Application of UML Associations and Their Adornments in Design Recovery

Ralf Kollmann & Martin Gogolla
University of Bremen
Department of Computer Science
PO Box 330440, D-28334 Bremen
{kollmann|gogolla}@informatik.uni-bremen.de

Abstract

Many CASE tools support reverse engineering and the UML. However, it can be observed that usually, only a subset of the UML notation is supported, namely those parts with a more or less direct code representation. Although a lot of research is done in this field, the more advanced features of UML notations are not commonly supported in reverse engineering.

In this paper, we show approaches to discover patterns in program code that can be represented by means of advanced notational features of UML class diagrams. We obtain the necessary information by reverse engineering Java programs with different methods. These have been implemented in a prototypical implementation.

Keywords: UML class diagrams, association, adornment, aggregation, composition, reverse engineering, design recovery.

1 Introduction

Many approaches to static and dynamic reverse engineering of object-oriented languages exist (e.g. [24][14][20]) and most modern CASE tools like Together [25] or Rose [19] support at least static structure reverse engineering, as well as the UML. However when examining class diagrams generated this way, two observations can be made. First, most tools use only the more basic notational features in reverse engineering, like class compartments, plain associations and specialization. And second, the use of the more advanced features, if implemented, is not always consistent between different tools. A uniform, standardized process for the redocumentation of Java programs with UML diagrams that furthermore aims at an extensive coverage of the notational features of the UML is still missing.

Another problem is caused by the large amount of information often produced by reverse engineering. When taking over this information without further processing, the resulting diagrams tend to become very large, which impairs lucidity and therefore understandability of the program architecture.

This paper is based partially on our previous work [11], where a metamodel-based translation scheme for the redocumentation of Java programs with UML is presented. After a brief discussion of UML association semantics, we present in this paper an approach to recognizing relationship features in Java programs and show rules on how and when to use association adornments for their representation. These rules constitute an extension of our set of basic mapping rules used to translate Java language constructs into UML diagrams.

But let us first explain the context of our work. According to Chikofsky and Cross [3], *design recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction [...] are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself. Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains.* This definition addresses some of the central issues presented below: The prototype that we used to elaborate the approaches discussed in this paper does not rely exclusively on automatic analysis techniques. In some cases, domain knowledge from the user is called in to make semantic-based decisions. The latter is not always possible for a software-system, as it is limited to implementation-specific analysis and deduction efforts.

The work described here has different main goals: as mentioned before, we aim for an extensive and correct utilization of UML notation features in reverse engineering. Features which have been recognized from the Java program code are then translated into UML diagrams by employing mapping rules on the metamodel layer. Additionally, we want to investigate, to what extent the inherent meaning of advanced notational elements can be employed to increase the "density of information" of a diagram: In the common fashion of using the basic notational features, a lot of information can not be represented or causes the diagram size to increase. Ideally, we want the size of the diagrams to remain constant or decrease in the process, while the content of conveyed information increases. Our motivation is to help improving the conciseness of diagrams as well as making them easier to read and understand.

This paper is organized as follows: In section 2, we discuss some preliminaries for our reverse engineering approach and issues concerning the handling of reverse engineered information. Section 3 describes approaches for discovery of UML association adornments from program code. Experiences made with these concepts in a prototypical implementation are presented in context of the respective notation elements, as they often have a strong impact on the presented results. We close with a short evaluation, followed by a summary and conclusion.

2 Design Recovery Preliminaries

2.1 On the Different Meaning of Descriptions from Forward and Reverse Engineering

The meaning of UML features is somehow shifted when comparing design descriptions created in forward engineering with those recovered by (program) reverse engineering. In the former case, rules and constraints are given that have to be enforced by the implementation. An essential difference to information gained by reverse engineering in this context is that the latter contains system structures and behaviour directly encountered in the given source code or program traces, but not always examples which would allow deduction of negative constraints and prohibitions [24, p.76ff]. For example, a trace of a given program may hint on a certain constraint holding for this program. To prove that that the constraint does not hold generally, a negative example has to be found. However, it is not always possible to determine after how many trace runs or under which conditions a negative example will occur at all.

Another issue is that a program may contain information that cannot be extracted from source code alone and may have a slightly varying structure and behaviour for every run. This can be due to external input, e.g. parameterizations or data read from a database. For example, an external input may determine the number of part-elements in a one-to-many association or may influence the results from observing sharing strategies and lifetime of resources (such details are important e.g. when analyzing associations for potential compositions).

This means that when doing runtime analyses, a single program run is often insufficient to get precise and complete information required for the aspired redocumentation. The reverse engineered information describe only observations made on a specific set of runs of a program, i.e. their universal validity cannot always be assured. A similar problem is discussed in [24, p.54], where a "sufficiently complete set of sequence diagrams" is a prerequisite to create statechart diagrams from a program.

2.2 Capturing Program Information

We use two different methods for capturing Java program information. The static structure and some behavioural aspects are extracted using a byte code parser. Additionally, we analyze the behaviour at runtime using a trace generator that bases on the *Java Platform Debugger Architecture* (JPDA) [23] from SUN.

The information gained from the trace generator is used to extend the existing class diagrams by object diagrams that represent the complete object state of the reengineered system. This information about concrete instances is used for different analyses: Firstly, it is the basis for determining dynamic object types and multiplicities. Further, by being able to trace the life lines of objects, it is possible to check for coincidence in lifetimes and discover objects that are referenced simultaneously from different sources. The examination of object sharing is used to check the strong ownership rules of compositions.

A drawback of the JPDA is that is requires debugging information compiled into the byte code. In contrast to static structure analysis with a byte code parser, it is necessary to have the source code present and recompile it. This is rather inconvenient when analyzing program parts that use APIs whose source is not generally present, e.g. Java core classes. However as the Java sources are freely available [13], this is at least in this case no severe problem.

Once the relevant information is extracted from the program, we use three methods to produce an abstract redocumentation. These are discussed below.

2.3 Syntactical Conventions

Using naming conventions for methods and attributes is one possible approach to recognize candidates for access methods of attributes. For further validation, the method's body is searched for read or write access (depending on whether the method name hints on a get- or a set-method) performed on the respective class attribute. Relying on such conventions requires discipline from the authors of the analyzed software, as only a consequent application of the conventions yields usable results in the analysis. This approach has been used before for example in the FUJABA project [15].

2.4 Modeling the Life cycles of Objects

When observing the life cycles of objects, different kinds of events have to be distinguished. The most important of these are construction and destruction of objects, establishment and removal of references as well as sending messages. Rather than measuring the exact time of their execution, a qualitative comparative relation in time between these events is observed. We differentiate three cases : an event e_1 can happen before, simultaneously to or after another event e_2. Not the exact time point of an event is important, but its relation in time to other events.

Obviously, this approach restricts the view on a program's behaviour and it is surely not suitable for all possible systems (especially some applications with real time requirements will require a more precise, quantitative measuring). However, this abstract view fits for most non-time critical applications and selects only the information required in this context. We use behavioural information obtained from a trace generator to reconstruct those structural features that make also assertions about the behaviour, like e.g. composition.

The concept of our view is very similar to (and partially motivated by) representation and reasoning with qualitative differential equations (QDE) [12]. But, as we are working with object-oriented program structures, the information can be represented more fittingly by means of UML object- and sequence diagrams than with the QDE representations. We use object diagrams to hold the system's present object state. With each change of the observed software system, our object model is updated and certain checks are performed on it (these will be explained in section 3). Examples for object diagrams can be found in figures 4 and 5. Sequence diagrams can be used to show the temporal context between different object states, but we won't discuss these here, as our recognition approaches base on the object model and comparison of temporally adjacent object states.

2.5 User-Interaction for Semantic-Based Decisions

Our prototype performs an analysis on the syntactical level, which is sufficient in some situations. Especially concerning elements with complex or ambiguous code representation like e.g. aggregation or n-ary associations, this is not always the case. Here, semantic properties decide whether the employment of complex notational elements is appropriate or if their application would change the meaning in the diagram. Lacking the semantic knowledge, the program cannot always distinguish the clear cases from the ambiguous ones. Consequently, this partitioning may require a decision from the user.

Our approach bases on a semi-automatic process. First, a reverse engineered program is searched for patterns that hint on the applicability of a certain notational element. For some elements, the semantics is formulated in a clear way so that the program is able to handle this on its own (examples for this group are qualifiers and composition). The second group has an imprecisely formulated semantics, so that the program only suggests candidates for the application to the user (e.g. aggregation). The last group contains elements that have no direct correspondent in an object-oriented programming language. Therefore, when using these elements to redocument object-oriented programs, the user has to decide whether the application is sensible and if the program's semantics remains unchanged by their application (e.g. n-ary association).

When reverse engineering very large and complex systems, the amount of user-interaction involved may easily take unacceptable dimensions. We have left this program aspect configurable, which allows to determine the amount of user-interaction. This can also be used to make first a coarse automatic system analysis on the implementation level and then refine the results semi-automatically.

3 Feature Discovery

In this section, we discuss the recognition of UML association adornments from program code or implementation level redocumentation. As the topic of association semantics and whole-part relationships has been discussed thoroughly in many research papers (e.g. [22] [8] [9]), we will give only a short survey of our semantical basis for each feature and proceed then to the actual recognition.

3.1 Inverse Associations

An association is a relationship among two or more specified classifiers that describes connections among their instances [21, p.152].

In our approach, the UML association notation is used primarily to distinguish reference-type class attributes from primitives. Navigation across plain binary associations in UML is implicitly bidirectional [1, p.143]. Object references in Java source code however, are unidirectional and usually represented by CASE tools using navigation to show their direction. The class diagram in figure 1 shows an example for joining two unidirectional associations into a plain bidirectional one.

Plain UML associations without explicit navigability are sometimes used incorrectly in reverse engineering. However, a good example for their deployment in forward engineering can be found in [5], where a correct implementation of bidirectional associations using pairs of class attributes is described. Our recognition of bidirectional associations bases on similar assumptions: we search the program structure for pairs of class attributes (and containers, to cover to-many associations). For containers, the classes of contained objects are determined by analyzing all calls to the container's access methods. This approach is also used in [15]. We use program traces here to get information about the dynamic type of the contained objects. If objects with different classes of the same inheritance tree are stored in a container, we use the tree's root class (except for class Object, which is the root of every Java inheritance tree).

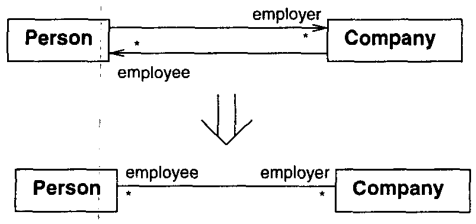

Figure 1. Joining Unidirectional Associations

Although there may be sensible reasons for doing so, adding objects that are totally unrelated concerning their inheritance tree to a single container may be a hint for an imprecise design or implementation. This issue can be solved by representing the container class by a set of 'virtual' aggregations to different classes.

Basically, if two classes have each an attribute of the respective other class, they are candidates for a bidirectional association. Especially if the number of references to the respective other class is greater than one, it is not obvious if and how references have to be joined. It may be that certain references are semantically related. Therefore, our prototype allows the user to interactively join unidirectional associations. For automatic detection, it is possible to search for syntactical conventions: if a pair of role names of two unidirectional associations hints on a semantic relationship, they can be selected for joining. Some of these name pairs are obvious, as e.g. (parent, child). We have found that such conventions often depend on the domain of application and should therefore be configurable.

Navigation is used, whenever the rules for inverse associations do not apply or when the user decides that in spite of a rule-match, the joining into a bidirectional association is not appropriate from a semantic point of view.

3.2 Multiplicities

We examine the contents of container classes to determine multiplicities. Although a container may hold a single element and thus have a multiplicity of 1 too, this value usually refers to a single class attribute. When counting the number of objects stored in a container to determine greater multiplicities, we make only observations. These depend on the number of program traces performed to obtain information from the program and – if possible – configuration settings with an impact on the observed multiplicity.

UML defines several multiplicities that are used primarily: 0..1, 1, 1..*, *. The CASE tool Together [25] for example provides these as default. Starting from a set of multiplicities obtained from program traces, we use three views on multiplicities with different granularities: unchanged (the entire set is shown), condensed as a range (the upper and lower boundaries of the set are given) and abstracted into one of the aforementioned general notations.

For example, supposed that a sequence of program traces yields the following set of multiplicities: {4..7, 5, 5..9, 8, 10}. The condensed range is 4..10 and for a more general view, the multiplicity is abstracted to *.

In the special case that multiple program runs yield a constant multiplicity, its value is not generalized but given unchanged.

3.3 Aggregation

The semantics used in this paper bases on statements in [8], which bases mainly on [18] and [17]. Here, aggregation is distinguished from a plain association by adding a whole-part relationship between aggregate and part and by imposing *forbidden instance reflexivity* on it, which *define[s] a transitive, antisymmetric relationship, i.e. instances form a directed, non-cyclic graph* [18, p.38].

The UML Reference Manual [21] agrees with this definition, but adds furthermore that only binary associations may be declared an aggregation.

The whole-part aspect of the aggregation is difficult to formalize and according to [1, p.146], this is not necessarily desired: *Simple aggregation is entirely conceptual and does nothing more than distinguish a "whole" from a "part".* As this kind of conceptual information is not reflected directly by the program code, it seems often necessary to leave the decision of deploying an aggregation to the user.

However, some special cases exist, when certain hints on a whole-part relationship can be derived from the program. We divide these into one-to-one and one-to-many associations. Special cases of the first kind are most commonly covered by composition and are discussed in the next section. Examination of cases of the second kind depends on the implementation: In Java, the most obvious way to implement a one-to-many association is by means of container classes [5][26].

Java provides a set of collection classes with well defined interfaces. One approach to discover aggregations is to search for these collection classes, which can be done by examining the system's static structure. Then, we try to determine the aggregation's target class, i.e. the class which takes the "part"-role in the aggregation. The approach is the same we described for inverse associations (see section 3.1), by examining all calls to the container's access methods.

Once a set of candidates for aggregation has been discovered, the rules for acceptance as aggregate depend on certain configuration parameters. The simplest way would be to use aggregation whenever container classes are encountered. Clearly, this is a rather imprecise approach. During experiments with a prototype implementation, it could be noticed that by using this behaviour, the user's idea of the aggregation notation tends to shift from "whole-part relationship" to "representation for container-class". This shifts semantics towards a different conceptual view and therefore has to be avoided. The container class is on the implementation level, while the whole-part relationship refers to semantic properties.

Letting aggregation entirely optional led often to discussions whether a certain object or bag of objects was actually in a whole-part relationship or not. After evaluating our results from the prototype implementation, we came to the following attitude towards recognition of aggregations from container classes:

- The automatic recognition of aggregations should stay optional, i.e. the user can determine whether to use it at all.

- When activating this feature, the user must have the opportunity to take a selective choice manually from the set of aggregation candidates.

Forbidden instance reflexivity (FIR) is a constraint encountered relatively seldom in source code. It applies only to associations where the type of the target end is identical to or a generalization of the source end's type. Some typical examples for FIR are shown in figure 2. This constraint does not help much in the discovery of aggregations, but rather in the disqualification of existing candidates for aggregation.

If the declared type of a class attribute is identical to or a generalization of the class owning the attribute, then the object on the 'part'-side of the association may neither directly nor indirectly aggregate the object on the 'whole'-side of the association. This can be formulated in the following OCL-expression, where C is an arbitrary class and `transitiveClosure` refers to the transitive closure of a one-to-many aggregation.

```
C.allInstances->forAll(c|not
(c.transitiveClosure->includes(c)))
```

We have implemented this by maintaining a table of object-IDs for each container and using an implementation of the container's transitive closure for identity checks. When examining artificially created large data amounts (about 300 classes with FIR applying and a DIT (*depth of inheritance tree* [2]) of 2-3), searching the transitive closure took too much time to be acceptable as response for an interactive request. However, this issue had no major relevance for the "real" applications we examined, as FIR did not occur often enough to cause performance losses. We observed from our results that only for a fraction of the aggregation candidates, FIR-checks can be used. (Giving precise numbers for the average occurrence of FIR in program code does not appear useful, as it depends heavily on design style).

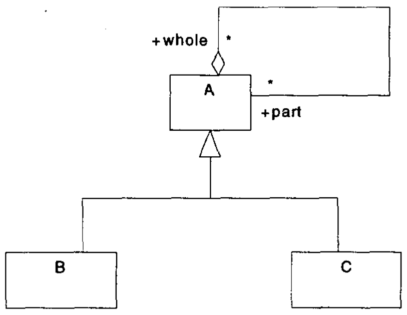

Figure 2. An Aggregation where FIR applies

Although there may surely be applications that make intensive use of this feature, we conclude from our observations that the relevance of FIR for recognition of aggregations in reverse engineering is generally limited.

3.4 Composition

Composition is distinguished from aggregation by requiring additionally *strong ownership* and *coincident lifetime* of the part [17, p.3-73].

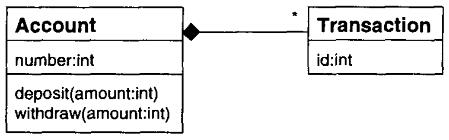

Figure 3. An Example for Composition

Strong Ownership (SO, also called *strong forbidden sharing* [8]) requires that the number of aggregates a part belongs to never exceeds one. The class diagram in figure 3 shows an example for composition, where an `Account` is related to a number of Transactions. A Transaction is created for each call to deposit(...) and withdraw(...) and has only a meaning in context to its Account (it cannot refer to a different account). If an Account is closed, its Transactions are deleted.

The object diagram in figure 4 shows a valid example for Strong Ownership, with two Accounts having different sets of Transactions. Note that the object diagram shows only a snapshot of the system. We can't conclude on the validity of SO until the life cycle of the owner object is finished, as only at this point we can be sure that no further violation of the rule can occur.

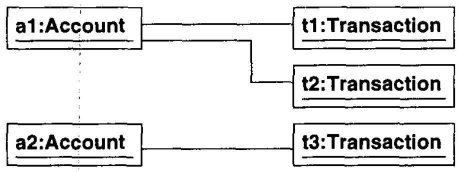

Figure 4. Valid Example for Strong Ownership

Strong ownership is broken, if the number of aggregates referencing a part exceeds one. This case is shown in the object diagram in figure 5, where the second account a2 has established a link to transaction t2, which belongs originally to account a1. We can stop the examination of this case, as a violation has been found.

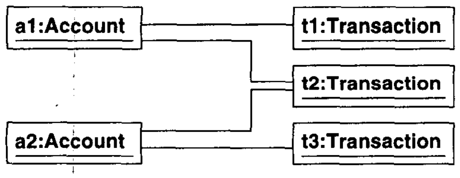

Figure 5. Invalid Example for Strong Ownership

Coincident Lifetime (CL) requires that the initialization of all parts of an aggregate did not occur prior to the aggregate's initialization and that their life cycle ends at the latest when the aggregates life cycle ends.

In [22], this is called a special case of *contained* rather than *coincident* lifetime binding, as the instantiation of the part must be finished simultaneously with *or after* the instantiation of the whole. Other "official" sources specify the semantics for a composition in a similar way:

Composition is a form of aggregation with strong ownership and coincident lifetime as part of the whole. Parts [...] may be created after the composite itself, but once created, they live and die with it. Such parts can also be explicitly removed before the death of the composite. [1, p.147]

Implementation To discover composition, we use checks for strong ownership and coincident lifetime, which are performed at different time points during the life cycle of an object. As they base on searching for violation of these constraints, the complete life cycle has to be examined to be sure that no violation has occurred. Conversely, we can stop our observations as soon as one of the constraints is broken for the first time.

Similar to the link-concept of the Unix-file system, we use a function $ref(o)$ to determine the size of the set of objects, which hold a reference to an object o. We use an extension of the object-ID tables deployed with aggregation here, which allows us to monitor the number of references to an object.

At instantiation of an object A, we examine if references to already existing objects are established and discard these from the group of potential composition parts. For those objects newly created at instantiation of A or afterwards, we check for establishment of references by other objects. Each object with $ref(o) \geq 2$ is discarded. At the end of A's life cycle, all objects with life lines that continue beyond that of A are discarded. The remaining group of parts is regarded as composition parts of A.

So far, we have succeeded in discovering composition in applications, which were known to implement composition according to the aforementioned rules. The source code of the applications we examined was available so that the results could be verified.

In a next step, we plan to examine larger software systems in a similar way as we did for aggregation, to obtain closer hints on the frequency of composition as well as the applicability of our approach for composition discovery when reverse engineering large systems.

3.5 N-ary Associations

According to [17, p.61], *an n-ary association is an association among three or more classes.* Concerning n-ary associations, *the multiplicity of a role represents the potential number of instance tuples in the association when the other N-1 values are fixed.*

In forward engineering, the reification of n-ary associations in design and implementation can be a reasonable move, but has to be considered well. The semantics expressed by an n-ary association differs significantly from that of multiple binary associations, not only because of the different semantics for multiplicities. However, it has been shown that n-ary associations can be translated into a set of binary associations [7]. We want to examine, if the presented rules can also be used for the discovery of n-ary associations in program code and translation in the inverse direction.

First, we search the class structure for classes that "hold three or more other classes together". This means that all of them have in common that they have an association to the central class. Multiplicity has to be 1 at the target end for each association leaving the central class. Classes which fulfill this constraint are considered candidates for transformation into an n-ary association. We use the following example from [4] for illustration.

The class diagram in figure 6 models the delivery of goods between a company, its stores and customers. A `Customer` is associated to a set of `Orders`, which are in turn associated to a `Company`. Companies are also associated to a set of `Stores`. The single class that interconnects all others is `Delivery`: One instance of it is related to exactly one instance of each of the other classes.

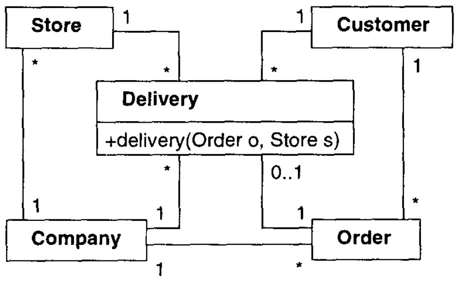

Figure 6. Candidate for N-ary Association

As the constraints for translation are fulfilled, `Delivery` is a candidate for an n-ary association. However, in reverse engineering, the employed translation rules leave open some important points. We will discuss these in context of figure 7, where a new class diagram is shown with `Delivery` transformed into a 4-ary association.

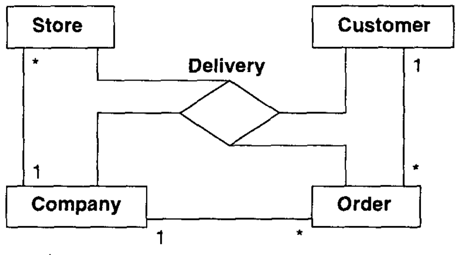

Figure 7. Representation as N-ary Association

Different semantics for n-ary multiplicities. As the semantics for multiplicities differ substantially between binary and n-ary associations, we cannot adopt them directly in the new diagram. In the binary associations, each class had a multiplicity at `Delivery`. As the former class has now turned into an association, there is no longer a target or location for them.

It is possible to solve this problem by representing the lost multiplicities with OCL expressions that are attached to the class diagram. However, as these expressions are not trivial [7], the diagram can quickly become much more complicated than prior to the translation. The original idea of making understanding of the underlying architecture easier by providing a graphical representation is lost to a certain extent, as the reader still has to understand rules given in a kind of programming language.

Furthermore, adding the '1'-multiplicities to the outer association ends of the n-ary association would require that each object of a participating class can be uniquely assigned to a fixed tuple of objects of the other participating classes. This would apply only if all multiplicities at the central class were "1".

Holding information from central class. In the most ideal – albeit rather theoretical – case, the central class has no own state and methods, but only attributes and access methods of references to the other classes in the n-ary association. As this kind of information is only used in context of accesses on the structure, it could be omitted here. If the user considers the information relevant, or if other attributes and methods exist, a means has to be found to represent them.

One solution for this problem is to put the information into an association class. Although not absolutely satisfactory (concerning the condensation of the class diagram), this can be considered an improvement, as the intention of introducing the different semantics of an n-ary association succeeded while still being able to hold the information from the former central class within the diagram. The issue of semantics is considered more important here than displaying a reduced class diagram and worth the additional class.

The bottom line of our experiences from the implementation is that n-ary associations can help to understand and emphasize the role of certain central classes in design. However, this feature should be employed with great care. Candidates for n-ary associations should not be adopted without consideration of the obvious and hidden changes in semantics and information presented in the diagram.

3.6 Qualifier

A Qualifier is an attribute or list of attributes whose values serve to partition the set of instances associated with an instance across an association [17].

As shown in [6], qualifiers may be used for representation of classes from the Java collections, e.g. Map and List. According to [21, p.398ff], qualifiers may also be used to model arrays, with the qualifier type being an integer range. This provides an alternative rendering to the textual field notation, where an array is distinguished from a normal field by adding its multiplicity in square brackets. However, as stated in section 3.1, we employ the field compartment of a class for primitive fields only and use the association notation for representation of relationships between objects. Note that since no "official" notation for integer ranges as types is specified in UML, we employ the notation for multiplicity ranges in square brackets (see figures 8 and 11).

Arrays. Recognition of arrays is straightforward, as all necessary information can be obtained directly from the program structure. The following snippet of pseudo code shows rules used to translate a Java array, represented as instance of our Java metamodel [11], into a UML qualifier, represented as a UML metamodel instance. As qualifier type, we use the array range, as suggested in [21]. The name of the class to which the qualifier is attached, is derived from the Java class that "owns" the respective array. The name of the class at the target side of the association is given by the declared type of the array.

```
//qualifier type
AssociationEnd.qualifier.type.name =
    [0..JMMArrayType.size-1]
// class at association source end:
AssociationEnd.type.name =
    JMMArrayType.owner.name
// class at association target end:
AssociationEnd.association.
                connection[1].type.name =
    JMMArrayType.contentType.name
```

Figure 8. Translating an Array into a Qualifier

Maps. The recognition and representation of the Map collection class depends on knowledge about methods used for handling its contents. The approach works analogous to the discovery of aggregation, by analyzing all calls to the access methods of the Map. The extraction of their parameter types yields the qualifier type and target. In the special case of a Map returning an array of objects (or another object container, for that matter), we render this in a condensed manner by showing its capacity as upper limit of the multiplicity at the association target end, as shown in figure 9. It can be seen that the changed semantics of qualifier multiplicity at the target end is well-suited to show the – otherwise more complicated to represent – indirect access to containers.

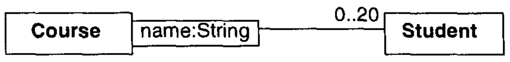

Figure 9. Qualified Access to Containers

3.7 Association Class

An association class [...] not only connects a set of classifiers but also defines a set of features that belong to the relationship itself and not any of the classifiers [18].

Like n-ary associations, association classes are not included in Java. We use them for an alternative representation to qualifiers to represent the utilization of a Map: Given an association between two classes. When there is exactly one instance of the potential association class for each instance at the target association end (i.e. there must be no key without a value in the Map), we can use an association class.

In the example in figure 10, this criteria is fulfilled if a Person object has exactly one reference to a Job object for each of its employers.

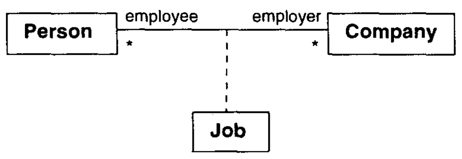

Figure 10. An Association Class

Our first recognition approach for this feature makes use of similar techniques as used before: we search the class structure for two classes with an association between them, where one class (the "source class") has a Map with instances of the other class (the "target" class) as keys. We establish a table of object IDs, which is updated by information from program traces. The IDs from the set of objects at the association target end (e.g. viewed from Person, this would be Company) are matched against each object pair that has been added to the map. If one map value is found for each object ID in the target set, then we can use the association class notation.

We did not encounter maps very often in this constellation in program code. However, since the map interface ensures a unique mapping of keys to values, the association class notation can also be be used as a direct representation for maps, even without the previously described underlying association. Actually, both are valid implementations of the association class, but in the latter case, the objects at the target class are held directly as a set of keys in the map.

3.8 Bound Elements of Templates

We discuss bound elements here although they are no association adornments and therefore are not directly in our scope. However, the problems that can be solved by employing them in reverse engineering are very similar to those encountered with qualifiers and therefore they belong to the same semantic context.

Focus of interest are associations with multiplicities whose values depend on parameterization. For example, the size of an array could be determined by a command line parameter. Thus, multiple runs of the program might yield different multiplicities while for each isolated run, only a static range is allowed.

This scenario would not allow numeric multiplicity-intervals, but only a general "many" multiplicity, as the multiplicity ranges can change with every run. As this is very similar to the template concept, we employ the notation for parameterized classes and binding to indicate the dependency between multiplicity and externally provided parameter values. Binding refers to *the assignment of values to parameters to produce an individual element from a parameterized element* [21, p.173].

Usually, binding is used at model time to create new elements. As the parameter values considered here are external and not available at model time, we use a restricted notation that shows only the parameter types and the dependency relationship without concrete values at the bind stereotype. A possible example for this notation is given in figure 11. Note that the integer range of the qualifier type is `0..k`, which actually allows polygons with only two points. This is due to the fact that our recognition algorithm would discover the parameterized array but has obviously no semantic knowledge of its utilization.

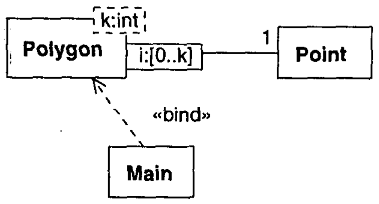

Figure 11. Binding external Parameters

Currently, we examine different methods to extract the required information for discovery of bound elements from the program code. We determine the types of the command line parameters by parsing the parameter types of the parameter array of the main method. Then, constructor signatures of classes called from main are matched against parameter types. It appears sensible to derive the direct correlation between both by extending the parsing algorithm with an annotation based approach as presented in [15].

4 Evaluation of our Approach

Because of the limited space in this paper, we present only a short examination of two software systems: the Java metamodel (as example for a data structure) and Cobalt[10], a group awareness system.

The metrics in figure 12 aim at giving a rough idea of the frequency of occurrence of the examined features. For both programs, about one third of the total associations have been modified, based on the patterns from section 3. For some of the features like inverse associations and aggregations, it is possible to eliminate a significant number of associations.

Based on future tests, we plan to determine more precisely the occurrence of UML features in program code and thus get an idea of their relevance for software design.

Metric	Java MM	%	Cobalt	%
Classes	75		97	
Associations	103		230	
Inverse assoc.	7	6.8	22	9.6
Aggregation	16	15.5	9	3.9
Thereof Composit.	6	5.8	2	0.9
Association Class	3	2.9	16	6.9
Bound Elements	0	0	1	0.43
N-ary associations	0	0	1	0.43
Qualifier	1	0.9	8	3.48
Modified assoc.	34	32.9	79	34.3

Figure 12. Exemplary Examination of two Software Systems

5 Summary and Conclusion

In this paper, we have examined how advanced notational elements of UML class diagrams can be employed in design recovery. Our motivation for this approach was to achieve an extensive and especially correct utilization of the more advanced UML notation features. Also, we wanted to investigate, which impact the utilization of advanced UML notation features has on size of diagrams and denseness of information obtained from reverse engineering. The remaining question is: Do advanced features of associations actually help to make diagrams smaller? From our view, this is not generally the case. However, it is possible to eliminate or modify a significant part of a class diagram's associations, thus increasing the denseness of information and decreasing the diagram interconnectivity.

We consider the additional information in class diagrams helpful for understanding undocumented programs, as well as for a better overview on the static structure. We hope that this work will contribute to a better understanding of how to employ UML association adornments in reverse engineering.

Acknowledgments

Thanks to Tarja Systä for providing valuable material on reverse engineering and to Rudolph Keller for pointing to relevant tools in the field. Discussions with Mark Richters about UML semantics and the comments from the anonymous referees have helped to improve this paper.

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions of Software Engineering*, 20(6):476 – 493, June 1994.

[3] E. J. Chikofsky and J. H. C. II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[4] G. Engels, R. Hücking, S. Sauer, and A. Wagner. UML Collaboration Diagrams and Their Transformation to Java. In R. France and B. Rumpe, editors, *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, volume 1723 of *Lecture Notes in Computer Science*, pages 473–488. Springer Verlag, 1999.

[5] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based in the unified modeling language. In *Proc. of the 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT 98), Paderborn, Germany*, volume 1764 of *LNCS*. Springer Verlag, 1998.

[6] M. Gogolla and R. Kollmann. Re-Documentation of Java with UML Class Diagrams. In E. Chikofsky, editor, *Proc. 7th Reengineering Forum, Reengineering Week 2000 Zürich*, pages REF 41–REF 48. Reengineering Forum, Burlington, Massachusetts, 2000.

[7] M. Gogolla and M. Richters. Equivalence Rules for UML Class Diagrams. University of Bremen, 1998. Internal report.

[8] M. Gogolla and M. Richters. Transformation Rules for UML Class Diagrams. In J. Bézivin and P.-A. Muller, editors, *Proc. 1st Int. Workshop Unified Modeling Language (UML'98)*, volume 1618 of *LNCS*, pages 92–106. Springer, Berlin, 1999.

[9] B. Henderson-Sellers and F. Barbier. Black and white diamonds. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 550–565. Springer, 1999.

[10] R. Kollmann. COBALT - Design and Implementation of a Group Awareness System for Facilitating Synchronous Interpersonal Communication. Master's thesis, University of Bremen, Computer Science Department, 1999.

[11] R. Kollmann and M. Gogolla. Capturing Dynamic Program Behaviour with UML Collaboration Diagrams. In P. Sousa and J. Ebert, editors, *Proc. 5th European Conference on Software Maintenance and Reengineering*, pages 58–67. IEEE, Los Alamitos, 2001.

[12] B. Kuipers. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. MIT Press, 1994.

[13] E. McNichols. Sun releases source code for the Java(TM) 2 platform, 1999. `http://java.sun.com/pr/1999/02/pr990224-01.html`.

[14] H. A. Müller, K. Wong, and S. R. Tilley. Understanding software systems using reverse engineering technology. In *Proceedings of the 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences (ACFAS '94), Colloquium on Object Orientation in Databases and Software Engineering*, (Montréal, PQ; May 16-17, 1994), pages 41–48, May 1994.

[15] U. A. Nickel, J. Niere, J. P. Wadsack, and A. Zündorf. Roundtrip engineering with FUJABA. In *Proc. of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*, Technical Report 8/2000, Fachberichte Informatik. Universität Koblenz-Landau, 2000.

[16] OMG, editor. *OMG Unified Modeling Language Specification, Version 1.3, June 1999*. Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 1999.

[17] OMG. UML Notation Guide. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [16], chapter 3.

[18] OMG. UML Semantics. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [16], chapter 2.

[19] Rational Software Corporation. Rose Enterprise Edition 2000e, 2000. `http://www.rational.com`.

[20] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In H. Yang and L. White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 13–22. IEEE, 1999.

[21] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[22] M. Saksena, M. Larrondo-Petrie, R. France, and M. Evett. Extending aggregation constructs in UML. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 434–441. Springer, 1999.

[23] Sun Microsystems, Inc. Java(TM) Platform Debugger Architecture, 1999. `http://java.sun.com/products/jpda`.

[24] T. Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, Department of Computer and Information Science, University of Tampere, Finland, 2000.

[25] TogetherSoft Corporation. Together 5, 2001. `http://www.togethersoft.com`.

[26] T. K. und Ulrich Nickel, J. Niere, and A. Zündorf. From UML to Java and back again. Technical report, University of Paderborn, 1999.