
```
class Util {
  int fat (int x){
    if(x == 0) return 1;
    else return x*fat(x-1);
}

interface Other{
  int test();
}
```

First, the following result sets for each internal VIEW TREE expression are calculated.

lvd:#local_variable_declaration	t:#type
Util u = new Util()	Util
Other z1	Other
Other z2	Other

td:#type_declaration	id:#identifier
class Util ...	Util
interface Other ...	Other

The entries of these tables are combined to produce the final result set: a table with four fields and three lines, which holds the selected tuples. Each field has a name and a type that corresponds to a p-reference for the corresponding piece of program.

lvd:#loc...	t:#type	td:#type_decl...	id:#i...
Util u = new Util()	Util	class Util{...}	Util
Other z1	Other	interface Other{...}	Other
Other z2	Other	interface Other{...}	Other

Since SELECT operation results in a table, it can be combined with other queries composing a bigger one. However, it must be used carefully because of the computational cost of the cartesian product operation.

Details on UPDATE, INSERT and DELETE operations can be found at [Oliveira, 2004].

4. Related work

Pattern-matching has already been largely adopted in functional programming languages. In [Sellink and Verhoef, 1998], pattern-matching based on the language grammar is used for source code analysis.

There are already some source code analysis tools based on iterators and queries proposed in the literature. JJTraveler is a Java framework used to combine visitors for analyzing source code [van Deursen and Visser, 2004]. The framework is generated with JJForester from SDF specifications of the language grammar. This approach is based on iteratively visiting tree nodes to perform code analysis. Astlog is a Prolog variant that aims locating and analyzing syntactic artifacts in C/C++ abstract syntax trees[Crew, 1997]. Astlog introduces many ad-hoc features that are adapted for the C/C++ languages. Even though this work manipulates only Java programs, our approach does not rely on static semantics of the target programming language, making easier the construction of plugins for other languages. Genoa is a code analysis tool whose language core is based on the notion of ASGs (abstract semantics graph) traversal[Devanbu, 1999], which differs from our proposal, both on the notion of iteratively traversing a structure, and also considering semantic information of the target language. PQL is a program query language based on modeling several program information needed to answer queries[Jarzabek, 1998]. The program model is the most difficult part to be constructed because it requires several semantic analyses. The result sets are not based on tables and cartesian products, such as in SCQL, but on tuple of lists. SCA is a source code algebra that permits users to express complex source code views and queries as algebraic expressions[Paul and Prakash, 1996]. Queries in SCA are performed over a object data model that are dependent on the semantics of the target language. JQuery is a language that augments Tyruba, a logic programming language, with a library of predefined predicates for querying Java source code units and the relationships between them[Janzen and de Volder, 2003]. The predicates implement semantic relationships between units, for instance, which methods call the others. Graphlog is a logic query language for visualizing and querying software systems modeled as directed graphs[Consens et al., 1992]. The construction of the graph involves semantic information of the target language. Queries are constructed drawing graph patterns with a graphical editor. Most of the information queried is based on dependency relationships. Omega is a language-based programming environment in which all calculated program information is represented as tables using relational database[Linton, 1984]. The underlying relational model takes into account only procedural languages, and relies both on syntactic and semantic information. Horwitz defines a model for adding relational query facilities to software development environments[Horwitz, 1990]. The model relies on the use of implicit relations, which are not stored as set of tuples, but instead computed as needed during query evaluation. This approach is similar to ours for query evaluation. The model also relies on calculating several ad-hoc functions for extracting the desired information from the software systems. This information goes from transitive closure of calling functions to the dynamic information of variable values during program execution.

5. Case study: static semantics graph

This section presents a case study to motivate the usefulness of MetaJ and SCQL. Complex program transformations, such as refactorings, would benefit from more expressive abstractions than syntax trees. For instance, a *Rename Class* refactoring for Java programs requires updating all the occurrences of the class name throughout the whole selected system, let be on an extends clause, on a object creation, on a variable declarator or even on a anonymous class definition. Finding all these occurrences in a syntax tree may require more than just traversing the nodes and applying the necessary changes. It may require considering issues such as scoping and visibility for general cases.

A meta-program that constructs a model that captures static semantics of the Java source code is now presented. The design of this model was devised considering its usefulness for writing refactoring functions. The model is a graph that directly represents

declarations, scopes and use of symbols as its nodes. This model is also linked with the syntax tree of the source code.

A scope node defines an area where declarations and uses are located. Each scope have three sets: a set of all declarations declared inside it, a set of all occurrences of symbols inside it, and a set of all directly nested scopes.

Declaration nodes are defined by an identifier, a type (either the type of a variables, or signature of methods, or the new declared type itself). From a declaration node it is possible to navigate to all its corresponding use nodes. This feature is not present in conventional symbol tables. Some declarations may open a new scope, for instance, a class body, a method body, or a local variable declaration.

Use nodes represent the occurrences somewhere in the source code of a previously declared symbol. From an use node it is possible to navigate to its corresponding declaration node.

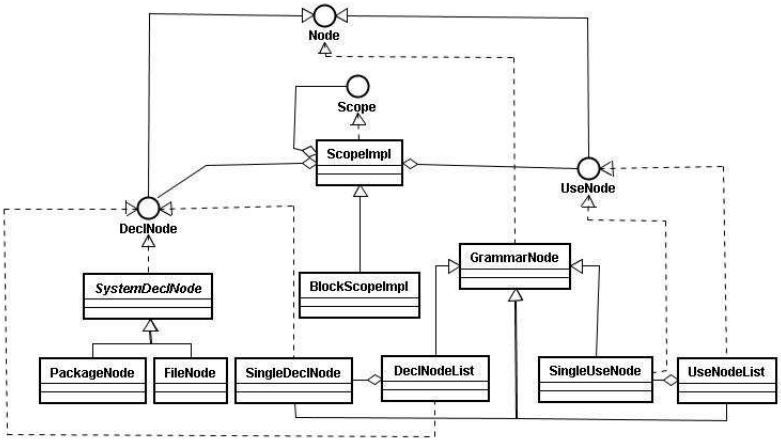


Figure 1: Static semantics model

Figure 1 shows classes and interfaces used to represent the static semantics model. The `Node` interface defines common operations for all declaration and use nodes. The `GrammarNode` class represents nodes that have references for source code, i.e., MetaJ references.

The `DeclNode` interface defines common operations for all types of declaration nodes, respectively, the `SystemDeclNode` that models elements of the file system which are not defined from the grammar (`PackageNode` and `FileNode`), the `SingleDeclNode` that models a single declaration, and `DeclNodeList` that models a list of declarations that have the same type. The only two kinds of references modeled by `DeclNodeList` are instance variable and local variable declarations.

The `UseNode` interface defines common operations for two types of use nodes, respectively, the `SingleUseNode` that models the occurrence of a single identifier previously declared and the `UseNodeList` that models a list of occurrences composing a single syntactic structure, for instance, a qualified class name that have the occurrence of package names and the class name itself.

The `Scope` interface defines common methods for two types of scopes, respectively, the `ScopeImpl` class that models scopes in which the order of declaration occurrences does not matter, for instance, inside a class body, and the `BlockScopeImpl` class that models scopes in which the order of declarations occurrences must be considered, for instance, inside a method body or a block statement.

The model creation process can be divided into three phases: preprocessing, parsing and linking.

5.1. Preprocessing Phase

In this phase, all new types declared in the system are collected. Firstly, all subdirectories of the system are traversed and for all files in each directory a MetaJ reference is created. After that, SCQL queries extract the corresponding desired type declarations. Each found type declaration is indexed by its qualified name. The algorithm for preprocessing is shown below.

```
preProcessSystem(File directory) {
  File[] subPack = Select all subpackages in directory
  for each package in subPack do
     preProcessSystem(package);
  File javaFiles = Select all Java Files in directory
  for each jFile in javaFiles do
    Reference r = createReference(jFile,"#compilation_unit");
    Reference classSet[] =
        "VIEW TREE r AS TABLE(#class_declaration NOTCOMPOSING #class_declaration)";
    for each reference in classSet do
      put reference into class declaration index.
    Reference interfaceSet[] =
        "VIEW TREE r AS TABLE(#interface_declaration NOTCOMPOSING #interface_declaration)";
    for each reference in interfaceSet do
      put reference into interface declaration index
}
```

5.2. Parsing phase

In this phase, all *declaration*, *scope* and *use* nodes are created. The parsing process is applied to all references collected in the previous phase. The process is to similar a recursive descent parser, using a method for each relevant nonterminal.

Each method decomposes an input *Reference* into its components. The method verifies if it can be produced either scope, declaration or use nodes. Inside each method, scope nodes are created for each file system item, and from syntactic structures, such as, *compilation_unit*, *type_declaration*, *method_declaration*, *block*, *local_variable_declaration*. Scope nodes are containers for declaration nodes, use nodes and nested scope nodes.

Declaration nodes are created when analyzing syntactic structures, such as, *class_declaration*, *interface_declaration*, *method_declaration*, *field_declaration*. For instance, in the method that analyzes a *class_declaration*, a `SingleDeclNode` object is created and added into the scope node previously created from the respective file.

Use nodes are created when analyzing syntactic structures that contain *identifiers*.

SCQL queries were used for extracting information from most syntactic structures which have none or simple recursive rules. For instance, consider parsing the *name* syntactic structure which rule is defined below.

name ::= *identifier*
| *name qualified_name*
qualifier_name ::= *identifier*

```
name(Reference r) {
    Reference[] ref = "VIEW TREE r AS TABLE(#identifier i)";
    for each Reference in ref do
       create a use node for this reference
       put this node in the current scope
}
```

An alternative way to do the same thing using templates is shown below.

```
template #name BaseName #{
    #identifier i
}#
template #name RecursiveName #{
    #name n #qualified_name qn
}#
template #qualified_name QualifiedName #{
    #identifier i
}#

name(Reference r) {
    BaseName bn = new BaseName();
    RecursiveName rn = new RecursiveName();
    if(bn.match(r))
       create a use node for the reference bn.i
       put this node in the current scope
    else if(rn.match(r))
       name(rn.n)
       qualifiedName(rn.qn)
}
qualifiedName(Reference r) {
    QualifiedName qn = new QualifiedName();
    if(qn.match(r))
       create a use node for the reference qn.i
       put this node in the current scope
}
```

It seems clear that the SCQL solution is much cleaner and simple. Indeed, in some cases SCQL could not handle with some recursive structures. For instance, consider the Java grammar rule *conditional_and_expression*. In this case, the solution should be based on templates.

conditional_and_expression ::= *inclusive_or_expression*
| *conditional_and_expression* "&&" *inclusive_or_expression*

```
template #conditional_and_expression
         BaseConditionalAndExpression #{
    #inclusive_or_expression ioe
}#
template #conditional_expression
         RecursiveConditionalAndExpression #{
    #conditional_and_expression cae "&&"
    #inclusive_or_expression ioe
}#

conditionalAndExpression(Reference r) {
   BaseConditionalAndExpression bcae = new BaseConditionalAndExpression();
   RecursiveConditionalAndExpression rcae = new RecursiveConditionalAndExpression();
   if (bcae.match(r))
       inclusiveOrExpression(bcae.ioe);
   else if(rcae.match(r))
       conditionalAndExpression(rce.cae);
       inclusiveOrExpression(bcae.ioe);
}
```

This solution is similar to a recursive descent parser. Nonetheless, it must be pointed that templates is just a simple interface to create this kind of analyzers, and can be applied in other applications, such as code generators or program transformations, using other strategies.

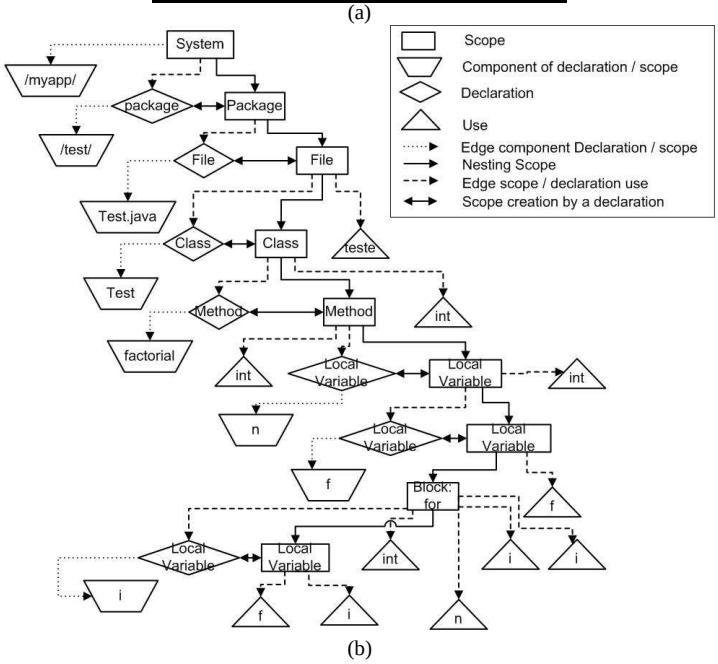
Figure 2 shows the resulting model (b) for a simple Java program (a).

5.3. Linking phase

At the beginning of this phase, all model nodes have been created. The linking phase links all declaration nodes to their respective uses and vice-versa. This phase is implemented by three visitors, respectively, `ExtendsLinkVisitor`, `DeclUseLinkerVisitor` and `QualifiedNameVisitor`, which do not use neither MetaJ nor SCQL mechanisms, but only the model produced so far. This shows the usefulness of having meta-programs

```
package test;
public class Test {
  public int factorial(int n) {
    int f = 1;
    for(int i = 1;i <= n;i++)
       f *=  i;
    return f;
  }
}
```

(a)



(b)

Figure 2: Result of the parsing phase

facilities embedded in an object-oriented language. Figure 3 shows the resulting model for the program shown in Figure 2.

The first visitor is specialized on visiting *extends* and *implements* clauses to link the corresponding class identifiers to their declaration. Its algorithm is shown below.

```
ExtendsLinkerVisitor()
  for each package scopes in System root
    for each class declaration in current package scope
      if current class has a extends clause
        // O(1) search in index from preprocessing phase
        Node n = Search class declaration of this clause
        if(! n.isNull())
          link use node of extends clause to its decl
      if current class has a implements clause
        Node decl[] = Search all the interfaces declaration of this clause
        Node use[] = Get all uses of implements clause
        for each Node in use do
          link the current use to his declaration
```

The second visitor links declarations and their uses that are non qualified uses. For instance, these cases correspond to local variables and non qualified field accesses and non qualified method calls. The third visitor is specific for linking uses occurring in method/constructor calls and instance variable use. This visitor traverse the current model, in preorder, driven the scope nodes, and whenever it encounters an use node not currently linked to its declaration and that it is not created from a qualified name, it verifies if the respective declaration is in the current scope, and if not it creates another iterator that searches the scope tree upwards until the root trying to find the corresponding declaration. The root scope contains references to all public declarations. This guarantees that all use nodes will be linked.

The third visitor links use nodes corresponding to qualified names occuring in syntactic structures such as, import declarations, method calls, and field accesses. These structures are represented in the model as `UseNodeList` nodes. The respective declaration of the element $i+1$ of this list can be found in the scope node created by the respective declaration of the element i of the list. The used algorithm is shown below.

```
QualifiedNameVisitor()
  for each scope in System root
    if(current use is qualified)
      Node decl = null;
      for each use in the current use list
        if(decl == null)
           decl = Search the declaration of the first use in the list
        else
           decl = Search the declaration of the current use in scope opened by decl
        link the current use with decl
```

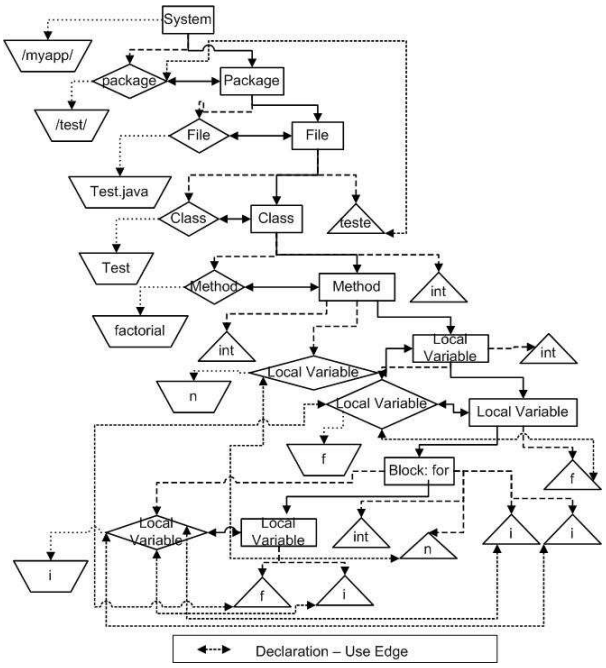


Figure 3: Result of the linking phase

6. Case study: refactorings

In this section, MetaJ and SCQL approaches are compared with each other. The methodology used was implementing two refactorings, namely Rename Local Variable (RLV) and Self Encapsulate Field (SEF) with MetaJ and with SCQL. The refactorings were applied to three artificial programs. A qualitative comparison on the writing style of the approaches, a quantitative comparison on the metrics of the refactoring implementations, and a performance comparison on the application of the refactorings are presented.

6.1. Qualitative Analysis

The following method was extracted from the implementation of the Rename Local Variable refactoring written with MetaJ. The fragment is called when there are name clashes between the new local variable name and an instance variable name.

```
void addThisToFieldAccess(String var, Reference ctx) {
   Iterator it = ctx.getIterator();
   while(it.hasNextIn()){
      it.nextIn();
      Reference r = it.get();
      if(r.isTypeOf("java.#postfix_expression") && r.toString().startsWith(varName))
         r.set("this." + r.toString());
   }
}
```

The same method written with SCQL is shown below.

```
String findFieldAcc =
 "VIEW TREE context AS TABLE (#postfix_expression pe
 FILTERED BY pe.toString().startsWith(outer.varName))";
void addThisToFieldAccess(String var, Reference ctx) {
   QueryFactory qf = SCQL.createQueryFactory("java");
   qf.add("varName", var); qf.add("context", ctx);
   ResultSet rs = qf.createQuery(findFieldAcc).getResultSet();
   while(rs.next()){
      Reference r = rs.getReference("pe");
      r.set ("this." + r.toString());
   }
}
```

The MetaJ approach defines a traversal on the code that filters a desired syntactic structure and applies some action on it. The SCQL approach defines declaratively all desired syntactic structures. The action is then performed on all of them. When this situation is scaled to larger specifications, the SCQL approach shows more directly which syntactic structures are being selected and what are the respective actions on them. The code with SCQL seems to be more organized because of the the specification of the desired structure is separated from actions performed on them. So, it is easier to reuse SCQL queries than the MetaJ loops.

6.2. Design Metrics Analysis

Design metrics of the implemented refactorings were collected. They are shown in Figure 6.2 and are, respectively, the number of lines of code, the number of classes, the total number of method declarations, and the total number of field declarations. The main goal of this analysis is to verify the size of implementations. The difference are more notably seen in the implementation of the Self Encapsulate Field (SEF) refactoring, where the SCQL approach has the half size. Each template used in the MetaJ implementation was counted as a class. Each field within the template was counted as an instance variable, and has respective get/set methods. But the number of lines collected from the template was not from the generated class but from the template itself. The idea was to measure the programmer's labor. The intensive use of templates for the SEF refactoring explains why the number of fields for SEF refactoring written with MetaJ is much bigger than that written with SCQL.

6.3. Performance Analysis

The performance comparison of the refactorings was done over three artificial programs. The following table shows the main characteristics of the programs. LCM is the number of line of code per method; #C, #M, #F are the number of classes, methods and fields, respectively; AAT is the total number of accesses and assignments to either local variables or instance variables, in each method; AAIV is the number of accesses and assignments to the instance variable that will be encapsulated by the SEF refactoring, in each method; and AALV is the number of accesses and assignments to the local variable that will be renamed by the RLV refactoring, in each method. All methods of a class are identical, so the SEF will affect all methods. The 60 methods in programs *P1* and *P2* are identical. Programs *P2* and *P3* have the same structure, but methods of *P3* are bigger.

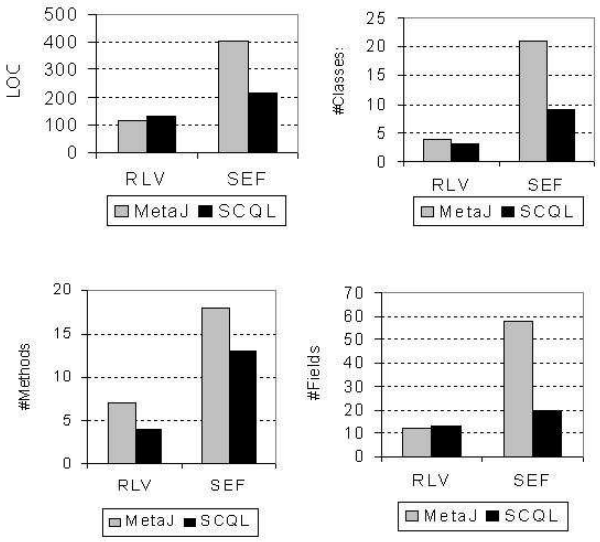
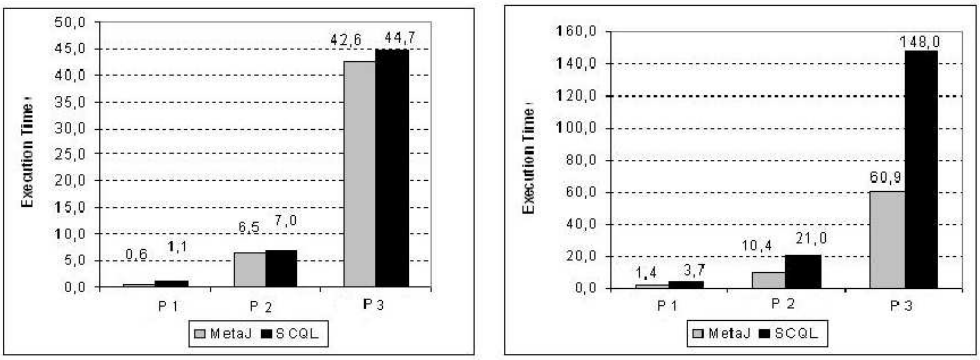


Figure 4: Metrics of the refactorings' codes

	LOC	#C	#M	#F	LCM	AAT	AAIV	AALV
P1	383	1	10	10	36	55/24	2 / 1	4/2
P2	1888	1	50	10	36	55/24	2 / 1	4/2
P3	3715	1	50	10	71	110/42	5 / 2	7/3

The input parameters for refactoring evaluation were chosen, such that, the RLV refactoring renames a local variable declared in the middle of a method. The local variable name was chosen such that it clashes with an instance variable name. The method chosen for the RLV refactoring is located at the middle of the class. Also, the SEF refactoring adds get/set methods for an instance variable, such a local variable with the same name is declared in the middle of all methods. The aim of such choices was try to capture a mean cost of refactoring evaluations. The refactorings were executed five times over each program and in Figure 6.3 the mean execution times are presented.



(a) (b)

Figure 5: Mean execution times (a) RLV refactoring (b) SEF refactoring

Queries and updates within the SEF refactoring implemented with SCQL were declared to execute on a specific scope. For the above results the scope was carefully designed to be a method. In a prior naïve implementation, the queries which calculated cartesian products over 5 sets were performed on the whole class every time, and in this case the execution time for the SEF refactoring over Program3 reached 16 hours. The execution time of SCQL refactorings is quite acceptable in all cases for the RLV refactoring. Special attention is needed for SEF refactoring which is more than twice slower.

7. Discussion

The presented approach for program analysis is based on the definition of several mechanisms for extracting syntactic information of the source code. These mechanisms were defined considering only the syntax of the object language. These mechanisms were used to produce more expressive (object-oriented) models than those based solely on the syntax. These models can that be used in more sofisticated analyses.

As shown in section 4, many other approaches, such as Genoa and SCA, have included ad-hoc mechanisms for constructing a knowledge base about the information of the software being analyzed, making easier to query semantic information. However, our approach is more stratified and reusable, if considered the construction of program analyzers for different languages.

Our approach has shown that different mechanisms ranging from procedural iterative algorithms to declarative template and queries could be used together within a single object-oriented program analyzer, benefiting from design practices of object-oriented programming.

The imperative resources of the approach were proved to be more efficient (they are computationally cheap), but indeed more difficult to use and reuse. The declarative resources are more readable and easier to use, but tend to be computationally expensive. Using templates and queries seems to be more intuitive than writing visitors like, for instance, JJTraveler. The parse phase for construction of the graph shown in section 5, if written with JJTraveler, would require a visitor that traverse syntax trees and the implemention operations for all nodes of the tree. Even if the visited node do not contain any significant information for the model, the operation should be implemented doing nothing. If the visited node represents a declaration, it is necessary capturing additional information, such as its identifier, type, modifiers, that requires specific visitors. This could be implemented more naturally with MetaJ templates.

However, it should be noted that when iterators and visitors are necessary with our approach, MetaJ does not offer an elegant mechanism for combining visitors such as JJTraveler. Implementing this parse phase with Yacc should presented the similar drawbacks, and additionally would require even more work to construct syntax trees.

8. Concluding remarks

This paper has presented an integrated use of iterators, templates and queries for analyzing and manipulating source code. These mechanisms are much more closer to average programmers than other tools based on rewriting systems. Developer teams may effectively consider writing meta-programs that help analyzing and manipulating source code during the maintenance process, because the presented tools, MetaJ and SCQL, have proved to be quite easy to use.

About, the expressive power of the tools, the underlying information model for MetaJ and SCQL is based solely on the syntax of the target language. More powerful queries would require incrementing the model with semantic information. Nonetheless, this design decision simplifies the construction of plugins for other languages, and the use of SCQL proved to be useful and simple to use in many situations.

The comparison of MetaJ and SCQL has shown that SCQL may be useful for producing more compact and elegant analysis of source code. However, the implementation still deserves more attention to optimization of query execution, and also there are situations pattern-matching and iterators are necessary, thus indicating the necessity to include mechanisms to compose iterators in MetaJ.

References

Consens, M., Mendelzon, A., and Ryman, A. (1992). Visualizing and querying software structures. In *Proc. of the International Conference on Software Engineering*, pages 138–156. ACM.

Cordy, J. R., Dean, T. R., Malton, A. J., and Schneider, K. A. (2002). Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 44(13):827–837.

Crew, R. F. (1997). ASTLOG: A language for examining abstract syntax trees. In *Proc. of the USENIX Conference on Domain-Specific Languages*, pages 229–242.

de A. Oliveira, A., Braga, T. H., de A. Maia, M., and da S. Bigonha, R. (2004). MetaJ: An Extensible Environment for Metaprogramming in Java. *Journal of Universal Computer Science*, 10(7):872–891.

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

Horwitz, S. (1990). Adding relational query facilities to software development environments. *Theoretical Computer Science*, 73:213–230.

Janzen, D. and de Volder, K. (2003). Navigating and querying code without getting lost. In *Proc. of the 2nd International Conf. on Aspect-oriented Software Development*, pages 178–187.

Jarzabek, S. (1998). Design of flexible static program analyzers with PQL. *IEEE Transactions on Software Engineering*, 24(3):197–215.

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

Linton, M. (1984). Implementing relational view of programs. In *Proc. of ACM SIGSOFT/SIGPLAN Software Engineering Symposium - Practical Software Development Environment*, pages 132–140.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

Oliveira, A. (2004). MetaJ - An environment for metaprogramming in Java. *in portuguese*. Master's thesis, Federal University of Minas Gerais, Brazil.

Paul, S. and Prakash, A. (1996). A query algebra for program databases. *IEEE Transactions on Software Engineering*, 22(3).

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.

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.

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