

# Objects in Context: An Empirical Study of Object Relationships

Stephanie Balzer

Alexandra Burns

Thomas R. Gross

Department of Computer Science  
ETH Zurich

## Abstract

Object collaborations are at the core of all object-oriented programming, yet current class-based object-oriented programming languages do not provide an explicit construct to capture the relationships between objects. This paper reports on an empirical study that investigates the occurrence of object collaborations to assess the need of intrinsic support for relationships in a programming language. We introduce a categorization of possible forms of object collaborations and their corresponding implementation patterns when using a traditional class-based object-oriented language (Java) and analyze 25 Java programs (ranging from 4 to 6275 classes) with the Relationship Detector for Java (RelDJ) to identify occurrences of these patterns. The empirical results show that object collaborations are indeed a frequent phenomenon and reveal that collaboration-related code does not remain encapsulated in a single class. These observations strongly support efforts to define language constructs to express object relationships: relationships allow the encapsulation of a frequently occurring phenomenon and increase program expressiveness.

## 1 Introduction

Class-based object-oriented programming languages have become mainstream in various domains, in commercial software development and scientific research alike. The abstraction of a class has proven to be viable to represent the notions of our modeling domains, and advanced concepts such as inheritance and polymorphism allow for adequate flexibility and extensibility.

Object collaborations are central to any object-oriented program and can be conveniently expressed in conceptual modeling languages (e.g., through associations in UML [13]). Class-based object-oriented languages, on the other hand, lack a corresponding counterpart to represent these collaborations, and programmers resort to employing references (and method invocations) to accommodate collaborations. As a result, the collaboration-relevant code distributes among several classes, and the intent of the collaboration gets buried in the implementation code. With regard to collaborations, the modularity of class-based object-oriented applications seems far from perfect.

A number of researchers have argued that object-oriented programming languages should provide explicit support to express object collaborations [23, 1, 3, 16, 22, 21, 2]. They introduce extensions of class-based object-oriented languages (i.e., *relationship-based programming languages*) that provide, in addition to classes, the programming language abstraction of a *relationship* to encapsulate the possible collaborations that emerge between instances of classes.

Unlike other extensions of object-oriented languages (e.g., aspect-oriented programming languages [14]) that have been put to test in various large-scale software systems by now, relationship-based programming languages are still in their early stage of development. As a consequence, real-world applications developed in such languages are not yet available.

The idea of an explicit *relationship* construct to express object collaborations is certainly appealing but since it has not been put to test so far, there is little empirical evidence regarding the need of support for explicit relationships in a programming language.

This paper aims to provide the basis for an assessment of the need of support for explicit relationships. It contributes an *empirical study* on the occurrence of object collaborations in a class-based object-oriented programming language (i.e., Java). To conduct the study, we implemented a fully automated tool — *RelDJ* (Relationship Detector for Java) — that identifies occurrences of object collaborations in a Java program. We used the tool to analyze a portfolio of 25 Java programs, ranging from the SPEC JVM98 [25] and the SPEC JBB2005 [26] benchmarks to open source Java projects comprising up to thousands of classes.

The implementation of RelDJ is based on the so called *collaboration code skeletons*. These skeletons represent a categorization of possible implementation patterns that programmers use to represent object collaborations in class-based object-oriented programs. By detecting instances of these collaboration code skeletons, RelDJ identifies the object collaborations of a Java program and thus uncovers the relationships inherent to the application.

The collaboration code skeletons constitute another, more technical contribution of this paper. Although we devised the skeletons for the purpose of relationship detection, they could also be helpful outside program analysis, for instance, in assisting programmers in implementing software systems specified by UML class di-

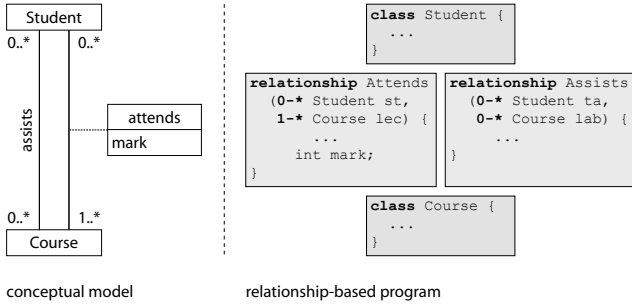


Figure 1: Relationship-based program with classes `Student` and `Course` and relationships `Attends` and `Assists`; relationships correspond one-to-one to the associations of the conceptual model (e.g., UML class diagram).

agrams.

On a broader scale, this empirical study provides insights that are of interest not only to researchers of relationship-based programming languages, but also to researchers of object-oriented languages in general. The study uncovers the internal structures that programmers use to implement the interactions between classes and their instances, respectively, and reports on the “shapes” of today’s Java programs.

The remainder of this paper is organized as follows: Section 2 provides a short introduction to the concept of explicit relationships. Section 3 introduces the collaboration code skeletons. Section 4 discusses the implementation of the tool RelDJ. Section 5 explains the setup of the study, including information on the analyzed programs and details on the metrics employed. Section 6 presents the experimental results. Section 7 discusses the related work, and Section 8 concludes the paper.

## 2 Relationships in a Nutshell

*Relationship-based programming languages* [23, 1, 3, 16, 22, 21, 2] allow a direct expression of the object collaborations identified during the design of a software system. Figure 1 provides an example. On the left, the figure depicts a UML class diagram modeling the possible collaborations between students and courses in a university. On the right, the figure depicts the corresponding implementation of the diagram in a relationship-based language. As relationship-based languages provide the additional abstraction of a *relationship*, the associations of the UML class diagram can be preserved from the design to the implementation stage: the association `attends` results in the relationship `Attends` and the association `assists` in the relationship `Assists`. A pure class-based object-oriented implementation on the other hand, would declare the same number of classes as the relationship-based implementation, but would establish references (instead of relationships) between the classes and possibly auxiliary classes to realize the collaborations.

The pseudocode on the right in Figure 1 syntacti-

cally resembles RelJ [3] but uses features of various relationship-based programming languages. We briefly review some of the most important features: To guarantee appropriate pairing of collaboration partners, a relationship must indicate the classes whose instances are allowed to participate in a collaboration. Relationship `Assists`, for instance, lists in parentheses the participating classes `Student` and `Course`. For convenience, some relationship-based languages allow to assign *role names* to the participating classes; in the `Assists` relationship, for instance, students play the role of teaching assistants (`ta`), and courses the role of lab sessions (`lab`). Some relationship-based languages further allow the definition of *consistency constraints* to restrict collaboration participation. In the example, the constraints reflect the multiplicities established in the UML class diagram. Relationship `Attends`, finally, demonstrates that relationships can declare their own members (i.e., attributes and methods). Unlike class members that operate on individual objects, relationship members operate on relationship instances and thus on pairs (or tuples) of collaborating objects.

## 3 Uncovering Relationships

After introducing the rationale underlying the empirical study, we discuss the collaboration code skeletons. We conclude this section with a survey of the various forms of collaborations that the collaboration code skeletons can represent.

### 3.1 Rationale

Early research in relationship-based languages dates back approximately 20 years [23], but it is only recently that new efforts [3, 16, 22, 21, 2] in this domain have been undertaken. Relationship-based programming languages are consequently still in their early stage of development: the concepts and semantics of such languages are being defined and appropriate compilers are being implemented. As a result, there are no real-world applications available that are coded in a relationship-based language.

The absence of relationship-based applications prevents any experimental evaluation of relationship-based languages that would allow us to verify the advertised benefits of explicit support for relationships. By investigating the large body of existing object-oriented applications, however, we can at least verify whether any explicit support for relationships is justified. An investigation of current object-oriented programs can shed light on questions like “*How prevalent are object collaborations?*” and therefore enables us to infer to which extent existing programs would change if an explicit relationship construct were available.

To address above question, we must measure occurrences of object collaborations, and consequently, must be able to identify such collaborations. In short, we have

to know the possible “representations of relationships” in class-based object-oriented code.

The possible forms of collaborations between instances of classes in class-based object-oriented programming languages are diverse. Guéhéneuc and Albin-Amiot [10] introduce comprehensive definitions of binary collaborations<sup>1</sup> based on a number of formal properties. In its most general form, a collaboration (i.e., association) exists whenever one instance of a class can send a message to another instance of a class. This collaboration form also includes collaborations that are established through method parameters and local variables. The two other more restrictive forms of collaborations (i.e., aggregation and composition) can only be established through field declarations and consequently exclude any temporary collaborations. Orthogonal to the categorization of binary object collaborations is the work on design patterns [8]. Rather than detailing how a collaboration is established between two classes, design patterns describe how several classes can be set up for a specific purpose.

It is possible that all above mentioned forms of collaborations — from temporary collaborations to design patterns — are representatives of explicit relationships. Further research and actual experience with relationship-based programs is needed to answer this question. For the experimental study, we chose a conservative approach: we regard only the collaborations that are established through fields as representatives of relationships and exclude temporary collaborations (established through method parameters and local variables) as well as design patterns. This is in line with the examples discussed in the relationship-based programming language literature and prevents overestimating the importance of collaborations (and relationships, respectively).

### 3.2 Collaboration Code Skeletons

In this subsection we introduce a set of so called *collaboration code skeletons*. These code skeletons constitute a categorization of possible class-based object-oriented implementations of object collaborations that we regard as representatives (as defined previously) of explicit relationships. We derived these collaboration code skeletons from an analysis of a number of real-world applications and have reviewed them with professional software engineers. The task of defining and identifying collaboration code skeletons poses challenges similar to the ones encountered in the reverse engineering of UML associations [12, 17, 10, 18]. However, there is a fundamental difference in intent between the identification of representatives of relationships and the reverse engineering of UML associations. Whereas a reverse engineering tool typically tries to hide how a particular association is implemented [12, 10], the aim of our study is to expose how relationships are represented in class-based

<sup>1</sup>What we call object collaboration in this paper, the authors call binary class relationship.

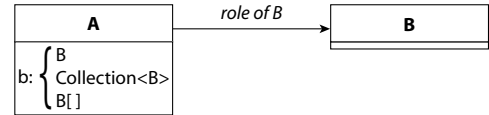


Figure 2: Direct binary unidirectional (DBU) collaboration code skeleton.

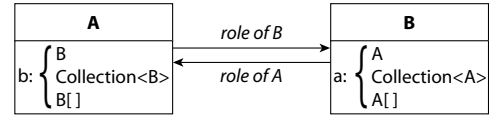


Figure 3: Direct binary bidirectional (DBB) collaboration code skeleton.

object-oriented code. As a result, the collaboration code skeletons reflect the implementation decisions taken by a programmer and detail, for instance, whether auxiliary classes were introduced in addition to the immediate collaborators.

#### 3.2.1 Direct Binary Unidirectional (DBU)

Figure 2 depicts the *direct binary unidirectional (DBU)* collaboration code skeleton. This skeleton represents a very straightforward class-based implementation of a relationship: one of the two collaborating classes establishes an instance variable pointing to its collaboration partner. As indicated, the reference can either be *single-valued* (the type of the reference is the class of the collaboration partner) or *multi-valued* (the type of the reference is either an array with the collaboration partner as element type or a collection with the collaboration partner as element type).

*Example:* We can implement association `assists` (Figure 1) using the DBU collaboration code skeleton. Since the DBU skeleton is unidirectional, we must select one of the collaboration partners (i.e., `Student` and `Course`) to become the main point of reference. We decide on class `Student` and therefore establish a reference to `Course` in that class. As a particular student can assist several courses, the declared field must be multi-valued (e.g., of type `Collection<Course>`).

#### 3.2.2 Direct Binary Bidirectional (DBB)

Figure 3 depicts the *direct binary bidirectional (DBB)* collaboration code skeleton. This skeleton attempts to maintain the bilateralism inherent to relationships by establishing instance variables pointing to the collaborator at both sides of the collaboration. In the figure, class `A` declares a reference to class `B` and vice versa. Again, the references can be either *single-valued* or *multi-valued*.

*Example:* If we want the `assists` association to become navigable in both directions, we must implement the DBB collaboration code skeleton. In contrast to the previous example, we now declare fields referring to the collaboration partner in both classes. Since association `assists` is a many-to-many collaboration (a particular student can assist various courses and a course can be assisted by various students), both declared fields must be multi-valued (e.g., of type

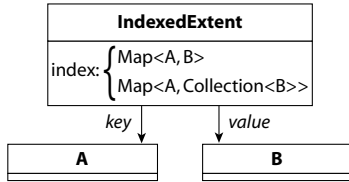


Figure 4: Indirect indexed extent (IIE) collaboration code skeleton.

Collection<Course> and of type Collection<Student>, respectively).

### 3.2.3 Indirect Indexed Extent (IIE)

Figure 4 depicts the *indirect indexed extent (IIE)* collaboration code skeleton. In addition to the actual collaborating classes A and B, this skeleton introduces an additional class IndexedExtent. In contrast to the direct code skeletons DBU and DBB, the instance variables referring to the collaborating classes are kept in the newly introduced class IndexedExtent and not in the collaborating classes themselves. The class IndexedExtent acts as a “repository” (extent) that stores all the objects that take part in the collaboration. To allow access of the collaborating objects, the class IndexedExtent provides an index that, given a key (participant A), retrieves all values (participant B) associated with that key. In a Java setting, the index is typically a descendant of Map. The reference representing the values of the index can be either *single-valued* or *multi-valued*.

*Example:* If we want support for storage and retrieval of students and assisted courses, we can implement association assists using the IIE collaboration code skeleton. In that case, we declare a class Assists in addition to the classes Student and Course. Furthermore, we declare an instance variable of type Map<Student, Course> in Assists to refer to all teaching assistants and courses assisted.

### 3.2.4 Indirect Pair Extent (IPE)

Figure 5 depicts the *indirect pair extent (IPE)* collaboration code skeleton. Like the IIE code skeleton, this skeleton introduces an indirection (class Pair) to remove the instance variables pointing to the participants of the collaboration from the participants themselves. Unlike the IIE skeleton, however, this skeleton represents the collaborating object pairs explicitly through their own class (class Pair) and thus separates the collaborating objects from the repository (class Extent) of the collaboration. The instance variables declared in the class Pair are all single-valued and of the type of the respective participating class. The instance variable pair declared in class Extent, on the other hand, must be multi-valued as it refers to a collection containing all Pair instances.

*Example:* Because of its member mark, association attends necessitates the use of the IPE collaboration code skeleton (see also discussion in Section 3.3). According to the IPE skeleton, we introduce the auxiliary classes Attends and AttendsExtent in addition to the classes Student and

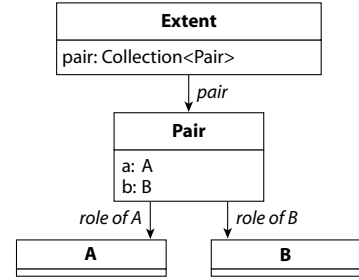


Figure 5: Indirect pair extent (IPE) collaboration code skeleton.

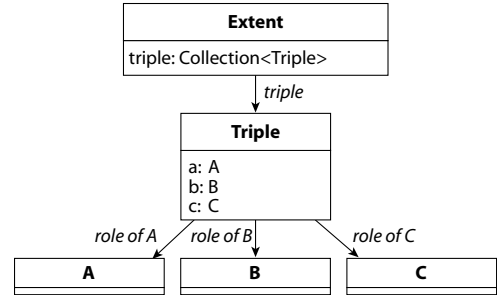


Figure 6: Indirect triple extent (ITE) collaboration code skeleton.

Course. Class Attends represents a particular student-course pair, and class AttendsExtent provides access to all instances of such student-course pairs. Besides declaring references to a particular student and to a particular course that the student attends, we also declare the collaboration attribute mark in class Attends.

### 3.2.5 Indirect Triple Extent (ITE)

The IPE collaboration code skeleton can be generalized to support n-ary relationships. Since the need for n-ary relationships is controversial (ambiguity in the interpretation of multiplicities [9]) and since n-ary relationships can be approximated through an appropriate number of binary relationships, we restrict the discussion to the implementation of ternary relationships. Figure 6 depicts the *indirect triple extent (ITE)* collaboration code skeleton. Like the IPE code skeleton, this skeleton represents the collaborating object triples explicitly (class Triple) and separates the collaborating objects from the repository (class Extent) of the collaboration. Unlike the IPE skeleton, the ITE skeleton declares three instance variables in class Triple, each referring to one collaborator.

## 3.3 Supported Collaboration Forms

The few collaboration code skeletons introduced in the previous section actually allow representing a surprising variety of collaborations. To provide a systematic survey of the forms of collaborations that a specific collaboration code skeleton can implement, we introduce a categorization of the various features a collaboration may exhibit and indicate for each collaboration code skeleton the features the skeleton supports. We distinguish the

Option <i>configuration</i>	Cardinality			Support	
	1:1	1:n	m:n	Member	Extension
<b>DBU</b>					
<i>sv</i>	-	●	-	○	-
<i>mv</i>	-	-	●	-	-
<b>DBB</b>					
<i>sv, sv</i>	⦿	-	-	○	-
<i>sv, mv</i>	-	⦿	-	○	-
<i>mv, mv</i>	-	-	⦿	-	-
<b>IIE</b>					
<i>sv</i>	⦿	●	-	-	●
<i>mv</i>	⦿	⦿	●	-	●
<b>IPE</b>	⦿	⦿	●	●	●

Table 1: Summary of the collaboration features that can be realized by the different code skeletons. Symbol ● indicates support of the feature, symbol ⦿ specifies that additional programmatic effort for the support of the feature is required, and symbol ○ marks conceptually suboptimal but technically achievable solutions.

following feature categories:

- *Cardinality of a collaboration* – corresponds to the right digit in UML multiplicity declarations and prescribes, for each side of a collaboration, the maximal number of participants. We distinguish one-to-one (i.e., 1:1), one-to-many (i.e., 1:n), and many-to-many (i.e., m:n) collaborations.
- *Support for collaboration members* – indicates whether a particular collaboration code skeleton supports the setup of collaboration members.
- *Provision of extensional view* – states whether the collaboration maintains a repository of the collaboration instances, allowing for an enumeration of the collaborating objects.

Table 1 gives an overview of the collaboration features that the different collaboration code skeletons support (symbol ●)<sup>2</sup>. Where a skeleton provides the choice between single-valued (*sv*) and multi-valued (*mv*) instance variables, the table lists an entry for each particular configuration (*configuration*) of that skeleton. Symbol ⦿ specifies that the enforcement of the feature requires programmatic effort. Symbol ○ indicates that the feature can be technically accommodated, but that the resulting solution is suboptimal from a design perspective. We briefly discuss each feature in turn:

*Provision of extensional view* The only collaboration code skeletons that provide access to all collaboration instances are the indirect skeletons IIE and IPE.

*Support for collaboration members* Only the IPE collaboration code skeleton allows the setup of collaboration attributes and methods. As the IPE skeleton represents the collaborating objects explicitly through the

<sup>2</sup>Because of its controversial interpretation of multiplicities [9], we exclude the ITE skeleton from this survey.

class `Pair`, any members that operate on collaboration instances can be declared as part of that class. One could argue that the DBU and DBB configurations for 1:1 and 1:n cardinalities can also accommodate collaboration members (symbol ○ in Table 1). However, the member would then have to be declared as part of a collaboration participant, that is, at either side of the collaboration for 1:1 collaborations or at the many-side for 1:n collaborations. This approach is not desirable as it intermingles the members of a participant of a collaboration with the members of the collaboration.

*Cardinality of a collaboration* The mere choice of a particular collaboration code skeleton and its respective configuration determines the cardinality of the collaboration. This cardinality inherent to the skeleton/configuration is indicated by the symbol ●. For example, if the DBU skeleton is employed with a single-valued instance variable then this configuration implements a 1:n collaboration: every object collaborates with at most one other object, which in turn can be referenced by other collaborators. For the IIE and IPE collaboration code skeletons, however, Table 1 lists in addition to the inherent cardinalities also cardinalities that must be programmatically enforced (symbol ⦿). As both the IIE and the IPE skeleton record all collaboration instances, it is possible to enforce constraints that are more restrictive than the inherent cardinalities by setting up the necessary checking code in the extent class of the collaboration. The DBB collaboration code skeleton, on the other hand, only lists programmatically enforceable constraints. This choice reflects the fact that the enforcement of the bilateralism (and therefore the cardinality) of the collaboration must be manually implemented.

From Table 1 we can infer that each collaboration code skeleton only supports a specific feature combination. Furthermore, we can see that the particular kind of feature combination supported is different for the various collaboration code skeletons and configurations, respectively. It is consequently not possible to exchange one skeleton configuration with another skeleton configuration without losing support for the previously guaranteed features. The selection of the appropriate collaboration code skeleton and configuration, respectively, is therefore crucial not only for the design of a system but also for its maintainability.

## 4 ReIDJ

This section introduces the program analysis tool ReIDJ that we developed for the purpose of this study. We detail the algorithm employed to identify occurrences of collaboration codes skeletons and discuss known limitations.

### 4.1 Overview

*ReIDJ* (Relationship Detector for Java) is a fully automated program analysis tool that we developed for this

study. Given a Java program as input, RelDJ identifies occurrences of the collaboration code skeletons (Section 3.2) and collects measurements. Although not its principle aim, RelDJ could be regarded as a reverse engineering tool since it identifies relationships between classes and thus reconstructs the associations present in the corresponding UML class diagram. In contrast to existing static model extraction tools [12, 17, 18] that try to hide representation details, RelDJ exposes the implementation decisions taken by a programmer and uncovers all the classes — collaborators and auxiliaries alike — that are involved in implementing a relationship.

RelDJ uses the *ASM Java bytecode manipulation framework* [4] to parse the Java class files constituting the application, and thus employs a static bytecode analysis. A static analysis is sufficient for the purpose of relationship detection as explicit relationships denote the possible, and not the actual collaborations between objects. A bytecode analysis furthermore offers the advantage that binary representations of applications can also be analyzed, where the source is not available or protected.

## 4.2 Algorithm

In this and the following section we describe the implementation of RelDJ in as much detail as is necessary to understand the empirical results of this study. Overall, the programming analysis techniques applied in the implementation are standard (see related work), and therefore described thoroughly in a technical report [5].

To identify the occurrences of the collaboration code skeletons in a Java program, RelDJ examines the reference structure of the program. RelDJ executes three passes over the input class files. In the first pass, RelDJ collects metadata about the program (e.g., classes and their inheritance structure) and records all declarations of instance variables that are of a reference type. In the second pass (see Algorithm 1), RelDJ analyses one class after the other and produces a preliminary identification of the collaboration code skeletons based on the local information available per class. In the third pass (see Algorithm 2), RelDJ completes the identification of code skeletons and refines some of the identification decisions taken during the preliminary phase.

As outlined by Algorithm 1, RelDJ considers only those reference instance variables that are relevant to the analysis (line 7). A reference instance variable is relevant only if either its type (case: single-valued reference) or the type of its elements (case: multi-valued reference) is an application class. With this check, RelDJ can exclude library classes from the analysis. Algorithm 1 also reveals how RelDJ distinguishes between the individual collaboration code skeletons. To distinguish the IIE skeleton from the remaining collaboration code skeletons, RelDJ treats instances that are descendants of `Map` separately (line 47). To differentiate between the DBU, IPE, and ITE skeletons, RelDJ tracks assignments to the involved references. If three references are assigned all together in the same method, the underlying collab-

oration is categorized as an ITE skeleton (line 22). Likewise, if two references are assigned together in the same method, the underlying collaboration is categorized as an IPE skeleton (line 31). In case of a single reference, finally, RelDJ discovers an occurrence of a DBU skeleton (line 47). Algorithm 2 further details how RelDJ identifies DBB code skeletons (line 4) and the `Extent` classes of the IPE (line 13) and ITE (line 21) collaboration code skeletons.

```

Procedure PreliminaryIdentification(inputClassFiles, skeletons):
1: for all class files  $c \in \text{inputClassFiles}$  do
2:   relevantFields  $\leftarrow \{ \}$ 
3:   relMapFields  $\leftarrow \{ \}$ 
4:   relNMapFields  $\leftarrow \{ \}$ 
5:   for all fields  $f \in c$  do
6:     /* Check whether f is relevant */
7:     if isRelevant(f) then
8:       relevantFields  $\leftarrow \text{relevantFields} \cup f$ 
9:     end if
10:  end for
11:  for all fields  $f \in \text{relevantFields}$  do
12:    /* Check whether f is a Map */
13:    if instanceof(f, Map) then
14:      relMapFields  $\leftarrow \text{relMapFields} \cup f$ 
15:    end if
16:  end for
17:  relNMapFields  $\leftarrow \text{relevantFields} \setminus \text{relMapFields}$ 
18:  for all methods  $m \in c$  do
19:    /* Indirect triple extent (ITE) */
20:    for all fields  $f_1, f_2, f_3 \in \text{relNMapFields}$  do
21:      /* Check whether f1, f2, f3 are all assigned in m */
22:      if isAssignedIn( $f_1, f_2, f_3, m$ ) then
23:        ite  $\leftarrow \text{initializeITE}(f_1, f_2, f_3)$ 
24:        skeletons  $\leftarrow \text{skeletons} \cup \text{ite}$ 
25:        relNMapFields  $\leftarrow \text{relNMapFields} \setminus \{f_1, f_2, f_3\}$ 
26:      end if
27:    end for
28:    /* Indirect pair extent (IPE) */
29:    for all fields  $f_1, f_2 \in \text{relNMapFields}$  do
30:      /* Check whether f1 and f2 are assigned in m */
31:      if isAssignedIn( $f_1, f_2, m$ ) then
32:        ipe  $\leftarrow \text{initializeIPE}(f_1, f_2)$ 
33:        skeletons  $\leftarrow \text{skeletons} \cup \text{ipe}$ 
34:        relNMapFields  $\leftarrow \text{relNMapFields} \setminus \{f_1, f_2\}$ 
35:      end if
36:    end for
37:    /* Indirect indexed extent (IIE) */
38:    for all fields  $f_1 \in \text{relMapFields}$  do
39:      /* Check whether key and value of f1 are assigned in m */
40:      if isAssignedIn( $f_1.\text{key}, f_1.\text{value}, m$ ) then
41:        iie  $\leftarrow \text{initializeIIE}(f_1)$ 
42:        skeletons  $\leftarrow \text{skeletons} \cup \text{iie}$ 
43:        relMapFields  $\leftarrow \text{relMapFields} \setminus \{f_1\}$ 
44:      end if
45:    end for
46:  end for
47:  /* Direct binary unidirectional (DBU) */
48:  for all fields  $f_1 \in \text{relNMapFields}$  do
49:    dbu  $\leftarrow \text{initializeDBU}(f_1)$ 
50:    skeletons  $\leftarrow \text{skeletons} \cup \text{dbu}$ 
51:    relNMapFields  $\leftarrow \text{relNMapFields} \setminus \{f_1\}$ 
52:  end for
53: end for

```

### Algorithm 1: Local pass.

A further issue in the identification of collaboration code skeletons is the interpretation of inherited refer-

```

Procedure FinalIdentification(inputClassFiles):
1: identifiedSkeletons  $\leftarrow$  { }
2: PreliminaryIdentification(inputClassFiles, identifiedSkeletons)
3: for all skeletons s1  $\in$  identifiedSkeletons do
4:   /* Direct binary bidirectional (DBB) */
5:   for all skeletons s2  $\in$  identifiedSkeletons do
6:     /* Check whether s1 and s2 are DBU skeletons and
7:     whether they refer to each other */
8:     if instanceOf(s1, DBU) and instanceOf(s2, DBU) and
9:     s1.A = s2.B and s1.B = s2.A then
10:      dbb  $\leftarrow$  initializeDBB()
11:      identifiedSkeletons  $\leftarrow$  identifiedSkeletons  $\cup$  dbb
12:      identifiedSkeletons  $\leftarrow$  identifiedSkeletons  $\setminus$  {s1, s2}
13:    end if
14:  end for
15:  /* Indirect pair extent (IPE) */
16:  for all skeletons s2  $\in$  identifiedSkeletons do
17:    /* Check whether s1 is a IPE and s2 a DBU skeleton
18:    and whether s2 refers to s1 */
19:    if instanceOf(s1, IPE) and instanceOf(s2, DBU) and
20:    s2.B = s1.Pair then
21:      updateIPE(s1, s2)
22:      identifiedSkeletons  $\leftarrow$  identifiedSkeletons  $\setminus$  {s2}
23:    end if
24:  end for
25:  /* Indirect triple extent (ITE) */
26:  for all skeletons s2  $\in$  identifiedSkeletons do
27:    /* Check whether s1 is a ITE and s2 a DBU skeleton
28:    and whether s2 refers to s1 */
29:    if instanceOf(s1, ITE) and instanceOf(s2, DBU) and
30:    s2.B = s1.Triple then
31:      updateITE(s1, s2)
32:      identifiedSkeletons  $\leftarrow$  identifiedSkeletons  $\setminus$  {s2}
33:    end if
34:  end for
35: end for

```

**Algorithm 2: Global pass.**

ences. One could either regard such inherited collaborations as new collaborations or as incarnations of one collaboration. In line with the literature on static model extraction [12, 17, 18], RelDJ takes the latter, more conservative interpretation. Rather than delivering an imaginary number of collaborations and collaboration incarnations, respectively, RelDJ collapses inherited reference instance variables and thus identifies as many collaborations as there would be relationships in a relationship-based implementation.

### 4.3 Limitations

The use of collections, essential to establishing  $m:n$  collaborations, poses a challenge for the identification of the collaboration code skeletons. As soon as a program declares non-generic collections, the actual element type of the collection is unknown. As a result, the corresponding participant of the collaboration is unknown too.

RelDJ employs an elementary *data-flow analysis* to deduce the type of a collection element, allowing the identification of the collaboration participant. The main idea of the analysis (similar to the one described by Jackson et al. [12] and Milanova [18] for the reverse engineering of UML associations) is to monitor invocations of non-generic collections and to deduce the el-

ement type from any information provided as part of the invocation. For instance, the retrieval of an element type from a collection is typically accompanied by a cast instruction, revealing the actual element type. When an element is added to a collection, on the other hand, the type of the provided argument indicates the actual element type. RelDJ is capable of inferring the least common supertype when several invocations happen on the same collection. For that purpose, RelDJ uses the collected metadata about the class inheritance structure. Of course, the data-flow analysis described requires RelDJ to know the methods to be monitored. RelDJ maintains a configuration file for that reason that lists for all Java library collection classes the corresponding methods for element addition and retrieval.

Unfortunately, the data-flow analysis presented is incapable of deducing the type of a collection element in all cases. More specifically, the analysis is only successful if the collections are actually operated on. The data-flow analysis currently implemented in RelDJ, furthermore, does only monitor “direct” modifications of collections where either the target or argument is the reference instance variable of concern. As soon as a collection is accessed indirectly, RelDJ is unaware of the relevance of the call and does not analyze it. As a result, the analysis of a program may yield *unidentified collaboration participants* and supposedly *non-collaborating classes*, respectively.

## 5 Empirical Study

This section provides information about the setup of the empirical study: we introduce the analyzed Java applications and discuss the metrics employed by RelDJ.

### 5.1 Analysis Portfolio

We analyzed a portfolio of 25 Java applications with RelDJ. Table 2 provides an overview of these applications. The portfolio ranges from the SPEC JVM98 [25]<sup>3</sup> and the SPEC JBB2005 [26] benchmarks to open source Java projects comprising up to thousands of classes and covers applications of various domains (e.g., application servers, graphical email clients, speech synthesizers, games, etc.). Additionally, we analyzed some in-house applications, such as the project sample solution for the compiler design lecture at ETH and the extended version of the university information system sketched in Figure 1.

Most Java applications typically make use of a large amount of library functionality. Since a considerable portion of the same library functionality is actually used by several applications, it is vital to separate the library classes from the core classes of an application and to analyze only these core classes. Otherwise, the analysis results are less meaningful. Whenever possible, we

<sup>3</sup>We excluded the `jvm_mtrt` SPEC JVM98 benchmark from the analysis since the benchmark consists only of two classes, one of which is the main class.

have built the applications from the sources to avoid the inclusion of library classes, as these are typically not contained in the source packages.

## 5.2 Basic Metrics

We use the following basic *metrics* to quantify the results of the program analysis:

**Number of classes** The total number of class files constituting the analyzed application. This number includes classes that do participate in collaborations as well as classes that do not participate in collaborations. As mentioned in Section 5.1, we tried to avoid the inclusion of library classes.

**Number of collaborations** The number of occurrences of identified collaboration code skeletons. This number includes only occurrences of collaborations that arise within the analyzed classes.

**Lines of bytecode (LOBC)** The number of lines of bytecode of the analyzed class files, one bytecode instruction per line. The number includes: class signatures, method signatures, field declarations, and the bytecode instructions of method and constructor bodies.

## 5.3 Custom Metrics

In addition to the basic metrics introduced in the previous section, RelDJ also employs custom metrics that were specifically constructed for the purpose of this study.

### 5.3.1 Collaboration-related Code

To assess the portion of an application that is involved in the representation of object collaborations, RelDJ computes the number of bytecode instructions that are collaboration-related. It is actually not possible to compute a precise measurement of that metrics without the intervention of a human expert. A fully automated tool like RelDJ can only identify those instructions that are directly related to collaborations (i.e., declarations and invocations of reference instance variables), but will miss any auxiliary instructions that are used in connection with these directly collaboration-related instructions and are consequently collaboration-related too. To provide a more meaningful calculation of the number of collaboration-related LOBC, we compute two measures, indicating the bandwidth of the number of collaboration-related LOBC:

**Lower bound** The number of LOBC that are *directly* related to collaborations.

**Upper bound** The number of LOBC that are both *directly* and *indirectly* collaboration-related.

The upper bound constitutes an approximation of the number of LOBC that are indirectly collaboration-related. It relies on the assumption that functionally associated instructions tend to occur together in a program. To compute the upper bound, RelDJ measures the total number of bytecode instructions of those methods that contain directly collaboration-related instructions.

The actual number of collaboration-related LOBC for an application lies in between its lower and upper bound. Both the lower and upper bound represent two extremes of the same measure as they are an underestimation and overestimation, respectively. A manual investigation of some of the small applications of the analysis portfolio actually revealed that the lower bound is a very conservative metric and that the actual number of collaboration-related LOBC was considerably above the lower bound for the investigated applications.

### 5.3.2 Collaboration Dispersion

In class-based object-oriented programs, collaborations inevitably involve several classes. Classes, on the other hand, can participate in several collaborations, and consequently, accommodate different collaboration-specific concerns. As a result, collaborations become dispersed. We introduce two metrics to measure the dispersion caused by collaborations: inter-class dispersion and intra-class dispersion. Whereas the *inter-class dispersion* measures the distribution of a collaboration *across* several classes, the *intra-class dispersion* measures the fragmentation *within* a class due to the accommodation of different collaboration concerns.

**Inter-class dispersion** Equation 1 shows how we compute the inter-class dispersion of an application:

$$\frac{\sum_{skel} \#occurrences_{skel} * \#constituents_{skel}}{\#collaborations} \quad (1)$$

The formula relies on the collaboration code skeletons and the distribution of their occurrences in an application. Intuitively, the formula computes the ratio between the cumulated number of classes that are constituents of collaborations and the total number of collaborations. The subscript *skel* enumerates the possible collaboration code skeletons. Expression  $\#occurrences_{skel}$  denotes the number of occurrences of a particular collaboration code skeleton in an application and expression  $\#constituents_{skel}$  indicates the number of classes that are needed to implement the collaboration code skeleton. The denominator of the fraction,  $\#collaborations$ , represents the number of collaborations in the application.

Equation 1 is devised such that the fraction becomes 1 if there is no dispersion. In the absence of inter-class dispersion, a collaboration remains encapsulated in one module. As a result, expression  $\#constituents_{skel}$  becomes 1 and the entire fraction becomes 1 too. However,

Application	Version	Description	Classes	Bundle	Origin
azureus	3.0.5.0	BitTorrent client	1905	Binaries in com/aELITIS/azureus	SourceForge
columba	1.4	Graphical email client	1702	Built from sources	SourceForge
compiler	SS07	Compiler design sample solution	116	Built from sources	ETH internal
findbugs	1.3.2	Program to find bugs in Java code	1577	Built from sources	SourceForge
freetts	1.2.1	Speech synthesizer	189	Built from sources	SourceForge
gruntsput	0.4.6 beta	Graphical CVS client	746	Built from sources	SourceForge
jasperreports	2.0.4	Reporting tool	1209	Binaries in net/sf/jasperreports	JasperForge
jbidwatcher	1.02.2	Auction site tracking tool	410	Built from sources	SourceForge
jboss	4.2.2 GA	JBoss applications server	6275	Binaries in server/default/org/jboss	SourceForge
jedit	4.3 pre13	Text editor	1008	Built from sources	SourceForge
jgraph	5.12.0.1	Graph visualization component	96	Built from sources	SourceForge
jhotdraw	6.0.1	Graphics framework for drawing editors	600	Binaries in org/jhotdraw	SourceForge
jtopen	6.1	Toolbox for client/server programming	3600	Binaries in com/ibm	SourceForge
jvm_compress	JVM98	Modified Lempel-Ziv method	13	Binaries	SPEC JVM98
jvm_db	JVM98	Memory resident database	4	Binaries	SPEC JVM98
jvm_jack	JVM98	Parser generator	57	Binaries	SPEC JVM98
jvm_javac	JVM98	Java Compiler	177	Binaries	SPEC JVM98
jvm_jess	JVM98	Java expert shell system	152	Binaries	SPEC JVM98
jvm_mpegaudio	JVM98	Audio file decompression	56	Binaries	SPEC JVM98
jvm_raytrace	JVM98	Raytracer	26	Binaries	SPEC JVM98
lectcomm	1.0	ETH lecture communicator	268	Binaries in lectcomm	SourceForge
megamek	0.32.2	Battletech board game	876	Built from sources	SourceForge
pmd	4.2rc1	Java program analyzer	924	Built from sources	SourceForge
spec_jbb	JBB05	Three-tier client/server system	90	Binaries	SPEC JBB2005
university	1.0	Example from Figure 1	10	Built from sources	ETH internal

Table 2: Java applications analyzed with ReLDJ. *Classes* indicates the number of analyzed class files; *Bundle* specifies whether the application has been built from the sources, or otherwise, specifies the location of the analyzed jar files.

as soon as a collaboration distributes across several modules, expression  $\#constituents_{skel}$  becomes greater than 1, raising the result of the fraction above 1. Therefore, a value greater than 1 for inter-class dispersion indicates the presence of dispersion.

**Intra-class dispersion** Equation 2 shows how we compute the intra-class dispersion of an application:

$$\sum_n \frac{\#classesParticipatingNTimes_n}{\#participatingClasses} * n \quad (2)$$

The formula relies on the frequency distribution of the number of collaborations classes participate in. Intuitively, the formula computes the weighted sum of the number of times classes participate in collaborations. The subscript  $n$  enumerates the possible number of times that classes participate in collaborations, starting at 1 (the measure considers collaborating classes only). The numerator of the fraction,  $\#classesParticipatingNTimes_n$ , denotes the number of classes that participate the subscripted-number of times in a collaboration. The denominator of the fraction,  $\#participatingClasses$ , represents the number of classes in the application that participate in collaborations.

Also Equation 2 is devised such that the fraction becomes 1 if there is no dispersion. In the absence of intra-class dispersion, each class participates in at most one collaboration. As a result, expression  $\#classesParticipatingNTimes_n$  becomes the number of participating classes for  $n = 1$  and 0 in all remaining cases (i.e., for  $n > 1$ ). However, as soon as a class participates in several collaborations, expression

$\#classesParticipatingNTimes_n$  becomes at least one also for cases where  $n \neq 1$ , yielding an overall value greater than 1. Therefore, a value greater than 1 for intra-class dispersion indicates the presence of dispersion.

## 6 Experimental Results

This section presents the results that ReLDJ delivers for the analysis portfolio.

### 6.1 Frequency of Collaborations

Table 3 lists the number of discovered collaborations per application (*Coll*). Considering that the number of collaborations stands for the number of occurrences of identified collaboration code skeletons (Section 5.2) and considering that each skeleton consists of at least 2 classes, Table 3 reveals that most applications involve overall more classes in collaborations than they declare classes. Some classes must therefore participate in several collaborations. Column *Par* in Table 3 indicates the average number of collaborations a class participates in. For some applications (e.g., *jgraph*, *lectcomm*, and *spec\_jbb*), classes participate in roughly 2.5 collaborations on average. Although the average collaboration participation is less pronounced for other applications (e.g., *jvm\_jess* and *pmd* with an average participation of 0.7 and 0.9, respectively), the total average for the analysis portfolio still amounts to 1.8.

The average collaboration participation of a class varies from one class to another. Some classes participate in numerous collaborations (up to 70) at a time

Application	Cl	Coll	Par	Not	Unknown
azureus	1905	1729	2.3	13 %	25 %
columba	1702	1295	1.8	31 %	13 %
compiler	116	60	1.2	47 %	7 %
findbugs	1577	1166	1.8	27 %	14 %
freetts	189	122	1.6	39 %	14 %
gruntspud	746	764	2.4	3 %	15 %
jasperreports	1209	847	1.7	38 %	13 %
jbidwatcher	410	277	1.6	30 %	13 %
jboss	6275	4840	1.8	33 %	17 %
jedit	1008	888	2.1	18 %	14 %
jgraph	96	99	2.6	1 %	28 %
jhotdraw	600	447	1.6	12 %	7 %
jtopen	3600	3222	2.1	32 %	17 %
jvm_compress	13	5	1.3	31 %	15 %
jvm_db	4	2	1.0	50 %	25 %
jvm_jack	57	29	1.2	37 %	25 %
jvm_javac	177	116	1.8	44 %	27 %
jvm_jess	152	42	0.7	75 %	7 %
jvm_mpegaudio	56	44	1.8	21 %	9 %
jvm_raytrace	26	23	2.2	12 %	12 %
lectcomm	268	294	2.5	0 %	12 %
megamek	876	858	2.3	21 %	24 %
pmd	924	371	0.9	58 %	9 %
spec_jbb	90	91	2.5	20 %	19 %
university	10	6	1.8	0 %	0 %
<b>Average</b>	883	705	1.8	28 %	15 %

Table 3: Program metrics: total number of classes (*Cl*), total number of collaborations (*Coll*), avg. number of collaborations a class participates in (*Par*), percentage of classes that do not participate in any collaborations (*Not*), and number of unidentified collaboration participants expressed in percentage of the number of classes (*Unknown*).

(Section 6.2), but there are also classes that do not participate at all. Table 3 lists the percentage of such non-collaborating classes (*Not*). This number roughly correlates with the average collaboration participation: the fewer classes that participate in collaborations, the lower the average participation. The percentage of non-collaborating classes, though, is very widespread (e.g. between 0 % and 75 %), and it is surprising that for some applications, there are so many “isolated” classes.

Actually, we cannot equate “non-collaborating” with “non-interacting”; it is possible that RelDJ categorizes a class as non-collaborating even if the class interacts in some way with the remaining classes of the application. This can happen in the following cases: (a) the class participates in a temporary collaboration, (b) the class is a subclass that participates in an inherited collaboration but does not declare any new reference instance variables, and (c) the class is a unidentified participant of a collaboration. In the first two cases, the categorization is as requested: temporary collaborations are ignored in this study (Section 3.1), and inherited col-

laborations are not counted anew (Section 4.2). In the last case, however, the class is wrongly categorized as non-collaborating (Section 4.3).

The last column (*Unknown*) in Table 3 reveals that for all except the university application<sup>4</sup> the number of non-collaborating classes is indeed overestimated. The column lists the number of unidentified collaboration participants in percentage of the number of classes. As a class can participate in several collaborations, there need not be as many wrongly categorized classes as there are unidentified participants. However, the number of unidentified participants constitutes an upper bound for the number of wrongly categorized classes. For the application *jasper\_reports*, for instance, this upper bound suggests that up to 13 % of the total number of classes are wrongly categorized, and therefore the number of non-collaborating classes can diminish down to 25 % (instead of 38 %). The occurrence of unidentified collaboration partners indicates in general that the number of non-collaborating classes is too high and the average number of collaborations per class too low (the total number of collaborations is not influenced by the occurrence of unidentified collaboration partners).

For the applications *jvm\_javac*, *compiler*, *jvm\_jack*, and *jasper\_reports* a substantial amount of the non-collaborating classes is due to the occurrence of unidentified collaboration participants. A manual investigation of the remaining applications with a pronounced occurrence of non-collaborating classes (e.g., *jboss* and *jvm\_jess*) reveals that the phenomenon mainly stems from a predominance of temporary collaborations for the former, and from the use of an excessive inheritance structure for the latter. The application *jvm\_db*, finally, consists only of 4 classes. One of the two non-collaborating classes is the class that declares the main method, a “driver” class typically maintaining local references.

## 6.2 Collaboration Dispersion

Table 4 lists the inter-class dispersion computed for every application in the analysis portfolio. A value of 1 indicates the absence of dispersion. For the analysis portfolio we observe a relatively pronounced presence of inter-class dispersion (2.44 on average). This result matches the data of the average distribution of occurrences of the collaboration code skeletons (Table 5): on average, 22 % of all collaborations distribute across at least 3 classes.

Table 4 lists the intra-class dispersion computed for every application in the analysis portfolio. Like with the inter-class dispersion, a value of 1 for the intra-class dispersion indicates the absence of dispersion. For the analysis portfolio we observe a relatively pronounced presence of intra-class dispersion (2.09 on average).

<sup>4</sup>As the university application employs generic collections, RelDJ is capable of identifying all collaboration participants.

Application	Inter-Class	Intra-Class
azureus	2.55	2.17
columba	2.32	2.15
compiler	2.35	2.18
findbugs	2.47	2.04
freetts	2.46	2.31
gruntsrud	2.38	2.06
jasperreports	2.38	2.14
jbidwatcher	2.34	2.03
jboss	2.36	2.04
jedit	2.33	2.17
jgraph	2.49	2.22
jhotdraw	2.13	1.69
jtopen	2.34	2.31
jvm_compress	3.40	1.67
jvm_db	2.00	1.50
jvm_jack	2.34	1.50
jvm_javac	2.68	2.34
jvm_jess	2.36	2.32
jvm_mpegaudio	2.32	1.98
jvm_raytrace	2.48	2.35
lectcomm	2.23	2.17
megamek	2.36	2.40
pmd	2.36	1.97
spec_jbb	2.48	2.82
university	3.00	1.8
<b>Average</b>	2.44	2.09

Table 4: Inter-class and intra-class dispersion.

Application	DBU	DBB	IIE	IPE	ITE
azureus	72%	-	5%	17%	5%
columba	84%	-	4%	9%	3%
compiler	77%	5%	2%	17%	-
findbugs	75%	-	8%	13%	5%
freetts	79%	-	3%	11%	7%
gruntsrud	83%	1%	-	12%	5%
jasperreports	76%	-	12%	9%	3%
jbidwatcher	80%	1%	8%	7%	4%
jboss	79%	1%	7%	10%	3%
jedit	82%	1%	3%	10%	3%
jgraph	75%	-	6%	14%	5%
jhotdraw	92%	-	3%	4%	1%
jtopen	81%	1%	5%	9%	4%
jvm_compress	40%	-	-	40%	20%
jvm_db	100%	-	-	-	-
jvm_jack	79%	-	10%	7%	3%
jvm_javac	65%	1%	8%	20%	7%
jvm_jess	71%	7%	10%	10%	2%
jvm_mpegaudio	82%	2%	5%	7%	5%
jvm_raytrace	83%	-	-	4%	13%
lectcomm	85%	2%	3%	8%	1%
megamek	80%	1%	6%	9%	4%
pmd	77%	1%	10%	11%	1%
spec_jbb	78%	-	1%	15%	5%
university	33%	17%	-	50%	-
<b>Average</b>	76%	2%	5%	13%	4%

Table 5: Relative occurrences of collaboration code skeletons.

### 6.3 Collaboration-Related Code

The code necessary to accommodate object collaborations in class-based object-oriented applications represents a noticeable portion of an application. This code includes declarations of reference instance variables and corresponding setter and getter methods, and in case of the indirect collaboration code skeletons, also the declaration of auxiliary classes. In addition to providing the code necessary for the setup of a collaboration, programmers must also provide the code necessary to preserve the consistency of the collaboration. For instance, if support for bilateralism is required and the DBB collaboration code skeleton is chosen, invocations of setter methods on either side of the collaboration must be coordinated. Otherwise, the bilateralism is readily destroyed.

Table 6 lists for every application in the analysis portfolio the absolute number of LOBC and presents the percentage of LOBC that is collaboration-related. For the collaboration-related portion of LOBC, both the lower and upper bound (Section 5.3) are provided. The portion of code that is collaboration-related tends to generally increase with an augmented occurrence of more complex collaboration code skeletons (i.e., DBB, IIE, IPE, and ITE) as well as with an augmented collaboration participation. For instance, the por-

tion of collaboration-related code for the applications `university` and `jasper_reports` is considerable as both applications exhibit a relatively pronounced occurrence of more complex collaboration code skeletons (i.e., DBB and IPE for the former and IIE for the latter), and both applications exhibit a collaboration participation above average. The portion of collaboration-related code for the application `jvm_compress`, on the other hand, is, despite the augmented use of complex collaboration code skeletons, below average since the collaboration participation of the application is below average too. The presence of unidentified collaboration participants, furthermore, diminishes the portion of collaboration-related code. For instance, the portion of collaboration-related code for the application `compiler` is surprisingly low considering the relatively pronounced occurrence of the ITE collaboration code skeleton and the relatively pronounced collaboration participation. This result is due to the high number of unidentified collaboration participants. For the analysis portfolio altogether, the portion of collaboration-related code lies between 18% and 46% of the total number of LOBC.

Application	LOBC	Lower	Upper
azureus	374627	22 %	59 %
columba	237132	25 %	56 %
compiler	32776	11 %	24 %
findbugs	312090	18 %	51 %
freetts	44991	14 %	41 %
gruntsnud	144105	27 %	63 %
jasperreports	290135	21 %	47 %
jbidwatcher	140164	11 %	32 %
jboss	1423590	18 %	55 %
jedit	269756	21 %	54 %
jgraph	43371	21 %	52 %
jhotdraw	87788	14 %	31 %
jtopen	1413736	20 %	51 %
jvm_compress	1911	9 %	32 %
jvm_db	1686	13 %	48 %
jvm_jack	19895	14 %	31 %
jvm_javac	46126	18 %	45 %
jvm_jess	21255	18 %	45 %
jvm_mpegaudio	35026	4 %	10 %
jvm_raytrace	7054	19 %	55 %
lectcomm	35519	28 %	61 %
megamek	525996	21 %	60 %
pmd	195886	11 %	37 %
spec_jbb	46191	16 %	43 %
university	1140	26 %	75 %
<b>Average</b>	230078	18 %	46 %

Table 6: *LOBC*: total number of lines of bytecode; *Lower* and *Upper* indicate the lower and upper bound, resp., in percent of collaboration-related LOBC.

## 6.4 Discussion

We can summarize the outcome of the empirical study of the analysis portfolio as follows:

- Object collaborations occur frequently. On average, the number of collaborations amounts to almost 80 % of the number of classes, and the average number of collaborations a class participates in amounts to 1.8.
- Object collaborations disperse both across classes and within classes. On average, 22 % of all collaborations distribute across at least 3 classes and the average dispersion amounts to 2.44 for the inter-class dispersion and to 2.09 for the intra-class dispersion.
- A noticeable portion of a program is concerned with setup and maintenance of collaborations. On average, between 18 % (lower bound) and 46 % (upper bound) of the total number of LOBC is collaboration-related.

The relatively pronounced presence of dispersion in the analysis portfolio questions the modularization of

the applications with regard to their support for object collaborations. Traditional software metrics, such as *coupling* and *cohesion* [24], were introduced to assess the modularization of software systems. The metrics of inter-class dispersion and intra-class dispersion we have introduced can be seen as refined versions of coupling and cohesion, respectively. Due to our focus on object collaborations, though, we have to apply a more fine-grained analysis than the one suggested by Chidamber and Kemerer [6]. Rather than applying the modularization metrics uniformly to all classes, attributes, and methods, we must first “factorize” the classes and their members with regard to their relevance to collaborations and then apply the modularization metrics onto these aggregations. In general the four metrics are related as follows: a high inter-class dispersion implies a high coupling between the collaborating classes, and a high intra-class dispersion implies a low cohesion within the participants of the collaborations.

Discussions on the modularity of object-oriented programs have been raised in the context of aspect-oriented programming [14, 15] too. In particular the results on modularity improvements [11] due to aspect-oriented implementations of certain design patterns [8] indicate that explicit treatment of conglomerate structures beyond class boundaries is required. Although aspects and explicit relationships are research areas addressing different goals (crosscutting recurring behavior as opposed to collaborative behavior), they share the desire to provide mechanisms to express separation of concerns. Space limitations prohibit a detailed discussion of this topic in this paper.

## 7 Related Work

The work presented in this paper intersects with efforts in basically three domains: static model extraction, dynamic model extraction, and relationship patterns. We introduce the domains in the order of their relatedness to our work and discuss the individual efforts of a particular domain in chronological order.

**Static model extraction** Jackson et al. [12] describe a tool Womble that allows the extraction of object models from Java programs. Like RelDJ, Womble performs a static bytecode analysis. Whereas Womble focuses on the identification of UML associations, RelDJ investigates the representation of such associations and exposes their implementation detail. Both Womble and RelDJ, apply data-flow analysis to deduce the element types of collections. In contrast to Womble, RelDJ takes advantage of generic declarations.

Our work differs along the same lines from the work by Matzko et al. [17]. The authors describe a tool to extract class diagrams from C++ source code and further provide a comparison of a set of tools targeting the same goal.

Guéhéneuc and Albin-Amiot [10] aim to provide clarification on the implementation of UML associations to

prevent discontinuity between the design and the implementation of an object-oriented system. The authors define a set of formal properties of associations and use these properties to distinguish different kinds of binary class relationships. The authors also describe a tool suite to detect binary class relationships in class-based object-oriented code. Unlike Guéhéneuc and Albin-Amiot, we exclude temporary associations (established through local variables and method parameters) from our study because of our focus on explicit relationships. However, the collaboration code skeletons, being implementation directives too, cover also bidirectional collaborations and collaborations that involve more than two classes.

Milanova [18] introduces an approach for identifying UML compositions in Java programs. The approach is based on a static object ownership inference and aims to verify the implementations of compositions. Since the separation of UML compositions from UML aggregations is not relevant for the identification of collaborations, RelDJ does not distinguish between compositions and aggregations.

**Relationship patterns** Noble [20] defines a set of patterns to model relationships in class-based object-oriented programming languages. The author provides valuable design guidelines on the appropriate use of the individual relationship patterns and further draws comparisons with the well-known design patterns [8] of object-oriented programming. Whereas Noble assumes the programmer’s awareness of relationships and therefore guides the programmer during system design, our work focuses on object-oriented programs that have been devised without relationships “in mind”. The collaboration code skeletons we introduced thus target actual implementations of object collaborations and focus on the technical aspects of the implementation.

**Dynamic model extraction** More loosely related to our work is the research in dynamic object model extraction. Mitchell [19] focuses on the identification of runtime structures of object-oriented programs to infer any adverse affect the structures may have on memory footprint. Flanagan et al. [7] present a dynamic approach to extract object models from legacy code.

## 8 Concluding Remarks

We presented an empirical study on the occurrence of object collaborations in Java programs. The study relies on the implementation of a fully automated program analysis tool (RelDJ) and on the collaboration code skeletons, a categorization of possible implementations of object collaborations. To quantify and evaluate the empirical results, we further introduced the metrics of inter-class dispersion and intra-class dispersion, measures related to the traditional metrics of coupling and cohesion, respectively, but allowing a more fine-grained, object collaboration-specific analysis.

The empirical study supports previous claims that object collaborations exist and are an important part of a program: code implementing such collaborations amounts to a considerable portion of an application. The study further shows that fairly “low-level” techniques (i.e., references) are used to implement object collaborations. Even if these techniques are common practice and their application is well understood, they fail to convey the underlying abstraction of what they represent. This situation in turn hinders program maintenance and evolution: since programmers must examine the code structure to identify and understand collaborations, they are likely to be misled by the implementation details and likely to introduce inconsistencies when updating the program.

The results of the study point to the benefits of introducing explicit support for collaborations at the programming language level. Preliminary studies with relationships as a model to express collaborations indicate that programs gain in expressiveness. However, there are many possible approaches to express collaborations and a number of research groups pursue this topic. Additional work will determine the most appropriate abstractions for collaborations. The next step is to find adequate language models to support explicit relationships on a large scale. With such a model, we can move towards capturing an important aspect of programs and expect a significant increase in our ability to understand and modify large-scale object-oriented programs.

## References

- [1] A. Albano, G. Ghelli, and R. Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *17th International Conference on Very Large Data Bases (VLDB’91)*, pages 565–575. Morgan Kaufmann Publishers Inc., 1991.
- [2] S. Balzer, T. R. Gross, and P. Eugster. A relational model of object collaborations and its use in reasoning about relationships. In *21st European Conference on Object-Oriented Programming (ECOOP’07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 323–346. Springer, 2007.
- [3] G. M. Bierman and A. Wren. First-class relationships in an object-oriented language. In *19th European Conference on Object-Oriented Programming (ECOOP’05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 262–286. Springer, 2005.
- [4] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. <http://asm.objectweb.org/>, November 2002.
- [5] A. Burns. The relationship detector: Uncovering hidden relationships in object-oriented programs. Technical Report 550, ETH Zurich, October 2006. Masterthesis.

- [6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering (TSE)*, 20(6):476–493, 1994.
- [7] C. Flanagan and S. N. Freund. Dynamic architecture extraction. In *1st Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification (FATES/RV'06)*, Lecture Notes in Computer Science, pages 209–224, 2006.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [9] G. Génova, J. Lloréns, and P. Martínez. The meaning of multiplicity of n-ary associations in UML. *Software and System Modeling*, 1(2):86–97, 2002.
- [10] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the uml cake. In *19nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 301–314. ACM, 2004.
- [11] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *17th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '02)*, pages 161–173. ACM Press, 2002.
- [12] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. *IEEE Transactions on Software Engineering (TSE)*, 27(2):156–169, 2001.
- [13] I. Jacobson, G. Booch, and J. E. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- [14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.
- [15] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *27th International Conference on Software Engineering (ICSE'05)*, pages 49 – 58. ACM Press, 2005.
- [16] Y. D. Liu and S. F. Smith. Interaction-based programming with classages. In *20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '05)*, pages 191–209. ACM Press, 2005.
- [17] S. Matzko, P. J. Clarke, T. H. Gibbs, B. A. Malloy, J. F. Power, and R. Monahan. Reveal: A tool to reverse engineer class diagrams. In *40th International Conference on Tools Pacific (TOOLS Pacific'02)*, pages 13–21. Australian Computer Society, Inc., 2002.
- [18] A. Milanova. Precise identification of composition relationships for uml class diagrams. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 76–85. ACM Press, 2005.
- [19] N. Mitchell. The runtime structure of object ownership. In *20th European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *Lecture Notes in Computer Science*, pages 74–98. Springer, 2006.
- [20] J. Noble. Basic relationship patterns. In *2nd European Conference on Pattern Languages of Programs (EuroPLoP'97)*, 1997.
- [21] D. J. Pearce and J. Noble. Relationship aspect patterns. In *11th European Conference on Pattern Languages of Programs (EuroPLoP'06)*, 2006.
- [22] D. J. Pearce and J. Noble. Relationship aspects. In *5th International Conference on Aspect-Oriented Software Development (AOSD '06)*, pages 75–86. ACM Press, 2006.
- [23] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 466–481. ACM Press, 1987.
- [24] S. R. Schach. *Classical and Object-Oriented Software Engineering*. McGraw-Hill Professional, 1996.
- [25] Standard Performance Evaluation Corporation. The SPEC JVM98 benchmarks. <http://www.spec.org/jvm98/>, 1998.
- [26] Standard Performance Evaluation Corporation. The SPEC JBB2005 benchmark. <http://www.spec.org/jbb2005/>, 2005.