XIRC: Cross-Artifact Information Retrieval [GPCE]

Michael Eichberg and Thorsten Schäfer
Software Modularity Lab, Department of Computer Science
Darmstadt University of Technology, Germany
{eichberg,mezini,ostermann,schaefer}@informatik.tu-darmstadt.de

ABSTRACT

In large scale software development projects, in particular in the field of Component-Based Software Engineering (CBSE), different kinds of a project's artifacts are used and related information is spread over the different artifacts. E.g., the transaction attributes ("Require", "Requires-New",etc.) of methods of an Enterprise Java Bean are defined in the deployment descriptor while the method bodies are defined in a Java class. If we want to put these information into relation, e.g., to find all methods with a specific transaction attribute, we have to use different search engines and have to map the information manually. It is not possible to execute "one query" that returns the desired result.

XIRC is a platform that enables to define queries over a uniform representation of all artifacts of a software project. XIRC maps all artifacts of a project to XML representations and stores the documents in a database. The database can be queried using XQuery, a functional query language for XML documents. XIRC can be used as a sophisticated search engine, as a tool to check implementation restrictions, to find errors or as a basis for further tools for code generation and visualization.

1. INTRODUCTION

Information engineering (IE), the interconnected processes of *information retrieval* and *information processing*, is an important and challenging task in software development. While many "proprietary" tools for retrieving and processing information from particular types of artifacts have been developed, most tools do not enable cross-artifact information retrieval.

To close this gap we present XIRC [3], an *open*, *cross-artifact* information engineering platform. The term *open* has two main facets. Firstly, the platform is open with respect to the kinds of information that can be searched for. Secondly, the platform supports a variety of processing techniques, e.g., to visualize information or to generate new artifacts. The term *cross-artifact* refers to information retrieval and processing *across* several heterogeneous software development artifacts. This is crucial since the types of the source documents of a software project are very different, including source and binary code, deployment descriptors, scripting and configuration files, etc., and the information stored in the documents are tightly related.

XIRC uses a uniform representation of all documents to enable cross-artifact information retrieval. All sources of information (artifacts involved in the software development process) are mapped to equivalent representations in XML. To query the XML documents we use the query language XQuery [1]. This approach enables to use a single query language for diverse artifacts.

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

2. THE XIRC PLATFORM

XIRC's approach to information engineering platforms has three main building blocks. The first one are the XML converters for different kinds of development artifacts. They convert the documents into XML and build up the repository. The second building block is the query engine, which is based on XQuery. The third block of the approach are various tools which build on top of the query engine and support retrieval and processing of the retrieved information in different ways.

Currently XIRC has support for retrieving artifact elements that share some properties, e.g., all persistent fields of a set of classes, or for discovering patterns in various artifacts that indicate violations of implementation restrictions, best practices, or design rules.

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.

The *application layer* is responsible for initializing and using the XIRC framework. As a first application layer, we implemented an Eclipse plug-in to integrate XIRC into the Eclipse IDE, which is fully functional and can be downloaded from:
www.st.informatik.tu-darmstadt.de/pages/projects/IRC
The plug-in provides functionality for tracking changes of artifacts, for defining queries, and for triggering their execution.

In order to be able to handle different kinds of artifacts, the application registers *input processors* responsible for producing the XML representation of a certain artifact type and *output processors*, which are Eclipse specific and responsible for the further processing of the results.

The *framework layer* manages artifact updates, query ex-

ecutions, and transformations of the artifacts. After every change of an artifact, the framework is notified by the application layer and all queries are re-evaluated.

The *data layer* is responsible for storing the XML documents representing involved artifacts and to evaluate the queries.

3. AN EXEMPLARY APPLICATION

In this section, we present an exemplary application of XIRC, namely the discovery of patterns in software artifacts which indicate that certain implementation restrictions, best practices and design guidelines are violated.

We will check two EJB programming restrictions [2]. On the one hand, an enterprise bean must not define the finalize() method, on the other hand "*An enterprise bean must not use thread synchronization primitives to synchronize execution of multiple instances*". 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 [4].

Patterns in the code structure that indicate the violation of these restrictions can be expressed as queries in XIRC. 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"])
```

Note, the sample restrictions presented so far can also be detected by other tools because the evaluation whether or not a restriction is violated can be done by analyzing a single artifact - Java source code / byte 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 "cross-artifact" detection of violations is supported by our approach, consider the following query which discovers beans with declarative transaction demarcation that also use transactions explicitly in their code.

```
declare function EJBsWithCMT() as element()*{
  let $ejbname := /ejb-jar/enterprise-beans
       /(session | message-driven)/transaction-type
       [./text() = "Container"]/../ejb-class/text()
  return /class[@name = $ejbname]
};
```

```
declare function methodsWithBMT() as element()*{
 //invoke[@declaringClassName =
         "javax.transaction.UserTransaction"]/ancestor::method
};
methodsWithBMT() intersect EJBsWithCMT()//method
```

Listing 1: 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 11) 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. For illustration, Fig. 1 shows the result of evaluating this cross-artifact query in our plug-in.

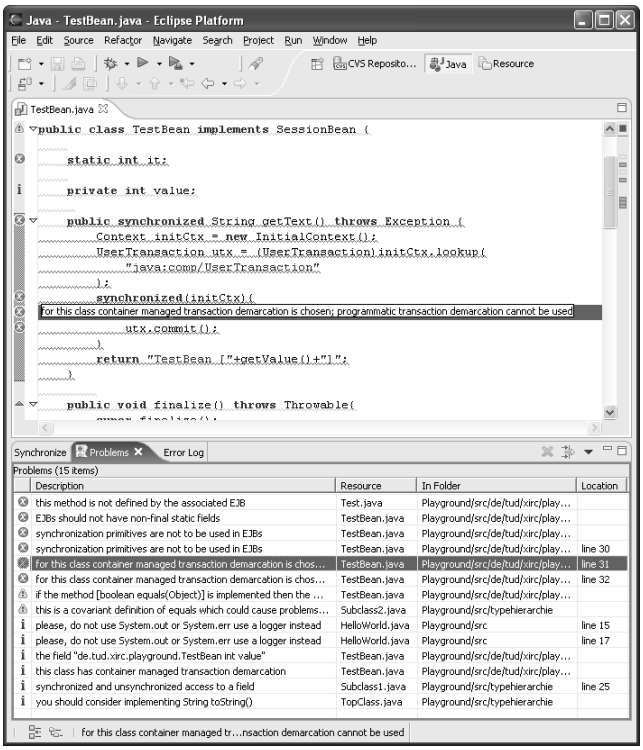

Figure 1: Presenting the result of multiple queries

4. REFERENCES

[1] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.

[2] Linda G. DeMichiel. *Enterprise JavaBeans Specification, Version 2.1*. SUN Microsystems, 2003.

[3] M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer. Xirc: A kernel for cross-artifact information engineering in software development environments. In *Proc. of WCRE 2004*. IEEE Computer Society.

[4] M. Eichberg, M. Mezini, T. Schäfer, C. Beringer, and K. M. Hamel. Enforcing system-wide properties. In *Proc. of ASWEC 2004*. IEEE Computer Society.