Enable an Automatic Validation of OCL Constraints in the Context of Java Data Objects

Matthias Merz und Matthias Gutheil

Discussion Paper 1/2007
January 2007

Arbeitspapiere in der Wirtschaftsinformatik

Matthias Merz
Department of Information Systems III
University of Mannheim
L 5,5, 68131 Mannheim, Germany
merz@uni-mannheim.de

and

Matthias Gutheil
Chair of Software Technology
University of Mannheim
A 5, 6, Part B, 68131 Mannheim, Germany
mg@informatik.uni-mannheim.de

Enable an Automatic Validation of OCL Constraints in the Context of Java Data Objects

Matthias Merz
Department of Information Systems III
University of Mannheim
L 5,5, 68131 Mannheim, Germany
merz@uni-mannheim.de

Matthias Gutheil
Chair of Software Technology
University of Mannheim
A 5, 6, Part B, 68131 Mannheim, Germany
mg@informatik.uni-mannheim.de

Abstract

In this paper we present an approach to validate OCL constraints in the context of Java Data Objects (JDO). Based on RECODER[1]*, a Java framework for source code metaprogramming, we have implemented a source code pre-compiler. Its objective is to modify the source code of a single class by adding the code to handle the validation of OCL constraints. To enable the JDO runtime environment to check these constraints e.g. before a persistent capable object is stored, the pre-compiler implements the methods defined in the* `InstanceCallbacks` *interface. Since JDO represents a data store independent abstraction layer, this approach ensures data integrity for any arbitrary data store. Moreover, as long as the constraints are loaded dynamically from a configuration file at runtime, it ensures also an easy adjustment of OCL constraints without the need of recompilation.*

1. Introduction and Research Overview

In the recent years, more and more tools have been emerged in the Java community to enable developers to implement their applications by using a model driven development (MDD) approach. One objective of MDD is to shift the software development process from a code-centric view to a model-centric perspective. Thus, using a tool like AndroMDA (http://www.andromda.org/) for example, enables an automatic generation of Java source code snippets by using a XMI representation of a UML class diagram. However, one disadvantage of UML is its semiformal syntax and semantics and consequently, the insufficient capabilities to define constraints and rules.

To remedy this situation the Object Management Group (OMG) has introduced the Object Constraint Language (OCL) [9]. OCL is a formal language that can be used to describe expressions on UML models. As specified in [9], these expressions "typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model". Today, in the modeling domain OCL is used for a wide variety of purposes starting from methods to increase the quality of distributed component based systems [1] to the automatic checking of OCL constraints in relational databases [6].

This paper focuses on an automatic validation of OCL constraints in the context of Java Data Objects [3, 4]. In contrast to our JDO example we developed in [2], this paper does not refer to meta model hierarchies for code generation. Instead, we present a new approach to validate OCL constraints based on a Java source code pre-compiler.

The remainder of this paper is organized as follows: Section two introduces the current JDO specification and briefly emphasizes the JDO architecture. Section three outlines the approach to validate OCL constraints in the context of JDO. To highlight the advantages of our approach, section four introduces a more comprehensive example. The last section gives a critical review and addresses areas for future research.

2. The Java Data Objects-Specification

JDO represents an industry standard for object persistence developed by an initiative of Sun Microsystems under the auspices of the Java Community Process [3, 4]. JDO 2.0 was introduced in May 2006 and enables application developers to deal with persistent objects in a transparent fashion way. Thus, JDO as a data store independent abstraction layer enables the mapping of domain object architectures to any type of data store.

The JDO specification defines two packages: The JDO Application Programming Interface (API) allows application developers to access and manage persistent objects.

[1] http://sourceforge.net/projects/recoder/

The classes and interfaces of the Service Providers Interface (SPI) are intended to be used exclusively by a JDO implementation.

The interfaces and classes of the JDO API are located in the package `javax.jdo` [4]. The `JDOHelper` class is used to get the initial `PersistenceManagerFactory`. This instance enables the construction of `PersistenceManager`, that serves as primary application interface and provides methods to control the life cycle of persistent objects. The `Query` interface allows to obtain persistent instances from the data store. Therefore, the JDO specification provides a Java oriented query language *JDO Query Language* (JDOQL).

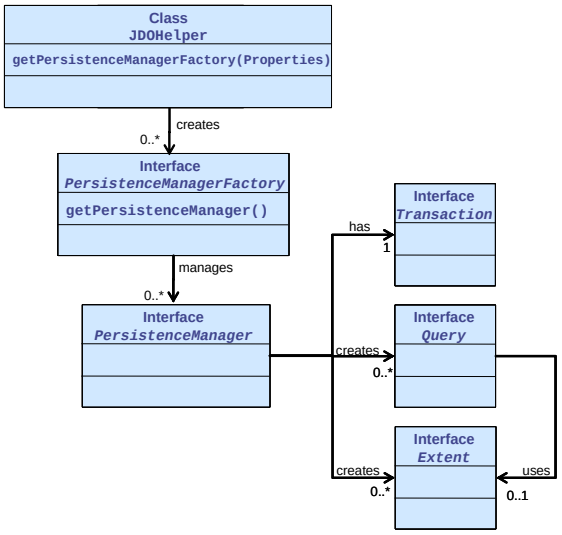

Figure 1. JDO-API

Every instance that should be managed by a JDO implementation has to implement the `PersistenceCapable` interface. As a part of the JDO SPI package, the `PersistenceCapable` interface has not to be implemented explicitly by an application developer. Instead, the JDO specification prefers a post-processor tool (*JDO-Enhancer*) that automatically implements the `PersistenceCapable` interface (cf. figure 2). It transforms regular Java classes into persistent classes by adding the code to handle persistence. A XML-based *persistence descriptor* has to be configured previously. The JDO-Enhancer evaluates this information and modifies the Java bytecode of these classes adequately. The JDO specification assures the compatibility of the generated bytecode for the use within different JDO implementations.

Although JDO provides a standardized, transparent and data store independent persistence solution including tremendous benefits to Java application developers, the JDO

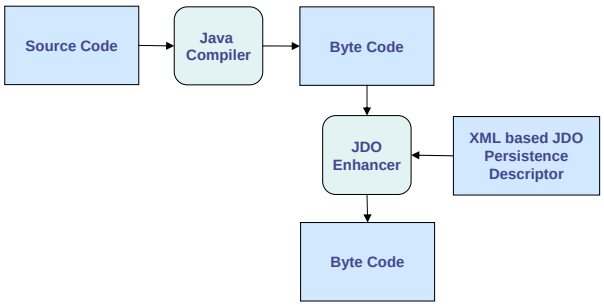

Figure 2. JDO Enhancement Process

specification has been discussed critically in the Java community. Beside technical details like the JDO enhancement process [10], the conceptual design as a lightweight persistence approach without role-based security has been also criticized [7, 8]. Nevertheless, in the Java world, JDO is still the persistence framework of choice, as long as data store independency is assumed.

3. The Validation of OCL Constraints in the Context of JDO

In this section we present our approach to validate OCL constraints in the context of JDO. The first subsection discuss how and when it is possible for a domain application to check constraints before an object will get persistent. The second subsection outlines the pre-compiler approach to modify the Java source code of the domain classes and the last subsection illustrates the preparation of the OCL constraints.

3.1. Intercepting JDO Calls to the Database

To enable the JDO runtime environment to check OCL constraints of an object e.g. before it becomes persistent, the `javax.jdo.InstanceCallbacks` interface has to be implemented for the according class. Instance callbacks provide a mechanism for persistent capable objects to take some action on specific JDO instance life cycle events [4]. In detail, the `InstanceCallbacks` interface defines the following four methods:

- `jdoPreClear()`
  Called before the values in an instance are cleared.
- `jdoPreDelete()`
  Called before an instance is deleted.
- `jdoPostLoad()`
  Called after the values are loaded from the data store into an instance.

- `jdoPreStore()`
  Called before the values are stored from an instance to the data store.

Consider a simple example in the finance domain, where it is necessary to ensure, that the age of a potential customer is not under 18. In this context a given OCL constraint "`context Customer inv: age >= 18`" can approximately be translated for a persistence capable object like in this way:

```
public void jdoPreStore() {
    if ( this.age >= 18) {
        throw new RuntimeException(
            "OCL constraint mismatch");
    }
}
```

When an application tries to make a new `Customer` object persistent, the `jdoPreStore()` method will previously be invoked from the JDO runtime environment. In this case, the age is checked and if a customer is less than 18 years old, a `RuntimeException` is thrown. Therefore, the application that tries to make an object persistent should be extended by a `try-catch`-block to detect potential exceptions.

To release a developer from dealing with code that handles `InstanceCallbacks` methods, we refer to the source code pre-compiler introduced in the following section.

3.2. The Source Code Pre-Compiler

The source code pre-compiler introduced in this section is intended to implement the JDO `InstanceCallbacks` methods to persistent capable objects (cf. section 3.1). As a result, the automatic validation of OCL constraints is possible e.g. before an object is stored to the database. In this context, the basic idea is not to implement the OCL constraints as static statements. Instead, we prefer a more dynamic approach where the OCL constraints will be loaded from a separat file at runtime. This leads to the advantage of a high flexibility and allows an update of OCL constraints without source code recompilation.

To develop a Java pre-compiler, many different solution are available. Our first approach was based on an application that uses regular expressions. However, after a few test scenarios it turns out, that this solution is not sufficient and also highly error-prone. Therefore, we have developed an pre-compiler, that could parse and analyze a source file, transform the source and write the results back to the file. To implement this approach, we use RECODER, a Java framework for source code metaprogramming [5]. The API include classes for Java source code analysis as well as transformation tools. RECODER is under joint development at the University of Karlsruhe and Forschungszentrum Informatik Karlsruhe, Germany. The current version 0.81 was released in May 2006 and supports Java 5. RECODER allows to parse the java source file and to construct an abstract syntax tree (AST). In contrast to the most other tools like ANTLR (ANother Tool for Language Recognition, http://www.antlr.org/) and it's Java grammar, RECODER allows easily to unparse the tree back to source by invoking the `toSource()` method of a `CompilationUnit` (figure 3).

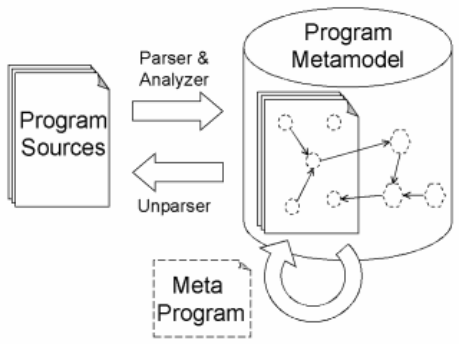

Figure 3. RECODER survey according to [5]

The next section describes how the OCL constraints were prepared and how the source code is modified in detail.

3.3. Preparing OCL Constraints

As described above, the OCL constraints will be loaded dynamically at runtime. Therefore the Java source code pre-compiler needs the information, where the constraints are located and which `InstanceCallbacks` method has to be used to implement the OCL verification. To provide only one meta information file for every class, we decide to enhance the XML based JDO persistence descriptor. As defined in the JDO specification, the `jdo.dtd` allows to add new `<extension>` tags including three attributes, namely `vendor-name`, `key`, and `value` to the persistence descriptor. As shown in the following source code snippet, the `value-name` has to be "OCLEnhancer" in this context, the `key` specifies the `InstanceCallbacks` method and the `value` refers to a file containing the OCL constraint:

```
<class name="Customer">
 <extension vendor-name="OCLEnhancer"
            key="jdoPreStore"
            value="d:\\constraint.ocl"/>
</class>
```

The `OCLEnhancer` we have developed, analyzes the JDO persistence descriptor by using a DOM

(Document Object Model) parser (Apache Xerces, http://xerces.apache.org/xerces2-j/). For every class detected in the XML descriptor the `<extension>`-tag is evaluated. As described in section 3.2 the `OCLEnhancer` uses the RECODER API to enhance the source code of the according class. To keep the lines of added code to a minimum, we have implemented a `OCLHelper` class with two static methods. The `getOCLConstraint()` method loads and parses the `ocl.file` whereas the `checkOCLConstraint()` method returns a `Boolean` whether the constraint is true or not. For the `<extension>` tag presented above, the `OCLEnhancer` adds the following code to the persistence capable class:

```
public void jdoPreStore() {
 String ocl = JDOOCL.OCLHelper.
    getOCLConstraint(
        "d:\\constraint.ocl", this);
 java.util.StringTokenizer t =
    new java.util.StringTokenizer(ocl);
 t.nextToken(" ");
 String attribute = t.nextToken(" ");
 if (! JDOOCL.OCLHelper.
        checkOCLConstraint(ocl,
            attribute, this)){
   throw new RuntimeException(
        "OCL constraint mismatch");
 }
}
```

One remaining problem in this context is to implement a full compatible Java OCL constraint checker. Thus, the current version of our approach uses a simplified constraint validator, that restrict the use of OCL constraints to simple invariants with only one variable. Nevertheless, by using the Beanshell API (www.beanshell.org/) the `checkOCLConstraint()` method is able to validate such constraints dynamically at runtime. To highlight the advantages of our approach in more detail, the next section provide a more comprehensive example.

4. Consumer Loan Example

In the previous sections, we have described our approach to validate simple OCL constraints in the context of JDO. As a proof of concept, we have implemented the following consumer loan example:

As outlined in figure 4, a consumer is represented by his name, first name, age, and a regular income. A consumer is associated with one address and could have one or more loans. In general, Banks use scoring models to rate credit applications and to decide whether or not a customer receive a loan. As result, the commitment of a loan is based on many different pieces of information, e.g. payment history, credit amount, or length of credit history. In this context, we use a simplified model. The rule whether or not a customer receive a loan is represented by an OCL constraint. As shown in figure 4, a consumer will only receive a loan, if his monthly income is more than 1,000 Euro.

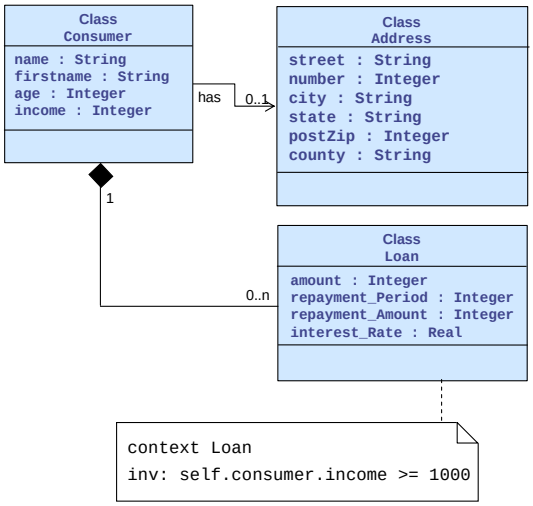

Figure 4. UML class representation of the credit scoring example

The consumer credit scoring example is implemented straightforward. We first implemented the three plain Java classes and defined the corresponding JDO persistence descriptor as follows:

```
<?xml version="1.0"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
 <package name="finance">
  <class name="Loan">
   <extension vendor-name="OCLEnhancer"
              key="jdoPreStore"
              value="d:\\loan.ocl"/>
  </class>
  <class name="Consumer">
  <class name="Address">
 </package>
</jdo>
```

After preparing the `loan.ocl` file as pictured in figure 4 we set-up the JDO environment. For our test scenario we use Java 5, the JDO implementation Xcalia Core (version 4.3.0, http://www.xcalia.com/), a MySQL database and the Java MySQL connector (a JDBC driver for MySQL needed by the JDO implementation). We also add an Apache Ant (version 1.6.5, http://ant.apache.org/)

`build.xml` file, to provide an convenient build mechanism. In the build-file we define to following tasks:

- clean
  Clean up the project directory.
- oclenhancer
  Run the `OCLEnhancer` to modify the Java source code of the `Loan` class.
- compile
  Compilation of the regular and the enhanced java source files.
- enhance
  Run the JDO Bytecode enhancer to transform the plain old java objects (POJO) to `PersistenceCapable` classes.
- defineSchema
  Run `xcalia.lido.DefineSchema` to automatically generate the table structure in the MySQL database.

As it becomes apparent, even the build process of this simple example is quite complex. Nevertheless, with the aid of an automatic build mechanism like Apache Ant, an application developer could still concentrate on the development of the domain classes.

In our test scenario the OCL constraint will be validated in the `Loan` class, before an object becomes persistent. Therefore, the `OCLEnhancer` implements the `jdoPreStore()` method as described in section 3.2 and 3.3. If a bank changes the rule whether or not a customer will receive a loan, one has only to update the OCL constraint in the according file. As in the real world, already committed loans will not automatically canceled only if the rule has changed. Instead, the persistent instances of the `Loan` class remains untouched in the data store. Only if the attributes of a loan contract have to be changed, a database update becomes necessary. Consequently, the `jdoPreStore()` method will be invoked in this case and the new OCL constraint takes effect.

5. Conclusion and Areas for Future Research

In this article we present an approach to validate OCL constraints in the context of JDO. With aid of a source code pre-compiler the JDO runtime environment is able to check OCL constraints e.g. before a persistent capable object is stored. One potential shortcoming of our approach is that the current version could handle at present only simple OCL invariants with just one attribute. In an upcoming version, we plan to enhance the OCL constraint validation, e.g. by using parts of the OCL4Java API (http://www.ocl4java.org/). Nevertheless, since JDO represents a data store independent abstraction layer, the automatic validation of OCL constraints ensures data integrity for any arbitrary data store. Moreover, as long as the OCL constraints are loaded dynamically at runtime, our approach ensures an easy adjustment of them without the need of source code recompilation.

References

[1] A. D. Brucker and B. Wolff. Checking OCL Constraints in Distributed Component Based Systems. Technical Report 157, Department of Computer Science, University of Freiburg, 2001.

[2] R. Gitzel and M. Merz. How a Relaxation of the Strictness Definition Can Benefit MDD Approaches With Meta Model Hierarchies. In *Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics*, pages p. 62–67, Orlando, Florida, USA, 2004.

[3] Java Community Process. JSR-012: Java Data Objects (JDO) Specification, Maintenance Draft Review, 2004.

[4] Java Community Process. JSR-243: Java Data Objects 2.0 - An Extension to the JDO specification, 2006.

[5] Ludwig, A. RECODER Java Framework Website. http://recoder.sourceforge.net/, 2006.

[6] U. Marder, N. Ritter, and H.-P. Steiert. A DBMS-based Approach for Automatic Checking of OCL Constraints. In *Rigourous Modeling and Analysis with the UML: Challenges and Limitations*, Denver, Co., 1999. OOPSLA 99-Workshop.

[7] M. Merz. JDOSecure: A Security Architecture for the Java Data Objects-Specification. 15th International Conference on Software Engineering and Data Engineering (SEDE-2006), Los Angeles, California, July, (accepted), 2006.

[8] M. Merz. Using the Dynamic Proxy Approach to Introduce Role-Based Security to Java Data Objects. Eighteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'06), San Francisco, USA, 5-7. July, 2006.

[9] Object Management Group. Object Constraint Language Specification, Version 2.0. http://www.omg.org/technology/documents/formal/ocl.htm, 2006.

[10] TheServerSide.COM. A Criticism of Java Data Objects (JDO). http://www.theserverside.com/news/thread.tss?thread_id=8571, 2003.