# Re-Documentation of Java with UML Class Diagrams

Martin Gogolla & Ralf Kollmann
University of Bremen
Computer Science Department
Postfach 330440, D-28334 Bremen
{gogolla|kollmann}@tzi.de

**Abstract**

*This paper describes the generation of class diagrams characterizing the static data and class structure of Java source code. To achieve such a diagrammatic representation, translation rules are defined that transform Java syntax into UML notation. Attention is paid especially to implicit and advanced properties of associations.*

*First, an implementation view class diagram of the Java source code is generated that reflects program-specific details of the source code. In a second, more involved step, we try to create a design view from the source code and also represent certain patterns as ODMG language features. The focus of our approach is on minimizing the resulting UML diagrams (minimal with respect to the number of elements in the diagrams) by recognizing certain patterns in the source code and transforming certain low-level syntactic constructs into high-level semantic UML features.*
**Keywords:** *Re-Documentation, Re-Engineering, Association, Java, UML Class Diagram.*

## 1 Introduction

Object-oriented design techniques using languages like OMT [20] or UML [17] and object-oriented implementation languages like C++ or Java are today well-established means for software construction. However changes and additions between the design and implementation task of the software development process occur in every realistic project. Two of the basic reasons for this discrepancy are that usually (1) the levels of abstraction in the different phases force design revisions and (2) different languages are used for the different phases. The fundamental motivation of the approach here is to lift results and modifications of the implementation phase again to the design level but respecting (ideally) all design revisions made in the implementation.

The benefits we see for such an approach are manyfold. By lifting implementation details again to the design level we recognize high-level semantic language features in given low-level programming code. By doing this, programming language code is easier to understand and to communicate. Typical patterns which are candidates for reuse in the development process could be detected by semi-automated means. Furthermore, especially when considering re-implementation of given software on a different platform — for example when one wants to re-arrange the persistent program data by migrating from a simple file system organisation to a database supported framework — it is advisible to recognize the structures which can be expressed within the given data model supported by the current database system and which had to be expressed before with low-level file system structures.

The more concrete scope of this paper is that we want to translate Java code into UML class diagrams. The features of the UML class diagrams we use are very close to the data model language features employed in the ODMG standard [4] for object-oriented databases. We concentrate on class relationships, which in UML terms are called associations between classes. We study in detail the detection of inverse associations and collection types. Both features cannot be directly and elegantly expressed in Java, but UML and ODMG provide language elements for them.

Our work has relationships to many other approaches found in the literature. Apart from the fundamentals of the field presented in textbooks [2, 1] and survey articles [14, 16, 5] we mention recent papers concentrating on design aspects. [22] uses high-level petri nets for a method to reverse engineer Java applets, and [6] concentrates on parametrized Java types. The Rigi environment, an implemented tool for reverse engineering, is used in [24, 23] for the generation of OMT-like scenario diagrams and for evaluating software metrics on Java code. Java applets and version control is one of the main motivations in [13]. Integration of existing information systems and newly developed ones are considered in [12] with UML as a central description language. [3] studies the connection between COBOL and OMT, [19] concentrates on relational databases and OMT, and [25] applies the system ROMEO to a concrete case study for reverse engineering of an application into OMT. The project Fujaba [10] also focusses on UML and Java having in mind a strong background in graph grammars. There are also various approaches [11, 15, 18] which tackle the reverse engineering task by viewing the program source as data in a database and by querying this data about certain properties. The results of this paper are also influenced by our earlier work on re-engineering of ODMG databases [7] and properties of UML class diagrams [8].

The rest of this paper is structured as follows. Section 2 shows how inverse associations can be detected in Java source code. Section 3 explains the details about treating one-to-many associations. The paper ends with concluding remarks in section 4.

## 2 Inverse Associations

In the first step of generating a syntactical representation from source code, references between classes are shown from a class-local point of view[1]. Our translation of the Java source code to UML yields a class diagram that shows only directed associations pointing from one class to another, representing object-valued attributes. If two classes A and B are connected with each other by separate attributes, this is shown as two separate associations, as in figure 1.


Figure 1: Implementation view of an inverse association.

This representation reflects only the implementation point of view: the association between both classes is realized by means of two object-valued attributes each of which is represented by a unidirectional association. As there is exactly one instance of each class on each side of the association, the multiplicities at both association ends do not exceed 1 (multiplicities are handled in more detail in the second part of this section, while one-to-many associations are dealt with in section 3).

[1] Associations are shown from the view of the respective class only. A view of the architecture as a whole is missing at this stage.

Certain semantic information that may have been available in the initial design is likely to get lost in the first stage of re-documentation, namely the UML-inherent information about inverse (or bidirectional) associations. In UML, associations are by default bidirectional: for each association from A to B, there is a respective inverse association from B to A (unidirectional associations are shown explicitly by navigation arrows). Those cases in which the two unidirectional references between A and B refer to a single bidirectional association as in figure 2 have to be identified and require further analysis for a complete and correct view of the architecture.

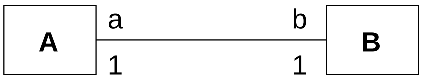


Figure 2: Design view of an inverse association.

### 2.1 Inverse Elements in ODMG

In contrast to the ODMG standard [4], information about inverse associations is neither documented nor expressible in the Java source code.

Some CASE tools, especially some of those that feature roundtrip-engineering, introduce tool-specific extensions to the produced Java code that serve as hints for further reverse engineering. However, the procedure that is described here is meant to provide a general basis for a re-documentation tool. Therefore, it is necessary to rely on information that is contained in the standard unmodified Java source code. The re-documentation algorithms have to work with arbitrary Java codes, not only those produced by a specific tool.

The ODMG-standard provides features to express inverse associations explicitly in class descriptions: classes A and B of figure 2 can be described in ODMG as follows:

```
class A{
    relationship B b inverse B::a;
};

class B{
    relationship A a inverse A::b;
};
```

To represent a Java class containing inverse associations in UML or ODMG notation, it is at first necessary to search the source code for patterns that indicate its existence. The respective source code has to fulfill certain conditions to make a such an association applicable.

## 2.2 Indicator Conditions for Inverse Associations

Since associations are implemented in Java by means of object-valued attributes, it is mandatory that the classes in question have each such an attribute of the type of the respective other class. The attributes have to be class-global[2] to ensure that they are visible for the entire class (and possibly from outside, depending on the visibility).

Furthermore, a hint has to be present that both attributes refer to the same association, i.e. it has to be ensured that bijectivity applies. For two classes A and B with attributes a and b of the type of the respective other class, the following two equations must be true:

$$A.b.a = id(A)$$
$$B.a.b = id(B)$$

Or in other words, viewed from class A,

```
this.b.a = this
```

has to be true.

These conditions are satisfied, if both classes pass a self reference to the other, most ideally from within a synchronized code block, as shown in the following piece of code.

**Pattern 1: A and B have equal rights.** *A and B are instantiated independently from each other by a third class C. The methods linkToA() and linkToB() are called to establish an inverse association between both.*

```
class A{
    B b=null;

    public A(){}
    public synchronized void linkToB(B b){
        if(this.b!=b){
            this.b=b;
            b.linkToA(this);
        }
    }
}

class B{
    A a=null;

    public B(){}
    public synchronized void linkToA(A a){
        if(this.a!=a){
            this.a=a;
            a.linkToB(this);
        }
    }
}

class C{
    public static void main(String[] args){
        // ...
        A a=new A();
        B b=new B();
        a.linkToB(b);
    }
}
```

[2]The term *class-global* is used here to distinguish attributes that are visible everywhere in a class (i.e. normal class attributes) from those that are defined locally in a method or code block. This distinction is particularly important for the concrete implementation of a re-documentation tool.

This approach shows two classes establishing an association by calling two separate methods `linkToA(...)` and `linkToB(...)` that are called with references to the respective other classes as parameter. This approach allows to establish the association at any time during the life of the class instances. Also, it is possible to trigger the establishment from outside, which gives an optional third class that instantiates A and B the opportunity to explicitly link them together.

Please note that the above code represents a pattern in the sense that the elements do not have to appear exactly in the same order they are given. For example, in procedure `main` (class C) from above, the first two statements may also appear in the other order.

Although this pattern shows a good example for the establishment of an inverse association, it has the disadvantage that it is rather unlikely to appear in an everyday scenario. The following example shows a simpler code that is also much more likely to occur.

**Pattern 2: A superior to B.** *A instantiates B and passes a self reference to B's constructor. B stores the reference in a class-global attribute. No further modifications of the attributes occur.*

```
class A{
    // ...
    B b=new B(this);
    public A(){}
}

class B{
    A a=null;
    public B(A a){
        this.a=a;
    }
}
```

The initialization of an attribute as part of the declaration (see class A) and as part of the constructor (see class B) are closely related, because both occur during the construction of the new object. If a third class calls the constructor of A, an instance of B and the association between A and B are created implicitly.

This pattern could also be considered an aggregation with A as whole and B as part, since B's constructor is called by A. Still, as both classes have an identical relation to each other once the constructors have been finished, this difference remains invisible in the respective class diagram. A sequence or collaboration diagram may reflect this superiority of A over B better.

Furthermore, a variant of this pattern is imaginable: A instantiates B in a method instead of the declaration and then passes the self reference to B's constructor, as before. This approach has the advantage that the establishment of the association is not limited to the construction time, but preserves the simplicity of the second pattern.

Summarizing the aforementioned equations for the applicability of bijectivity, it is now possible to construct a generalized rule for the indication of inverse associations, which applies for all patterns:

*If two classes A and B have each a class-global (object-valued) attribute of the respective other type (named a and b, respectively) and instances of both classes are passed a reference to the respective other instance, such that*

$$A.b.a = id(A)$$
$$B.a.b = id(B)$$

*is true, then a bidirectional association exists between both, with A.b and B.a being inverse elements.*

### 2.3 Example

The following source code shows an example implementation of pattern 2 that also demonstrates the difference between unidirectional and inverse associations.

Classes `Controller` and `Dialog` have references to each other that are both initialized at construction time. `Controller` first instantiates `Dialog` and then passes a self reference to it. A slight difference to pattern 2 is the fact that the attribute `Controller.requester` is not initialized in the declaration but in the constructor. This makes no difference in this example, however.

Furthermore, `Dialog` has an attribute of type `Info` that has no reference to its owner and therefore no means to access it.

```
class Controller{
    Dialog requester=null;
    public Controller(){
        requester=new Dialog(this);
    }
    public void processData(Info dm){
        // ...
        // do operation on data
    }
}

class Dialog{
    Controller owner;
    Info newData;
    public Dialog(Controller m){
        this.owner=m;
        // ...
        // create data model
    }
    void update(){
        owner.processData(newData);
    }
}

class Info{

    public Info(){}

}
```

The references between `Controller` and `Dialog` match pattern 2 closely and are therefore represented in the class diagram as a single inverse association: not only do both classes have access to each other, it is also possible to navigate from an instance of one class to an instance of the other and then to return to the original instance by traversing along the inverse association.

The reference to `Info` in class `Dialog` has been established in one direction only. This is shown by a navigation arrow pointing from `Dialog` to `Info` and reflects that `Info` has neither information about nor means to access its owner, `Dialog`.

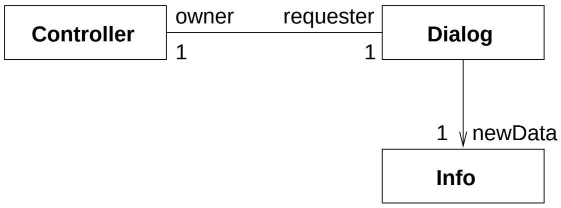


Figure 3: Inverse vs. unidirectional association

### 2.4 Abstraction of Generalizations

A variation of pattern 2 is imaginable, which will be explained by means of the example in section 2.3. Class `Dialog` does not necessarily have to have a reference of the type of the superior class `Controller`. Instead, the reference may point to an abstract super class of `Controller` to keep `Dialog` more general: every class that is a subclass of the new abstract class may now have an inverse association with `Dialog`, not only `Controller`.

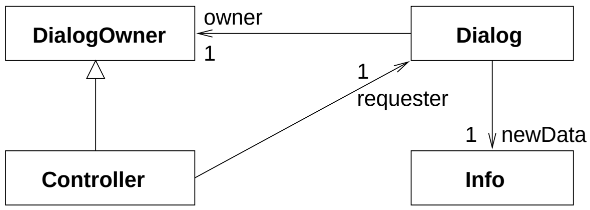


Figure 4: Implementation view of the extended example

This detail is shown in figure 4, with the introduced abstract class named `DialogOwner`. Showing this detail is useful in the implementation view but should be suppressed in the design view, where emphasis is laid on the association between the participating classes, not the way this association is realized in the source code. To discover such a kind of association, it is required that information about all super classes of `Controller` is available. This makes it possible to recognize that `DialogOwner` is a super class of `Controller` and therefore that `Dialog.owner` may point to an instance of `Controller` instead of `DialogOwner`.

### 2.5 Multiplicities

Depending on how and when the self references are passed and if the value of the attributes is modified at a later time or not, the multiplicity at both association ends may vary.

In the simplest case, both attributes are initialized in the declaration block or in the constructor and are not touched afterwards. Here, both association ends are assigned a multiplicity of 1.

As mentioned in section 2.2, we consider initialization of an attribute in a declaration block and in the constructor similar enough to deem it unnecessary to distinguish between both, since the instantiation of a class is not finished until both declaration and constructor have been completed.

If a modification occurs at a later time during the life of the objects, the new multiplicity depends on the assigned value. If null is assigned, the multiplicity becomes 0..1. If a non-void value is assigned, the multiplicity does not change, since exactly one object exists always on the respective side of the association. The fact that this role may be occupied by different objects at different times is irrelevant.

Assuming that an attribute is initialized with null in the declaration block and no non-void value is assigned in the constructor, the multiplicity would have the invalid value 0 at this time. If a complete analysis of the source code provides no further assignments to the attribute in question, the respective association may be removed at least in one direction, as the attribute is not used. If a non-void value is assigned at any time after return from the constructor call, the multiplicity takes the value 0..1.

## 3 One-to-Many Associations

All of the aforementioned cases have in common that not more than one object can be related to another at any time during the life of both. If the multiplicity takes values larger than 1 to allow several objects to be simultaneously in association with a single one, this cannot be modeled through single object-valued attributes. Instead, container classes like those provided by the Java Collections Framework can be used.

The Collections Framework provides three different interfaces for the modeling of object groups with different properties: sets, lists and maps. The interfaces

Set and List are specializations of Collection, the root of the interface hierarchy. The Collection interface represents a group of objects with no further restrictions [9].

By using certain features of associations (both implicit and explicit) it is possible to represent the container classes provided by the Collections Framework in such a way in a UML class diagram that the critical properties of these classes are shown clearly without increasing the complexity of the diagram unnecessarily.

The Set interface models a mathematical set, i.e. an unordered collection that contains no duplicate elements. To represent the Set interface in UML, the utilization of a simple association is sufficient, as stated in the UML Notation Guide:

*If the multiplicity [at one association end] is greater than one, then the set of related elements can be ordered or unordered. If no indication is given, then it is unordered (the elements form a set)* [17].

The multiplicity is specified as *, since a set poses no restrictions on the number of contained elements. By default, navigation is only possible from the container class to its elements, which is shown by a navigation arrow.

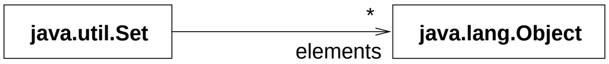


Figure 5: Representation of a set

The List interface represents an ordered collection (also referred to as sequence) [9]. Each element can be accessed through a positional index. A list may contain duplicate elements, which is achieved through the introduced ordering: by assigning multiple indices to a single element, it appears to be contained multiply in the list. As all indices refer to the same object, modifications to the first reference also apply to all others. As we will see, this feature is important for the UML representation, because if a modification would not apply to all references of an object, then it would implicitly create distinguishable clones of the object at all other links.

Because of the way lists work, it is possible to use a qualified association for its representation: the qualifier is used to select an object from the set of objects that the qualified object is related to, i.e. it functions as an index on the set of related objects. Therefore, it reflects closely the model provided by the List interface.

A qualifier has been used for the representation of an array, with the qualifier type being an integer range [21]. However, since a list does (theoretically) not limit the number of elements contained in it, the type int is used here. As a qualifier acts always on associations whose multiplicity is many on the target range [21], it is only possible to limit the range of related elements through the qualifier type.

By specifying the multiplicity at the target association end as 0..1, it is allowed to have indices with no assigned element. This reflects that in an implementation, the number of elements contained in a list is always finite.

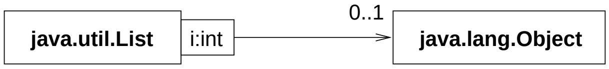


Figure 6: Representation of a list

The Map interface models a mapping from a set of keys to a collection of values [9]. Duplicate keys are forbidden while duplicate values are allowed in the same sense as for lists. The elements are ordered (each element can be accessed directly through its corresponding key).

Concerning the UML representation, the Map interface works very similar to the List interface, but on a more general level: there is no constraint concerning the type of the qualifier. The class diagram of a map may be considered the general case, while the class diagram of a list is a specialization that is limited to mapping from integer qualifiers to objects.

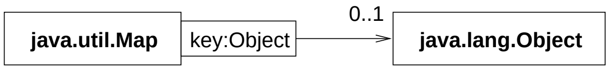


Figure 7: Representation of a map

### 3.1 Relation to ODMG Collection Types

The collection hierarchies of ODMG and Java feature almost identical types, with respect to both syntax and semantics (with the introduction of the Java Collection Framework, the class Dictionary has been obsoleted by the Map interface that has a similar but more extensive functionality).

| Java | ODMG |
|---|---|
| Set | Set |
| List | List |
| Map | Dictionary |

Through the provided collection hierarchies, the conversion from Java to ODMG for storage of persistent objects appears to be relatively easy.

### 3.2 From Implementation to Design View

In this section, we show the differences between the implementation and design views of a one-to-many association exemplary for the interface `java.util.Set`.

In a first step, the implementation level view is generated from Java code. This view reflects directly implementation-specific details. It can be seen that class `Aggregate` has an association to `java.util.Set` (using an aggregation here simply emphasizes that `java.util.Set` is a part of `Aggregate`). Also, we see the default representation of a mathematical set: a simple unidirectional association from `java.util.Set` to its elements, with the elements being from the most general type `java.lang.Object`. The elements that are actually stored in the set are of type `Part`, which is a subclass of `java.lang.Object` (like all Java classes), shown here as a generalization.

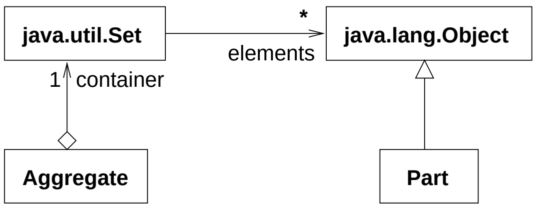


Figure 8: Implementation view of a set

Figure 8 shows the exact relations between all involved classes, which makes it more complicated than necessary to present the set-relation between classes `Aggregate` and `Part`. In the second step, the goal is to suppress all details that are not immediately necessary to understand this set-relation.

As we could see in the previous section, all that is needed to represent a set in UML is a simple association, as it implies all the features that are characteristic for a mathematical set. The associations from `Aggregate` to `Set` and from there to `java.lang.Object` may be suppressed, as they convey no additional semantic information. The resulting class diagram in figure 9 shows only the classes `Aggregate` and `Part` and conveys the implicit information that `Aggregate` is associated to a set of `Parts` which can be accessed via the attribute `Aggregate.container`. Also, the navigation arrow implies that no `Part` element may access its `Aggregate`.

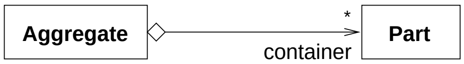


Figure 9: Design view of a set

### 3.3 Example

This section shows an example re-documentation of Java source code that uses the interface `java.util.Map` from the Collections Framework. The class `Store` implements a very simple database for storage of personal information. For each person, an object of type `Person` is created that stores her name. When storing objects in the database, a mapping is established from a unique identification number to the respective `Person` object. The mapping is implemented in the source code by using the class `java.util.HashMap` which is a subclass of `java.util.AbstractMap`, which implements the interface `java.util.Map` from the Collections Framework.

As in section 2.4, it is important to have access to the entire super class tree of the class in question to be able to recognize that it realizes the `Map` interface or sub-classes a class that does so. In this example, the `Map` interface is easy to recognize, as it is used directly.

```
import java.util.*;

class Store{
    Map file=new HashMap();
    public Store(){
        //..
        Person p=new Person("Joan Smith");
        String ID="JS-081154";
        file.put(ID, p);
    }
    public static void main(String[] args){
        new Store();
    }
}

class Person{
    String name;
    public Person(String name){
        this.name=name;
    }
}
```

The result of the first step of generating an implementation view is shown in figure 10. Although the general type of the qualifier for the `Map` interface is `java.lang.Object`, the type `String` is displayed here, as it is used as qualifier in the present source code. The UML notation may become difficult at this point, since it is possible in Java to create an object without giving it a name. With respect to the re-documentation of the Java source code, only the type of the qualifier is important here. If no identifier is available, a generic term has to be used.

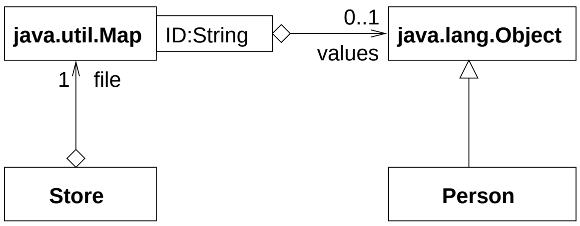


Figure 10: Implementation-view representation of a map

In the second step, the design view class diagram is created, which shows only the immediate relation between `Store` and `Person`: the class `Store` establishes a mapping from `String` values to `Person` objects with at most one `Person` object mapped to each `String` (there are strings whom no person is assigned to, since the number of stored persons is finite).

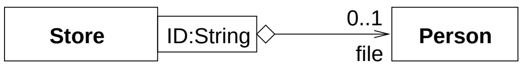


Figure 11: Design-view representation of a map

### 3.4 On Inverse One-To-Many Associations

All one-to-many associations that have been introduced in the preceding sections were by default unidirectional, as they represented only the deposition of a group of objects in a certain container class. To make these associations bidirectional, the conditions for inverse elements as described in section 2.2 have to apply for each element in the container. Once the objects have been instantiated, the respective references have to be established and the part elements have to be added to the container. It is irrelevant for the resulting class diagram, whether this last step occurs before or after the establishment of the references. To show this detail in a graphical manner, collaboration or sequence diagrams could be preferred. This is subject to future considerations.

## 4 Conclusion

In this paper we have studied the generation of UML diagrams from Java code. We have been concentrating on inverse associations and collection types. The basis for handling inverse associations were special patterns in the Java source code. Our treatment of collection types induced part/whole relationships on the considered classes. We also discussed the close connection of the generated UML class diagrams to equivalent language features of the ODMG standard for object-oriented databases. Our general aim was to discover high-level semantic features in low-level programming language code which seems to be possible in a lot of practical cases.

Future work will touch further properties expressible in UML class diagrams like n-ary associations, qualifier, or further treatment of part/whole relationships, i.e. aggregation and composition. Especially n-ary associations seem to be very interesting from the database point of view. Apart from questions dealing with structural properties we will also try to reveal UML statechart diagrams on the basis of given programming language code.

## Acknowledgements

We thank O. Radfelder and M. Richters for fruitful discussions.

## References

[1] P. Aiken. *Data Reverse Engineering: Slaying the Legacy Dragon*. McGraw-Hill, 1995.

[2] R.S. Arnold. *Software Reengineering*. IEEE Computer Society Press, 1993.

[3] J. Bizivin, Y. Lennon, and C. Nguyen Huu Nhon. From cobol to omt: A reengineering workbench based on semantic networks. In *Tools USA' 95 (Technology of Object-Oriented Languages and Systems)*, pages 137–152, 1995.

[4] R. G. G. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.

[5] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[6] Dominic Duggan. Modular type-based reverse engineering of parameterized types in Java code. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 97–113, 1999.

[7] Martin Gogolla, Anne Kathrin Huge, and Bodo Randt. Stepwise Re-Engineering and Development of Object-Oriented Database Schemata. In Roland R. Wagner, editor, *Proc. 9th Int. Workshop Database and Expert Systems Applications (DEXA'98)*. IEEE, Los Alamitos, 1998.

[8] Martin Gogolla and Mark Richters. Transformation Rules for UML Class Diagrams. In Jean Bezivin and Pierre-Alain Muller, editors, *Proc. 1st Int. Workshop Unified Modeling Language (UML'98)*, pages 92–106. Springer, Berlin, LNCS 1618, 1999.

[9] Sun Microsystems Inc. *Java 2 Platform API Specification*, 1999. Electronically available at http://java.sun.com/products/jdk/1.2/docs/api/index.html.

[10] J.-H. Jahnke and A. Zündorf. Implementation of a Distributed Planning and Information System for Courses based on Story Driven Modelling. In *Proc. 9th Int. Workshop on Software Specification and Design*, pages 77–86. IEEE Computer Society, 1998.

[11] Stan Jarzabek. Design of flexible static program analyzers with *pql*. *IEEE Transactions on Software Engineering*, 24(3):197–215, March 1998.

[12] Ulrike Kolsch. Object-oriented reengineering of information systems in a heterogeneous distributed environment. In *Working Conference on Reverse Engineering*, pages 104–114, Hawai, USA, October 1998. IEEE Computer Society, IEEE Computer Society Press.

[13] Jeffrey Korn, Yih-Farn Chen, and Eleftherios Koutsofios. Chava: Reverse engineering and tracking of Java applets. In *Proceedings: Sixth Working Conference on Reverse Engineering*, pages 314–325. IEEE Computer Society Press, 1999.

[14] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[15] A. Mendelzon and J. Sametinger. Reverse engineering by visualizing and querying. *Software—Concepts and Tools*, 16(4):170–182, 1995.

[16] Melody Moore. Representation issues for reengineering interactive systems. *ACM Computing Surveys*, 28(4es), December 1996.

[17] OMG. UML Notation Guide. In OMG, editor, *OMG Unified Modeling Language Specification, Version 1.3, June 1999*, chapter 3. Object Management Group, Inc., Framingham, Mass., Internet: http://www.omg.org, 1999.

[18] S. Paul and A. Prakash. Supporting queries on source code: A formal framework. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):325–348, 1994.

[19] W. Premerlani and M. Blaha. An approach for reverse engineering of relational databases. In *Proceedings of the 1st Working Conference on Reverse Engineering*, pages 151–160, 1993.

[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs (NJ), 1991.

[21] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[22] Giovanna Di Marzo Serugendo and Nicolas Guelfi. Using object-oriented algebraic nets for the reverse engineering of java programs: A case study. In *Proceedings of the International Conference on Application of Concurrency to System Design (CSD'98)*, pages 166–176. IEEE Computer Society Press, 1998. Also available as Technical Report (EPFL-DI No 98/267).

[23] T. Systä. On the relationships between static and dynamic models in reverse engineering Java software. In *Proceedings: Sixth Working Conference on Reverse Engineering*, pages 304–313. IEEE Computer Society Press, 1999.

[24] Tarja Systä and Ping Yu. Using OO metrics and rigi to evaluate java software. Technical Report A-1999-9, Department of Computer Science, University of Tampere, 1999.

[25] Lantzos Theodoros, Helen M. Edwards, Anthony Bryant, and Neil Willis. Romeo: Reverse engineering from oo source code to omt design. In *Working Conference on Reverse Engineering*, pages 191–200, Hawai, USA, Oct 1998. IEEE Computer Society, IEEE Computer Society Press.