Building Flexible Refactoring Tools with XML

Nabor C. Mendonça[1], Paulo H. M. Maia[2], Leonardo A. Fonseca[1], Rossana M. C. Andrade[2]
[1]Mestrado em Informática Aplicada – Universidade de Fortaleza (UNIFOR)
Av. Washington Soares, 1321 – Fortaleza – CE – Brasil
[2]Departamento de Computação – Universidade Federal do Ceará (UFC)
Campus do Pici, Bloco 910 – Fortaleza – CE – Brasil
e-mail: [nabor, leonardo]@unifor.br, [pauloh, rossana]@lia.ufc.br

Abstract

Refactoring, i.e., the process of changing a software system to improve its internal quality while preserving its external behavior, is gaining increasing acceptance among software developers. Even though many refactoring tools are now available for a variety of programming languages, most of them rely on their own, closed mechanisms for representing and manipulating source code information, which makes them difficult to customize, extend and reuse. This paper makes three contributions towards the development of more flexible refactoring tools. Firstly, it proposes an XML-centric refactoring process in which XML is used as a standard way to represent, analyze and modify source code information. Secondly, it presents a concrete realization of that process, in the form of a refactoring framework, called RefaX, which builds on existing XML-based source code models and processing technologies to facilitate the development, extension and reuse of refactoring tools. Finally, it demonstrates the applicability of the proposed process and framework through two RefaX-based refactoring prototypes for Java and C++, respectively.

1. Introduction

The costs and complexity of software maintenance are widely recognized. It is estimated that 50% of a software engineer's time is dedicated to browsing and comprehending existing code [20], and that in the last three decades maintenance activities took over 60% of all software development costs in most organizations [23].

As an attempt to reduce the costs and complexity of maintaining existing software, a number of techniques have been proposed, including metrics, testing and verification, reverse engineering, and reengineering. A more recent technique, *refactoring*, i.e., the process of changing a software system to improve its internal quality while preserving its external behavior [11], is gaining increasing acceptance among software developers. Its main goal is to reduce code complexity, which increases rapidly as the code evolves, by making it more extensible, modular, reusable and maintainable [21]. The adoption of refactoring has been boosted by the fact that it is one the pillars of *Extreme Programming* [3], an emerging and increasingly popular software development methodology, and by the wide dissemination of the refactorings catalog created by Fowler [11].

As with other code changing techniques, refactoring is most effectively performed by means of automated tools. This avoids the risk of introducing new errors due to manual intervention, and minimizes the amount of tests needed every time the code is changed. A considerable number of refactoring tools are now available for a variety of programming languages, including, for example, Refactoring Browser [25], for Smalltalk, JFactor [15], for Java, and C# Refactoring Tool [4], for C#. Nonetheless, it is virtually impossible to find a refactoring tool, or a suit of such tools, that will fully satisfy the needs of every software developer. The

main reason is those tools only provide a fixed set of refactoring operations, which may not be adequate for the maintenance task at hand. In addition, most of the available tools fall short of customization support, making it difficult – if not impossible – for a developer to create new refactoring operations, or even to adapt an existing one. The need for more flexible refactoring tools has already been recognized in [21].

One of the main factors that hinder customization in current refactoring tools is the lack of open, standardized ways to represent and manipulate the source code elements of the target language, with each tool using its own data model and processing technology. In an attempt to address this issue, several source code representations have been recently proposed for different programming languages and paradigms (e.g., [19][1][13][2]). Most of those representations share the fact that they rely on XML [31] as their underlying data model. The main advantage of using XML to represent source code information is that software maintenance tool developers and users could both benefit from the abundance of XML processing tools currently available, including XML processing API's [30][26], query engines [32][33], databases [35][9][27] and transformation tools [34][36].

Even though maintenance and related tools are already benefiting from XML as a standard way to represent and exchange source code information (e.g., [19][6][13]), those tools do not yet fully explore the potential of current XML-based technologies towards better customization, extension and reuse. In particular, we are unaware of any refactoring tool that relies on XML to improve the flexibility of it underlying code transformation mechanism.

This paper makes three important contributions towards the development of more flexible refactoring tools. As the first contribution, it proposes an XML-centric refactoring process in which XML is used as a standard mechanism for representing, analyzing and modifying source code information. The second contribution is a concrete realization of that process, in the form of a refactoring framework, called RefaX, which builds on existing XML-based source code models and processing technologies to facilitate the development, extension and reuse of refactoring tools. In particular, RefaX makes it possible to implement refactoring operations that are independent of source code model, programming language and manipulation mechanism. In practice, this means that a RefaX tool can be reusable across different source code representations and processing technologies. The third and final contribution is a demonstration of the applicability of the proposed process and framework through two RefaX-based refactoring prototypes for Java and C++, respectively.

We believe that such an open approach for refactoring tools production, as advocated in this paper, can be beneficial not only for tools developers, who will have a powerful environment upon which to build, extend and reuse refactoring operations, but also to refactoring users themselves, whose tools will be more easily customized to their specific maintenance needs.

The rest of the paper is organized as follows. In the next section we give an overview of existing XML-based source code representations and processing technologies. We then introduce the XML-centric refactoring process (section 3), present the RefaX framework in more details (section 4), and report on the development of the two refactoring prototypes (section 5). We follow with a discussion of our research with respect to related work (section 6), and conclude the paper by summarizing our main results and suggesting some directions for future work (section 7).

2. Representation and Manipulation of Source Code Information Using XML

In this section we give a brief overview of existing XML data models for representing source code information, and discuss how this information can be manipulated using current XML processing standards and technologies.

2.1. XML-Based Source Code Representations

As mentioned previously, there have been a number of proposals for using XML as a common data model to represent source code information. In [19], Mamas and Kontogiannis propose three of such models: JavaML, CppML and OOML. JavaML and CppML represent object-oriented source code written in Java and C++, respectively, in the form of an Abstract Syntax Tree (AST). OOML is a generalization of the other two models and represent only those syntactic elements that are shared by both. Badros proposes another XML source code model for Java, also called JavaML [1]. Just like the previous models, Badros's JavaML also represents source code information in the form of an AST. Compared to Mamas and Kontogiannis's JavaML, though, Badros's model manages to capture the same information in a semantically richer yet more concise way. Another example of an XML-based source code model for Java is XJava [2].

The srcML source model [18] follows a slightly different approach. Instead of converting the source code to a different format, srcML "annotates" the original source code artifacts with structural information represented in the form of XML tags, thus preserving non-syntactic elements, such as blanks, comments and indentation marks. Compared to other source code models based on an AST representation, srcML has the advantage that the annotation process does not require a complete parsing of the source code, which simplifies the development of code-to-XML conversion tools. On the other hand, the resulting XML document will end up with a potentially large amount of non-essential code information, which may land it difficult to manipulate using current XML processing technologies.

GXL [13] is another source code model that was originally proposed as a common software exchange format to promote interoperability among software maintenance and reverse engineering tools. In GXL, the code structure is captured in the form of a directed graph using only two types of syntactic elements: nodes and edges. All types of syntactic information presented in the source code have to be represented as attributes of those two basic elements. The lack of a syntactically richer source code representation makes GXL an unsuitable format for specifying refactorings, as most of the update operations would have to be expressed in terms of element attributes. This not only would make the refactoring specification more cumbersome, but would also slow down the execution of query and update expressions on large documents.

It is worth noting that, from a strictly syntactic point of view, there are significant differences among all those models, with some models being considerably more complete and/or concise than others with respect to the kinds of syntactic information they represent. Although syntactic differences are an important issue to consider when choosing an XML-based source code representation upon which to implement refactoring operations, other factors may also be relevant – for example, whether or not there is an appropriate conversion tool available. In any case, different source code models may suit different tool developers under different circumstances. Therefore, the choice of the best model should better be left to the developer, and not be restricted by the development environment at hand.

2.2. XML Processing Standards and Technologies

Current technologies for processing XML data can be grouped into two main categories: query processors and update and transformation tools. XPath [32] and XQuery [33] are the two W3C (*World-Wide Web Consortium*) standards for querying XML data. Xpath is a simple query language to access parts of an XML document through the specification of path expressions. Due to its simplicity, XPath is most commonly used as a basis upon which to build more sophisticated XML processing tools, such as XSLT [34]. XQuery in turn is a

direct extension of XPath. In addition to the concept of path expressions, XQuery supports query expression in terms of the so called FLOWR (FOR, LET, WHERE, ORDER BY and RETURN) clauses. These clauses bring extra power and flexibility to the language, as they allow the specification of multiple nested expressions in the same query. Other advanced features added by XQuery include a number of built-in operators for processing query results, such as *sort*, *count* and *average*; support for conditional queries through the logical operators IF/THEN/ELSE; and support for recursive queries through user-defined functions, which is XQuery's mechanism for query encapsulation and abstraction. This latter feature can be of particular interest to the specification of XML-based refactoring operations, as the relationships among source code elements in most source code models are recursive in nature.
In contrast to query processing, currently there is no standard way of updating XML data. So far several XML update languages have been proposed, including XQuery Update [17], the one proposed by Tatarinovi *et al.* [28], and XUpdate [36]. XQuery Update is a direct extension of XQuery but which, along with the language proposed by Tatarinovi *et al*, suffers from the lack of adequate tool support. Support for XUpdate, on the other hand, is continuously increasing, with many professional XML databases adopting it as their native update language.
XML transformation tools, such as XSLT [34], can also be seen as an updating technology for XML data, for they can be applied to transform the original XML document into a new document that reflects the desired changes. Although some types of refactoring operations can be implemented this way, such an approach would be highly ineffective when only a small fraction of the code is actually affected by the refactoring – which is likely to be the case for most refactoring operations used in practice. In addition, it is our experience that refactorings expressed as document transformations tend to be more cumbersome (and thus more difficult to specify) than an equivalent specification in a proper update language, for example, XUpdate.
The next section describes how some of the above source code models and technologies can be integrated as part of a novel software refactoring process based on XML.

3. An XML-Centric Refactoring Process

Most existing refactoring tools follow a common refactoring process, in which refactoring operations are applied to a program's source code according to well-defined steps. In general, those steps include [21]:

1. Detecting where to apply refactorings in the source code.
2. Determining which refactorings to apply to each selected source code location.
3. Guaranteeing that the chosen refactorings preserve program behavior.
4. Applying the chosen refactorings to their selected source code locations.
5. Verifying that program behavior is actually preserved after the refactorings had been applied.

Our proposal for an XML-centric refactoring process focuses on steps 3 through 5 of that general process. We assume that the refactoring tool user is responsible for steps 1 and 2. Those two steps can be carried out either manually, with the maintainer exploring her own experience and knowledge of the code, or with the help of semi-automated techniques, such as metrics and reverse engineering. Metrics, in particular, can offer a promising solution to automate the task of detecting refactoring opportunities in an existing system source code. A recent work in this direction is described in [5].

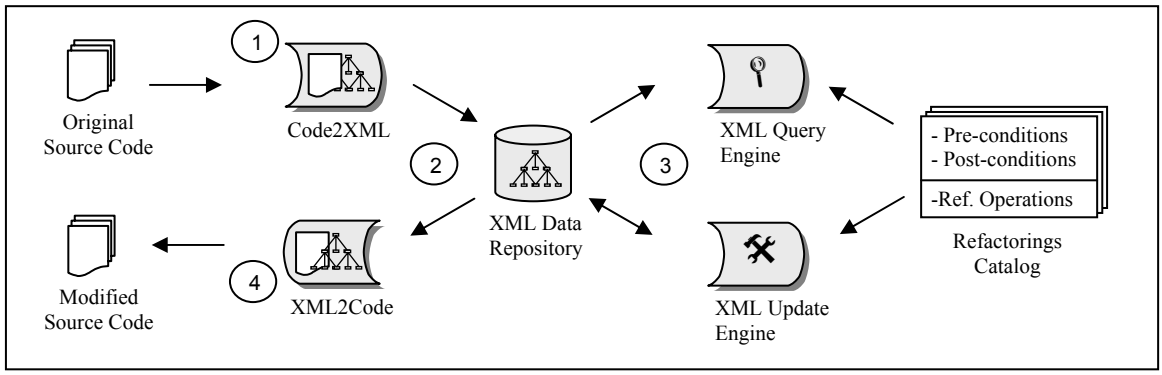

Figure 1: An XML-centric software refactoring process.

In our process, steps 3 to 5 were further refined to accommodate the necessary suit of XML data models and processing technologies. The resulting process was then reorganized into the four new steps, as follows:

1. Conversion of source code to XML.
2. Storage of the generated XML data.
3. Refactoring via XML data manipulation.
4. Conversion of XML to source code

Figure 1 illustrates the relationships among these four steps as part of our XML-centric refactoring process. The following subsections describe each of these steps in more details.

3.1. Conversion of Source Code to XML

In this step, the user selects the source code files to be converted into the chosen XML representation. The conversion process is usually carried out by means of automated tools, such as language parsers or string processing tools. We refer to the tool responsible for this step as an *XML converter*.

3.2. Storage of the Generated XML Data

Once the selected source code files have been converted to the corresponding XML representation, that representation should be stored in some form of repository, so that it can be effectively accessed and manipulated in later steps. The repository can range from simple flat files to a fully-fledged commercial XML database. The actual choice depends on the types of XML processing technologies used in the refactoring step, as well as on several other factors, such as the amount of source code information to be manipulated, and the level of performance and/or scalability expected from the refactoring tool.

3.3. Refactoring via XML Data Manipulation

In this step, the user selects which refactorings should be applied to each source code file converted to XML in the first step. This step is further decomposed into three sub-steps, namely *test of pre-conditions, applying refactoring operations*, and *checking behavior preservation*. These are described below.

3.3.1. Test of Pre-Conditions. Refactoring pre-conditions are responsible for guaranteeing the refactoring operation legitimacy. A given refactoring can have multiple pre-conditions. In the case that at least one of them is not satisfied, the refactoring will not be applied. The pre-condition concept was introduced by Opdyke [22] and later extended by Roberts [25].

Roberts defines pre-conditions as combinations of *Analysis Functions* (AFs), which describe relationships among source code entities (classes, methods, fields, etc). AFs can be grouped into two categories: primitive AFs and derived Afs. As an example, consider the primitive AFs *IsClass(className)*, which checks whether *className* already exists as a defined class in the source code, and *Superclass(className)*, which returns the superclass of *className*, if there is such a class defined in the source code.
A derived AF can be specified in terms of one or more primitive AFs. For example, the derived AF *AllSuperclasses(className)* returns the set of all superclasses of *className*, and can take one of the following values:

- Ø, when *Superclass(className)* = Ø; or
- *Superclass(className)* U *AllSuperclasses(Superclass(className))*, otherwise.

Since AFs only access source code information, without actually changing it, in our XML-centric refactoring process they can be expressed using an appropriate XML query language.

3.3.2. Applying Refactoring Operations. Once all preconditions have been validated, the next activity consists of applying the refactorings themselves. A single refactoring may be specified in terms of multiple refactoring operations. This feature not only improves refactoring modularity and reuse, but also provides a powerful abstraction mechanism which allows higher-level refactorings to be described in a source code model and programming language independent way. Since each of refactoring operation has the effect of changing the code, in our XML-centric refactoring process they should be expressed using an appropriate XML update or transformation language.

3.3.3. Checking Behavior Preservation. After each refactoring operation has been executed, it is necessary to verify that the modified code still preserves its original external behavior. To this end, Roberts also introduced the concept of refactoring post-conditions, which are conditions that must be valid after all refactoring operations of a refactoring have been applied. As with pre-conditions, refactoring post-conditions vary according to the type and purpose of each refactoring. Since post-conditions are also limited to access source code information only, in our process they can also be implemented in terms of AFs expressed using an appropriate XML query language.

3.4. Conversion of XML to Source Code

Once all refactorings have been applied to the appropriate source code elements in the repository, their results can be reflected back to the original source code artifacts. This is usually done using an XML transformation tool. This tool processes the selected source code elements in the repository according to their target XML scheme, generating an equivalent textual representation for those elements in their original programming language. We refer to this second conversion step, from XML back to source code, as *reversion*, and to the conversion tool used as an *XML reverter*.

4. RefaX: An XML-Based Refactoring Framework

The RefaX framework is a *proof-of-concept* realization of the XML-centric refactoring process presented in the previous section. In this section, we first discuss the framework's main requirements, then elaborate on some of its selected implementation details, and finally illustrate its use through an example.

4.1. Requirements

The design of RefaX was largely influenced by the requirements of FAMIX [29], a meta-model for representing object-oriented software entities and their relationships in a language-independent way. In addition to language independence, in developing RefaX we had also to consider further requirements specific to the use of XML, such as independence of XML data model (or scheme), and independence of XML processing technology. These and other requirements are discussed below:

Scheme-independence – RefaX was designed to support different XML data models for the same target language. This means that any refactoring operation written for a particular programming language (say Java), using a specific data model (say XJava), should be easily applied to source code elements of the same language represented in a different data model (say JavaML or BuiltyJ).

Language-Independence – RefaX supports the specification of reactorings in varying levels of abstraction. This makes it easier for the developer to design refactoring operations abstract enough to be applicable to different programming languages. This requirement is particularly important in practice, since different languages belonging to the same paradigm usually have many types of source code entities in common, thus offering a number of opportunities for refactorings reuse across those languages. In general, the deeper the differences between two or more programming languages, the higher the level of abstraction necessary for writing common refactorings to them.

Technology-Independence – RefaX was built following a conscious architectural design, in which several *hot spots* are provided so that a developer can easily decouple refactorings implementation from the suit of XML processing technology (converter, storage medium, update engine, etc) necessary to execute them. This frees the developer from committing to any specific technology early in the development process, and also makes it easier to replace one of more of the chosen technologies for new ones in later stages.

Reliability – Through the verification of pre- and post-conditions, RefaX refactorings are guaranteed to preserve the external behavior of the code – insofar as those conditions have been properly defined! In practical terms, this means that new refactorings should always be designed with great care, especially with regards to their pre- and post-conditions elements.

Scalability – A RefaX refactoring should be effectively applied to systems of varying sizes, from a few lines of code to millions of lines of code. In practice, the performance of any RefaX-based tool will largely depend on the scalability of its underlying XML processing technologies. Since RefaX was designed from the ground-up in a technology-independent way, tool developers should be able to easily switch to new, more scalable technologies as they become available.

Other important requirements, such as easy of customization and extension, are intrinsic to the framework's object-oriented design, which is described next.

4.2. Selected Implementation Details

At the architectural level, RefaX is organized into a hierarchy of layers, as shown in Figure 2. The first layer, *RefaX Tool*, contains all code developed specifically for a RefaX application (or instance). In this layer, developers are free to use and/or extend any of the framework's basic refactoring services. The second layer, *RefaX Facade*, as the name says, implements the Facade design pattern [12]. It offers a unified interface for the suit of services implemented in the layer below. Those services are a *Conversion Manager*, which abstracts from the details of any model or language specific conversion technology; *RefaX Core*, which encompasses a number of other services for applying refactoring operations and verifying pre- and post-

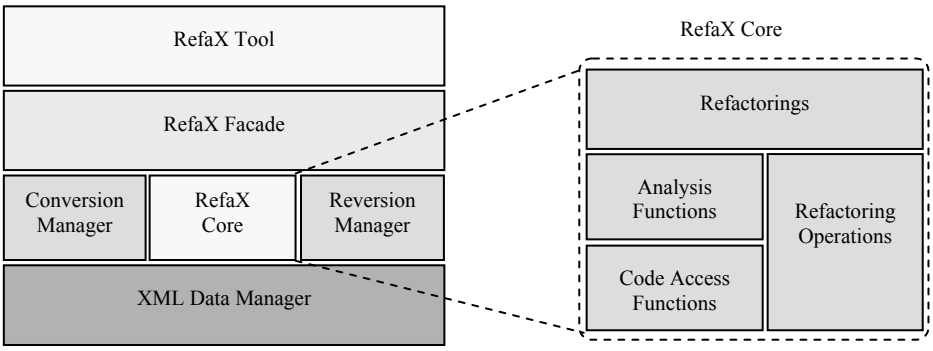

Figure 2. RefaX layered architecture.

conditions; and a *Reversion Manager*, which abstracts from the details of the reversion technology. Finally, the last layer, *XML Data Manager*, abstracts from the details of all XML processing technologies used, including the storage medium, query processor, and update engine. All three "manager" layers implement one or more instances of the Abstract Factory design pattern [12].

The RefaX Core services are at the heart of the framework, as they implement some of its most important requirements, namely, scheme-independence, language-independence, and reliability. Scheme-independence is achieved by encapsulating all XML data queries that access source code information directly in the form of scheme-specific XQuery functions called *Code Access Functions* (CAFs). To give but two examples, the CAF *getClass($doc)* returns all XML nodes that correspond to class definitions in a given XML document *$doc*, and the CAF *getClassName($class)* returns the name of the class represented by a given XML node *$class*.

Because CAFs embody knowledge on how source code elements are defined and represented in XML, as it is the case with *getClass* and *getClassName*, which embody knowledge on class definition, their implementation is always bound to a specific source code model. On the other hand, functions that call one or more CAFs will be completely ignorant to (and hence isolated from) the knowledge they embody, thus improving their reusability.

The set of all CAFs for a given source code model constitutes a scheme-specific layer upon which different sets of scheme-independent analysis functions (AFs) can be specified. In addition, the framework supports the definition of new AFs as a composition of existing AFs. This feature, which complies with the notion of primitive and derived analysis functions, as proposed by Roberts [25], makes it possible to define higher-level AFs in terms of common language features that are completely independent of a particular programming language. Since AFs only access source code information, though at a higher abstraction level than CAFs, they are also implemented by means of XQuery functions.

Refactoring operations in turn are scheme-dependent, as they need to access and change the source code structure directly. Therefore, they should be implemented using an XML update language. In the current version of RefaX, we implement refactoring operations as XUpdate expressions.

Finally, at the source code level, RefaX consists of a Java API (*Application Programming Interface*) with the necessary classes and interfaces to implement refactoring tools according to the XML-centric process described in the previous section. Our choice for Java as the main implementation language is due to, among other reasons, its platform-independence nature and, more importantly, the fact that nearly all XML conversion and processing technologies currently available offer a Java API as a built-in feature.

4.3. An Example Using the *Add Attribute* Refactoring

RefaX currently supports several of the language-independent refactorings built upon the FAMIX meta-model [29]. To illustrate how new refactorings can be implemented using the RefaX framework, here we briefly discuss the implementation of a particular FAMIX refactoring, namely *AddAttribute(className, attributeName)*. This refactoring has the effect of adding an attribute named *attributeName* to the definition of a given class named *className*. Applying the *addAttribute* refactoring requires that the following pre-conditions be satisfied:

- There must be no attribute named *attributeName* in the inheritance hierarchy of the given class.
- There must be no class named a*ttributeName.*

These two pre-conditions were expressed using four AFs, namely *IsClass*, *AllSuperclasses*, *AllSubclasses* and *DefinesAttribute*, whose language-independent implementation in XQuery is shown Figure 3.

From their XQuery expressions we can see that those AFs in turn call five code access functions (CAFs), namely *getClass*, *getClassName, getSuperclassName*, *getAttribute*, and *getAttributeName*. Since they are all scheme-dependent, those CAFs have to be provided with different XQuery implementations for each of the source code models at hand.

The refactoring itself was implemented through a single refactoring operation, expressed in XUpdate, which inserts a new attribute element, named *attributeName*, into the XML-based

```
declare function IsClass ($doc as element(),
$className as xs:string) as xs:boolean
{
  for $c in getClass($doc)
  return
    getClassName($c) = $className
};

declare function Superclass ($doc as element(),
$className as xs:string) as element()
{
  for $c in getClass($doc)
  where getClassName($c) = $className
  return
    getSuperclassName($c)
};

declare function AllSuperclasses ($doc as element(),
$className as xs:string) as element()
{
  let $d := $doc
  return
    if (Superclass($d, $className) = string("")) then
      string("")
    else
      Superclass($d, $className)
};

declare function DefinesAttribute ($doc as element(),
$attributeName as xs:string) as xs:boolean
{
  for $c in getClass($doc),
      $a in $getAttribute($c)
  return
    getAttributeName($c) = $attributeName
};
```

Figure 3. XQuery analysis functions used in the *AddAttribute* refactoring.

definition of the class *className* in the repository. That refactoring operation, along with its corresponding implementation in Xupdate, was then encapsulated as a new Java class. The decision to encapsulate every refactoring operation as a new class has the disadvantage of increasing the total number of classes necessary to implement a refactoring in RefaX. On the other hand, having each operation implemented as a separate class also has the advantage that it facilitates refactorings composition and further improves their reusability.
Finally, to guarantee that the *AddAttribute* refactoring does preserve the external behavior of the original program, the following post-conditions must be satisfied:

- The *className* class defines an attribute named *attributeName.*
- There is no access to the *attributeName* attribute.

These post-conditions were implemented in XQuery by calling the following AFs: *DefinesAttribute*, which checks whether a given class defines a given attribute; and *MethodsThatAccessAttribute* and *ConstructorsThatAccessAttribute*, which return all class methods and all class constructors that access a given class attribute, respectively.
It is worthy noting that the *Add Attribute* refactoring, as it is the case of all other refactorings supported by the RefaX framework, is specified only in terms of syntactic code constructs. In practice, syntactic information alone may be insufficient to express a number of other types of refactoring operations, such as those that require semantic knowledge and dynamic flow information [21].

5. RefaX Tools

To demonstrate the viability of the RefaX framework, we conducted a case study in which we used RefaX to implement a refactoring prototype for Java, called *RefaX4Java*, and other for C++, called *RefaX4C++*. Theses two prototypes were useful to investigate the level of reusability offered by the framework, and to show that its tools satisfy the requirements discussed in section 4.1. In this section, we describe the set of source code models and technologies used for each prototype, and discuss some of the issues involved in their implementation.

5.1. RefaX4Java

We developed two versions of RefaX4Java: one applicable to the JavaML source code model proposed by Badros 1, which we call Badros's JavaML, and the other applicable to the JavaML source code model proposed by Mamas and Kontogiannis [19], which we call M&K's JavaML. With these two versions, we intend to illustrate, in a more concrete way, how the framework supports the requirements of scheme-independence and technology-independence.
We chose the two JavaML models because both have converters and reverters freely available. Moreover, both define XML representations upon which one can easily specify code analysis functions and refactoring operations, with Badros's version offering an extra advantage, since it represents source code entities using shallower node hierarchies. This characteristic is particularly attractive as it facilitates specification of node removal operations.
The framework hot spots for each version of RefaX4Java were implemented with the following technologies:

- *Code access functions*: code access functions were provided for each JavaML model according their XML schema;
- *Converter*: for Badros's JavaML we used Jikes, an adaptation of IBM's Java compiler provided Badros himself, while for M&K's JavaML we used RET4J [24], a toolset

for analysis and transformation of Java programs;

- *Query engine*: the IPSI-XQ query processor [14] was used for both versions, as it supports the latest XQuey specification;
- *Refactoring specification language*: XUpdate [36] was used for both versions, as it is a "real" XML update language that makes it easier to specify update queries;
- *XML data repository*: Xindice [35], a freely available native XML database, was used for both versions, as it supports the latest XUpdate specification;
- *Update engine*: we used the update engine embedded in the XIndice both cases;
- *Reverter*: for Badros's JavaML we used XSLT transformations provided by Badros himself, while for M&K's JavaML we again used RET4J.

5.2. RefaX4C++

The RefaX4C++ prototype was developed with the intent of showing that the RefaX framework satisfies the language-independence requirement. However, in contrast to the Java prototype, this prototype was much more difficult to implement due to various reasons. First of all, we found only two XML-based representations for C++, both named CppML. One was proposed by Mamas and Kontogiannis [19], while the other is provided as part of the Columbus CAN reverse engineering tool [8]. Of those two representations, only the former had a converter available, but with no reverter. Since developing a new reverter from scratch was beyond our goals, we did not implement any support for the reversion step in RefaX4C++. Another difficulty was due to the fact that the C++ grammar is considerably more complex than the Java grammar, which required the specification of a greater number of analysis functions, code access functions, and refactoring operations to cover the particularities of the language. Refactoring operations were also more difficult to implement, due to the increased complexity of the underlying XML scheme.

The framework hot spots for RefaX4C++ were implemented with the following technologies:

- *Code access functions*: the code access functions were implemented according to the XML schema used in Columbus CppML model;
- *Converter*: we used the one provided by Columbus CAN;

For all the other hot spots, with the exception of the reverter, which was not available, we used the same set of technologies used to implement RefaX4Java.

5.3. An Example

To give a more concrete example of how the requirements of scheme-independence and language-independence can be achieved with the framework, Table 1 shows the XQuery implementation of the set of code access functions used in the *Add Attribute* refactoring for each of the two JavaML models and CppML. It should be noted that those functions only return references to XML nodes or strings, which in turn can be manipulated by higher-level analysis functions without concern about their internal structure.

As described in section 4.2, refactoring operations are scheme-dependent and thus need to be implemented separately for each source code model at hand. Figure 4 shows the XUpdate expression for the *Add Attribute* refactoring operation in Badros's JavaML representation. An example of the results of applying this operation to a Java code fragment, represented in Badros's JavaML, is given in Figure 5. Due to space restrictions, we omit the XUpdate implementation of that operation for M&K's JavaML and CppML. It should be noted that this particular refactoring only requires modifications to a single class. Refactorings that involve modifying multiple classes, such as the *Rename Class* refactoring, can also be easily expressed using XUpdate's node match and update capabilities.

Table 1. Examples of XQuery code access functions and their implementation in three different XML-based source code representations.

Code Access Function	Representation-specific Implementation		
	JavaML (Badros)	JavaML (M&K)	CppML
getClass($doc)	$doc//class	$doc//ClassDeclaration	$doc//Class
getClassName($cls)	$cls/@name	$cls/UnModifiedClassDeclaration/ @Identifier	$cls/@name
getSuperclass($cls)	$cls/superclass	$cls/UnModifiedClassDeclaration/ Name	(unsupported)
getSuperclassName($scls)	$scls/@name	$scls/@Identifier	$scls/@name
getAttribute($cls)	$cls//field	$cls//FieldDeclaration	$cls//Object
getAttributeName($att)	$att/@name	$att/VariableDeclaratorId/ @Identifier	$att/@name
getMethod($cls)	$cls/method	$cls//MethodDeclaration	$cls//Function [@kind="fnkNormal"]
getConstructor($cls)	$cls/constructor	$cls//ConstructorDeclaration	$cls//Function [@kind="fnkConstructor"]

```
<xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:insert-after select="/java-source-program/java-class-
file/class[@name='className']/superclass">
    <xupdate:element name="field">
      <xupdate:attribute name="name">attributeName</xupdate:attribute>
      <xupdate:attribute name="visibility">private</xupdate:attribute>
      <xupdate:element name="type">
        <xupdate:attribute name="name">Object</xupdate:attribute>
      </xupdate:element>
    </xupdate:element>
  </xupdate:insert-after>
</xupdate:modifications>
```

Figure 4. XUpdate implementation of the *AddAttribute* refactoring operation using Badros's JavaML representation.

6. Related Work

Even though refactoring techniques have only been recently proposed, at least when compared to other more traditional maintenance techniques, there is already a considerable body of work available in this area. A recent survey can be found in [21]. Here we focus only on those works that are more closely related to ours.

In [19], Mamas and Kontogiannis describe an integrated software maintenance environment, called ISME, upon which developers can build new maintenance tools via the integration of existing CASE tools. The ISME environment can be regarded as an extra abstraction layer sitting between the source code being maintained and the maintenance tools. Although we share some of Mamas and Kontogiannis's research goals and techniques, particularly the use of XML-based source code representations to improve tool customization and reuse, their work differs from ours in that they do not apply XML technologies to actually manipulate source code information. Instead, they use XML only as a standard software exchange format to facilitate integration among ISME's constituent tools. For source code manipulation they still rely on the internal (i.e., closed) representations and processing mechanisms provided by an external manipulation tool.

```
package client;
import clientFunctions;
public class Client {
  private String name;
  private Object attName;
  public String getName() {
    return name;
  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<java-source-program>
<java-class-file name="Client.java">
<package-decl name="client"/>
<import module="clientFunctions"/>
<class name="Client" visibility="public" line="5" col="0"
end-line="12" end-col="0">
  <superclass name="Object"/>
  <field name="attName" visibility="private" line="6" col="2"
end-line="6" end-col="21">
    <type name="Object"/>
  </field>
  <field name="name" visibility="private" line="7" col="2"
end-line="6" end-col="21">
    <type name="String"/>
  </field>
  <method name="getName" visibility="public" id="Client_mth-20"
line="7" col="2" end-line="9" end-col="2">
    <type name="String"/>
    <formal-arguments/>
    <block line="7" col="26" end-line="9" end-col="2">
      <return><var-ref name="name"/></return>
    </block>
  </method>
</class>
</java-class-file>
</java-source-program>
```

Figure 5. A Java code fragment (top right) and its corresponding representation in Badros's JavaML (bottom), as modified by the *AddAttribute* refactoring operation. The added code is highlighted in boldface.

In [29], Tichelaar *et al.* describe FAMIX, a language-independent source code meta-model. FAMIX has been used to develop a variety of language-independent maintenance tools, including a refactoring engine, named Moose, for both Java and Smalltalk programs. The work on FAMIX differs from ours in that they focus exclusively on the language independence requirement. In particular, Moose relies on a set of in-house source code models and manipulation mechanisms, which may be difficult to reuse outside their own execution environment.

Another similar work has been recently reported by Collard in [7], where he proposes an infrastructure to support (semi) automated construction of refactorings via a fine-grained syntax level differencing approach. The general approach is built on top of an XML representation of the source code, specifically srcML. The underlying idea is that, by comparing the syntactic differences between two scrML versions of the same code, with one version representing the code in its original state and the other representing the code after it has been manually modified by a developer, it should be possible to *infer* which refactoring operations have been applied and how. Collard suggests using XSLT [34] to capture that information, so that the detected refactorings could be easily checked and re-executed. That work differs from our work on RefaX in several points. First, Collard uses a fixed set of source code representation and transformation technology (srcML and XSLT, respectively) while we leave that decision open for the developer, so that she can choose the representation and technology that best suit her needs. Second, Collard does not consider pre and post conditions as part of the refactoring detection process, which may comprise the applicability of the detected refactorings in a real-world maintenance scenario. Another significant

difference is that Collard's approach focuses on detecting new refactorings from the syntactic differences between XML representations derived from two different versions of the same source code, while RefaX promotes implementation and reuse, in a flexible way, of well-known refactorings already described in previous works.

7. Conclusions and Future Work

This paper presented RefaX, an XML-centric refactoring framework aimed at facilitating the development of flexible refactoring tools. RefaX provides refactoring tool developers with a number of services to support, among other activities, conversion of source code artifacts into an XML representation; storage of the converted artifacts in the form of an XML-based repository; verification and execution of refactoring operations upon the repository data; and conversion of the (updated) XML representation back to its original textual format. Our main results can be summarized as follows:

- Developing refactoring tools with RefaX not only proved feasible, as we have shown with the two refactoring prototypes for Java and C++, but also increased our confidence that using XML as a standard way to decouple process from technology can be an effective contribution towards the development of more flexible software maintenance tools.
- The use of open, XML-based models and processing standards, in a technology independent way, can offer developers with a greater degree of flexibility in choosing an appropriate set of tools at each step of the refactoring process.
- The requirements of scheme and programming language independence, as supported by the RefaX framework, bring an additional contribution towards promoting refactorings reuse across different source code models, different programming languages, and even different software maintenance environments.

We are currently working to increase the number of refactorings available in the framework, so as to assess the adequacy and pragmatic useability of the proposal. Of particular importance is to investigate how difficult it is to define and implement correct refactoring operations. We also intend to build more refactoring prototypes for other programming languages, particularly those with a well-defined XML-based source code representation already available. Furthermore, it is our interest to apply RefaX-based tools (such as RefaX4Java and RefaX4C++) to systems of varying sizes and application domains, so as to investigate the performance and scalability of their underlying XML processing tools, and to compare them with existing refactoring tools for the same target languages. Another natural line for future work is to integrate RefaX tools with existing IDE's, so as to improve their usability. Finally, we believe that our set of XQuery-based analysis and code access functions could also be useful as a basis upon which to develop other types of maintenance tools, including tools for metrics, program analysis, and reverse engineering. Work in this direction is underway [10].

References

1. Badros, G. J. JavaML: A Markup Language for Java Source Code. Proc. of the 9th International World Wide Web Conference (WWW9), Amsterdam, Netherlands, May 2000.
2. BeaultyJ Home Page. Available at http://beautyj.berlios.de/beautyJ.html. Accessed on 04/03/2004.
3. Beck, K. Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999.
4. C# Refactoring Tool home page. Available at http://dotnetrefactoring.com/. Accessed on

04/03/2004.
5. Carneiro, G., Mendonça, M. Relacionando Refactorings e Métricas de Código Fonte – Um Primeiro Passo para Detecção Automática de Oportunidades de Refactoring. Proc. of the XVII Simpósio Brasileiro de Engenharia de Software (SBES 2003), Manaus, Amazonas, Brasil, October 2003, pp. 51–66.
6. Collard, M. L, Kagdi, H. H., Maletic, J. I. An XML-Based Lightweight C++ Fact Extractor. Proc. of the 11th IEEE International Workshop on Program Comprehension (IWPC '03), Portland, USA, May 2003.
7. Collard, M. L. An Infrastructure to Support Meta-Differencing and Refactoring of Source Code. Proc. of the 18th IEEE International Conference on Automated Software Engineering, Doctoral Symposium, Montreal, Quebec, Canada, October 2003.
8. Columbus Home Page. Available at http://www.frontendart.com/. Accessed on 28/04/2004.
9. Exist Home Page. Available at http://exist.sourceforge.net. Accessed on 04/03/2004.
10. Fonseca, L. A., Mendonça, N. C., and Maia, P. H. M. Towards Reusable Code Analysis Tools Using Standard XML Technologies. Proc. of the I Workshop de Ciências da Computação e Sistemas da Informação da Região Sul (WORKCOM-SUL), Palhoça, SC, May 2004.
11. Fowler, M. Refactoring: Improving the Design of Existing Programs. Addison-Wesley. 1999.
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
13. Holt, R. C., Winter, A., and Schürr, A. GXL: Toward a Standard Exchange Format. Proc. of the 7th Working Conf. on Reverse Engineering (WCRE'00), Brisbane, Australia, November 2000, pp. 162–171.
14. IPSI-XQ Home Page. Available at http://www.ipsi.fraunhofer.de/oasys/projects/ipsixq/index_e.html. Accessed on 04/03/2004.
15. JFactor Home page. Available at http://www.instantiations.com/jfactor/. Accessed on 04/03/2004.
16. Jikes Home page. Available at http://www-124.ibm.com/developerworks/oss/jikes/. Accessed on 04/03/2004.
17. Lehti, P. Design and Implementation of a Data Manipulation Processor for an XML Query Language. Ph.D. Thesis, Technical University of Darmstadt, Germany, 2001.
18. Maletic, J.I., Collard, M.L. and Marcus, A. Source Code Files as Structured Documents. Proc. of the10th Int. Workshop on Program Comprehension (IWPC '02), Paris, France, June 2002, pp. 289–292.
19. Mammas, E. and Kontogiannis, C. Towards Portable Source Code Representations Using XML. Proc. of the 7th Working Conf. on Reverse Engineering (WCRE'00), Brisbane, Australia, November 2000, pp. 172–18.
20. Mayrhauser, A. and Marie Vans, A. Program Comprehension During Software Maintenance and Evolution. IEEE Computer, Vol. 28, No. 8, pp. 44–55, August 1995.
21. Mens, T. and Tourwé, T. A Survey of Software Refactoring. IEEE Transactions on Software Engineering, Vol. 30, No. 2, February 2004.
22. Opdyke, W. F. Refactoring: A Program Restructuring Aid in Designing Object-Oriented Aplications Framework. Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, 1992.
23. Pressman, Roger S. Software Engineering – A Practitioner's Approach. McGraw-Hill, 5th Edition, 2000.
24. Ret4J Home Page. Available at http://www.alphaworks.ibm.com/tech/ret4j. Accessed on 04/03/2004.
25. Roberts, D. B. Practical Analysis for Refactoring. Ph.D. Thesis, Univ. of Illinois at

Urbana-Champaign, 1999.
26. SAX Home page. Available at http://www.saxproject.org. Accessed on 04/03/2004.
27. Tamino Home Page. Available at http://www.softwareag.com/tamino. Accessed on 04/03/2004.
28. Tatarinovi, I., Ives, Z. G., Halevy, A. Y., Weld, D.S. Updating XML. Proc. of ACM Special Interest Group on Management of Data (SIGMOD 2001), California, USA, May 2001.
29. Tichelaar, S. *et al*. A Meta-model for Language-Independent Refactoring. Proc. of the International Symposium on Principles of Software Evolution (ISPSE 2000), Kanazawa, Japan, November 2000.
30. W3C. Document Object Model (DOM). Available at http://www.w3.org/DOM. Accessed on 04/03/2004.
31. W3C. Extensible Markup Language (XML). Available at http://www.w3.org/ TR/xml. Accessed on 04/03/2004.
32. W3C. XML Path Language (XPath) Version 1.0. W3C Recommendation November 1999. Available at http://www.w3.org/TR/xpath. Accessed on 04/03/2004.
33. W3C. XQuery 1.0: An XML Query Language. W3C Working Draft 12 November 2003. Available at http://www.w3.org/TR/xquery/. Accessed on 04/03/2004.
34. W3C. XSL Transformations (XSLT) Version 1.0. W3C Recommendation November 1999. Available at http://www.w3.org/TR/xslt. Accessed on 04/03/2004.
35. XIndice Home Page. Available at http://www.xreftech.com/. Accessed on 04/03/2004.
36. XML:DB. XUpdate – XML Update Language Working Draft. Available at http://www.xmldb.org/xupdate/. Accessed on 04/03/2004.