XIRC: A Kernel for Cross-Artifact Information Engineering in Software Development Environments

Michael Eichberg, Mira Mezini, Klaus Ostermann and Thorsten Schäfer
Software Modularity Lab, Department of Computer Science
Darmstadt University of Technology, Germany

E-mail: {eichberg,mezini,ostermann,schaefer}@informatik.tu-darmstadt.de

Abstract

We describe XIRC, a tool and architecture that enables to define queries over a uniform representation of all artifacts of a software project. These queries can be used for general cross-artifact information retrieval or for more special applications like checking implementation restrictions or conformance to style guides. XIRC is also a good basis to implement a broad range of tools for refactoring, generators, aspect-oriented programming and many other domains on top of it.

1. Introduction

The size and complexity of today's software systems, the ever present need to accelerate the turnover of developers and the constantly changing requirements of software systems make their development, testing, maintenance, re-engineering, and evolution increasingly challenging. In this context, reverse engineering, which we understand in this paper as the process of analyzing a system to *identify the system's components and their relations*, and to *create representations of the system* in various forms supporting different levels of abstraction [8], becomes more important. We use the term *information engineering* (IE) to denote the interconnected processes of *information retrieval* and *information processing* needed to support the identification of a system's elements of interest and their relations, the creation of different representations of these elements and their relations, respectively.

Many "proprietary" tools for retrieving information from particular types of artifacts have been developed in the past. For example, the Eclipse IDE [12] or the Squeak development environment [39] support different search tasks in source code, such as finding all callers of a method or displaying the subclasses of a given class. Other tools, such as Lint [24] try to find potentially erroneous places in the code. Preprocessors, or documentation generators such as Javadoc integrate search engines to find preprocessor statements or documentation comments. Yet other tools are geared towards discovering bug patterns in code [13, 20, 21, 37].

In this paper we make a case for replacing these special-purpose tools by an *open*, *cross-artifact* information engineering platform as an integral part of software development environments. In the following, we will explain what we mean by *open* and *cross-artifact* and outline the advantages of such a platform over a plethora of proprietary tools.

The term *open* has three facets. First, such a platform should be open with respect to the kinds of information that can be searched for. A tool for finding bugs in the code, for example, is typically not open to also find, say, all callers of a method. More open tools like a text search tool, on the other hand, are not powerful enough to find information that depends on the logical structure of the artifacts, because the artifacts are only treated as flat text. Secondly, *open* means that an IE platform should be easily configurable with specific, predefined IE tasks such that the functionality of the aforementioned special-purpose tools becomes a matter of configuring the platform accordingly. For example, the platform should be easily configurable such that it enforces the coding guidelines of a company. Thirdly, such a platform should support a variety of processing techniques. The techniques used for the representation or processing of the retrieved information may be very different depending on the activity to be supported. For example, the visualization techniques needed for detecting patterns that indicate the presence of bugs or violations of best practices differ from those needed to generate documentation or to visualize the structure of a program to help with program understanding.

The term *cross-artifact* refers to information retrieval and processing *across* several heterogeneous software development artifacts. Cross-artifact information engineering is crucial, since the types of the source documents of a software project are very different, including source and binary code, XML deployment descriptors, scripting and configuration files, etc., and the information stored in the documents is tightly related. For example, information about a software developed with Sun's Enterprise JavaBeans (EJB) technology is spread around several different artifacts, such as XML deployment descriptors and their referenced Java classes.

Given the preceding requirements, a key point to make is that a *uniform approach* to all kinds of cross-artifact information retrieval should be provided. That is, the platform should have a built-in generic mechanism by means of which the developers can specify new arbitrary kinds of artifact types such that the search engine and search requests can uniformly work over all different types of artifacts.

None of the aforementioned tools is open and allows for cross-artifact information retrieval. Generally, they only find information in one kind of artifact, and it is not possible to aggregate information that is spread around different types of source artifacts. Furthermore, most of the tools are not open for user-defined retrievals, or do not support information processing in an appropriate way. None of them employ a uniform approach to information retrieval, they are rather specialized on specific kinds of information retrieval.

The goal of the work presented in this paper is to advance the state of the art in this area by making two main contributions: an *architecture* for information engineering platforms and the *design and implementation* of an information engineering platform based on this architecture.

In our *architecture*, all sources of information (artifacts involved in the software development process) are mapped to equivalent representations expressed in a common language: the Extensible Markup Language (XML) [7]. We have chosen XML as our *lingua franca* because it is well suited to represent hierarchical data as found in most documents related to software development projects. For instance, the representation of source code as an XML document is equivalent to the representation of the abstract syntax tree (AST) as an XML document [4]. Furthermore, there is wide support for mapping different types of formats to XML.

Search capabilities over XML documents are needed as the means for information retrieval. We propose the use of the query language XQuery [6] for this purpose, because XQuery is extremely powerful, and several ready-to-use query engines exist for it. This approach enables us to use a single query language on documents as diverse as binary files, source files and XML documents, such that the user is not confronted with many different special purpose query languages [23].

Our *implementation* of this platform, called XIRC (the acronym stands for XML-based Information Retrieval and Conversion), is an open framework that can be extended with new artifact types by registering respective converters to XML, new queries, and new tools for processing the information retrieved by queries. We will outline the design of XIRC and demonstrate its benefits by means of search tasks and checking of implementation restrictions and best practices in the context of the Eclipse IDE. However, it is important to note that the XIRC platform is not restricted to these applications; it is a kernel for supporting a range of reverse engineering tasks such as constructing different views on the program structure, including polymetric views as proposed in [28], refactoring, mining or construction of aspect-oriented language processors. We will outline this view of XIRC as a kernel for reverse engineering and aspect-oriented programming after having presented its architecture and demonstrated its usage by simple examples.

The remainder of this paper is structured as follows. In Sec. 2, we will shortly present XML and XQuery to establish the needed background for understanding the rest of the paper. In Sec. 3, we will present the basic concepts of our approach and show different applications of it. Sec. 4 elaborates on the architecture, implementation, and extensibility of XIRC. Sec. 5 discusses related work. Sec. 6 concludes and outlines areas of future work.

2. XML and XQuery

2.1. XML - Extensible Markup Language

Since it became a recommendation of the World Wide Web Consortium (W3C) in 1998 XML has been successful as a format for data interchange. As opposed to other markup languages like HTML XML allows the specification of user-defined markup tags, thus, it is able to represent arbitrary data. An XML document consists of plain text marked up with tags enclosed in angle braces, which together form so-called elements. The content of an XML element can itself contain other elements, making the structure of an XML document inherently hierarchical. An XML document is called well-formed if it conforms to the syntax requirements of the W3C's XML 1.0 Recommendation. A more stringent property is validity. A document is valid if and only if it is well-formed and adheres to its specified schema. In the following, we only consider valid XML documents.

2.2. XQuery

XQuery [6] is a query language for XML data sources. While XQuery is a functional language comprised of several kinds of expressions that can be nested and composed with full generality, we will only elaborate on the features relevant to this paper. The most important among them is the notion of *path expressions*[1]. In a nutshell, a path expression selects nodes in a (XML-)tree.

For illustration, consider the XML document in Listing 1 representing a simple session bean class named `TestBean` with a default constructor (the method named `init` in line 7).

```
 1 <class name="de.tud.xirc.playground.TestBean"
 2   visibility="public">
 3   <inherits>
 4      <class name="java.lang.Object"/>
 5      <interface name="javax.ejb.SessionBean"/>
 6   </inherits>
 7   <method name="<init>" visibility="public">
 8     <signature>
 9       <returns type="void"/>
10     </signature>
11     <code>
12       <load index="0" />
13       <invoke
14           declaringClassName="java.lang.Object"
15           methodName="<init>">
16         <signature>
17           <returns type="void"/>
18         </signature>
19       </invoke>
20       <return />
21     </code>
22   </method>
23   ...
24 </class>
```

Listing 1: XML representation of a Java class file

We can parse this document by accessing the top-level document node (`class`) of the corresponding tree. Then the path expression `/class/method/code/invoke` selects the `invoke` nodes, resulting in the node spanning line 13 to line 19 in Listing 1.

In general, a path expression consists of a series of *steps*, separated by the slash character. The previous path expression has three steps, namely the *child* steps `method`, `code`, and `invoke`. The result of each path expression is a sequence of nodes. XQuery supports different directions in navigating through a tree, called *axes*. In the path expression above, we have seen the *child* axis. Other axes that are relevant for this paper are the *descendant axis* (denoted by "`//`"), the *parent axis* (denoted by "`..`"), the *ancestor axis* (denoted by "`ancestor::`") and the *attribute axis* (denoted by "`@`"). Using the descendants/ancestor axis rather than the child/parent axis means that one step may traverse multiple levels of the hierarchy. For example, the above query could be rewritten as: `//invoke`.

The attribute axis selects an attribute of the given node, whereas the parent axis selects the parent of a given node. For example, the path expression `//code/../@name` selects all `name` attributes of all method nodes that have a `code` child, i. e., which are not abstract methods. Another important feature of XQuery is its notion of *predicates* – (boolean) expressions, enclosed in square brackets, used to filter a sequence of values. For instance, the query `//method[@name="main"]` selects all methods with the name `main`. One can bind query results to variables, which in XQuery are marked with the `$` character, by means of a `let` expression, as illustrated below.

```
let $concreteMethods := //code/..
return $concreteMethods/[@name = "main"]
```

The `for` construct has the same syntax as `let` but it iterates over all values of the sequence returned by the query.

XQuery also offers a number of operators to combine sequences of nodes, namely `union`, `intersect`, and `except`, with the usual set-theoretic denotation, except that the result is again a sequence with a specific order. The last relevant feature of XQuery is its notion of a function definition. For illustration, the function `diff` is shown below which, being passed two sets `$m1` and `$m2` of method elements (the `*` in `as element()*` stands for "zero to many"), returns the result of the set subtraction operation applied to them.

```
declare function diff
  ( $m1 as element()*, $m2 as element()* )
  as element()* {
  $m1/.. except $m2/..
}
```

3. Cross-Artifact Information Retrieval

Fig. 1 gives an overview of our approach to information engineering platforms which we envisage to be integrated into software development environments. There are three main building blocks of the proposal.

The building block shown on the left-hand side of Fig. 1 consists of the XML converters for different kinds of software development artifacts. The resulting documents are stored in a repository. The second important building block is the query engine, shown in the

1 This subset of XQuery is a separate standard called XPath [10]

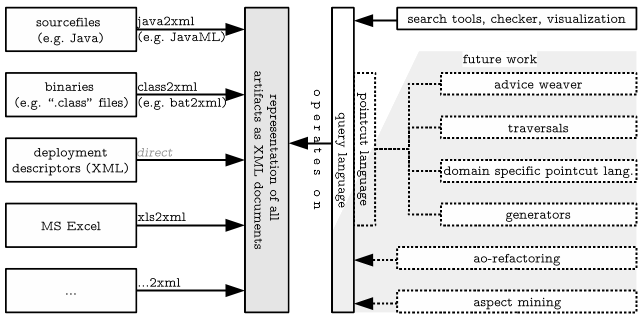


Figure 1. Overview

center of Fig. 1. As already mentioned, the current implementation is based on XQuery, which can be used in two different modes: (a) *selection mode queries* select nodes in existing documents, while (b) *construction mode queries* construct new documents by using information from existing documents. The third building block of the approach consists of various tools, shown on the right-hand side of Fig. 1, which build on top of the query engine and support retrieval and processing of the retrieved information in various ways.

The current instantiation of the approach has only limited support for the last building block. The shadowed area in Fig. 1 shows tools that could be added in the future but are not currently supported. Currently available tools support (a) searching for artifact elements that share some properties, e. g., all persistent fields of a set of classes, and (b) discovering patterns in various artifacts that indicate violations of implementation restrictions, best practices, or design rules (also called "bad smells" in the following). In the remainder of this section we will present the uniform approach for information retrieval underlying XIRC by means of these two kinds of information retrieval tasks, which use selection mode queries only.

For selection mode queries, we provide support to put the resulting elements in relation to the original source documents in order to give direct feedback on the selected elements. Queries in construction mode offer exciting new possibilities, e. g., generation of documentation or support for refactoring tools, and come very handy for the construction of processing tools. The exploration of construction mode queries remains for future work.

3.1. Browsing Through Software

In this subsection, we will introduce XIRC by the role it would play in modern integrated development environments (IDEs) for supporting common search-related tasks that are usually provided by proprietary tools. We first show how two basic search queries as provided by the Eclipse IDE [12] are implemented in XIRC.

The first query retrieves all field declarations in Java files which have a specific type. For this purpose, we declare the function `fieldsByType` below, which selects all field declaration nodes and applies a predicate to their type attribute.

```
declare function fieldsByType($type as xs:string) {
  /class/field[@type=$type]
};
```

For instance, calling this function with the parameter `java.lang.String` selects all field nodes that have this type. This set can be displayed as it is or used for further processing. For example, it is possible to retrieve the declaring classes of the selected field nodes by navigating to their respective parent node.

As a second example, consider the following query which works over a completely different artifact, the Eclipse plug-in descriptors, to show all available extensions to a given extension point.

```
declare function eclipseExtensions($point) {
  /plugin/extension[@point=$point]
};
```

One of the most important features of XIRC is that a single query can span multiple artifacts of different types. For illustration, we show a query that selects all fields of a Java class that are persisted by means of the Hibernate persistence framework [19]. In Hibernate, meta-data required for the object/relational mapping are specified in XML files, as illustrated by the following extract from such a file. This file maps the class `User` to a database table with the same name (line 2). The fields `id`, `username`, `password`, and `email` (lines 3 to 8) are declared persistent, of which the field `id` acts as a primary key.

```
 1 <hibernate-mapping>
 2  <class name="de.tud.xirc.User" table="User">
 3   <id name="id" column="id">
 4    <generator class="native"/>
 5   </id>
 6   <property name="userName" column="name"/>
 7   <property name="password" column="password"/>
 8   <property name="email" column="mail"/>
 9  </class>
10 </hibernate-mapping>
```

The descriptor only states that the `User` class contains these four fields without making explicit whether they have been declared in this class or in one of its super-classes. A developer might be interested to browse over the declarations of the persistent fields of `User`. While the information about the declared fields of `User` is contained in the XML representation of the corresponding Java class file and its super-classes, the information about which fields are persistent resides in the hibernate mapping file. Hence, in order to determine the declarations of persistent fields, one has to query over both kinds of artifacts, as shown below.

```
for $pf in /hibernate-mapping//(property|id)
return /class/field[
    @name=$pf/@name
    and @class=supertypes($pf/../@name)
  ]
```

In the above query, `supertypes` is a simple recursive function that selects all super-types of a given set of types; this functionality is general enough to be provided as a function in a library that can be imported in other arbitrary higher-level queries. We consider the ability to make queries reusable in the form of functions that can be organized in libraries an important feature for creating powerful queries.

The simple introductory examples discussed so far have illustrated three features of XIRC. The first two examples demonstrated XIRC's ability to uniformly retrieve information from different kinds of artifacts. Although the artifacts over which the second query works – Eclipse plug-in descriptors – are structurally organized in a completely different way as compared to Java class files used in the first example, we can uniformely express our retrieval semantics in the same query language in both cases. A query developer just has to know XQuery and the schema of the XML representations of the artifacts used in the queries; (s)he does not need to learn several proprietary query mechanisms for specific kinds of artifacts.

Such an uniform approach is not supported even by modern IDEs such as Eclipse. The latter indeed offers support for finding diverse structures in different kinds of artifacts. For example, there is support for finding information in the help documentation by a simple text search, a Java search offers support for finding Java elements such as types, fields, or methods, and a plug-in search module retrieves information in plug-in descriptors. However, each search module is implemented in a proprietary way; the help search uses the text search engine Lucene [2], while the Java and the plug-in search delegate the queries to the Java, respectively the plug-in model.

The third example illustrated how a single query can obtain information from several different artifacts, and how XIRC would enable IDEs to provide a generic search functionality enabling users to specify their own queries instead of exposing a fixed set of predefined queries for common search tasks. Whenever, using a specific framework such as e. g., Hibernate, a developer could be interested in information that cannot be retrieved with the provided search functionality.

3.2. Detecting "Bad Smells"

Another application of XIRC's information retrieval functionality is to discover patterns in software artifacts which indicate that certain implementation restrictions, best practices and design guidelines are violated. E. g., the EJB specification [11] states that "*The class [an enterprise bean] must not define the finalize() method*" or that "*An enterprise bean must not use thread synchronization primitives to synchronize execution of multiple instances*", etc. Altogether, there are 17 such restrictions in the specification and quite some others not explicitly stated in the specification, the violation causes more or less severe problems which are often very hard to detect before runtime [13]. Other examples of "bad smell" patterns of Java programs are defined in [5,43] and [44]. Such patterns can be naturally expressed as queries in XIRC.

For instance, the following two queries can be used to detect violations of the two exemplary EJB programming restrictions mentioned above. The first query selects all method nodes of all subclasses of the class `EnterpriseBean` whose signature is `void finalize()`. A non-empty result set of this query indicates a violation of the first restriction above and delivers all locations in the bean classes where finalize methods are declared. The second query detects synchronized methods and the usage of the `synchronized` statement in Java source code of any class inheriting from `EnterpriseBean`.

```
subtypes(/class[@name="javax.ejb.EnterpriseBean"])
  /method[
    @name = "finalize"
    and .//returns/@type = "void"
    and not(.//parameter)
  ]
```

```
subtypes(/class[@name="javax.ejb.EnterpriseBean"])
  //(monitorenter | method[@synchronized="true"])
```

A simple example of a Java best practice is: "*Always implement* `public String toString()`" [5]. The following query expresses the semantics for retrieving violations of this guideline by selecting all non-abstract classes (`/class[@abstract="false"]` in line 8) not (`except` operator) in the set of classes that define `toString()` (determined by `toStringMethods()/..`).

```
1 declare function toStringMethods() as element()*{
2   /class/method[
3       @name ="toString"
4       and .//returns/@type="java.lang.String"
5       and not(.//parameter)
6     ]
7 };
8 /class[@abstract="false"] except toStringMethods()/..
```

The sample "bad smells" presented so far can also be detected by other tools such as [3,13,20,37]. These examples are characterized by the fact that the evaluation whether or not a restriction is violated can be done by analyzing a single artifact - Java source code. However, in general, checking for implementation restrictions requires the analysis of different types of artifacts. For example, the EJB specification also states that: *A session bean or a message-driven bean can be designed with bean-managed transaction demarcation or with container-managed transaction demarcation. (But it cannot be both at the same time.)* In order to detect violations of this restriction it is necessary to analyze an EJB's deployment descriptor to determine the chosen transaction demarcation method - `Container` (CMT) or `Bean` (BMT) - and only if it is `Container` the bean's implementation has to be searched for the prohibited usage of programmatic transactions. Such cross-artifact detection is not supported by the tools mentioned above.

To illustrate how such cross-artifact detection of the violations of implementation restrictions, respectively design guidelines, is supported by our approach, consider the following query which discovers beans with declarative transaction demarcation that also use transactions explicitly in their code.

```
1  declare function EJBsWithCMT() as element()*{
2    let $ejbname := /ejb-jar/enterprise-beans
3          /(session | message-driven)/transaction-type
4          [./text() = "Container"]/../ejb-class/text()
5    return /class[@name = $ejbname]
6  };
7  declare function methodsWithBMT() as element()*{
8   //invoke[@declaringClassName
9        = "javax.transaction.UserTransaction"]/
10   ancestor::method
11 };
12 methodsWithBMT() intersect EJBsWithCMT()//method
```

Listing 2: Detecting the co-existence of declarative and programmatic transaction management

The first function (line 1) returns the set of all classes for which declarative transactions are specified. The second function (line 7) returns all methods that call a method to begin or commit transactions. The query itself (line 12) simply returns the intersection between the set of all methods of all classes with specified container-managed transactions and the set of methods using programmatic (bean-managed) transactions.

To see the query from Listing 2 "in action" consider the example source documents in Listing 3 and Listing 4, presenting the source code of a session bean that creates and commits transactions in its code (class `TestBean`) and an excerpt of the corresponding deployment descriptor, where the bean is declared with the transaction type `Container`. The query from Listing 2 detects that the method calls in line 5 and 7 of Listing 3 violate an EJB implementation restriction, based on the information found in the `<transaction-type>` tag in the deployment descriptor in Listing 4 (line 7).

```
1  public class TestBean implements SessionBean {
2    public String getText() throws Exception {
3      UserTransaction utx = ...;
4      String s;
5      utx.begin();
6      ...
7      utx.commit();
8      return s;
9    }
10   ...
11 }
```

Listing 3: Excerpt of a SessionBean's implementation using programmatic transaction demarcation

```
1  <ejb-jar>
2    <enterprise-beans>
3      <session>
4        <ejb-name>HelloWorldTestBean</ejb-name>
5        <ejb-class>TestBean</ejb-class>
6        ...
7        <transaction-type>Container </transaction-type>
8      </session>
9    </enterprise-beans>
10   ...
11 </ejb-jar>
```

Listing 4: Excerpt of a deployment descriptor.

Another example again concerns declarative transaction demarcation. While being a powerful means to facilitate the development of transactional functionality, declarative transaction demarcation also implies some subtle dependencies between the transaction attributes of methods in a call chain, which are not checked for statically. E. g., calling a method `m` declared with the transaction attribute `requiresNew` from within another method `n` which is running in a transactional context will cause a runtime exception to be thrown. In order to detect erroneous patterns of transaction attributes, we need both information of the call graph, which is available via static analysis of Java classes, and information about the transactional attributes of different methods involved in a call graph, which is available from the deployment descriptors.

4. The XIRC Platform

In order to meet the requirements on a platform for information engineering integrated in software development environments, the architecture of XIRC is organized in three layers: application, framework, and data layer, as shown in Fig. 2.

The *application layer* is responsible for initializing and using the XIRC framework. As a first application layer, we implemented an Eclipse [12] plug-in to integrate XIRC into the Eclipse development environment, which is fully functional and can be downloaded from [14].

In order to be able to handle different kinds of artifacts, the application registers so-called processor mappings which aggregate a single input and output processor. An *input processor* is responsible for producing the XML representation of a certain artifact type. *Output processors* on the other hand are responsible for the further processing of the results, i. e. to map the resulting XML nodes of a query to elements of the original artifacts. To enable developers to extend the set of processors by custom implementations, we provide a so-called extension point to which other plug-ins can contribute processors. Input processors, while being specific for a certain type of artifact, are application-independent. On the contrary, output processors are written for a specific application (e. g., for Eclipse) in most cases. For instance, an output processor which is responsible for the visualization of code locations is application-specific, because every application has different capabilities to present these locations of interest in a user-friendly manner.

For using the XIRC framework, the application provides functionality for defining queries and for triggering their execution. The application layer notifies the framework layer about changes of the artifacts. Eclipse has built-in functionality – so-called builders – for tracking such changes. Additionally, the application-specific behavior, such as providing views of the query results for source code in IDEs, is contained in this layer.

The *framework layer* manages artifact updates, query executions, and transformations of the artifacts. The central component of the framework layer is the resource manager which acts as the interface between the application and data layers. After every change of an artifact, the framework is notified by the application layer and all queries in the query manager get re-evaluated. The queries are persisted as serialized objects in a special project folder by the query manager. This enables the session-spanning use of queries as well as the team-wide sharing of queries via a version control system.

The *data layer* is responsible for storing the XML documents representing involved artifacts and to evaluate the XQuery queries. For XIRC, we decided to use Saxon [29], a collection of open source tools for processing XML documents, as the backend. In addition to an XSLT 2.0 and an XPath 2.0 processor, Saxon provides an XQuery 1.0 processor which can be invoked from a Java application by means of an API.

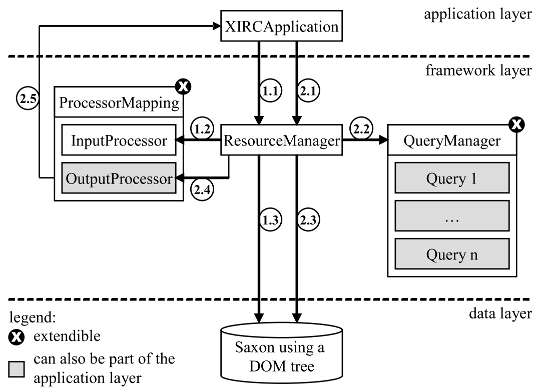


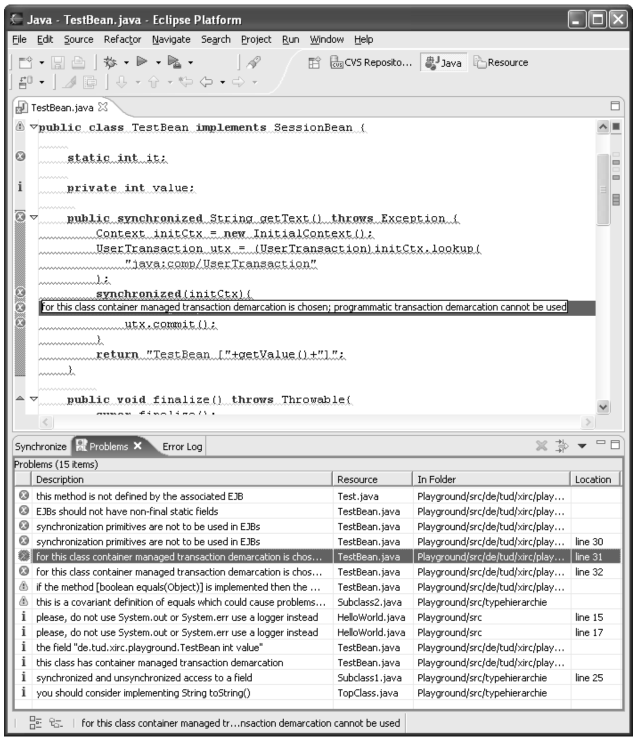
Figure 2. Overview of XIRC's architecture.

We consider some basic usage scenarios of XIRC to understand the interaction between these layers in the following. First, let us see what happens when new artifacts are defined. Assume that we have two input files, e. g., a Java class file and an XML descriptor. The application keeps track of each artifact and calls the resource manager (step 1.1 in Fig. 2) in case of an artifact change. In response to this call, the resource manager searches for input processors that can handle this artifact and delegates the call to the appropriate input processors (step 1.2), which in turn creates an XML representation of the artifact. Finally, the resource manager stores the resulting XML documents in the data store (step 1.3).

New queries are defined using the XIRC GUI. To execute them, the application sends a corresponding request to the resource manager (step 2.1) which obtains the list of all active (selected) queries from the query manager (step 2.2) and forwards each query to the data store (step 2.3). The result of the query is analyzed by the resource manager to determine the correct output processor to use for its further processing (step 2.4). Finally, the selected output processor transforms the results, e. g., it generates new artifacts or maps the elements retrieved by the query execution into corresponding elements in the original artifacts and calls the application to visualize them (step 2.5).

For illustration, Fig. 3 shows the result of evaluating the cross-artifact query discussed in Sec. 3.2. Recall that the purpose of this query was to retrieve locations in the code of EJB components where transactions are used in a programmatic way although the bean is defined with declarative transaction demarcation in the corresponding deployment descriptor. Fig. 3 also illustrates how the result of multiple queries is shown in the current implementation of XIRC.

These queries will be automatically executed whenever a source document changes. As soon as a new violation occurs, the text specified along with the query is shown in the problems view and a marker is added to the source code. The other way around, changing the "erroneous" artifact such as to address the violation, will cause the information and error icons to disappear.


Java - TestBean.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
CVS Reposito... Java Resource
TestBean.java
Synchronize Problems Error Log

```
public class TestBean implements SessionBean {

    static int it;

    private int value;

    public synchronized String getText() throws Exception {
        Context initCtx = new InitialContext();
        UserTransaction utx = (UserTransaction)initCtx.lookup(
            "java:comp/UserTransaction"
        );
        synchronized(initCtx){
for this class container managed transaction demarcation is chosen; programmatic transaction demarcation cannot be used
            utx.commit();
        }
        return "TestBean ["+getValue()+"]";
    }

    public void finalize() throws Throwable{
```

Problems (15 items)

Description	Resource	In Folder	Location
this method is not defined by the associated EJB	Test.java	Playground/src/de/tud/xirc/play...	
EJBs should not have non-final static fields	TestBean.java	Playground/src/de/tud/xirc/play...	
synchronization primitives are not to be used in EJBs	TestBean.java	Playground/src/de/tud/xirc/play...	
synchronization primitives are not to be used in EJBs	TestBean.java	Playground/src/de/tud/xirc/play...	line 30
for this class container managed transaction demarcation is chos...	TestBean.java	Playground/src/de/tud/xirc/play...	line 31
for this class container managed transaction demarcation is chos...	TestBean.java	Playground/src/de/tud/xirc/play...	line 32
if the method [boolean equals(Object)] is implemented then the ...	TestBean.java	Playground/src/de/tud/xirc/play...	
this is a covariant definition of equals which could cause problems...	Subclass2.java	Playground/src/typehierarchie	
please, do not use System.out or System.err use a logger instead	HelloWorld.java	Playground/src	line 15
please, do not use System.out or System.err use a logger instead	HelloWorld.java	Playground/src	line 17
the field "de.tud.xirc.playground.TestBean int value"	TestBean.java	Playground/src/de/tud/xirc/play...	
this class has container managed transaction demarcation	TestBean.java	Playground/src/de/tud/xirc/play...	
synchronized and unsynchronized access to a field	Subclass1.java	Playground/src/typehierarchie	line 25
you should consider implementing String toString()	TopClass.java	Playground/src/typehierarchie	

for this class container managed tr...nsaction demarcation cannot be used

Figure 3. Presenting the result of multiple queries

5. Related Work

The related work can be categorized according to the three architectural layers of XIRC. We first discuss work on XML representations of software development artifacts. Next, we discuss related work on query capabilities over XML documents. Finally, work related to XIRC applications will be presented.

In software development projects the most important artifact is the source code. Numerous papers describe means and advantages of representing source code in XML [30, 38, 45]. There are also tools to represent source code of specific programming languages such as Java [1,15], ANSI C [17], or C++ [31] in XML.

Apart from source code, many other artifacts are generated during the different phases of a software development project. Examples of such artifacts include UML diagrams, stored mostly in the XMI [35] format, or documentation artifacts, which are also often saved in an XML dialect [33, 34, 40]. These XML files can be saved directly in the data store or can be previously transformed using, e. g., XSLT [9].

These approaches show that XML is supported as a *lingua franca* for different artifacts and the availability of tools to create XML representations of those artifacts facilitates the development of input transformers for XIRC.

To retrieve information from the different artifacts, we use the query language XQuery [6] that incorporates XPath [10]. XIRQL [16] is a new XML query language which incorporates imprecision and vagueness for structural and content-oriented query conditions. These features could be used, e. g., to mine possible aspects out of legacy applications. A similar proposal based on XPath is described in [18].

The third field of research we discuss is related to applications of the XIRC platform. In Sec. 3.2 we discussed the use of XIRC for detecting "bad smells" in software. Several tools like JiveLint [41], J2EE Code Validation Tool [21], CodePro Advisor [22], or Assent [42] can be successfully used to find a fixed set of interesting locations in the source code. However, they can not be used to retrieve information in a generic way like XIRC.

There are other tools which are extensible due to more powerful query capabilities, which we discuss in the following. The approach proposed in [27] and applied in [36] uses AspectJ to find interesting locations in code. PMD [37] is a static source code analysis tool which uses lexical analysis to find violations of programming rules. Currently PMD supports more than 80 rule specifications and can be extended by user-defined rules. Such rules can be specified via XPath expressions over the abstract syntax tree or by writing a new Java rule class using the visitor pattern. FindBugs [20] is a similar tool which enables developers to implement restriction checks using the visitor pattern in Java. In contrast to PMD, FindBugs works on the bytecode instead of the source code. IRC [13] is a static analysis tool which uses a bytecode pointcut framework. This enables developers to express filters on class-, method-, and instruction-level properties of code elements. These filters can be applied to all project classes resulting in all elements which match the given filters. This enables an efficient description of interesting patterns in bytecode. MJ [3] is an approach in which checking routines are compiled into compiler extensions. Unlike the tools presented before, MJ provides language support to define checkers.

All these tools but PMD require the checking routines to be implemented programmatically. PMD allows to specify rules declaratively, but only for checks that do not span several classes. XIRC has a declarative approach for defining checks, which can span a single as well as several classes. Developers can provide definitions of new checks via XQuery expressions, using the power of a fully functional programming language. More importantly, to the best of our knowledge, none of the other approaches support cross-artifact information retrieval.

6. Conclusions and Future Work

This section is organized in two subsections. The first part summarizes XIRC features by evaluating them against the requirements we posed in the introduction. The second part gives an outlook into possible further developments of XIRC in the future.

6.1. Summary and Evaluation

This paper made two main contributions. We presented a uniform approach to cross-artifact information retrieval to support information engineering in software development projects and especially reverse engineering based on XML as a common language for representing various artifacts involved in the software development process, and query capabilities on XML documents as the language for specifying arbitrary semantics for retrieving information across the XML representations of different artifacts. In addition, we presented an implementation of this approach, called XIRC, that enables to define queries over a uniform representation of all artifacts of a software project. We have discussed an Eclipse-based instantiation of XIRC and have shown how it can be used for general information retrieval on development artifacts and for checking of implementation restrictions, or best-practice patterns.

While selected individual search-related tasks on which we focused so far might be well supported by individual existing tools, the power of XIRC is that it provides a generic information retrieval kernel across artifact boundaries for a uniform treatment of any retrieval task.

As such, XIRC fulfills the requirements from the introduction. It supports cross-artifact information retrieval and is open for user-defined custom retrievals. In doing so, it provides a single uniform means for specifying such retrievals over multiple heterogeneous artifacts, avoiding the need for the developer to get acquainted with a plethora of diverse tools. Due to its modular architecture, XIRC is customizable to project-specific needs, allowing to select from the available tools and to be incrementally extended with new information processing tools.

6.2. Outlook

Due to its modular architecture, XIRC serves as a good basis to implement a whole chain of processing tools on top of it. Many tasks in the context of software development and reverse engineering require not only to retrieve information of interest, but also to process it in some meaningful way. Tasks like pre-processing, generation of documentation (e. g., Javadoc), refactoring, generating metrics etc., all require sophisticated (cross-artifact) search tools. For example, in the persistence example from Sec. 3.1, we might want to trigger some kind of action after an access to any of the persistent fields happens. An XIRC-aware refactoring tool could process the information gained by a query and do the appropriate modifications in all places selected by the query.

This example indicates that XIRC is a good basis for implementing aspect-oriented tools and languages on top of it. The whole notion of a pointcut [26] can be seen as an abstraction to find information in a source- (static pointcuts) or dynamic call-tree (dynamic pointcuts). We belive that all static pointcuts available in AspectJ [25] can be expressed with XIRC. In order to show the feasibility of this approach we plan to use XIRC as the basis to implement the pointcut model of the aspect-oriented language Caesar [32].

From the discussion above several areas of future work naturally derive. First of all, the aforementioned workbench of tools is a part of our future work. We are currently considering the implementation of two specific tools, namely an aspect-oriented weaver and a refactoring tool that can modify code at search result places. The possibility to let queries return completely new documents instead of selecting existing nodes is also a possibility that has not yet been fully explored. Finally, we plan to explore the benefits of using other XML-related technologies like XSLT in the context of XIRC.

References

[1] A. Aguiar, G. David, and G. Badros. JavaML 2.0: Enriching the Markup Language for Java Source Code, 2004.

[2] Apache Software Foundation. Jakarta Lucene. http://jakarta.apache.org/lucene/.

[3] G. Back and D. Engler. MJ - A System for Constructing Bug-Finding Analyses for Java. Technical report, Stanford University, September 2003.

[4] G. J. Badros. JavaML: A Markup Language for Java Source Code. In *Proc. of the 9th international World Wide Web conference on Computer networks*, pages 159–177. North-Holland Publishing Co., 2000.

[5] J. Bloch. *Effective Java Programming Language Guide.* Addison-Wesley, 2001.

[6] S. Boag, D. Chamberlin, M. F. Fernndez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML Query Language W3C Working Draft 12 November 2003. http://www.w3.org/TR/2003/WD-xquery-20031112/.

[7] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210.

[8] E. Chikofsky and J. Cross. Reverse Engineering and Design Recovery: A Taxonomy. In *IEEE Software*, pages 13–17. IEEE Press, 1990.

[9] J. Clark. XSL Transformations (XSLT) Version 1.0 W3C Recommendation 16 November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[10] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0 W3C Recommendation 16 November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[11] L. G. DeMichiel. *Enterprise JavaBeans Specification, Version 2.1.* SUN Microsystems, 2003.

[12] Eclipse. http://www.eclipse.org.

[13] M. Eichberg, M. Mezini, T. Schäfer, C. Beringer, and K. M. Hamel. Enforcing System-Wide Properties. In A. B. Chaudri, M. Jeckle, E. Rahm, and R. Unland, editors, *Proc. of ASWEC 2004*, pages 158–167. IEEE Computer Society Press.

[14] M. Eichberg and T. Schäfer. XIRC Project. http://www.st.informatik.tu-darmstadt.de/XIRC/.

[15] G. Fischer and J. W. v. Gudenberg. Simplifying Source Code Analysis by an XML Representation. *Softwaretechnik-Trends*, 23(2):49–50, May 2003.

[16] N. Fuhr and K. Großjohann. XIRQL: An XML query language based on information retrieval concepts. *ACM TOIS*, 22(2):313–356, 2004.

[17] K. Gondow and H. Kawashima. Towards ANSI C Program Slicing using XML.

[18] T. Grabs and Hans-Jörg. ETH Zürich at INEX: Flexible Information Retrieval from XML with PowerDB-XML. In *Proc. of the INEX 2002 Workshop.*

[19] Hibernate Project. Hibernate - Object/Relational Mapping and Transparent Object Persistence for Java and SQL Databases. http://www.hibernate.org/.

[20] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. http://findbugs.sourceforge.net/docs/findbugsPaper.pdf, 2003.

[21] IBM. J2EE Code Validation Preview for WebSphere Studio. http://www-106.ibm.com/developerworks/websphere/downloads/j2ee_code_validation.html.

[22] Instantiations, Inc. CodePro Advisor. http://www.instantiations.com/codepro/advisor.htm.

[23] D. Janzen and K. De Volder. Navigating and Querying Code Without Getting Lost. In *Proc. of AOSD 2003*, pages 178–187. ACM Press.

[24] S. C. Johnson. Lint, a C Program Checker, 1978. Unix Programmer's Manual.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. of ECOOP 2001*, pages 327–355. Springer.

[26] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. of ECOOP 1997*, pages 220–242. Springer.

[27] R. Laddad. *AspectJ in Action.* Manning, 2003.

[28] M. Lanza and S. Ducasse. Polymetric Views - A Lightweight Visual Approach to Reverse Engineering . In *IEEE Transactions on Software Engineering 29(9)*, pages 782–795. 2003.

[29] M. H. Kay. Saxon Project. http://saxon.sourceforge.net/.

[30] J. I. Maletic, M. L. Collard, and A. Marcus. Source code files as structured documents. In *Proc. of the 10th International Workshop on Program Comprehension*, page 289. IEEE Computer Society, 2002.

[31] E. Mamas and K. Kontogiannis. Towards Portable Source Code Representations using XML. In *Proc. of WCRE 2000*, pages 172–182. IEEE, 2000.

[32] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *Proc. of AOSD 2003*, pages 90–99. ACM Press.

[33] Microsoft Corporation. Microsoft Office 2003 XML Reference Schemas. http://www.microsoft.com/office/xml.

[34] OASIS. DocBook XML 4.2. http://www.oasis-open.org/docbook/xml/4.2/index.shtml.

[35] Object Management Group. XML Metadata Interchange (XMI) Specification Version 2.0, 3 May 2003. http://www.omg.org/docs/formal/03-05-02.pdf.

[36] Pattern Testing Project. http://patterntesting. sourceforge.net/.

[37] PMD. http://pmd.sourceforge.net.

[38] H. Simic and M. Topolnik. Prospects of Encoding Java Source Code in XML. In *Proc. of the ConTel 2003.* IEEE, 2003.

[39] Squeak. http://www.squeak.org.

[40] SUN Microsystems. *OpenOffice.org XML File Format 1.0*, 2 edition, December 2002.

[41] Sureshot. JiveLint v1.22. http://www.sureshotsoftware.com/javalint/.

[42] TATA Consultancy Services. Assent. http://www.tcs.com/0_products/assent/.

[43] B. A. Tate. *Bitter Java.* Manning, March 2002.

[44] B. A. Tate, M. Clark, B. Lee, and P. Linskey. *Bitter EJB.* Manning, June 2003.

[45] Y. Zou and K. Kontogiannis. Towards a Portable XML-based Source Code Representation. In *ICSE 2001 Workshops of XML Technologies and Software Engineering (XSE)*, May 2001.