# The relationship detector
## Uncovering hidden relationships in object-oriented programs

# The Relationship Detector

**Uncovering Hidden Relationships in Object–oriented Programs**

Alexandra Burns

Master Thesis

**Supervised by**
Stephanie Balzer
Prof. Dr. Thomas Gross

October 2006

Laboratory for Software Technology
Department of Computer Science
ETH Zurich

**Laboratory for Software Technology**

inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

## Abstract

While modelling languages, such as UML, support the notion of relationships between classes, there is no conceptual equivalent in object–oriented programming languages. Instead, relationships are implemented using the elements of a class–based approach, with the relationship code entangled with the class code. We analyse how relationships are represented and propose a tool, the *relationship detector*, to uncover the hidden relationships in object–oriented programs, and suggest a transformation to a relationship–based implementation. Finally, we test the relationship detector with a set of examples and show how object–oriented programming languages could benefit from introducing relationships as first–class concepts.

# Acknowledgments

First of all, special thanks go to my supervisor at the ETH, Stephanie Balzer. Without her constant support, complete dedication, constructive criticism and great cups of coffee, this thesis would not have been possible.
I am grateful to my supervising Professor Th. Gross, for the opportunity to write my thesis at the Laboratory for Software Technology and for his guidance and contribution.
Many thanks go to Andreas Scherrer and Barbara Scheuner for giving me insightful comments and reviewing my report. I would also like to thank H. Wegener at SwissRe for taking time to discuss object relationships.
Last but not least, I thank my parents, for their unconditional love and support.

# Contents

# Chapter 1

# Introduction

## 1.1 Overview

Object–oriented programming languages are recognised to ease the development of software systems thanks to the abstraction of a class – real world notions can be directly mapped onto classes – and thanks to the concept of inheritance, which together with polymorphism and dynamic binding make software adaptable and extendible. Recent endeavours in science have pointed out the absence of conceptual support of object relationships in object–oriented programming languages. Although the identification of the relationships that emerge from collaborating objects is considered to be a crucial step during system design, and therefore supported by various conceptual modelling languages, such as the Unified modelling Language (UML) [9] and Entity–Relationship (ER) diagrams [11], object–oriented programming languages do not provide a programming language counterpart to accommodate these relationships. This results in losing the abstraction of a relationship; the information about object relationships is instead distributed across several classes.

## 1.2 Scope

The goal of this master thesis is to develop a method for uncovering hidden object relationships in existing object–oriented programs. The method is based on a set of rules specifying both how to identify hidden object relationships in object–oriented code and how to change the existing code to accommodate these relationships using a first–class module. In addition, these rules are implemented in a tool, the relationship detector, that given a Java program applies the rules and identifies the buried relationships. The relationship detector furthermore makes suggestions on how to adapt the existing code when providing first–class support for these relationships.

## 1.3 Contributions

This master thesis contributes the following results:

- **Types of relationships in modelling object–oriented programs**
  We determine the types of relationships that occur in conceptual modelling. For this purpose, we extract a set of dimensions according to which we can classify the relationships. Furthermore, we analyse the research into relationships in the database community, and match it to the object–oriented programming concepts.

- **Relationships in class–based implementations**
  We identify how the different types of relationships are implemented in Java programs and how prevalent they are. In chapter 5 we test the relationship detector with a set of examples.

- **Validation of relationships as first–class concepts**
  Based on the evaluation of the relationship detector results, we show what benefits can be gained from introducing relationships as first–class concepts.

## 1.4 Outline

In chapter 2, we investigate the types of relationships in the design of software systems. These results provide the basis to ensure that all types of relationships are found with the relationship detector. Chapter 3 covers the transition from class– to relationship–based concepts. In chapter 4 we describe the design and implementation of the relationship detector tool. We then evaluate our results in chapter 5. In particular, we show what benefits can be gained from introducing relationships in object–oriented programs as first-class concept. Chapter 6 concludes this thesis with an outlook of possible extensions.

# Chapter 2

# Object Relationships

## 2.1 Introduction

The object–oriented paradigm models the world as objects that interact by exchanging messages. Most object–oriented programming languages use classes to describe these objects. As part of the design process, the programmer identifies the classes of the objects to be modelled. While mapping real–world entities to objects is a straightforward step in the design process, modelling collaborations of these objects is much less so. Common class–based object–oriented programming languages do not provide the necessary abstractions to define object collaborations. Instead, the object collaborations have to be introduced at implementation level, often using unilateral references. As a result of this approximation, a global view of the collaboration is not provided. With unilateral references, access to a relationship can only be gained through one of the relationship members. Additionally, information about a relationship may be distributed across multiple classes. Consistency conditions of a relationship, such as its irreflexiveness or symmetry, can also only be enforced at implementation level. Contracts [23] allow to establish the consistency conditions between collaborating classes in terms of specifying the mutual obligations between client and supplier classes. However, due to their focus on classes, the specification of collaborations remains object–centered, providing only the view of one collaborator. Object–oriented design issues, such as the dependent delegate dilemma [24] for example, provide evidence that such unilateral descriptions are insufficient for capturing object collaborations appropriately.

In the following sections, we first approach the subject of object relationships from the angle of the database research community. The background material on entity–relationship modeling provides some insight into how relationships are analysed. Next, we approach the subject from the angle of the object–oriented programming languages and concepts community. In particular, we introduce a framework for relationships as first–class concepts [5]. This framework provides the foundation of our thesis.

## 2.2 ER Modeling

In this section, we present the background material of the entity relationship model [11] that is relevant to our thesis. In particular, we are interested in the different types of relationships that are analysed in the database research community. These relationships are a key aspect to identifying hidden relationships: to be able to discover relationships in existing programs, we first need to determine the types of relationships we are looking for.

### 2.2.1 Relationship Dimensions

Elmasri et al. [12] identify the following properties of relationships in ER modeling:

- **Relationship Type**
  A relationship type R among n entity types $E_1, E_2, ..., E_n$ defines a set of associations among entities from these types. So it is a mathematical relation on $E_1, E_2, ..., E_n$.
- **Relationship Instance**
  Relationship instances $r_i$ associate n individual entities $(e_1, e_2, ..., e_n)$ and each entity $e_j$ in $r_i$ is a member of entity type $E_j$.
- **Degree**
  The degree of a relationship is the number of participating entity types. A relationship of degree two is called binary, while one of degree three is called ternary. The term n–ary is used for relationships with n $>$ 3.
- **Recursiveness**
  A recursive relationship relates entities of the same type to each other. A relationship can be recursive (also known as loop relationship) or not. When not specified, a relationship is always non–recursive.
- **Role**
  Each entity type that participates in a relationship type plays a role in that relationship. The role name signifies what role the participating entity is playing in a particular relationship instance, and thus helps to explain what the relationship means.
- **Constraints**
  Constraints limit the possible combinations of entities that may participate in a given relationship. The following constraints are defined:
  - **Participation**
    The participation constraint specifies whether an entity must participate in a certain relationship or not. The differentiations are: partial and total. Total means that every entity must participate in the relationship, while partial means that there are some entities that do not participate.
  - **Cardinality ratio**
    The cardinality ratio specifies the number of relationship instances that an entity can participate in. The relevant differentiations are: one–to–one, one–to–many and many–to–many for binary relationships. For ternary relationships, the constraints are 1:1:1, 1:1:N, 1:N:N, and N:N:N.

  Together, these constraints are also called structural constraints. However, we will use this term to denote constraints that go beyond these two concepts (see section 2.3).

## 2.3 Relationship Framework

This section contains a short overview of the work by Balzer et al. [5]. The paper proposes a framework for relationships as first–class concepts, and is the foundation of this thesis. In particular, the framework provides the basis for the transformation rules introduced in the following chapter.

### 2.3.1 Terminology

The framework uses the following terminology:

- **Classes:** Like other class–based, object–oriented approaches, classes act as foundries for objects. They provide an abstract description of both the data and the behaviour of a collection of similar objects. Classes portray that description by means of members, which can be attributes (data) or methods (behaviour). Each object, created by instantiating a class, exhibits specific values for its attributes, constituting the object's state.
- **Relationships:** Relationships are the abstractions encapsulating object collaborations. They are not meant to replace classes, but rather complement them. Like classes, relationships declare members, which can be attributes or methods. Relationships further can declare invariants specifying the invariant behaviour of object collaborations. Analogously to the differentiation between classes and instances of classes, the framework distinguishes a *relationship*, denoting the type of a relationship, from a *relationship instance*, denoting an instance of a relationship.
- **Participants:** Relationships describe the behaviour emerging from the collaboration of objects. The participants of a relationship are the defining classes of the objects involved in the collaboration. The participating objects are the actual objects being part of a particular relationship instance.
- **Roles:** Roles can be attached to the participants of relationships. The role mechanism allows to indicate the respective role of a participant in a relationship.

### 2.3.2 Member Interposition

Member interposition is concerned with how to couple the relationship–dependent members of classes with the relationships where the classes are participants of. This coupling is achieved by the concept that relationships can declare members, i.e. attributes and/or methods.

Members of relationships can be declared at relationship–level or participant–level. This mechanisms allows the programmer to declare attributes and methods describing objects, outside the defining classes. Member interposition is restricted by only permitting it as part of a relationship declaration, when the target of interposition is at the same time a participant of the relationship.

### 2.3.3 Invariants

Relationship invariants allow specifying the behaviour of a relationship. As the traditional invariants that have been introduced for class–based programming languages [23], relationship invariants are boolean conditions that should always hold. The difference between the two lies in the point of reference: Class–based invariants refer to class instances, while relationship invariants are predicates on relationship instances and their participating objects.

Two types of invariants are introduced:

- **Value–based** invariants are boolean predicates on the values of relationship instances and their participants. They are imposed on the members that are declared as part of the relationships or as part of the participating classes.

- **Structural invariants** are independent of the values that relationship instances or participating objects exhibit for their members. They restrict the participation of objects in relationships based on the occurrence of objects. For example, a structural invariant could enforce that a particular object participates with at most one other object in a given relationship. Structural invariants can be expressed in terms of mathematical relations:
  - **function**
    A binary relation over a set X and a set Y is functional if for all x in X there is no more than one element y in Y related to it.
  - **surjective**
    A binary relation R over a set X and a set Y is surjective if for all x in X there is a y in Y related to it. Figure 2.1 shows a surjective relation.
  - **injective**
    A binary relation R over a set X and a set Y is injective if, for every y in Y, there is at most one x in X so that x is related to y. Figure 2.2 shows an injective relation.
  - **partial**
    A partial function is a relation that associates each element x of a set X with at most one element y of set Y. In particular, this means that some elements of X may not be associated with any element of Y.
  - **total**
    A binary relation R over a set X and a set Y is total if it holds for all a in X and b in Y that a is related to b or b is related to a (or both) [28].
  - **symmetric**
    A binary relation R over a set X is symmetric if it holds for all a and b in X that if a is related to b then b is related to a [28].
  - **not symmetric**
    A binary relation R over a set X is not symmetric if there exists an a and b in X where a is related to b and b is not related to a.
  - **antisymmetric**
    A binary relation R on a set X is antisymmetric if, for all a and b in X, if a is related to b and b is related to a, then a = b [28]
  - **reflexive**
    A reflexive relation R on set X is one where for all a in X, a is R–related to itself [28].
  - **not reflexive**
    A not reflexive relation R on set X is one where there exists an a in X, that is not R–related to itself.
  - **irreflexive**
    An irreflexive relation R is one where for all a in X, a is never R–related to itself [28].

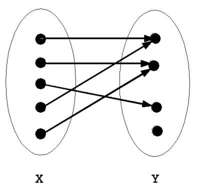

Figure 2.1: surjective relation

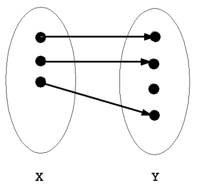

Figure 2.2: injective relation

# Chapter 3

# From Classes to Relationships

In the previous chapter, we introduced the foundation of this thesis, namely the concept of relationships as first–class objects. Common object–oriented programming languages (such as Java, C++) do not offer the abstraction of relationships. While real–world entities (such as persons, buildings etc.) can be easily mapped to objects, modelling the collaborations between these entities requires the user to employ approximations. This section explores these approximations and aims to identify the elements that constitute building relationships in existing object–oriented programming languages. That is, the programming elements that are used to model a relationship, without having the concept of relationships as first–class objects at hand. We explore the possibilities of transforming an implementation without relationships as first–class objects (the *source code*) into an implementation with relationships (the *target code*). This transformation is then represented as a set of transformation rules that can be applied to the source code, resulting in the target code.

In the first section, we give an overview of the existing work in the areas that are relevant to this thesis. Then we introduce a set of running examples. Each example is modelled first without first–class relationships (using *class–based concepts*), then with first–class relationships (using *relationship–based concepts*). Relying on the running examples, we attempt to identify those programming elements of a class–based approach that are candidates for hidden relationships and consequently will disappear in a relationship–based implementation. We call these programming elements in the class–based implementation *collaboration elements*, while the programming elements of a relationship–based implementation are named *relationship elements*. Next, we define the collaboration elements that are to be identified in the source code. These collaboration elements constitute the program elements of the class–based implementation, that point to a hidden relationship. They are used to *identify* the collaborations in a class–based implementation and provide the basis for the transformation rules. Finally, we then propose a set of transformation rules to take an implementation using class–based concepts to an implementation that incorporates relationship–based concepts. The collaboration elements identified in the previous step are then used to transform the implementation of a program from a class–based to a relationship based one.

## 3.1 Existing Work

To our knowledge, there is currently no research that covers the transformation from class–based to relationship–based implementations. However, there are a number of efforts in the area of relationships, both in the database and object–oriented programming communities. We provide a short overview of the aspects that are relevant to this thesis.

### 3.1.1 On Mapping ER and Relational Models into OO Schemas [25]

Badri Narasimhan et al describe a set of rules to map ER and relational models to object–oriented schemas. The object–oriented schemas include the conventional class–based concepts. In particular, the rules 6–8 for mapping *relationships* in ER schemas to object–oriented schema are relevant to our work:

- **Rule 6**
  Map each relationship with bidirectional pointers between the two classes involved, making sure that data in both the classes are consistent.
- **Rule 7**
  Map each 1:N relationship in an association between a parent class and a multivalued attribute.
  A 1:N relationship corresponds to what is called a one–to–many relationship in this thesis, while a multivalued attribute corresponds to a *collection object*.
- **Rule 8**
  Map each M:N relationship as an association between classes involving two multi–valued attributes.
  A M:N relationship corresponds to what is called a many–to–many relationship in this thesis.

The paper also contains a set of rules on how to map constraints in the ER schema to a object–oriented schema. The approach is based primarily on introducing a constraint class hierarchy with root class `Constraint`, with a subclass for every class in the schema. These subclasses contain all constraints of classes that have been instantiated. So, for each new class that is instantiated, a subclass with its constraints is appended to the `Constraints` class.
In relation to our work, the paper refers to conventional class–based concepts in the object–oriented schemas and does not include relationship–based concepts. However, the rules to map ER relationships help identify the elements in class–based implementations that point to collaborations between the participating objects. In terms of the constraints, we will be following the more extensive definitions of structural and value–based invariants, as defined by Balzer et al. [5].

### 3.1.2 Basic Relationship Patterns [26]

James Noble et al. approach the topic of relationships from a different angle. They propose a set of relationship patterns for object–oriented languages without first–class relationship support. Relationships are modelled using class–based concepts and are classified according to complexity and cardinality. For instance, a small, simple relationship is implemented by adding an attribute to represent the relationship. A large one–to–many relationship is implemented by adding a collection object (such as a list or array). Not all patterns are mutually exclusive, some patterns can be used to refine each other. The following patterns are introduced:

- **Relationship as Attribute**
  J. Noble et al. suggest using this first pattern for simple relationships. The idea is to create an attribute to present the relationship between two classes. For example, a `class A` has an attribute of type `B` to represent its relationship with `class B`.

- **Relationship Object**
  For more common or complex relationships, the authors suggest creating a relationship object containing all information related to the relationship. For instance, to associate classes `A` and `B`, create a new `class C` to manage the relationship.

- **Collection Object**
  The collection object pattern is used for one–to–many relationships, i.e. to associate one element with many others. This is done by creating a collection object, such as a list or array.

- **Active Value**
  An active value object is an object that contains a single value, with get and set methods accessing it. It is used to connect two other objects that are both dependent on this particular value.

- **Mutual Friends**
  Mutual friends is the name for a relationship that is managed on both sides. This means that the relationship can be accessed through both participating classes. The key issue here is maintaining consistency: the relationship has to be managed the same way on both sides. The steps the authors suggest are: first, split the relationship into two one–way relationships. Then keep the relationship consistent.

The paper provides valuable insight into the possibilities of modelling relationships without first–class support and therefore points to some of the relationship elements mentioned in the introduction of this chapter. J. Noble et al. provide suggestions on how collaborations should be implemented in a consistent and extendible fashion. In comparison, our work focuses on how programmers actually implement relationships, without the knowledge and application of these patterns. As we aim to identify relationships in class–based implementations that were written without regard to the relationship subject, we need to focus on how programmers implement relationships in practice, and not on how it should ideally be done.

### 3.1.3 Other Papers

Tetsuro Katayama et al. [18] propose a set of transformation rules to generate UML diagrams from existing UML diagrams, e.g. to generate an object diagram from a class, usecase, sequence, and collaboration diagram. Although the transformation between UML diagrams is not directly relevant to our approach, the methodology of deriving the transformation rules is considered in this thesis.

Another two papers are concerned with transforming UML diagrams into Java code:
Mathupayas Thongmak et al. [37] suggest design rules to transform UML sequence diagrams into Java code, while Gregor Engels et al. [13] propose the transformation of UML collaboration diagrams into Java code. Both approaches are connected to the method of this thesis, with an inverted mapping procedure. While our approach maps Java code to relationship information, the approach of the teams above is to map UML information to Java code. The work is especially relevant for the elements of UML that are concerned with object collaborations.

## 3.2 Running Examples

In this section, we aim to derive the key elements of a class–based program that point to a hidden relationship. To this end, we introduce a set of running examples. First, we implement the modelling example in a class–based fashion using Java. Then we perform the same using relationship–based concepts. The relationship–based implementation follows the terminology and notation defined in [5].

In addition, we use the following terminology to describe the elements of the implementations:

- **Collection classes**: Collection classes are container and collection classes such as Arrays, Lists and Trees. This also includes Java specific library classes such as LinkedList and Vector.
- **Collection objects**: Collection objects are the instances of the collection classes.
- **Field**: A field is a data member of a class (also known as property or attribute).
- **Collaboration**: The term collaboration refers to a hidden relationship in a class–based implementation. It points to an *implicit* declaration of a relationship, such as a field in a class. The term *collaboration field* refers to the field that contains the collaboration reference. If the collaboration is one–to–one, its collaboration field is a single–valued field. If it is one–to–many, the collaboration field is multi–valued, i.e. contains a collection object.
- **Relationship**: The term relationship refers to an explicit relationship in a relationship–based implementation. It points to an *explicit* declaration of a relationship, as found in [5] and [8].

### 3.2.1 Relationship Categories

To analyse the running examples, we will be looking at a set of relationship categories. Our approach combines the relationship dimensions described in section 2.2, with the structural invariants described in the section 2.3. We classify a relationship based on

- **Degree**
  As defined in 2.2
- **Recursiveness**
  As defined in 2.2
- **Invariants**
  The invariants include cardinality constraints as defined in 2.2 and value–based and structural invariants as proposed in 2.3.

As the debate over whether higher–order relationships are necessary is still ongoing, we will consider binary and ternary relationships only. Further, we introduce the term *composite relationship* for relationships composed of a number of binary or ternary relationships. The cardinality constraints differ according to the number of participants on each side of the relation, i.e. one or many.
The running examples cover all collaboration categories and invariants, as defined in the previous section. Based on these examples we extract the collaboration elements and show how the different constraints are enforced in the class–based implementation.

### 3.2.2 Authors

The entity–relationship (ER) diagram [11] model in figure 3.1 describes a simple relationship between an entity PERSON and an entity BOOK. A PERSON has the attributes NAME and AGE, while a BOOK has a TITLE and a PUBLISHER. While not every PERSON takes part in the relationship AUTHOR, every BOOK has at least one PERSON associated with it.

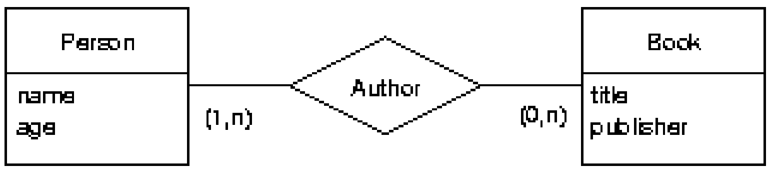

Figure 3.1: ER Diagram of Author collaboration

**Collaboration category:** Binary many–to–many
Surjective

```
import java.util.*;

public class Person {

    private LinkedList books;

    public void writeBook(Book book){
           books.add(book);
           book.setAuthor(this);
    }

}

import java.util.*;

public class Book {

    private LinkedList authors;

    protected void setAuthor(Person person){
           authors.add(person);
    }

}
...
Person ernestHemingway   = new Person();
Book theOldManAndTheSea  = new Book();
Book theSunAlsoRises     = new Book();

ernestHemingway.writeBook(theOldManAndTheSea);
ernestHemingway.writeBook(theSunAlsoRises);
```

Listing 3.1: Class-based implementation of Author collaboration

The cardinality constraints are specific to this example. However, they do not change the collaboration elements of the class–based implementation in a substantial way. While the "one" side of a relationship is modelled by a single–valued field of the class (e.g. a PERSON field), the "many" side is represented by a multi–valued field, i.e. a collection object field. The cardinalities that include a zero (e.g. (0,n)) are not handled differently to the cardinalities including one (e.g.

(1,n)). They just imply that the collaboration collection field or object does not have to be instantiated.

In this model, an author can write several books and a book can have several authors. The collaboration is therefore a many–to–many collaboration. The collaboration elements are:

- The collection object fields `books` and `authors`.
  If a book could only have one author (i.e. the collaboration were one–to–many) then the `authors` Collection object field would be replaced by a single AUTHOR field.
- The method calls on the collection objects:
  `books.add(book)` and `authors.add(person)`. These calls add an object instance to the collection object, therefore adding the object instance to the one–to–many collaboration. In general, adding an object instance to a one–to–one collaboration entails an assignment to the single field. For example, `author = ernestHemingway`. Adding an object to a one–to–many collaboration entails adding the object instance to the collection object. This is done by calling the relevant methods of this collection object, such as `add()` for LinkedLists. Implicitly, adding an element to a collection item translates to an assignment as well: `books = books plus the new element`.
- Class `Person` has a field `books`. The objects contained in this collection object are of type `Book`. This points to a one–to–many collaboration between classes Person and Book. Class `Book` has a field `authors`. The objects contained are `Person` objects. This points to a one–to–many collaboration between `Book` and `Person`. Together, these two collaborations add up to a binary many–to–many collaboration.
- The invariants are enforced by the following code sequence. Surjectivity is maintained by always adding the author to the corresponding book object: `book.setAuthor(this)`

```
relationship Author {
    participants(Person person, Book book);
    invariant
           surjective(person, book);
}

Person ernestHemingway    = new Person();
Book theOldManAndTheSea   = new Book();
Book theSunAlsoRises      = new Book();

Author.add(ernestHemingway, theOldManAndTheSea);
Author.add(ernestHemingway, theSunAlsoRises);
```

Listing 3.2: Relationship-based implementation of Author collaboration

The relationship–based implementation creates a relationship associating `Person` with `Book`. Each `Person` or `Book` object can take part in multiple `Author` relationships. The surjectivtiy constraint is enforced by specifying an invariant in the relationship declaration.

### 3.2.3 Yin and Yang

The example in figure 3.2 is taken from [5]. It describes the relationship between entities PERSON. Not every person is married, but each person can be married to one person only, and not to him–/herself. Additionally, both yin and yang must be at least 18 years of age.

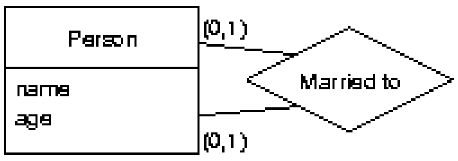

Figure 3.2: ER Diagram of Yin Yang collaboration

| | |
|---|---|
| **Collaboration category:** | Binary recursive one–to–one |
| | Irreflexive |
| | Symmetric |
| | Partial injective |

```
public class Person {

    private String name;
    private int age;
    private Person spouse;
    ...

    void getEngaged(Person p){
           if ((spouse != this) && (p.age >= 18) &&
                  (this.age >= 18)){
                  this.spouse = p;
           }
    }

    void marry(Person p){
           if((p != this) && (spouse == null) &&
             (p != null) && (p.spouse == null)) &&
             (p.age >= 18) && (this.age >= 18)){
                  getEngaged(p);
                  p.getEngaged(this);
           }
    }
}
```

Listing 3.3: Class-based implementation of Marriage collaboration

In the original paper, the class–based implementation includes contracts. Our class–based implementation in Java has to explicitly ensure the invariants are enforced by including code in the methods. The collaboration elements are:

- The Person field `spouse`
  This points to a one–to–one collaboration between PERSON and PERSON. As the relationship is a partial injection (as opposed to a total one), not every Person has a value assigned to the `spouse` field.
- The invariants are enforced by the following code sequence:
  Symmetry is enforced by calling both `getEngaged(p)` and `p.getEngaged(this)` in the method

`marry(Person p)` method. Partial injectivity is implicitly enforced by having a single–valued collaboration field `spouse`, as opposed to a collection item field. This means that only one person can be assigned as a spouse. It is further explicitly enforced by ensuring that no value was assigned before, i.e. the value was null: `(spouse == null) && (p.spouse == null)`.

```
relationship Marriage {

    participants(Person yin, Person yang);
    invariant
           partialInjection(yin, yang) &&
           symmetric(yin, yang) &&
           irreflexive(yin, yang) &&
           yin.age >= 18 &&
           yang.age >= 18;
}
```

Listing 3.4: Relationship-based implementation of Marriage collaboration

### 3.2.4 Narcissists

The model in figure 3.3 describes the relationship between NARCISSIST entities. Every narcissist loves him–/herself and him–/herself only.

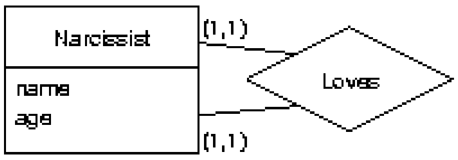

Figure 3.3: ER Diagram of Narcissist collaboration

| | |
|---|---|
| **Collaboration category:** | Binary recursive one–to–one |
| | Reflexive |
| | Total bijection |

```
public class Narcissist {

    private Narcissist iLove;

    public Narcissist(){
          this.iLove = this;
    }
}
```

Listing 3.5: Class-based implementation of Narcissist collaboration

The class–based implementation enforces reflexivity by ensuring in the constructor that the collaboration field `iLove` is assigned the value of `this`.

```
relationship LoveMyself {
    participants (Narcissist me, Narcissist myself);
    invariant
          reflexive(me, myself);
          totalBijection(me, myself);
}
```

Listing 3.6: Relationship-based implementation of Narcissist collaboration

The relationship–based implementation allows the programmer to specify the constraints, without having to explicitly enforce them.

### 3.2.5 Employees and Managers

The model in figure 3.4 describes a recursive relationship between entities EMPLOYEE. An employee is managed by an employee and manages other employees (except for the top– and low–level employees).

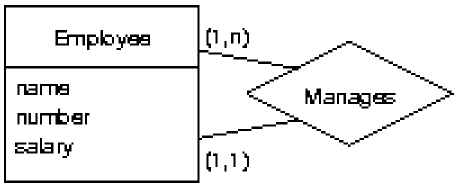

Figure 3.4: ER Diagram of Employee collaboration

**Collaboration category:** Binary recursive one–to–many
Antisymmetric
Surjective total function

```
import java.util.*;

public class Employee {

    private String name;
    private int number;
    private int salary;

    private LinkedList employees;
    private Employee manager;

    public Employee(Employee manager){
           if (manager.manager != this) {
                  this.manager = manager;
           }
    }

    public boolean addEmployee(Employee employee){
           if (employee != this.manager){
                  employees.add(employee);
                  return true;
           }
           else
                  return false;
    }
}
```

Listing 3.7: Class-based implementation of Employee collaboration

This implementation pattern associates `Employee` objects with other `Employee` objects. The class–based implementation needs to include code in both the constructor and the method `addEmployee` to enforce certain invariants. These invariants are: no employee can be his/her own manager and no two employees can have each other as managers.
The collaboration elements are:

- The Employee field `manager`
  This points to a one–to–one collaboration between Employee and Employee
- The LinkedList collection object `employees`
  Again, the collection object points to the fact that the collaboration is one–to–many between

Employee and Employee. Together, this adds up to a binary one–to–many collaboration between Employee and Employee.

- The constructor `public Employee(Employee manager)` with the assignment `this.manager = manager`
- The method call `employees.add(employee)`
- The code that ensures the invariants of the collaboration are satisfied: the if–statement in the constructor and the if–statement in the `addEmployee` method. `if(manager.manager<>this)` enforces antisymmetry, while surjectivity is enforced by the assignment `this.manager=manager` in the constructor.

```
relationship Manages {

    participants(Employee employee, Employee manager);
    invariant
           antisymmetric(employee, manager) &&
           surjectiveTotalFunction(employee, manager) &&
    ...
}

Employee manager1 = new Employee();
Employee manager2 = new Employee();
Employee employee1 = new Employee();
Employee employee2 = new Employee();
Employee employee3 = new Employee();

Manages.add(employee1, manager1);
Manages.add(employee2, manager1);
Manages.add(employee3, manager1);
Manages.add(manager1, manager2);
```

Listing 3.8: Relationship-based implementation of Employee collaboration

The code that was necessary in the class–based implementation to enforce the invariants is not present in the relationship–based implementation. It is replaced by a declaration of the invariants: the relationship is antisymmetric and surjective.

### 3.2.6 Operas, Singers and Parts

The model in figure 3.5 describes a ternary relationship: the Singers of an opera ensemble perform different parts in different operas. For instance, a soprano voice singer may perform the part of “Floria Tosca” in “Tosca” and of “Angelica” in “Orlando” in the same season of the opera house.

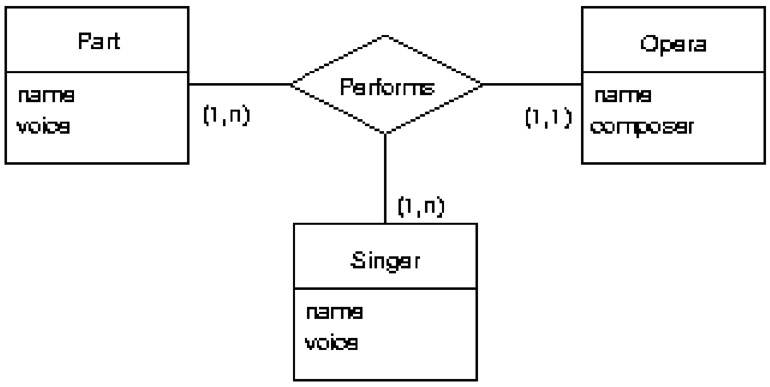

Figure 3.5: ER Diagram of Opera collaboration

**Collaboration category:** Ternary many–to–many

---

```
public class Opera {

    private String name;
    private String composer;

}
public class Singer {

    private String name;
    private String voice;
}
public class Part {

    private String name;
    private String voice;
}

import java.util.*
public class OperaPerformance {

    private Opera opera;
    private HashMap roles;

    ...
}

Opera      tosca                = new Opera();
Opera      laTraviata           = new Opera();
Singer     mariaCallas          = new Singer();
Singer     joseCarreras         = new Singer();
Part       floriaTosca          = new Part();
Part       alfredo              = new Part();
```

```
OperaPerformance royalOperaHouse1974 = new OperaPerformance();
royalOperaHouse1974.setOpera(laTraviata);
royalOperaHouse1974.addRole(joseCarreras, alfredo);

OperaPerformance coventGarden1964  = new OperaPerformance();
coventGarden1964.setOpera(tosca);
coventGarden1964.addRole(mariaCallas, floriaTosca);
```

Listing 3.9: Class-based implementation of Opera collaboration

The class–based implementation of the Opera collaboration introduces a new class: `OperaPerformance`, a so–called relationship object [26]. To remain consistent with our terminology, we will be referring to such an object as *collaboration object*. This means that it contains and maintains the collaboration references. One of these references is a single–valued collaboration field `opera`, which points to a one–to–one collaboration. The `HashMap roles` field associates the Singer with his or her Part in the Opera. So the collaboration elements are:

- The collaboration object `OperaPerformance`
- The collaboration fields it contains: `opera` and `roles`.

```
relationship OperaPerformance {

    participants(Singer singer, Opera opera);
    participants(Part part, Opera opera);
    participants(Singer singer, Part part);

}

OperaPerformance.add(mariaCallas, tosca);
OperaPerformance.add(floriaTosca, tosca);
OperaPerformance.add(mariaCallas, floriaTosca);

OperaPerformance.add(joseCarreras, laTraviata);
OperaPerformance.add(alfredo, laTraviata);
OperaPerformance.add(joseCarreras, alfredo);
```

Listing 3.10: Relationship-based implementation of Opera collaboration

The relationship–based implementation breaks down the ternary relationship into three binary relationships: *SingsIn, SingsAs,* and *HasPart.*

### 3.2.7 Students, Lecturers and Courses

The model in figure 3.6 describes some example elements of a university. STUDENTS attend COURSES, which are taught by LECTURERS and take place at LOCATIONS.

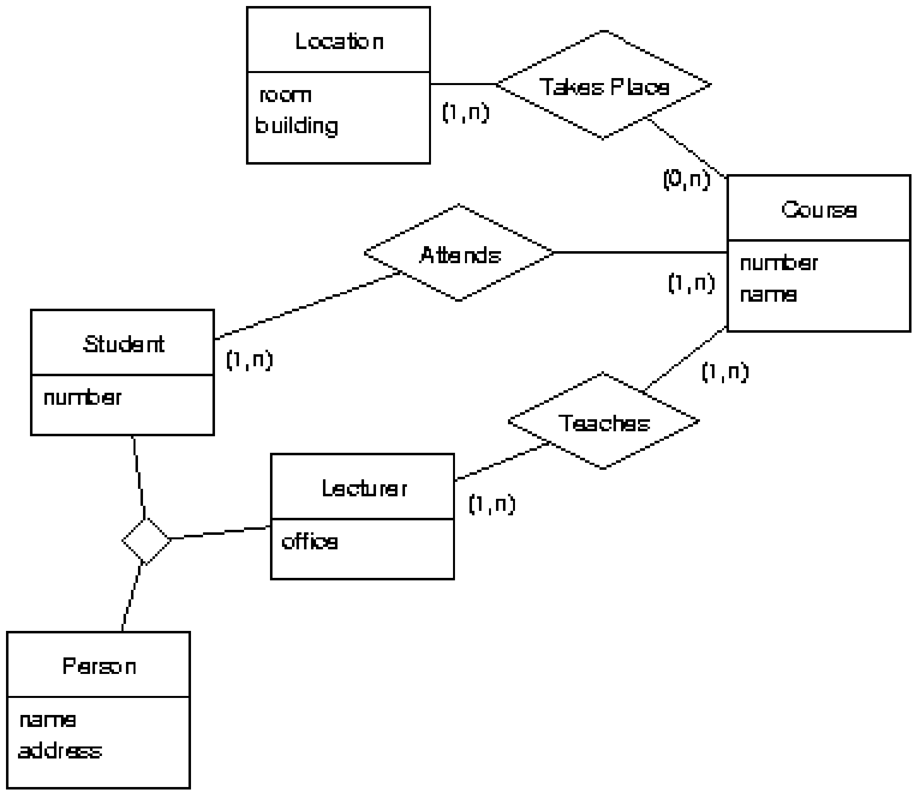

Figure 3.6: ER Diagram of Student collaboration

| | |
|---|---|
| **Collaboration category:** | Composite relationship composed of two binary many–to–many relationships and one binary one–to–many relationship. |
| Teaches | Total surjective relation |
| Attends | Value based invariants: each Course has maximum number of Students associated with it, each Student has a maximum number of Courses associated with it. Total surjective relation |
| Takes Place | Total relation |

```
public class Person {

    private String name;
    private String address;

    public Person(String name, String address){
           this.name = name;
           this.address = address;
    }
}

import java.util.*;
public class Lecturer extends Person {

    private Location office;
    private LinkedList courses;

    public Lecturer(Location office, String name, String address){
           super(name, address);
           this.office = office;
    }

    public void addCourse(Course course){
           courses.add(course);
    }
}

public class Person {

    private String name;
    private String address;

    public Person(String name, String address){
           this.name = name;
           this.address = address;
    }
}

public class Location {

    private Building building;
    private Room room;

    public Location(Building building, Room room){
           this.building = building;
           this.room = room;
    }
}

import java.util.*;
public class Course {

    private Lecturer lecturer;
    private Location location;
    private LinkedList participants;
    private int maxStudents;
    private String name;
    public Course(Lecturer lecturer, Location location){
           this.lecturer = lecturer;
           this.location = location;
    }

    public boolean enrolInCourse(Student student){
           if (participants.size() + 1 <= maxStudents){
                  participants.add(student);
                  return true;
           }
           else
                  return false;
    }
}
```

Listing 3.11: Class-based implementation of Student collaboration

The collaboration elements are:

- The LinkedList collection field `courses` in class Lecturer
  This points to a one–to–many collaboration between Lecturer and Course
- The LinkedList collection field `courses` in class Student
  Again, the collection object points to the fact that the collaboration is one–to–many between Student and Course.
- The LinkedList collection field `participants` in class Course
- The three collaborations mentioned above add up to many–to–many binary collaborations between Student – Course and Lecturer – Course.
- The method calls `courses.add(course)` and `participants.add(course)`.
- The code that ensures the invariants of the collaboration are satisfied: the if–statements on the collection fields (e.g. `if (participants.size()+1 <= maxStudents` )

```
relationship Teaches(){
    participants(Person lecturer, Course course);

    //member interposition
    String {Person} office;

    invariant
           totalSurjectiveRelation(lecturer, course);
}

relationship Attends(){
    participants(Person student, Course course);

    //member interposition
    String {Person} number;
    int {Person} numberOfCourses;
    int {Course} numberOfStudents;

    invariant
           ((numberOfStudents <= course.maxStudents) &&
           (numberOfCourses <= student.maxCourses));
           totalSurjectiveRelation(student, course);
}

relationship TakesPlace(){
    participants(Course course, Location location);
    invariant
            totalRelation(course, location);
}
```

Listing 3.12: Relationship-based implementation of Student collaboration

The implementation using relationship–based concepts has one major difference to the one using class–based concepts: The inheritance relationships between `Person` – `Student` and `Person` – `Lecturer` are missing. Instead, there is one class `Person`. Depending on whether a `Person` in a relationship has the role of a `Student` or a `Lecturer`, the necessary fields are added using *member interposition*. The principle of modelling a generalisation (inheritance) hierarchy by using relationships is also advocated by [6]. When mapping an ER model to the relational model (e.g. to create database tables), generalisation hierarchies have to be removed. Batini et al. suggest three possibilities. The terms *superentity* and *subentity* refer to the entities participating in the generalization hierarchy:

1. Collapse the generalization hierarchy into a single entity. This is done by adding all the attributes of all subentities to the superentity.
2. Remove the superentity but keep the subentities. The inherited attributes must be propagated into the subentities.
3. Retain all entities and establish explicit relationships among the super– and subentities.

The advantage of the third option is that it is always feasible. In our work, we will be following this approach.

## 3.3 Identifying Collaborations

In this section, we derive some general concepts from the running examples described in the previous section. We show what elements constitute a collaboration in a class–based implementation and break these elements down into different categories. The collaboration elements we identify provide the basis for the transformation rules introduced in the next chapter. These elements will be extracted and used to generate a new relationship–based implementation of the original problem.

### 3.3.1 Collaboration Elements

Resulting from the study of various real–life class–based implementations, we now derive the following collaboration elements. We use a color schema to visualise the different elements. To illustrate this section, we analyse an exercise example to identify the collaboration elements (Note to Stephanie: the Calendar.java example). The following collaboration elements are ordered according to how closely they are connected to the collaboration (starting with the strongest connection). Collaboration fields are the collaboration elements that are most strongly connected to the collaboration, as they constitute the link to the other participating objects.

1. **Collaboration fields:**
   These fields are either single–valued fields of a reference type or collection item fields containing reference types.

2. **Collaboration field assignments:**
   In the case of the single–valued collaboration fields, the assignment is explicit: `coll_field = element`. With the collection item fields, the assignment is implicit by using certain method calls: `linkedList.add(element)`. To this purpose, calls on the collection items have to be analysed on a lower level to check whether an assignment to the field takes place or not. For example, `linkedList.add(element)` contains such an assignment, while `linkedList.contains(element)` does not.

3. **Collaboration–related methods and constructors:**
   Methods and constructors containing assignments to collaboration fields. This property is recursive, so any method (or constructor resp.) calling a method (or constructor) containing a collaboration field assignment is added to the set of methods and constructors containing collaboration field assignments.

4. **Structural invariants:**
   Structural invariants can be implicitly enforced by the structure of the collaboration fields (see section 3.3.2 or explicitly enforced by code elements. The structural invariant collaboration elements are such code patterns.

5. **Value–based invariants:**
   To identify value–based invariants (see section 3.3.2), we need to search for if–clauses that contain a collaboration field assignment.

6. **Collaboration–related control code:**
   This is the code that calls collaboration–related methods and/or constructors.

7. **Code on collaboration field elements**:
   This type of code is not directly associated with the collaboration, but includes non–assigning method calls on the collaboration field elements. As these collaboration fields will be removed from the class in the class–based implementation and added to a new relationship in the relationship–based implementation, the method calls have to be moved as well. For instance, `linkedList.size()` cannot be called in the class, if it no longer contains the collaboration field. The code may also be subject to optimisations in the relationship–based code, meaning that parts of it may no longer be necessary.

8. **Methods with code on collaboration field elements**:
   These methods contain code as described above.

9. **Collaboration–participating classes**:
   These are the classes that contain one or more collaboration fields, appear as type of a collaboration field of another class or are direct or indirect collaboration classes.

### 3.3.2 Collaboration Implementation Options

There are different options to model collaborations in class–based implementations. To identify the different collaboration categories (e.g. binary collaborations), the following options need to be taken into account.

- **Binary collaboration between two classes A and B**

  1. **Collaboration controlled by class A**
     Class A has a collaboration field of type B (single–valued or collection item, depending on cardinality). The advantage of this solution is its simplicity. Figure 3.7 shows this option.

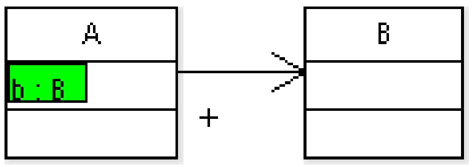

Figure 3.7: Collaboration controlled by class A

  2. **Collaboration controlled by class B**
     Class B has a collaboration field of type A (single–valued or collection item, depending on cardinality). Again, the advantage of this solution is its simplicity. Figure 3.8 shows this option.

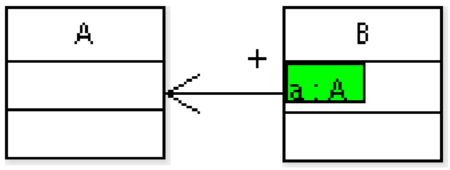

Figure 3.8: Collaboration controlled by class B

3. **Collaboration controlled by both class A and B**
   Both classes each have a collaboration field of type B or A resp. That means the collaboration is maintained on both sides. The advantage of this solution is that the collaboration can be accessed through both participating objects. Figure 3.9 shows this option.

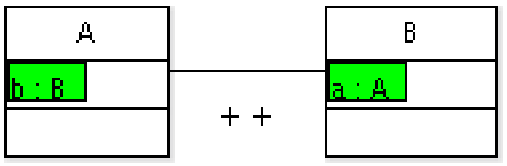

Figure 3.9: Collaboration controlled by classes A and B

4. **Direct collaboration object**
   This collaboration object ABCol contains a two–dimensional collection item that associates instances of A and B. For instance, HashMap, EnumMap etc. The advantage of this solution is that classes A and B are not affected by the collaboration, meaning that they contain no collaboration–related code. This means in particular that neither of the participating instances has control over the collaboration and the other participating object. Figure 3.10 shows this option.

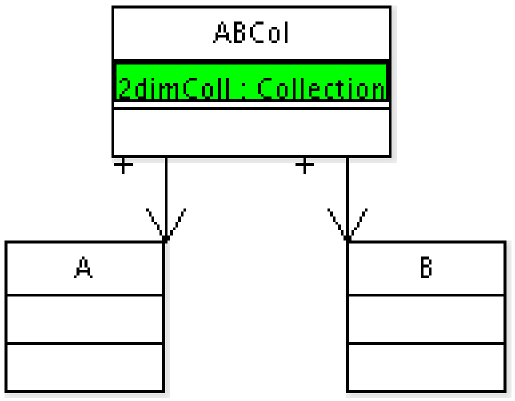

Figure 3.10: Collaboration controlled by direct collaboration object

5. **Indirect collaboration object**
   ABCol is a class with two single–valued collaboration fields of type A and B. For each collaboration between A and B, there exists an instance of ABCol. Additionally, a class ABColCollection stores the references to the ABCol instances in a collection item. Again, the advantage of this solution is that classes A and B have no knowledge of the collaborations they participate in. The additional indirection with the ABCol and ABColCollection classes provides more points of access to the collaboration elements. In particular, it also allows browsing the collaboration instances by retrieving them from the collection. However, the relationship retrieval property still has to be explicitly declared and implemented using two collaboration classes. In the framework suggested by Balzer et al. [5], object retrieval is provided by retrieval operators and no longer has to be explicitly implemented by the user. Figure 3.11 shows this option.

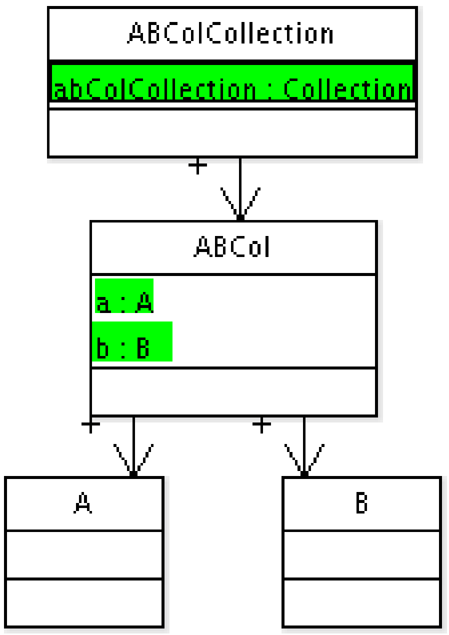

Figure 3.11: Collaboration controlled by indirect collaboration object

- **Ternary collaboration between three classes A, B and C**

1. **Indirect collaboration object**
   ABCCol is a class with three single–valued collaboration fields of type A, B and C. For each collaboration between A, B and C there exists an instance of ABCCol. Additionally, a class ABCColCollection stores the references to the ABCol instances in a collection item. For ternary relationships, a direct collaboration object is not an implementation option: there are no three–dimensional collection items offered by Java. Of course, programmers can introduce their own three–dimensional data structure. This case, however, corresponds to the same indirect collaboration object pattern with the data structure object acting as the ABCCol object. Figure 3.12 shows this option.

2. **Decomposed ternary collaboration**
   The ternary collaboration is decomposed into three binary collaborations A–B, B–C and A–C. For each of these three collaborations, the options mentioned for the binary collaborations apply.

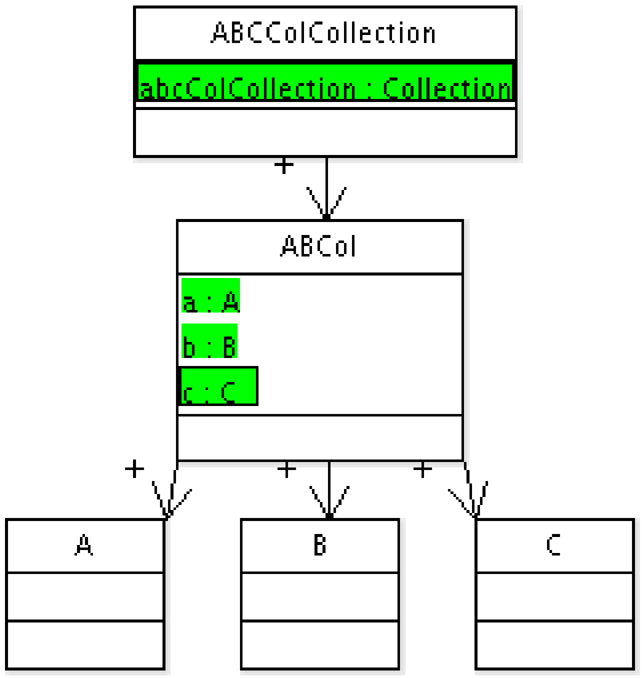

Figure 3.12: Collaboration controlled by indirect collaboration object

### Collaboration Invariants

While there are various options to implement the collaboration categories, there are also options to enforce invariants (structural and value–based) in class–based implementations. However, not all structural invariants can be enforced and not all value–based invariants can be positively identified. An *implicit* enforcement of an invariant means that the invariant is not explicitly implemented by the programmer. It is rather implied by the structure of the implementation. An *explicit* enforcement of an invariant means that the implementation includes explicit code to enforce it.

#### Structural Invariants

Table 3.1 shows how the structural invariants of a collaboration occur in a object–oriented implementation.

| | |
|---|---|
| **total** | explicitly enforced by assigning the collaboration field upon creation in the constructor, thus ensuring that the field is not uninitialized. |
| **partial** | not enforced. Collaboration fields can be uninitialised. |
| **function** | implicitly enforced by representing the collaboration field as a single–valued field. This ensures that only one participant can be associated with it. |

| | |
|---|---|
| **surjection** | enforced by setting the collaboration field upon creation of the participating object. For instance, when creating a book, set the author at the same time.<br>1. possibility: include the second object participating in the collaboration in the constructor of the first object participating in the collaboration. This ensures that the field is set and not null.<br>2. possibility: before each method call on the object participating in the collaboration, check that the collaboration field has been assigned a value. If not, handle the error. |
| **injection** | implicitly enforced by representing the collaboration in a single–valued field (as opposed to a collection object field) in both participants. This ensures that only one value can be assigned. Injectivity is further enforced explicitly by ensuring (upon assignment) that no other value was assigned before |
| **bijection** | enforces both surjection and injection, with the possibilities to do so described above. |
| **symmetric** | 1. possibility: allow addition to the collaboration through one of the participants only. In the running example “Marriage”, when yin gets married to yang, yang also gets married to yin.<br>2. possibility: explicitly check the collaboration fields of the participants. For example, to ensure that the collaboration MARRIAGE is symmetric, check that the spouse of the spouse points to the original class. These checks become more complicated the more participants are involved in a relationship: for a many–to–many collaboration, each collection object on both sides has to be checked. |
| **antisymmetric** | explicitly enforced: for a one–to–one collaboration, test that the single–valued collaboration field refers to a class that itself does not refer back; for a one–to–many collaboration, test that the collection object field does not include a reference to the class which refers back. |
| **reflexive** | explicitly enforced by ensuring that the collaboration field points to itself. |
| **irreflexive** | explicitly enforced. For one–to–one and one–to–many relationships: test before adding the reference to the collaboration field that the collaboration field does not include a reference to the class itself. |

Table 3.1: Structural Invariants

### Value–based Invariants

Value–based invariants define conditions on whether a participant is added to a collaboration or not. These conditions check for values and may occur in many different patterns. We analyse these patterns in more detail in the implementation section in 4.3.1.

## 3.4 Transformation Rules

After identifying the collaboration category and the constraints, we now have the basis to transform the class–based implementation to a relationship–based one. For each of the collaboration options described in section 3.3.2 we provide a transformation pattern.

This section defines the transformation patterns to convert a class–based implementation into a relationship–based implementation. With the collaboration elements described in the previous sections, the collaboration category is first identified. Based on this pattern, the transformation to a relationship–based implementation is then described. Each pattern is illustrated with a running example.

We use the following naming convention: a relationship is named after the two participating classes plus the appendix Relationship. For instance, a relationship between `class Client` and `class Server` is named `ClientServerRelationship`.

### 3.4.1 Binary collaboration controlled by class A

Situation: The class–based implementation has been identified as binary collaboration between classes A and B, that is controlled by class A, meaning that only class A maintains the references to class B. The following steps describe how the class–based implementation is transformed into a relationship–based implementation.

1. Create a new relationship `ABRelationship`. Substitute A and B for the class names.
   `relationship ABRelationship { }`
2. Add a participant clause
   `participants(A a, B b);`
3. Remove the collaboration field from class A
4. Add the invariant clauses to the relationship, depending on the collaboration category and constraints, for example:
   `surjective(a, b);`
5. Move the collaboration–related methods of class A to the `ABRelationship`.
   - Remove the collaboration field assignments in the relationship code.
   - If the collaboration–related method is now empty, remove it from the relationship code.
6. Remove the collaboration–related methods from the class–based implementation of class A
7. Remove collaboration field assignments from the collaboration–related constructors of class A, if applicable.
   - If a collaboration field was previously assigned a value passed to the constructor, update the constructor signature by removing the value from the signature.
8. Update the collaboration–related control code:
   - For all calls of a collaboration–related method in class A: replace the object of type A with the relationship instance and prefix the collaboration–related method with an `a_`.
   - For all instantiations with a collaboration–related constructor: update the constructor call with the new signature.
9. Move all methods of class A with code on collaboration field elements to the relationship, if applicable.
   - Update any calls on these methods by replacing the object of type A with the relationship instance.

### 3.4.2 Binary collaboration controlled by class B

Situation: The class–based implementation has been identified as binary collaboration between classes A and B, that is controlled by class B, meaning that only class B maintains the references to class A. The pattern is the same as for a binary collaboration controlled by class A. Substitute A for B in the pattern above and vice versa.

### 3.4.3 Binary collaboration controlled by classes A and B

Situation: The class–based implementation has been identified as binary collaboration between classes A and B, that is controlled by both classes. The following steps describe how the class–based implementation is transformed into a relationship–based implementation.

**1.** Create a new relationship `ABRelationship`. Substitute A and B for the class names.
```
relationship ABRelationship { }
```

**2.** Add a participant clause
```
participants(A a, B b);
```

**3.** Remove the collaboration fields from the classes A and B

**4.** Add the invariant clauses to the relationship, depending on the collaboration category and constraints, for example:
```
surjective(a, b);
```

**5.** Copy the collaboration–related methods of classes A and B to the `ABRelationship`.

- To avoid naming conflicts, prefix an `a_` to methods of class A and a `b_` to methods of class B.
- Remove the collaboration field assignments in the relationship code.
- If the collaboration–related method is now empty, remove it from the relationship code.

**6.** Remove the collaboration–related methods from the class–based implementation of classes A and B

**7.** Remove collaboration field assignments from the collaboration–related constructors of classes A and B, if applicable.

- If a collaboration field was previously assigned a value passed to the constructor, update the constructor signature by removing the value from the signature.

**8.** Update the collaboration–related control code:

- For all calls of a collaboration–related method in class A: replace the object of type A with the relationship instance and prefix the collaboration–related method with an `a_`. Do the same for class B.
- For all instantiations with a collaboration–related constructor: update the constructor call with the new signature.

**9.** Move all methods of class A and B with code on collaboration field elements to the relationship, if applicable.

- To avoid naming conflicts, prefix an `a_` to methods of class A and a `b_` to methods of class B.
- Update any calls on these methods by replacing the objects of type A or B respectively with the relationship instance.

### 3.4.4 Binary collaboration controlled by direct collaboration object

Situation: The class–based implementation has been identified as binary collaboration between classes A and B, that is controlled by a direct collaboration object. The following steps describe how the class–based implementation is transformed into a relationship–based implementation. In particular, the classes A and B are not affected by the changes.

**1.** Create a new relationship `ABRelationship`. Substitute A and B for the class names.
`relationship ABRelationship { }`

**2.** Add a participant clause
`participants(A a, B b);`

**3.** Add the invariant clauses to the relationship, depending on the collaboration category and constraints, for example:
`surjective(a, b);`

**4.** Copy all methods of the direct collaboration class to the `ABRelationship`.

- Remove the collaboration field assignments in the relationship code.
- If the collaboration–related method is now empty, remove it from the relationship code.

**5.** Copy the constructor code from the direct collaboration class to the `ABRelationship`.

- Remove the collaboration field assignments in the relationship code.

**6.** Remove the direct collaboration class.

**7.** Update the collaboration–related control code:

- For all calls on the direct collaboration object: replace with the `ABRelationship` instance.

### 3.4.5 Binary collaboration controlled by indirect collaboration object

Situation: The class–based implementation has been identified as binary collaboration between classes A and B, that is controlled by an indirect collaboration object. The following steps describe how the class–based implementation is transformed into a relationship–based implementation. In particular, the classes A and B are not affected by the changes.

**1.** Create a new relationship `ABRelationship`. Substitute A and B for the class names.
`relationship ABRelationship { }`

**2.** Add a participant clause
`participants(A a, B b);`

**3.** Add the invariant clauses to the relationship, depending on the collaboration category and constraints, for example:
`surjective(a, b);`

**4.** Copy all methods of the collaboration class `ABCol` (as defined in 3.3.2 to the `ABRelationship`.

- To avoid naming conflicts, prefix an `col_` to the methods
- Remove the collaboration field assignments in the relationship code.
- If the collaboration–related method is now empty, remove it from the relationship code.

**5.** Copy all methods of the collaboration class `ABColCollection` (as defined in 3.3.2 to the `ABRelationship`.

- To avoid naming conflicts, prefix an `coll_` to the methods

- Remove the collaboration field assignments in the relationship code.
- If the collaboration–related method is now empty, remove it from the relationship code.

**6.** Copy the constructor code from the collaboration class `ABCol` to the `ABRelationship`.

- Remove the collaboration field assignments in the relationship code.

**7.** Remove the `ABCol` collaboration class.

**8.** Remove the `ABColCollection` collaboration class

**9.** Update the collaboration–related control code:

- For all calls on the `ABCol` object: replace with the `ABRelationship` instance and prefix the method calls with a `col_`
- For all calls on the `ABColCollection` object: replace with the `ABRelationship` instance and prefix the method calls with a `coll_`.

### 3.4.6 Ternary collaboration controlled by indirect collaboration object

Situation: The class–based implementation has been identified as binary collaboration between classes A, B and C that is controlled by an indirect collaboration object. The following steps describe how the class–based implementation is transformed into a relationship–based implementation. In particular, the classes A, B and C are not affected by the changes.

**1.** Create a new relationship `ABCRelationship`. Substitute A, B and C for the class names.

```
relationship ABCRelationship { }
```

**2.** Add two participant clauses

```
 participants(A a, B b);
 participants(B b, C c);
```

**3.** Copy all methods of the collaboration class `ABCCol` (as defined in 3.3.2 to the `ABCRelationship`.

- To avoid naming conflicts, prefix an `col_` to the methods
- Remove the collaboration field assignments in the relationship code.
- If the collaboration–related method is now empty, remove it from the relationship code.

**4.** Copy all methods of the collaboration class `ABCColCollection` (as defined in 3.3.2 to the `ABCRelationship`.

- To avoid naming conflicts, prefix an `coll_` to the methods
- Remove the collaboration field assignments in the relationship code.
- If the collaboration–related method is now empty, remove it from the relationship code.

**5.** Copy the constructor code from the collaboration class `ABCCol` to the `ABCRelationship`.

- Remove the collaboration field assignments in the relationship code.

**6.** Remove the `ABCCol` collaboration class.

**7.** Remove the `ABCColCollection` collaboration class

**8.** Update the collaboration–related control code:

- For all calls on the `ABCCol` object: replace with the `ABCRelationship` instance and prefix the method calls with a `col_`
- For all calls on the `ABCColCollection` object: replace with the `ABCRelationship` instance and prefix the method calls with a `coll_`.

### 3.4.7 Supertyping

The transformation of a class to a relationship–based implementations affects the inheritance hierarchy as well. In current software engineering practices (source: Interview with H. Wegener), inheritance hierarchies are kept as flat as possible. Instead, interfaces are used to model the different roles that classes can play. The relationship framework [5] provides the mechanism of *member interposition*. We believe that inheritance hierarchies are not longer necessary with this approach: instead of inheriting fields and methods from a supertype and adding a subtype for each role, we only need to keep the supertype class. In each relationship, the fields and methods of a subtype are then added, using member interposition. For an example of this idea, please refer to the running example "Students, Lecturers and Courses" in section 3.2.7.

1. Keep the supertype of each inheritance hierarchy only
2. For each collaboration, remember the subtype participating
3. In each new relationship, add the fields of the participating subtype and the methods using member interposition

# Chapter 4

# The Relationship Detector

We now apply the results of how to identify and transform collaborations to build a tool, the *relationship detector*. The relationship detector extracts the elements in a class–based implementation that constitute a collaboration. It also shows how the class–based implementation would be affected if it is transformed into a relationship–based implementation.

## 4.1 Overview

The relationship detector performs a statical analysis, taking a java class file and extracting hidden object collaborations, as shown in figure 4.1. In a first step, the relationship detector searches for the collaboration elements that constitute a collaboration in a class–based implementation. We then apply an algorithm, the *collaboration decision algorithm*, to identify the collaboration implementation option. For each of these options, we introduce a collaboration class, that provides access and storage for the relevant collaboration elements. Finally, we provide the user with an output of the program analysed, showing which parts of the code are related to the hidden collaboration.

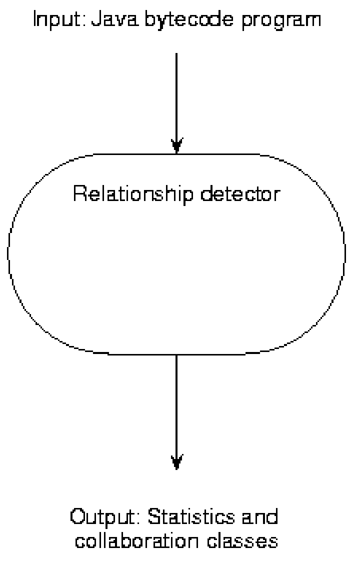
Input: Java bytecode program
Relationship detector
Output: Statistics and collaboration classes

Figure 4.1: The relationship detector

## 4.2 Third-party Tools

The relationship detector uses the ASM Bytecode Framework [10] to parse the bytecode of the java input files. We first give a brief introduction to the Java bytecode terminology that is relevant for our work, as introduced by Lindholm et al. [22]. Then, we summarise the main concepts of the ASM framework and show how we use the concepts in our approach.

### 4.2.1 Overview of the Java Virtual Machine (JVM) Specification

We use the following terminology and concepts as defined in [22]:

- **Class file format**
  In Java, source code (`.java` files) is compiled into bytecode, which is stored in the form of class files (`.class`). Each class file contains one Java type, either a class or an interface. The class file format is important to the relationship detector, as we analyse the bytecode using the ASM framework.

  A class file contains a single `ClassFile` format, each consisting of a sequence of bytes.

```
ClassFile {
       unsigned 4 byte magic;
       unsigned 2 byte minor_version;
       unsigned 2 byte major_version;
       unsigned 2 byte constant_pool_count;
       cp_info constant_pool[constant_pool_count-1];
       unsigned 2 byte access_flags;
       unsigned 2 byte this_class;
       unsigned 2 byte super_class;
       unsigned 2 byte interfaces_count;
       unsigned 2 byte interfaces[interfaces_count];
       unsigned 2 byte fields_count;
       field_info fields[fields_count];
       unsigned 2 byte methods_count;
       method_info methods[methods_count];
       unsigned 2 byte attributes_count;
       attribute_info attributes[attributes_count];
   }
```

  Listing 4.1: The java classfile format

  - `constant_pool[]`
    Is a table of variable-length structures that represent constants, such as string constants, class names and field names. These are referred to within the classFile structure. The number of entries in the `constant_pool` are stored in `constant_pool_count`.
  - `constant_pool_count`
    Gives the number of entries in the constant pool table of the class file.
  - `super_class`
    Defines the super class, that must be either a valid index into the constant pool table or zero. If it is zero, then this class file must represent the class java.lang.Object (the only class or interface without a superclass).
  - `interfaces[]`
    Each value must be a valid index into the constant pool table.
  - `interfaces_count`
    Gives the number of direct superinterfaces of the class or interface type.

- `fields[]`
  The fields are the variables of a class type. The fields structure does not include items representing fields that are inherited from superclasses or superinterfaces. A field structure has an access modifier, a name, descriptor, signature and value.
- `fields_count`
  Gives the number of all fields (class variables and instance variables) declared by this class or interface type.
- `methods[]`
  Includes all methods - both instance methods and static methods (for classes). It does not include, however, methods inherited from superclasses or superinterfaces. A method contains a name and descriptor (among other elements).
- `methods_count`
  Gives the number of methods.
- `attributes[]`
  Contains all attributes of the class. The *source file attribute* contains the source code for a single Java method. The *code attribute* contains the instructions and auxiliary information for a single Java method, instance initialisation method or class or interface initialization method. The *exceptions attribute* indicates which checked exceptions a method may throw. Attributes in this case do not include class fields.
- `attributes count`
  Gives the number of attributes of the class.

### 4.2.2 Bytecode Analysis Tools

Among the other bytecode analysis tools, such as BCEL[30] and SOOT[31], we chose ASM for its ease of use. ASM applies a visitor-based approach to generate bytecode and transform existing classes. It also allows a user to pull out significant details about existing classes. The ASM framework hides bytecode complexity by allowing users to avoid dealing directly with constants pools and offsets. ASM can both read and transform Java bytecode. The ASM tutorial [29] gives the following (4.2) schematic overview of the class file:

```
[1]-------------------------------------------+
 | Header and Constant Stack                  |
 +--------------------------------------------+
 | [*] Class Attributes                       |
[2]------------+------------------------------+
 | [*] Fields  | Field Name, Descriptor, etc  |
 |             +------------------------------+
 |             | [*] Field Attributes         |
[3]------------+------------------------------+
 | [*] Methods | Method Name, Descriptor, etc |
 |             +------------------------------|
 |             | Method max stack and locals  |
 |             |------------------------------|
 |             | [*] Method Code table        |
 |             |------------------------------|
 |             | [*] Method Exception table   |
 |             |------------------------------|
 |             | [*] Method Code Attributes   |
 |             +------------------------------|
 |             | [*] Method Attributes        |
 +-------------+------------------------------+
```

Figure 4.2: High-level diagram of the class file format

## 4.3 Collaboration Detection Algorithm

In this section we devise the algorithm that the relationship detector employs to identify collaborations. The goal is to identify the collaboration elements and the collaboration options, as introduced in section 3.3.

The elements in the bytecode that are most closely connected to the collaboration are reference fields - the collaboration fields. They provide the link between the classes that is necessary to define the collaboration and are therefore the key to identification. Not all reference types are relevant to the collaboration detection. Certain reference that are not directly connected to the application logic are excludedd, for instance graphical user interface classes such as buttons. The relevant and non-relevant references are defined by the relationship detector configuration. In the following sections we refer to the relevant ones as *relevant references.*

The algorithm assumes complete knowledge of the other collaboration implementation options. The implementation however has to operate in various phases to acquire the knowledge that is necessary. Some elements of the algorithm can be identified locally, i.e. from each class without knowledge of the other collaboration implementation options. Other elements require a global view and therefore are identified in a later phase of the algorithm.

Figure 4.3 shows how the collaboration detection algorithm can determine the different collaboration options, starting with the relevant references of a class.

The algorithm starts with a class containing relevant references. These references are the collaboration fields, and can be either single or multi–valued. The goal of the algorithm is to determine the collaboration implementation option. A class may participate in more than one collaboration - for instance, a `class A` may be involved in a single–valued collaboration with `class B` and a multi–valued collaboration with `class C`.

The following paragraphs reference the labels that are defined in figure 4.3.

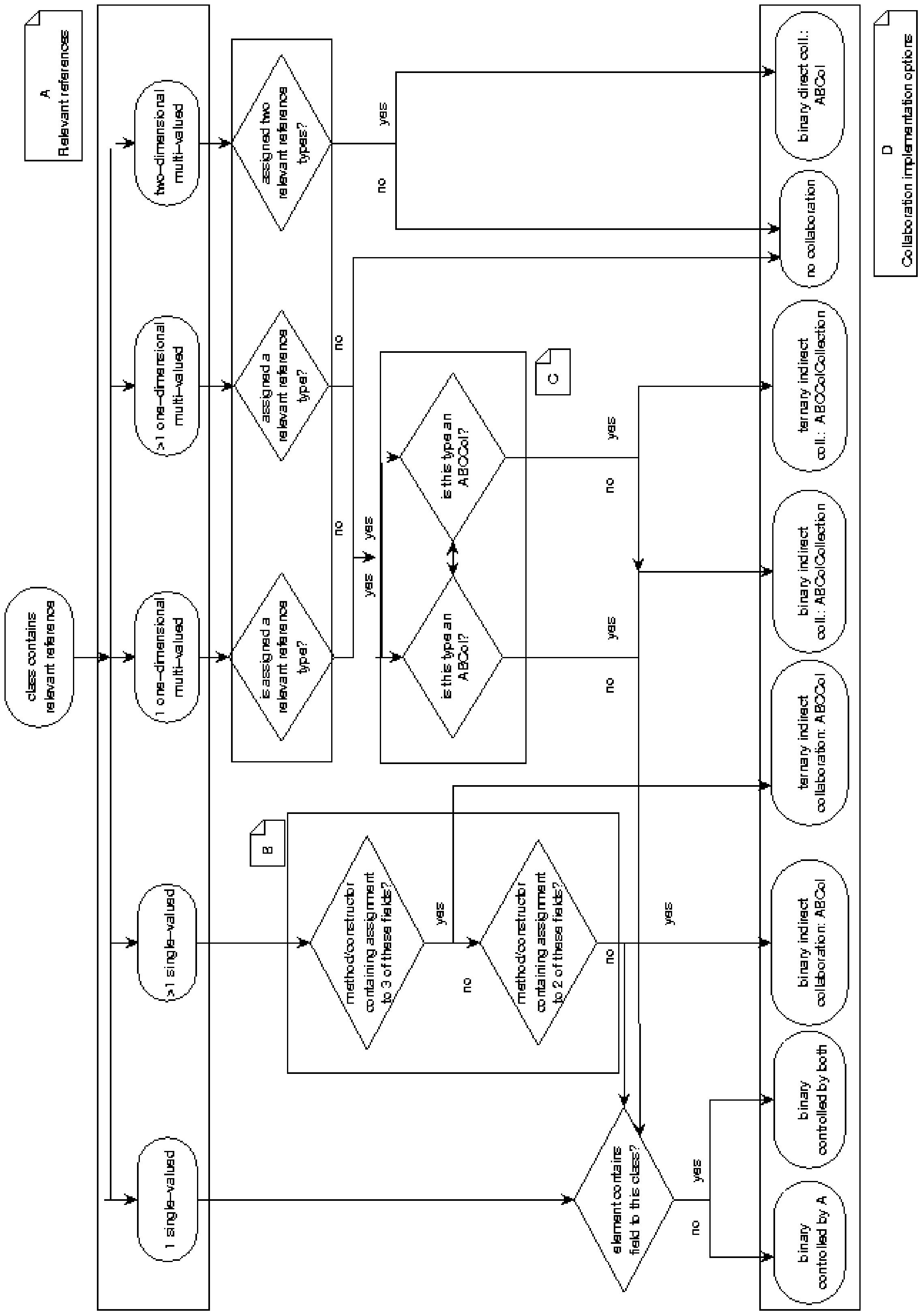
A
Relevant references
class contains relevant reference
two–dimensional multi–valued
>1 one–dimensional multi–valued
1 one–dimensional multi–valued
>1 single–valued
1 single–valued
assigned two relevant reference types?
assigned a relevant reference type?
is assigned a relevant reference type?
yes
no
is this type an ABOCol?
is this type an ABCol?
C
B
method/constructor containing assignment to 3 of these fields?
method/constructor containing assignment to 2 of these fields?
element contains field to this class?
binary direct coll.: ABCol
no collaboration
ternary indirect coll.: ABOColCollection
binary indirect coll.: ABColCollection
ternary indirect collaboration: ABOCol
binary indirect collaboration: ABCol
binary controlled by both
binary controlled by A
D
Collaboration implementation options

Figure 4.3: The collaboration detection algorithm

$\boxed{\text{B}}$ As a class may participate in more than one collaboration, we need a way to separate the following two options:

1. class A participating in two (separate) collaborations with B and C, i.e. with collaboration fields B and C. This collaboration implementation option is shown on the far left of the diagram.
2. an indirect collaboration class for the collaboration between B and C, i.e. with collaboration fields B and C. This collaboration implementation option is shown as *binary indirect collaboration: ABCol*.

In case 1, the collaborations are separate. We therefore assume that participants are not added to the collaboration at the same time, i.e. that collaboration fields are not assigned values at the same time. For case 2, the opposite is true: the participants of a collaboration must be added (i.e. assigned) at the same time. A direct or indirect collaboration class associates two (for binary collaborations) or three (for ternary collaborations) participant classes. It would make no sense to add these participant classes separately, e.g. to add all classes `A` in one method and all classes `B` in another. It would not be clear which participants are associated with each other. So, for our algorithm, we separate case 1 from case 2 by applying the following rule: if there is a collaboration–related method or constructor that contains assignments to all collaboration fields, then the class belongs to an indirect collaboration, i.e. is ABCol or ABCCol respectively. If not, then the collaborations are assumed to be separate. We therefore need to keep track not only of the references but also the collaboration–related methods and constructors.

$\boxed{\text{A}}$ For each type and number of collaboration fields, there is a series of checks that identifies the corresponding implementation option:

- **one single–valued collaboration field**
  A class that contains a single–valued collaboration field is either participating in a binary collaboration controlled by the class itself, or in a binary collaboration controlled by both participants. Let `class A` denote the class containing the reference, and `class B` the class that is referenced. To determine which collaboration implementation option is the case, `class B` needs to be analysed. If it contains a reference to `class A`, then the collaboration is controlled by both `A` and `B`. If not, then the collaboration is controlled by `A` only. Figure 4.3 shows the corresponding path in the collaboration detection algorithm.

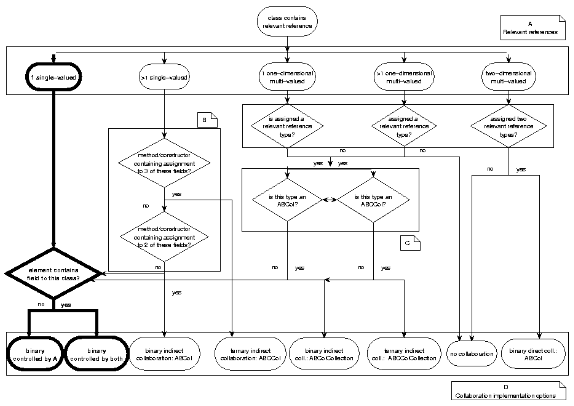
class contains relevant reference
A Relevant references
1 single–valued
>1 single–valued
1 one–dimensional multi–valued
>1 one–dimensional multi–valued
two–dimensional multi–valued
B
method/constructor containing assignment to 3 of these fields?
yes
no
method/constructor containing assignment to 2 of these fields?
no
yes
is assigned a relevant reference type?
assigned a relevant reference type?
assigned two relevant reference types?
no
no
yes
yes
no
yes
is this type an ABCol?
is this type an ABCCol?
C
no
yes
no
yes
element contains field to this class?
no
yes
binary controlled by A
binary controlled by both
binary indirect collaboration: ABCol
ternary indirect collaboration: ABCCol
binary indirect coll.: ABColCollection
ternary indirect coll.: ABCColCollection
no collaboration
binary direct coll.: ABCol
D Collaboration implementation options

Figure 4.4: Collaboration detection for a single–valued field

- **more than one single–valued collaboration field**
  A class containing more than one single–valued collaboration fields could be participating in more kinds of collaboration options. Specifically, the algorithm needs to differentiate

between a class participating in multiple collaborations and class that is a direct or indirect collaboration class. Again, we follow the assumption that participants to direct or indirect collaboration classes are added to the collaboration at the same time. In the following, we separate the ternary from the binary collaboration options:
If the class has a method or constructor that assigns three collaboration fields, then the class is the indirect collaboration object `ABCCol` of a ternary collaboration.
If the class has a method or constructor that assigns two of these collaboration fields, then the class is the indirect collaboration object `ABCol` of a binary collaboration.
If none of the two above apply, then handle each single–valued collaboration field separately, i.e. follow the path for the one single–valued collaboration field. The same is valid for any single–valued collaboration fields that are not included in one of the cases above - for instance, if there are four single–valued collaboration fields and a method/constructor assigning three of them, then the fourth reference is handled separately. Figure 4.3 shows the corresponding path in the collaboration detection algorithm.

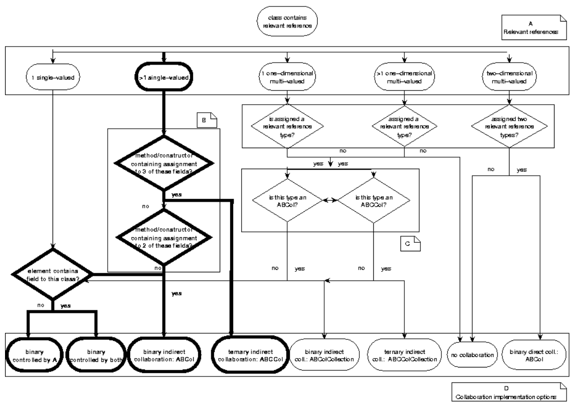
class contains relevant reference
A Relevant references
1 single–valued
>1 single–valued
1 one–dimensional multi–valued
>1 one–dimensional multi–valued
two–dimensional multi–valued
B
method/constructor containing assignment to 3 of these fields?
yes
no
method/constructor containing assignment to 2 of these fields?
no
yes
element contains field to this class?
no
yes
is assigned a relevant reference type?
assigned a relevant reference type?
assigned two relevant reference types?
no
no
yes
yes
no
yes
is this type an ABCol?
is this type an ABCCol?
C
no
yes
no
yes
binary controlled by A
binary controlled by both
binary indirect collaboration: ABCol
ternary indirect collaboration: ABCCol
binary indirect coll.: ABColCollection
ternary indirect coll.: ABCColCollection
no collaboration
binary direct coll.: ABCol
D Collaboration implementation options

Figure 4.5: Collaboration detection for multiple single–valued fields

- **one multi–valued collaboration field**
In this case, the algorithm checks whether the multi–valued collaboration field is assigned relevant references. If not, then the reference is not collaboration related. If yes, then the type of the reference is checked:
$\boxed{\text{C}}$ If the type is `ABCol`, then the class is the corresponding `ABColCollection` for a binary indirection collaboration.
If the type is `ABCCol`, then the class is the corresponding `ABCColCollection` for a ternary indirect collaboration.
If none of the two above apply, then check the referenced class. Again, let `class A` denote the class containing the reference, and `class B` the class that is referenced. If `B` contains a reference to `A`, then the binary collaboration is controlled by both `A` and `B`. If not, then the collaboration is controlled by `A` only. Figure 4.3 shows the corresponding path in the collaboration detection algorithm.

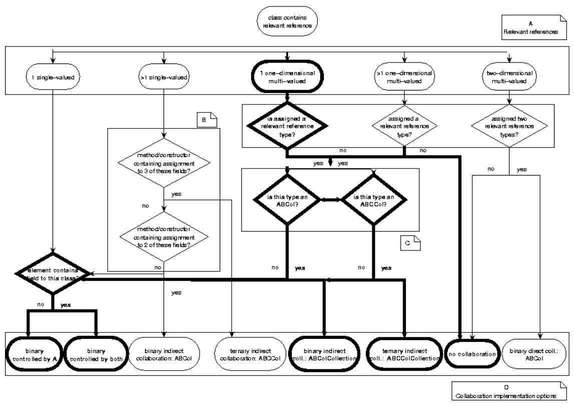
class contains relevant reference
A Relevant references
1 single-valued
>1 single-valued
1 one-dimensional multi-valued
>1 one-dimensional multi-valued
two-dimensional multi-valued
B
is assigned a relevant reference type?
assigned a relevant reference type?
assigned two relevant reference types?
method/constructor containing assignment to 3 of these fields?
method/constructor containing assignment to 2 of these fields?
is this type an ABCol?
is this type an ABCCol?
C
element contains field to this class?
binary controlled by A
binary controlled by both
binary indirect collaboration: ABCol
ternary indirect collaboration: ABCCol
binary indirect coll.: ABColCollection
ternary indirect coll.: ABCColCollection
no collaboration
binary direct coll.: ABCol
D Collaboration implementation options

Figure 4.6: Collaboration detection for a one–dimensional multi–valued field

- **more than one multi–valued collaboration field**
  Again, the algorithm checks whether the multi–valued collaboration fields are assigned relevant references. If not, then the reference is not collaboration related. If yes, then check the referenced class:
  $\boxed{\text{C}}$ Let `class A` denote the class containing the reference, and `class B` the class that is referenced. If `B` contains a reference to `A`, then the binary collaboration is controlled by both `A` and `B`. If not, then the collaboration is controlled by `A` only. Figure 4.3 shows the corresponding path in the collaboration detection algorithm.

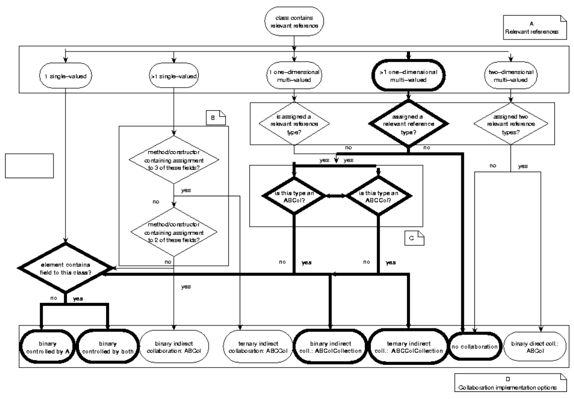
class contains relevant reference
A Relevant references
1 single-valued
>1 single-valued
1 one-dimensional multi-valued
>1 one-dimensional multi-valued
two-dimensional multi-valued
B
is assigned a relevant reference type?
assigned a relevant reference type?
assigned two relevant reference types?
method/constructor containing assignment to 3 of these fields?
method/constructor containing assignment to 2 of these fields?
is this type an ABCol?
is this type an ABCCol?
C
element contains field to this class?
binary controlled by A
binary controlled by both
binary indirect collaboration: ABCol
ternary indirect collaboration: ABCCol
binary indirect coll.: ABColCollection
ternary indirect coll.: ABCColCollection
no collaboration
binary direct coll.: ABCol
D Collaboration implementation options

Figure 4.7: Collaboration detection for multiple multi–valued fields

- **two–dimensional multi–valued collaboration field**
  Each two dimensional multi–valued field is handled separately. The algorithm checks if the field is assigned two relevant references (for instance, a relevant key and value for a `java.util.HashMap`).
  If yes, then the class is a direct collaboration class participating in a binary direct collaboration.
  If not, then the class is not collaboration–related.
  Figure 4.3 shows the corresponding path in the collaboration detection algorithm.

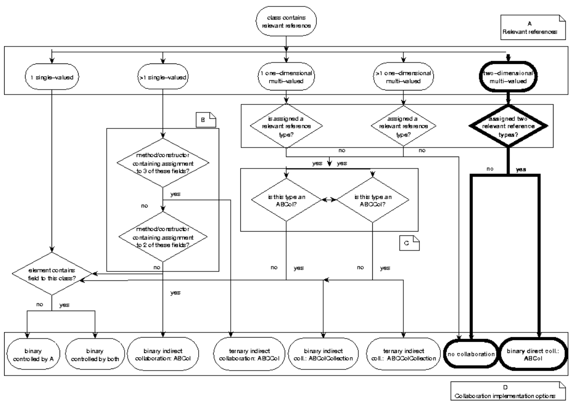
class contains relevant reference
A Relevant references
1 single–valued
>1 single–valued
1 one–dimensional multi–valued
>1 one–dimensional multi–valued
two–dimensional multi–valued
B
is assigned a relevant reference type?
assigned a relevant reference type?
assigned two relevant reference types?
method/constructor containing assignment to 3 of these fields?
yes
no
method/constructor containing assignment to 2 of these fields?
no
yes
no
no
yes
yes
is this type an ABCol?
is this type an ABCCol?
C
no
yes
no
yes
no
yes
element contains field to this class?
no
yes
binary controlled by A
binary controlled by both
binary indirect collaboration: ABCol
ternary indirect collaboration: ABCCol
binary indirect coll.: ABColCollection
ternary indirect coll.: ABCColCollection
no collaboration
binary direct coll.: ABCol
D Collaboration implementation options

Figure 4.8: Collaboration detection for two–dimensional multi–valued fields

For the collaboration detection algorithm the following collaboration elements are relevant:

- Collaboration fields
- Assignments to collaboration fields
  In class–based implementations, a participant is added to a collaboration by assigning the corresponding collaboration field.
- Collaboration-related methods and constructors
  The methods and constructors are relevant to provide the differentiation between multiple separate collaborations and the indirect/direct collaboration classes.

### 4.3.1 Invariant Detection

The invariant detection uses the concepts defined in section 3.3.2. The algorithm needs to recognise the patterns that constitute a structural or value–based invariant.

#### Structural Invariants

Some structural invariants are defined by the following code elements in class–based implementations, others are implicitly defined by the type of the collaboration field. These patterns occur before the collaboration field is assigned, i.e. in a collaboration–related method or constructor. Figure 4.3.1 shows the structural invariants.

| | |
|---|---|
| **total** | The collaboration field is assigned a value in a collaboration–related constructor. |
| **partial** | Not enforced, is the default value if totality is not enforced. |
| **surjection** | For single–valued collaboration fields: check `if(collaboration_field <> null){...}` For multi–valued collaboration fields: check for each collaboration element: `if(element <> null){...}` |
| **injection** | Check the class that is referenced by the collaboration field For a class with a single–valued collaboration field: check `if(collaboration_field <> null{...})` For a class with a multi–valued collaboration field: check for each collaboration element: `if(element <> null{...})` |
| **bijection** | If both injection and surjection apply |
| **symmetric** | For single–valued collaboration fields: check `if(collaboration_field.collaboration_field == this{...})` For multi–valued collaboration fields: check for each collaboration element: `if(element.collaboration_field == this{...})` |
| **antisymmetric** | For single–valued collaboration fields: check `if(collaboration_field.collaboration_field <> this{...})` For multi–valued collaboration fields: check for each collaboration element: `if(element.collaboration_field <> this{...})` |
| **reflexive** | For single–valued collaboration fields: check `if(collaboration_field == this{...})` For multi–valued collaboration fields: check for each collaboration element: `if(element == this{...})` |
| **irreflexive** | For single–valued collaboration fields: check `if(collaboration_field <> this{...})` For multi–valued collaboration fields: check for each collaboration element: `if(element <> this{...})` |

Table 4.1: Structural invariants

### Value-based Invariants

Value-based invariants may occur in many different patterns. The common characteristic they share is that they occur as as if statements in a method that assigns a collaboration field. Value-based invariants are usually checked in class–based implementations before a participant is added. Depending on the collaboration implementation option, the value–based condition is checked at a different location in the code - wherever participants are added to a collaboration. The following code patterns point to a value–based invariant:

- **Binary collaboration controlled by class A**
  The controlling `class A` contains the following code pattern. The value–based condition contained in the `if` clause is an example, as are the method name and the argument type. The pattern shows the assignment for a single–valued collaboration field. For a multi–valued collaboration field, the assignment is replaced by the corresponding method call, e.g. `list.add(b)` for a `List` collection item.

```
class A{
    public void addToCollaboration(B b) {
          if(observedValue == targetValue) {
                ...
                collaboration field = b
                ...
          }
    }
}
```

Listing 4.2: Invariant pattern: binary controlled by A

- **Binary collaboration controlled by class B**
  The controlling `class B` contains the following code pattern. The value–based condition contained in the `if` clause is an example, as are the method name and the argument type. The pattern shows the assignment for a single–valued collaboration field. For a multi–valued collaboration field, the assignment is replaced by the corresponding method call, e.g. `list.add(a)` for a `List` collection item.

```
class B{
    public void addToCollaboration(A a) {
          if(lowerTargetRange <= observedValue <= upperTargetRange) {
                ...
                collaboration field = a
                ...
          }
    }
}
```

Listing 4.3: Invariant pattern: binary controlled by B

- **Binary collaboration controlled by both class A and B**
  Both classes `A` and `B` contain the following code pattern. Depending on the class–based implementation, some of the invariants may be redundant, once the invariants of both classes are combined (for instance, both classes A and B may check the same conditions). Again, the value–based conditions are examples, as are the method names and the parameter types and the assignment applies to single–valued collection fields.

```
class A{
    void addToCollaboration(B b) {
           if(observedValue == targetValue) {
                  ...
                  collaboration field = b
                  ...
           }
    }
}

class B{
    public void addToCollaboration(A a) {!
           if (observedValue <= targetValue)) {
                  ...
                  collaboration field = a
                  ...
           }
    }
}
```

Listing 4.4: Invariant pattern: binary controlled by A and B

- **Binary collaboration controlled by direct collaboration object**
  The direct collaboration object (named `ABCol` in the previous section) contains the following code pattern. Classes `A` and `B` do not contain collaboration–related code. `2dimCollection` refers to a two-dimensional collection item that associates instances of `A` and `B`.

```
class ABCol{
    public void addToCollaboration(A a, B b) {
           if(observedValue >= minimumValue) {
                  ...
                  2dimCollection.add(a,b)
                  ...
           }
    }
}
```

Listing 4.5: Invariant pattern: binary direct collaboration

- **Binary collaboration controlled by indirect collaboration object**
  In the case where the collaboration is controlled by an indirect collaboration object, there are two options where value–based invariants may be checked:

  - **In the `ABCol` object**
    In this case, the value–based invariant is checked after creating the `ABCol` instance for the collaboration and before adding the two participants to the two collaboration fields:

```
class ABCol{
    public void addToCollaboration(A a, B b) {
           if(lowerTargetRange <= observedValue <= upperTargetRange) {
                  ...
                  collaboration field A = a
                  collaboration field B = b
                  ...
```

```
            }
        }
    }
```

Listing 4.6: Invariant pattern: binary indirect collaboration

- **In the `ABColCollection` object**
  The other possibility is to check the value–based invariants after creating the `ABCol` instance and adding the two collaboration fields `a` and `b`. The following code pattern points to this version:

```
  class ABColCollection{
       public void addToCollaboration(ABCol abCol) {
          if((lowerTargetRange <= abCol.getObservedValueA() <= upperTargetRange)
          && (lowerTargetRange <= abCol.getObservedValueB() <= upperTargetRange)){
                 ...
                 abColCollectionItem.add(abCol)
                 ...
          }
       }
  }
```

Listing 4.7: Invariant pattern: binary indirect collaboration

- **Ternary collaboration controlled by indirect collaboration object**
  As with the binary collaboration controlled by an indirect collaboration object, there are two possibilities where to check value–based invariants:

  - **In the `ABCCol` object**
    In this case, the value–based invariant is checked after creating the `ABCCol` instance for the collaboration and before adding the three participants to the three collaboration fields:

```
   class ABCCol{
       public void addToCollaboration(A a, B b, C c) {
             if(observedValue == targetValue) {
                    ...
                    collaboration field A = a
                    collaboration field B = b
                    collaboration field C = c
                    ...
             }
       }
   }
```

Listing 4.8: Invariant pattern: ternary indirect collaboration

  - **In the `ABCColCollection` object**
    The other possibility is to check the value–based invariants after creating the `ABCCol` instance and adding the three collaboration fields A, B and C. The following code pattern points to this version:

```
   class ABColCollection{
       public void addToCollaboration(ABCol abcCol) {
          if((lowerTargetRange <= abCol.getObservedValueA() <= upperTargetRange)
                && (lowerTargetRange <= abCol.getObservedValueB() <= upperTargetRange)
                && (lowerTargetRange <= abCol.getObservedValueC() <= upperTargetRange)){
                       ...
                       abcColCollectionItem.add(abcCol)!}}
                       ...
                }
           }
      }
```

Listing 4.9: Invariant pattern: ternary indirect collaboration

Depending on the structure of the if-clauses that contain the patterns above, the constraints may be connected in a logical AND operator or in a logical OR operator.

| | |
|---|---|
| `AND` | If the if-clauses containing a certain collaboration field assignment are nested (i.e. one is contained within the other), then the value–based invariant results from the `AND` relation of both if-conditions. This property is recursive. |
| `OR` | If the if-clauses containing a certain collaboration field assignment are on the same level (i.e. `if(..){...} else if(...) {...} else {}`), then the value–based invariant results from the `OR` relation of their if-conditions. |

## 4.4 Design

In the following section, we provide an overview of the design of the relationship detector. First, we show the components and their contribution to the relationship detector. Then we describe the problems and implementation decisions we faced upon designing the tool.

### 4.4.1 Components

Figure 4.4.1 shows the different packages that constitute the relationship detector. Each package groups the classes and interfaces that provide a particular service, such as the statistics service in the `statistics` package. In the following we will therefore refer to these packages also as *modules*. The implementation section 4.5 describes the relationship detector in more detail.

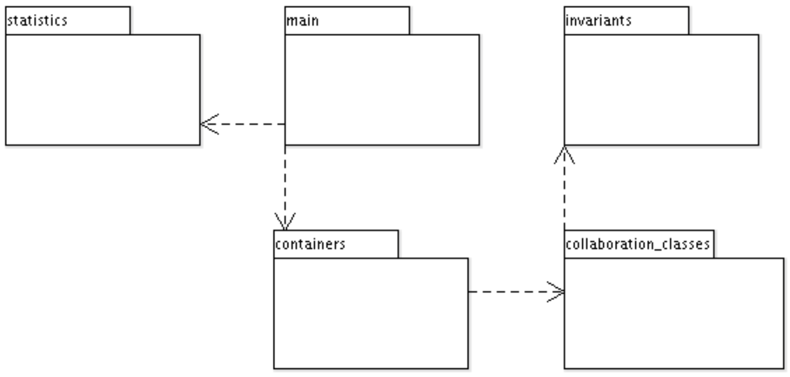

Figure 4.9: Package level view of the relationship detector

#### main

The main package contains the classes that form the core elements of the relationship detector. This module performs the relationship detection and controls the other modules. Figure 4.4.1 shows the main package.

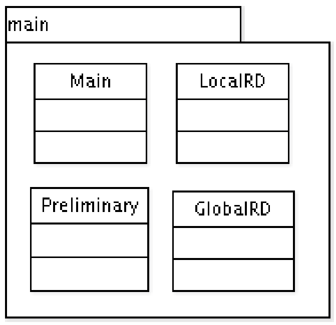

Figure 4.10: main package

#### containers

The containers package includes the data structure classes for the relationship detector. The algorithms in the main module use these containers to store intermediate and final results. The module provides storage for classes, field, methods and inheritance structures. Figure 4.4.1 shows the container package.

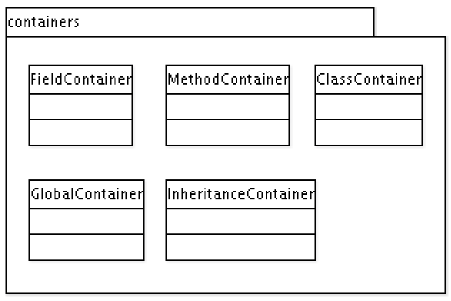

Figure 4.11: containers package

### collaboration_classes

This module contains the the collaboration classes: for each collaboration implementation option, there is a corresponding collaboration class - for instance, `BinaryIndirectCollaboration`. Whenever the collaboration algorithm detects a collaboration, it instantiates a new collaboration class and stores the relevant information in it. The collaboration classes share a common interface, to provide uniform access to the collaboration classes - without having to know the implementation option. Figure 4.4.1 shows the collaboration_classes package.

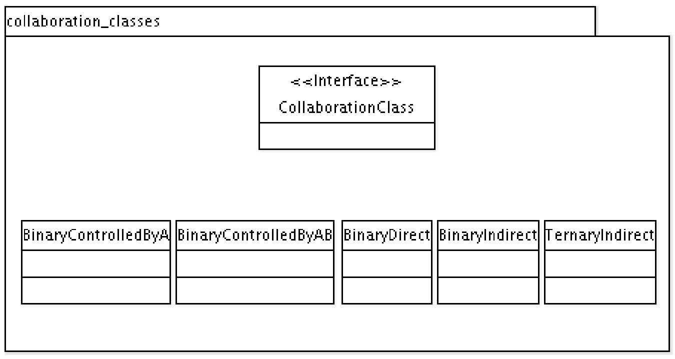

Figure 4.12: collaboration_classes package

### statistics

The classes in this module gather and store statistical information about the Relationship Detection. For instance, information such as the number of methods that are collaboration related. The statistics also show how a program would be affected by transforming it into a relationship–based implementation - for instance, how many instructions would become obsolete. Figure 4.4.1 shows the statistics package.

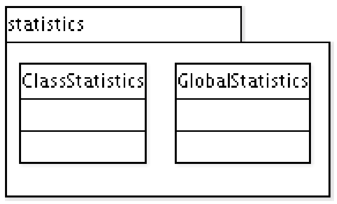

Figure 4.13: statistics package

#### invariants

This package contains the data structures for the different types of invariants - structural and value–based. Whenever an invariant is detected, the algorithm adds the information to the invariant data structures. The collaboration classes have a reference to their invariants. Figure 4.4.1 shows the invariants package.

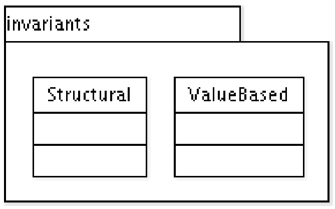

Figure 4.14: invariants package

### 4.4.2 Problems and Implementation Decisions

In the following sections we describe the problems we encountered in the relationship detector implementation. Some of these problems are inherent to static bytecode analysis, others are due to limitations within the bytecode analysis framework. We then present our solutions to the problems, grouped into strategies and heuristics. Strategies work correctly for the cases where they apply, but are not able to cover the whole problem domain (i.e. they only offer a solution in some cases). Heuristics are approximations that are used whenever a straightforward solution or strategy is not available. We explain why our strategies and heuristics apply and show where the limitations lie.

#### Problem: Missing Instance Name Information

The relationship detector performs a static analysis of the java bytecode. This means that no dynamic information is available, such as the actual values that are passed. For instance, we can derive that an integer `i` is increased by one: `i++`, but not the actual value of `i` before and after the instruction. In certain cases, the collaboration detection algorithm needs the names (and with them the possibility to derive the type) of elements such as fields, methods, variables and arguments. Some bytecode instructions provide such information, as for example a `GETFIELD` instruction: the owner type and the field name are available. Others, as for example an `INVOKEVIRTUAL` instruction provide neither the name of the target reference nor the name of the arguments that are passed to it. For this purpose, we need to derive the missing names from the context of the previous instructions.

#### Solution

For this purpose, we introduce an additional data structure: the *operand expression stack*. When a bytecode instruction is executed at runtime, it removes a certain number of operands from the stack, and pushes the results back to the stack. These changes to the stack happen in a predefined way for each bytecode instruction - as defined in the virtual machine specification [22]. For instance, an `INVOKEVIRTUAL` bytecode instruction (a method instruction) pops the target reference and the arguments from the stack and pushes nothing back. The appendix gives a list of the bytecode instructions covered and shows the operand stack rules. For each of the bytecode instructions, the stack follows the same rules as for the regular operand stack: elements are popped from the stack and new elements are pushed to the stack, according to these rules. The difference is that while the operand stack is a runtime concept and contains *values*, the operand expression stack is static and contains *expressions*. As an example, lets have a look at an int instruction: `IADD`. The operand stack is an element of the JVM specification, while the expression stack is part of our implementation. The rules for the operand stack are the following:

**1.** pop integer values `value1` and `value2` from the operand stack

**2.** compute the addition `result = value1 + value2`

**3.** push the result to the operand stack.

Figure 4.4.2 shows the operand stack before and after an example of a `IADD` operation.

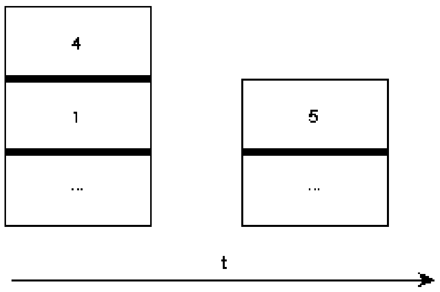

Figure 4.15: operand stack before (left) and after (right) the IADD instruction

For the operand expression stack, the following rules apply:

**1.** pop integer expressions `exp1` and `exp2` from the expression stack

**2.** generate the addition expression `exp1 + exp2`.

**3.** push the addition expression to the expression stack.

Figure 4.4.2 shows the operand expression stack before and after an example of a `IADD` operation.

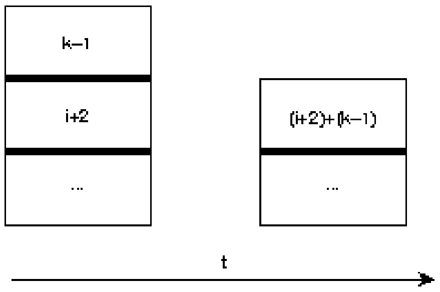

Figure 4.16: expression stack before (left) and after (right) the IADD instruction

The operand expression stack provides the additional information that is necessary for the relationship detector. Due to compiler optimisations, there are cases where the operand expression stack does not provide the correct information. Where there is a definite solution that does not rely on the operand expression stack, we prefer that solution. The operand expression is used, whenever other approaches fail to provide a solution.

### Problem: Collection Element Type Inference

As shown in the previous sections, collaboration fields can be either single or multi–valued. Whereas single–valued collaboration fields can contain references to at most one object at run-time, multi–valued collaboration fields can reference several objects simultaneously.

For single–valued collaboration fields, determining their type in bytecode is straightforward: it can be derived directly from the declaration of the field. However, for multi–valued collaboration fields, the situation is more complex. While the type of the field (i.e. a collection item type) is easily determined, it is the type of the elements stored in the collection item that is needed to deduce the collaboration partner. The element type is the type of the parameter that is added to the collection item. This is done by calling specific methods of the collection item, for example, to add an element to a `java.util.LinkedList` the method `java.util.LinkedList.add()` is called. This is where the key problem emerges: The library collection items (such as `java.util.List`, `java.util.TreeMap`, etc.) provide storage for all kinds of classes and therefore have an interface that is compatible to all by taking `java.lang.Object` arguments for their methods. In Java bytecode, adding an element to a collection item is represented by a method instruction.

Java 1.5 SDK [2] introduces several extensions to the Java programming language, one of which is the notion of *generics*. Generics allow abstractions over types in a similar (but not equivalent) fashion to C++ templates. For instance, generics can be used to specify the type of elements that a container, such as a list, may store. The following code snippet (taken from the Java generics tutorial [38]) shows the application of generics for a list type:

```
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

In principle, generics would provide a solution to the problem stated above - namely inferring the element types of a collection item. However, this information is not directly visible in the Java bytecode. This is due to the way generics are handled by the compiler. Generics are implemented in the Java compiler as a concept called *erasure*. Erasure is a front-end conversion of the source, i.e. the Java source code containing generics is translated into Java source code without generics. In principle, the following steps are executed:

- All parametrised type information is discarded, e.g. `List<Integer>` is translated to `List`.
- Any remaining type variables that are used are replaced by the upper bound of the type variable - usually `java.lang.Object`
- Type correctness is ensured by introducing casts.

Accordingly, the `List` example is translated to the following source code:

```
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

To determine the element types of collection items, we implement the strategies in tables 4.2 and 4.3 and the heuristics in tables 4.4 and 4.5.

### Solutions

| **Strategy** | |
|---|---|
| Applies when | A collection element is retrieved from the collection item |
| Goal | Determine the element type of a collection item |
| Case A | The collection item is one-dimensional |
| | When a collection element is retrieved from a collection item, it needs to be cast to the actual element type. If not, then the collection element would be of type `java.lang.Object`. We therefore monitor all casts to elements that are retrieved from a one-dimensional collection item. The element type of the collection item is then set to the type of the cast instruction. For instance, if there is an instruction `(ElementTypeA)list.get(i))`, then the element type of `list` is set to `ElementTypeA`. If there are multiple cast instructions to the same collection item: then the element type is set to the most general. For example, if `class Super` is a super class of `class Sub` and collection item retrievals from `list` are cast to `Super` and to `Sub`, then the element type of `list` is set to `Super`. This is due to the concept of polymorphism, where a sub class may be passed where a super class is expected, but not vice versa. For this purpose, a tree data structure is generated for the class inheritance hierarchy. Each class has its superclass as parent and its inheriting classes as children. |
| Succeeds | If there is a retrieval of a collection element, then the strategy correctly identifies the element type. |
| Fails | If there is no retrieval, the collection element type cannot be determined. |

Table 4.2: Strategy: one dimensional collection item element types

| **Strategy** | |
|---|---|
| Applies when | A collection element is retrieved from the collection item |
| Goal | Determine the element type of a collection item |
| Case B | The collection item is two-dimensional |
| | The same strategy applies as for the one-dimensional collection items. Additionally, there needs to be a differentiation between casts to the two element types: the key and the value. We use the notation from the notion of HashMaps: the *key* is the primary element in the two-dimensional collection item, while the *value* is the secondary element that is associated with the primary one. For this purpose, we separate the retrieval methods that retrieve the key from those that retrieve the value. For each of these, we separately look at the cast instructions. If there are multiple cast instructions to the same collection item: then the element type is set to the most general of the two. For example, if `class Super` is a super class of `class Sub` and collection item retrievals from `list` are cast to `Super` and to `Sub`, then the element type of `list` is set to `Super`. This is due to the concept of polymorphism, where a sub class may be passed where a super class is expected, but not vice versa. |
| Succeeds | If there is a retrieval to both key and value collection element, then the strategy correctly identifies both element types. If there is a retrieval to only key or only value, then only that element type can be determined. |
| Fails | If there is no retrieval, the collection element type cannot be determined. |

Table 4.3: Strategy: two dimensional collection item element types

| **Heuristics** | |
|---|---|
| Givens | The correct operand expressions are on the stack for the method instruction. This may not apply in the case of compiler optimisations. |
| Goal | Determine the element type of a collection item |
| Case A | The collection item is one-dimensional |
| | For each method instruction that is declared by a one-dimensional collection item, we first determine the relevant argument, i.e. the one that contains the element to be added. For example, for the method instruction `insertElementAt(Object obj, int index)` declared by `Vector`, the first argument contains the element to be added to the vector. Then we remove the operands from the expression stack, according to the rules for the method instructions: For an `INVOKEVIRTUAL`, `INVOKEINTERFACE` or `INVOKESPECIAL` instruction: remove the arg . For an `INVOKESTATIC` instruction: remove the arguments from the expression stack (last argument is at the top). If the relevant argument is a field or a local variable, then determine its type. If the type is a relevant reference type, set the element type of the collection item to it. If the relevant argument is any other expression (i.e. another method call), then do nothing. |
| Succeeds | If the relevant argument is a field or a local variable and the condition stated in Givens applies. |
| Fails | If the compiler performs code optimisations, then the operand expression stack may not have the correct expressions for the method instruction. For instance, compiler optimisations may prevent a certain field from being loaded twice, i.e. the second load instruction will be absent from the bytecode. This means that the relevant field or variable is also not pushed to the operand expression stack (because the instruction is absent) and that the following operation does not have the correct operand expressions on the stack. |

Table 4.4: Heuristics: one dimensional collection item element types

| **Heuristics** | |
|---|---|
| Givens | The method instruction has the correct operand expressions on the expression stack. |
| Goal | Determine the element type of a collection item |
| Case A | The collection item is two-dimensional |
| | For each method instruction that is owned by a two-dimensional collection item, we first determine the relevant arguments, i.e. the ones that contain the two elements to be added. Following the terminology for Map data structures, the first element is denoted by *key*, while the second is denoted by *value*. For example, for the method instruction `put(Object key, Object value)` owned by `HashMap`, the first argument is the key, the second the value. Then we remove the operands from the expression stack, according to the rules for the method instructions: For an `INVOKEVIRTUAL`, `INVOKEINTERFACE` or `INVOKESPECIAL` instruction: remove the arguments from the expression stack (last argument is at the top), then the target reference. For an `INVOKESTATIC` instruction: remove the arguments from the expression stack (last argument is at the top). If the relevant arguments are fields or local variables, then determine their types. If the types are relevant reference types, set the element types of the collection item to them. If the relevant arguments are any other expressions (i.e. method calls), then do nothing. |
| Succeeds | If the relevant arguments are fields or local variables and the condition stated in Givens applies. |
| Fails | In any other case. |

Table 4.5: Heuristics: two dimensional collection item element types

### Problem: Determining Target Name of Method Call

This problem refers to determining on which instance a certain method instruction is called. Again, this is relevant for the collection items, where collaboration participants are implicitly assigned by calling a method that adds an element to the collection item. As an example, the method call `list.add(element)` is compiled to the following bytecode statement:
`INVOKEVIRTUAL java/util/LinkedList.add (java/lang/Object;)` From the method instruction above, we can infer the type of the field on which the method is invoked (i.e. the *owner*), in this case java/util/LinkedList. The name of the field cannot be determined from the method instruction in the bytecode.
We address this problem by applying the heuristics defined in table 4.6

### Solution

| **Heuristics** | |
|---|---|
| Givens | The method instruction has the correct operand expressions on the expression stack. In particular, the target reference expression is correct. |
| Goal | Determine name of the field on which a method instruction is called |
| Case | |
| | For each method instruction that is owned by relevant reference type, we remove the operands from the operand expression stack, according to the rules for the method instructions: For an `INVOKEVIRTUAL`, `INVOKEINTERFACE` or `INVOKESPECIAL` instruction: remove the arguments from the expression stack (last argument is at the top), then the target reference. For an `INVOKESTATIC` instruction: the target reference is the owner type. |
| Succeeds | If the conditions in Givens apply. |
| Fails | If the conditions in Givens do not apply. |

Table 4.6: Heuristics: determine name of target reference

### Problem: Invariant Detection

Both structural and value based invariants occur as if statements in a method that assigns a collaboration field. The corresponding bytecode instructions are a set of tests for certain conditions, for example a `IF_ACMEQ` tests whether two references are equal, while `IFNONNULL` tests whether a reference is equal `null`. The invariant detection needs to be able to distinguish *positive invariants* from *negative invariants*. Positive invariants are invariants that add an element to a collaboration if the condition evaluates to true, while negative invariants are the opposite. A compiler does not necessary translate a positive invariant into a positive invariant in bytecode. Instead, it may invert the condition and switch the corresponding code. For instance, an instruction

```
    ... code part 1
}
else {
    ... code part 2
}
```

may be optimised to

```
    ... code part 2
}
else {
    ... code part 1
}
```

The jump instruction bytecode instructions that the if statements are translated to refer to labels. These labels are positions to code blocks in the bytecode, to which the program execution jumps if a certain condition is fulfilled. For instance, an instruction `IF_ACMEQ L2` jumps to label `L2` if the two references are equal. Otherwise execution continues in the next code block.
We solve this problem by applying the strategy defined in 4.7

**Solution**

| **Strategy** | |
|---|---|
| Givens | The algorithm has detected an invariant expression in a collaboration–related method. The strategy uses the operand expression strategy to determine the operand names of the jump instructions. |
| Goal | Determine the positive invariant, i.e. the one that leads to adding an element to a collaboration |
| Case | |
| | In the preliminary visitor, store the sequence of labels for each method. For each collaboration–related method, store the label in which the collaboration field is assigned. When an invariant pattern `invariant_pattern` is encountered in a collaboration–related method, extract two labels: the label `LTrue` to which execution jumps if the condition evaluates to true and the label `LFalse` which follows (i.e. the one execution continues to if the condition evaluates to false). If the `LTrue` label contains the assignment to the collaboration field, then the invariant was not inverted by the compiler and is the original `invariant_pattern`. If the `LFalse` label contains the assignment, the compiler has inverted the pattern. In this case, the `invariant_pattern` is reverted. For example, in a pattern the operand `<=` is translated to `>`. |
| Succeeds | The strategy finds all invariant patterns, as defined in the section 3.3.2. |
| Fails | If the operand expression strategy fails to find the correct operand names, for example due to compiler optimisations. |

Table 4.7: Strategy: determine positive invariant

## 4.5 Implementation

The relationship detector carries out its analysis in four different phases, with each performing a specific task, as shown in figure 4.5. Each phase builds upon the results of the previous phases and provides the foundation for the following ones. The relationship detector uses the ASM framework to traverse the different bytecode elements. It relies on the visitor pattern: there are visiting methods for the bytecode elements, from the most top-level (i.e. class level) to the most low-level (i.e. instructions). The framework provides the local information for each of these elements, but not the global context information. For instance, the ASM framework gives the following information for a method instruction: the type of instruction, owner, name, and number and types of arguments. However, it does not provide the global context information, such as the method and class containing the particular method instruction. One of the key elements of the relationship detector is therefore to build data structures that store the context information.

The **configuration** phase loads the configuration settings for the relationship detector to work with. The second phase performs a **preliminary analysis**, preparing data structures for the main relationship detection phases. The **local relationship detection** phase follows the detection algorithm and extracts all elements that can be determined locally. The **global relationship detection** then completes the relationship detection from a global point of view. The results of these stages are then combined into statistical information and other output files.

### 4.5.1 Phase One: Configuration

In the first phase of the relationship detector, the configuration settings are included. These settings are stored in a configuration file and are loaded when the relationship detector is started. The advantage of this solution is that adaptations to the configuration can be executed without changing and recompiling the source code. The configuration file indicates the collection item classes that we consider relevant for uncovering hidden relationships, classified as one-dimensional and two-dimensional. These library classes, such as `java.util.LinkedList`, are used to store multi–valued collaboration references. These library classes are subject to changes with each new Java SDK version and need to be kept up to date accordingly. The configuration file also stores the addition and retrieval methods for each of the collection items and defines the classes that are excluded from the relevant references. For example, `javax.swing.*` classes are excluded, as they define the user interface and are therefore not application specific. However, this configuration can be adapted if preferred. For a complete list of the collection item configuration, please refer to the appendix C.

### 4.5.2 Phase Two: Preliminary Analysis

The next phase prepares some of the data structures for the relationship detection phases. A preliminary visitor visits each input class file, using the ASM framework. For each class file, it instantiates a class container data structure. It also generates and stores the data structures for each method in the class, and adds an index of the local variables. Each of the class containers is then inserted into a global data structure, the global class container. In addition, the preliminary analysis phase generates the inheritance hierarchy structure for the class containers. The inheritance data structure consists of a tree, with the `java.lang.Object` class as a root. Each class container node has its super class as parent node, and any inheriting class container nodes as child nodes. The tree structure allows different types of queries that are required in further steps. For instance, the path from a class container node to the root node defines the class inheritance hierarchy for the given class. It is also possible to determine the lowest common superclass of two classes, by finding the lowest common ancestor in the tree. The inheritance hierarchy is important for various reasons: It is necessary to determine the most common superclass, if elements of collection items are cast to different types. This problem is described in more detail in section 4.4.2. On the other hand, the inheritance hierarchy is essential for the transformation of class–based

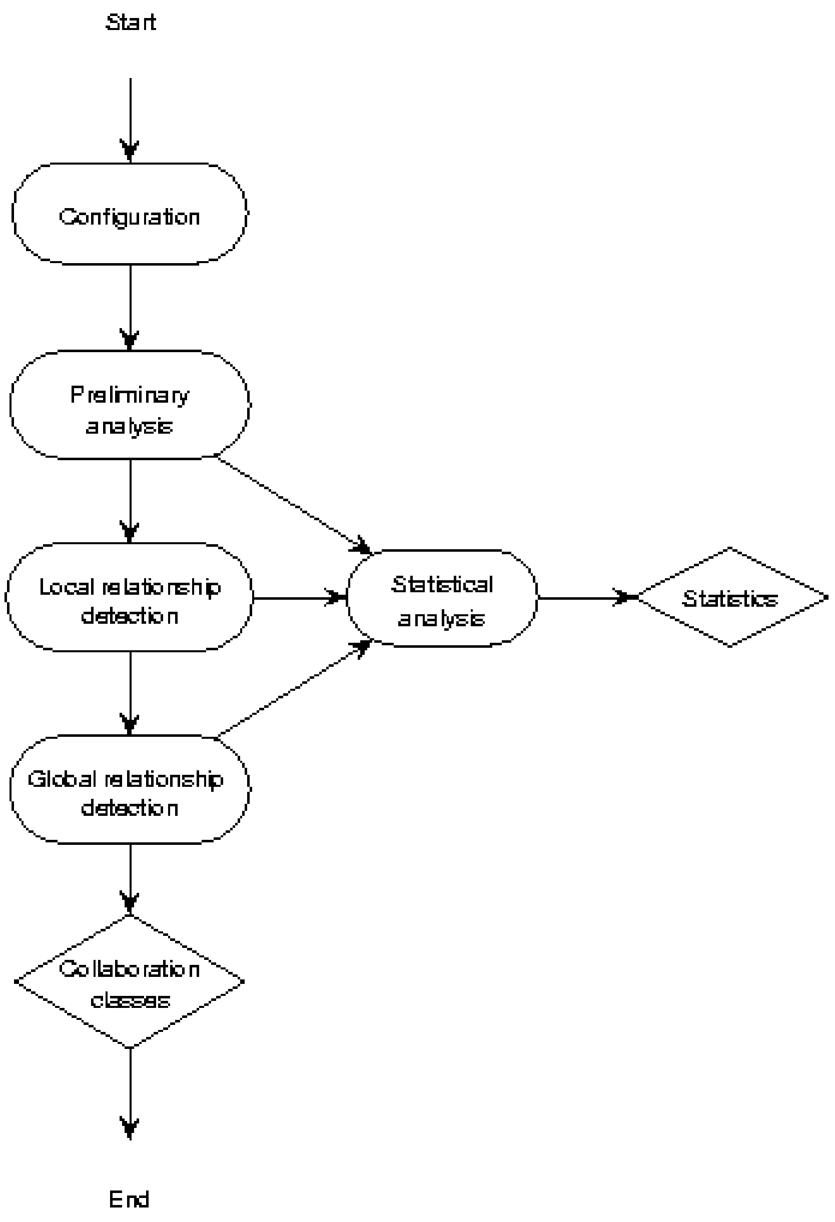

Figure 4.17: relationship detector phases

to relationship–based implementations. As part of the transformation, inheritance hierarchies are replaced by the its super class, with *member interposition* providing the concept of roles. For more details, please refer to section 3.4.7.

### 4.5.3 Phase Three: Local Relationship Detection

This phase constitutes the main part of the relationship detector. In a first part, a local visitor performs the analysis necessary for the second part. The ASM framework provides the interface to visit a class, field, method and other declarations and the corresponding bytecode instructions. The visitor performs the following steps:

- **Classify the global references: relevant or not**
  The key to the relationship detection algorithm, and also its starting point, are the relevant references. The local visitor has to determine, which references are relevant. Single-valued references are relevant if there type is not on the list of classes to exclude, as defined in the configuration. Multi-valued references are relevant if their element types are relevant references.

- **Determine the methods that assign relevant references**
  This is essential to later determine the collaboration implementation option. For instance, a binary indirect collaboration class `ABCol` must have at least one method that assigns both relevant references `A` and `B`.

- **Identify possible invariants**
  The visitor also collects code patterns that point to possible invariants. There are specific patterns for structural invariants, as for the value–based invariants. The concept behind these ideas is defined in section 3.3.2.

The second part consists of applying the relationship detection algorithm, as defined in figure 4.3 at the beginning of this chapter. Essentially, the local relationship detector performs the detection that can be done locally, i.e. from within a given class. Almost the whole algorithm can be covered, with the exception of the matching collaboration classes. For instance, the local relationship detection can determine that there are two binary collaborations *Binary collaboration controlled by A* with reference to B and *Binary collaboration controlled by B* with reference to A. However, it cannot infer that together they constitute a *Binary collaboration controlled by A and B*, i.e. that the two binary collaboration are *matching collaboration classes.* This is done in the next phase. The local relationship detection determines the collaboration option and instantiates a *collaboration class* instance. For each collaboration implementation option, there is aseparatee collaboration class. This collaboration class contains all necessary information to identify the collaboration and provides the basis for the transformation to relationship–based code.

### 4.5.4 Phase Four: Global Relationship Detection

As mentioned above, the global relationship detection performs the steps of the collaboration detection algorithm that cannot be performed by the local relationship detection. This restriction is due to the fact that the local detection does not know the other collaboration classes. For example, if it is the first class to be analysed, then it is not possible to determine if there is a matching collaboration class. The global relationship detection however has access to all collaboration classes. It iterates over them and finds the matching collaboration classes. The following items show the collaboration classes and the matching collaboration classes that constitute them:

- **Binary collaboration controlled by A and B**
  Binary collaboration controlled by A with reference to B and Binary collaboration controlled by B with reference to A.

- **Binary indirect collaboration**
  Binary collaboration controlled by A with a multi–valued reference to an `ABCol` class and a `ABCol` class with references to A and B.

- **Ternary indirect collaboration**
  Binary collaboration controlled by A with a multi–valued reference to an `ABCCol` class and a `ABCCol` class with references to A, B and C.

### 4.5.5 Output

The relationship detector provides a set of output files:

- **Statistics**
  The statistics gather information about the input and the corresponding collaboration classes. They are the basis for further analysis, as described in the evaluation chapter of this thesis. The statistics compare the total occurrence of a certain concept, with the occurrence of the collaboration–related concept - for instance, it computes the percentage of collaboration–related methods to the method total. It also shows the number of new classes that would be generated upon transformationn from class to relationship based implementations.

- **Collaborations**
  The collaborations output shows the different types of collaborations, the participating types and information such a collaboration–related methods, structural and value–based invariants.

- **Relationships**
  The relationship output gives the structure of the relationships that would be generated for the transformation from class- to relationship–based implementation. Each relationship contains the participant classes, the methods moved from the classes, and the value–based and structural invariants.

- **Inheritance structure**
  This output file shows the class inheritance structure of the input class files.

# Chapter 5

# Evaluation

In this chapter, we evaluate the results of the relationship detector tool. For this purpose we analyse a set of programs and compare the output of the tool with our analysis of the code. The examples are taken from different areas: starting with a set of small Java programs, we then progress onto more complex applications, such as a graphical framework. The results provide a basis to discuss the occurrence of collaborations in object–oriented programs. Then we put our results into a broader perspective. Based on the collaborations that the relationship detector extracts, we show what benefits can be gained from introducing relationships as first-class concepts.

## 5.1 Relationship Detector Results

We compare the results of the relationship detector with our manual extraction of the collaborations in the code. Our qualitative analysis is based on the application of the following quantitative metrics:

**Classes**

- **Number of classes**
  The total number of java class files that the relationship detector receives as input.
- **Number of collaboration classes**
  The number of collaborations that the relationship detector identifies. We also specify the number of collaborations for each of the collaboration implementation options. The total number of collaboration classes may exceed the number of classes. This is due to the circumstance that a class can participate in multiple collaborations.
- **Number of collaboration control classes**
  This is the number of classes that contain code where collaboration classes are instantiated or where method calls to such objects are made.

**Methods**

- **Number of methods**
  The total number of methods in all classes.
- **Number of collaboration–related methods**
  This is the number of collaboration–related methods for all classes. This provides a hint as to how many methods are moved to the relationship in the transformation to a relationship–based implementation - or become obsolete.

- **Percentage of collaboration–related methods**
  This metric relates the collaboration–related methods to the total number of methods. Again, this provides an insight into how many methods are connected to the collaborations.

**Lines**

- **Lines of bytecode**
  As the relationship detector analyses bytecode, we use the number of bytecode instructions as our metric, instead of lines of sourcecode. Usually, the lines of bytecode exceed the number of lines of sourcecode and would not allow a comparison of the two. However, the ratio of collaboration related lines vs. total lines should yield approximately the same values for bytecode and sourcecode comparisons.

- **Lines of collaboration–related bytecode**
  The number of lines of collaboration–related bytecode is computed as follows:

  - declarations of collaboration fields;
  - assignments to single–valued collaboration fields
    the corresponding `PUTFIELD` instruction takes the target reference (owning the field) and the value from the stack, then executes the instruction. Therefore, such an assignments amounts to three collaboration–related bytecode instructions.
  - assignments to multi–valued collaboration fields
    following the same argument as above, the number of instructions depends on the number of arguments. The `INVOKEVIRTUAL`, `INVOKESPECIAL` and `INVOKEINTERFACE` amount to `numArguments + 2` (= arguments + target reference + the instruction itself), the `INVOKESTATIC` to `numArguments + 1` (no target reference).
  - method calls with a collaboration field as target reference.
  - field instructions on a collaboration field.
  - collaboration control code instructions
    such as calls to collaboration–related methods

- **Lines of collaboration control bytecode**
  Instantiations of collaboration classes and method calls to collaboration–related methods.

- **Lines of bytecode not necessary after transformation**
  The number of lines that become obsolete after a transformation to a relationship–based implementation is computed in a conservative manner. These lines consist of the instructions that can be positively identified as not necessary after transformation, i.e. the definite candidates. This is the case for collaboration field declarations, collaboration field assignments and retrievals and structural invariant code.

### 5.1.1 University Example

The university example is a slightly modified version of the example given in section 3.2.7. In particular, we added some structural and value–based invariants to the code and some extra classes. For the complete code, please refer to the appendix C. Figure 5.1.1 shows the UML model for the university example.

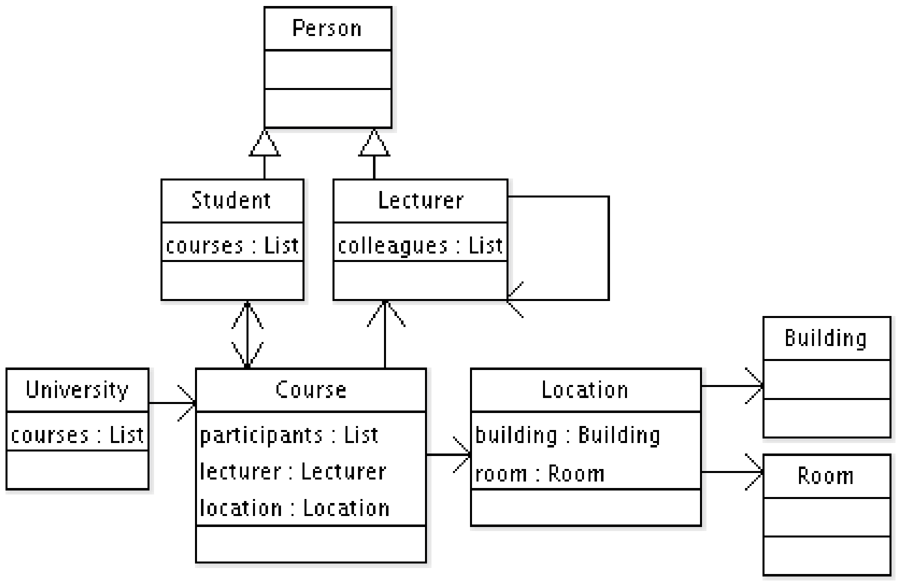

Figure 5.1: University example

#### Overall Program Metrics

| | Total |
|---|---|
| Classes | 8 |
| Methods | 19 |
| Lines of bytecode | 343 |

#### Collaborations

The relationship detector identifies the following four collaborations. Each collaboration corresponds to one of the collaboration implementation options.

- **Binary indirect collaboration**

| Coll. option element | Class | |
|---|---|---|
| Class A | `Lecturer` | single valued |
| Class B | `Location` | single valued |
| Class ABCol | `Course` | |
| Class ABColCollection | `University` | |
| Structural invariants | `Partial function(A, B)` | |
| | `Partial function (B, A)` | |
| Value based invariants | none | |

- **Binary indirect collaboration**

| Coll. option element | Class | |
|---|---|---|
| Class A | `Building` | single valued |
| Class B | `Room` | single valued |
| Class ABCol | `Location` | |
| Class ABColCollection | `Lecturer` | |
| Structural invariants | `Partial function (A, B)` | |
| | `Partial function (B, A)` | |
| Value based invariants | none | |

- **Binary collaboration controlled by Course and Student**

| Coll. option element | Class | |
|---|---|---|
| Class A | `Course` | multi valued |
| Class B | `Student` | multi valued |
| Structural invariants | `Partial relation (A, B)` | |
| | `Partial relation (B, A)` | |
| Value based invariants | `participants.size()+1 <= maxStudents` | |
| | `courses.size()+1 <= maxCourses` | |

- **Binary collaboration controlled by Lecturer and Lecturer**

| Coll. option element | Class | |
|---|---|---|
| Class A | `Lecturer` | single valued |
| Class B | `Lecturer` | multi valued |
| Structural invariants | `Partial irreflexive relation (A, B)` | |
| | `Partial irreflexive relation (B, A)` | |
| Value based invariants | none | |

- **Collaboration-related control code**
  There is collaboration–related control code in the following classes:

```
Student
University
```

### Relationships

The relationship detector generates the basic structure for the four relationships:

```
relationship LecturerLocationRelationship(){
      participants(Lecturer lec, Location loc)
      ...
}
relationship BuildingRoomRelationship(){
      participants(Building b, Room r)
      ...
}
relationship CourseStudentRelationship(){
      participants(Course c, Student s)
      ...
}
relationship LecturerLecturerRelationship(){
      participants(Lecturer lec1, Lecturer lec2)
      ...
}
```

These are the classes that are associated in the class–based implementation by using the various collaboration implementation options. After transformation to the relationship–based implementation however, the implementation option that was present in the class–based implementation, is no longer visible. So, for example, formerly binary indirect collaborations and collaborations controlled by A are no longer distinguishable. The relationships are named according the the naming convention, with the two participating classes and a suffix.
Seven of the total 19 methods are moved from classes to the relationships, 36% of the original number of methods. The transformation renders 49 lines of bytecode obsolete, which corresponds to 23% of the total lines of bytecode.

### Assessment

Table 5.1 summarises the relationship detector results for the university example.

The relationship detector recognises all collaborations, structural invariants and value–based invariants.

In comparison to the relationships, the collaborations in the class–based implementation affect more classes: seven classes together create four collaborations. For instance, to associate the classes

| | Total | % |
|---|---|---|
| Classes | 8 | |
| Collaborations | 4 | |
| Methods moved | 7 | 36% |
| Collaboration lines | 120 | 35% |
| Obsolete lines | 49 | 23% |

Table 5.1: Relationship detector results for the university example

`Lecturer` and `Location`, the indirect collaboration also needs the classes `Course` and `University`. The two participants could also be associated directly in a binary collaboration controlled by one of them (or both). However, the advantage of the indirect collaboration is that it also provides an easy access to all participants in a collaboration, through the `ABCColCollection` class, in this case `University`. The relationship–based implementation achieves the same effect with much less code and complexity. The relationship `Course` associates the participants `Lecturer` and `Location` and provides a retrieval mechanism to access all elements that does not have to be implemented by the programmer.
The complexity mentioned above is also apparent in the number of methods that are collaboration–related. Almost half at least partly serve the purpose of adding or retrieving participants from a collaboration. Approximately a fourth of the lines of bytecode become obsolete in the classes, by introducing the notion of relationships.

### 5.1.2 Calendar Example

Figure 5.1.2 shows the structure of the calendar example. The example is taken from the course "Introduction to programming in the large" at the ETH in Zurich. The calendar application contains appointments, days and months, and provides a view for each of these. For the complete code, please refer to the appendix C.

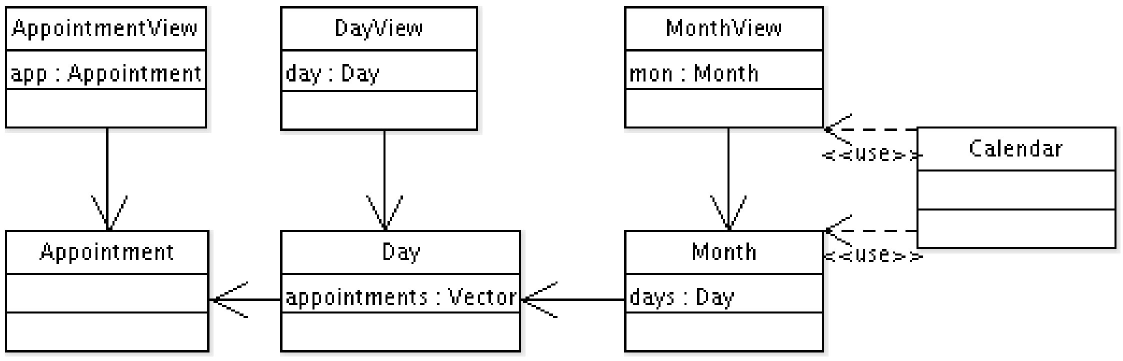

Figure 5.2: Calendar example

#### Overall Program Metrics

| | Total |
|---|---|
| Classes | 7 |
| Methods | 35 |
| Lines of bytecode | 1510 |

### Collaborations

The relationship detector identifies the following 5 collaborations:

- **Binary collaboration controlled by AppointmentView**

| Coll. option element | Class | |
|---|---|---|
| Class A | `AppointmentView` | single valued |
| Class B | `Appointment` | single valued |
| Structural invariants | `Partial function (A, B)` | |
| Value based invariants | none | |

- **Binary collaboration controlled by DayView**

| Coll. option element | Class | |
|---|---|---|
| Class A | `DayView` | single valued |
| Class B | `Day` | single valued |
| Structural invariants | `Partial function (A, B)` | |
| Value based invariants | none | |

- **Binary collaboration controlled by MonthView**

| Coll. option element | Class | |
|---|---|---|
| Class A | `MonthView` | single valued |
| Class B | `Month` | single valued |
| Structural invariants | `Partial function (A, B)` | |
| Value based invariants | none | |

- **Binary collaboration controlled by Appointment**

| Coll. option element | Class | |
|---|---|---|
| Class A | `Day` | single valued |
| Class B | `Appointment` | multi valued |
| Structural invariants | `Partial relation (A, B)` | |
| Value based invariants | `if(i==` `appointments.size()){` | |

- **Binary collaboration controlled by Month**

| Coll. option element | Class | |
|---|---|---|
| Class A | `Month` | single valued |
| Class B | `Day` | multi valued |
| Structural invariants | `Partial relation (A, B)` | |
| Value based invariants | none | |

- **Collaboration-related control code**
  The following classes contain collaboration related control code:

```
AppointmentView
Calendar
Day
DayView
MonthView
```

### Relationships

The relationship detector generates the basic structure of five relationships:

```
relationship AppointmentViewAppointmentRelationship()(){
       participants(AppointmentView view, Appointment app)
       ...
}
relationship DayAppointmentRelationship(){
       participants(Day day, Appointment app)
       ...
}
relationship DayViewDayRelationship(){
       participants(DayView view, Day day)
       ...
}
relationship MonthDayRelationship(){
       participants(Month mon, Day day)
       ...
}
relationship MonthViewMonthRelationship(){
       participants(MonthView view, Month mon)
       ...
}
```

These are the classes that are associated in the class–based implementation by using the binary controlled by A collaboration option.
Six of the total 35 methods are moved from classes to the relationships, 17% of the original number of methods. The transformation renders 38 lines of bytecode obsolete, which corresponds to 9% of the total lines of bytecode.

### Assessment

Table 5.1.2 summarises the relationship detector results for the calendar example.

The relationship detector recognises all collaborations, structural invariants and value–based invariants.

| | Total | % |
|---|---|---|
| Classes | 7 | |
| Collaborations | 5 | |
| Methods moved | 6 | 17% |
| Collaboration lines | 302 | 20% |
| Obsolete lines | 28 | 9% |

Table 5.2: Relationship detector results for the calendar example

The calendar example contains a series of binary collaborations controlled by one class only. All collaborations either have the structural invariant partial relation or partial function. This means that totality and other properties (such as injection) are not enforced – the function property is implicitly enforced by a single–valued collaboration field. All collaborations implement the most basic collaboration option, with binary collaborations controlled by A only. This means that collaboration–related code is not as prevalent as in other examples. Out of the total number of lines, 302 are collaboration–related, a ratio of 20%.

### 5.1.3 Compiler Design Example

This example is the result of the semester project for the class "Compiler Design I" at the ETH in Zurich. The compiler works for a subset of Java and uses the Yylex and CUP frameworks for the lexical analysis and parser generation. For the complete code and further information, please refer to [16].

#### Overall Program Metrics

| | Total |
|---|---|
| Classes | 24 |
| Methods | 284 |
| Lines of bytecode | 12104 |

#### Collaborations

The relationship detector identifies 11 collaborations in the compiler design example. In the following, we will provide a summary, instead of a complete coverage of the implementation options.

| Total collaborations | | 11 |
|---|---|---|
| Binary collaborations | | 10 |
| | Controlled by A | 7 |
| | Controlled by AB | 1 |
| | Direct | 1 |
| | Indirect | 1 |
| Ternary collaborations | | 1 |
| | Indirect | 1 |

In addition, there are 6 classes that contain collaboration–related control code. This includes, for example, instantiations of collaboration classes and calls to collaboration–related methods.

#### Relationships

The relationship detector defines the structure of the relationships that would be generated when transforming the compiler example from a class–based to a relationship–based implementation. The structure includes the participants, the invariants and the methods that would be moved from the classes. 32 of the total 284 methods are moved from classes to the relationships, which is equivalent to 11% of the total number of methods. The transformation renders 293 lines of bytecode obsolete, which corresponds to 6.0% of the total lines of bytecode.

#### Assessment

Table 5.3 summarises the relationship detector results for the compiler design example.

| | Total | % |
|---|---|---|
| Classes | 24 | |
| Collaborations | 11 | |
| Methods moved | 31 | 11% |
| Collaboration lines | 1142 | 10% |
| Obsolete lines | 293 | 6% |

Table 5.3: Relationship detector results for the compiler example

In this example, the relationship detector fails to identify two elements:

- **Collection element type**
  In `class SemanticAnalyzer` there is a `java.util.LinkedList` instance called `methods_`. However, no elements are added or retrieved from this list, nor is there any other code containing this reference. The relationship detector fails in this case due to missing information.

- **Collection class for ternary collaboration**
  The `class MethodEnv` is the ABCCol class for a ternary collaboration. There is no ABCColCollection class in this example, so the relationship detector cannot recognise it.

The compiler design example does not contain many collaborations, as compared to the other examples. However, the collaborations have 31 methods that at least partly serve the purpose of adding and retrieving from the collaboration. In addition, 10% of the lines of bytecode are collaboration–related. Both the methods and the collaboration-lines point to the fact that, although the collaborations are not as numerous as in other examples, they are more involved, with more code necessary to implement them. Among the 11 collaborations, the relationship detector extracts a ternary indirect collaboration. It associates the classes `Symbol` `Node` and `Table`.

### 5.1.4 JHotDraw Example

JHotDraw [14] is a graphics framework written entirely in Java. It provides a graphics library to be used by graphics editors for technical and structured graphics. JHotDraw is available under the GNU library or lesser general public license (LGPL). The framework is unrelated to the topic of object collaborations and therefore provides some insight into what kinds of collaborations are found in such a collaboration–unrelated example.

#### Overall Program Metrics

| | Total |
|---|---|
| Classes | 600 |
| Methods | 5475 |
| Lines of bytecode | 130218 |

#### Collaborations

Due to the amount of classes, we will provide some core results, instead of covering all collaboration classes.

| Total collaborations | | 442 |
|---|---|---|
| Binary collaborations | | 441 |
| | Controlled by A | 417 |
| | Controlled by AB | 0 |
| | Direct | 9 |
| | Indirect | 15 |
| Ternary collaborations | | 1 |
| | Indirect | 1 |

In addition, there are 221 classes that contain collaboration–related control code. This includes, for example, instantiations of collaboration classes and calls to collaboration–related methods.

### Relationships

For each of the collaborations, the relationship detector gives the structure of the relationship, with the participants, the invariants and the methods that would be moved from the classes. 709 of the total 5475 methods are moved from classes to the relationships, 11% of the total number of methods. The transformation renders 2925 lines of bytecode obsolete, which corresponds to 4.0% of the total lines of bytecode.

### Assessment

Table 5.4 summarises the results of the JHotDraw example.

| | Total | % |
|---|---|---|
| Classes | 600 | |
| Collaborations | 442 | |
| Methods moved | 709 | 12% |
| Collaboration lines | 8831 | 7% |
| Obsolete lines | 2925 | 4% |

Table 5.4: Relationship detector results for the JHotDraw example

In this example, the relationship detector fails to identify some elements. We therefore first assess these issues, then analyse the results as in the previous examples.

- **Collection item type inference**
  One of the main challenges for the relationship detector is inferring the element types of binary direct collaborations. This collaboration implementation option uses a two dimensional collection item (such as `java.util.HashMap`) to associate one collaboration participant with the other. In one of the direct collaborations, the relationship detector fails to recognise the type of the second participating class:

| | |
|---|---|
| Class A | `org.jhotdraw.framework.Figure` |
| Class B | `unknown element type` |
| Class ABCol | `org.jhotdraw.standard.AlignCommandUndoActivity` |
| Structural invariants | `Partial relation(Figure, unknown)` |
| Value based invariants | none |

The relationship detector cannot infer the value type information from the following instruction:

```
protected void addOriginalPoint(Figure f) {
    myOriginalPoints.put(f, f.displayBox().getLocation());
}
```

This is due to the restriction that the relationship detector cannot recognise the return type of nested method invocations.

Collection item inference is also relevant to the other collaboration implementation options, where a collaboration field is multi valued. Of the 22 multi–valued references, the relationship fails to determine the collection item type in four cases:

```
org/jhotdraw/util/ReverseListEnumerator
org/jhotdraw/standard/CreationTool
org/jhotdraw/standard/ReverseFigureEnumerator
org/jhotdraw/contrib/CustomToolBar
```

In all four cases, the classes act as a wrapper class. They are passed a list, to which the own lists are assigned. The relationship detector fails to infer the type of the collection item, because there are no additions of elements to the list nor retrievals from the list. The collection items are handled in their entirety (i.e. as lists).

- **Indirect collaboration classes** The 15 binary and one ternary indirect collaboration classes have no `ABColCollection` (`ABCColCollection` respectively) classes assigned. Again, this is due to the fact that the JHotDraw example is a framework. While it provides the indirect collaboration data structures, it does not instantiate them and store the instances as `ABColCollection` elements.

While there are many collaborations in this example, the collaboration–related code is not as prevalent. This may be due to the fact that the JHotDraw is an example of a framework. It provides a set of classes to a user to implement an application, using the framework. This also means that much of the instantiation and therefore collaboration code happens within the user application, where the framework classes are instantiated, associated and called. Interestingly, the indirect and direct collaboration options are used where there are many collaboration elements. For the direct collaboration, the two-dimensional data structure provides fast access to the participants: by iterating over the key participants (according to the naming of `Maps`) and retrieving the corresponding value elements. The indirect collaborations allow fast access to the participants through the collection classes `ABColCollection` (binary) or `ABCColCollection` (ternary).

### 5.1.5 Jasper Reports Example

Jasper Reports [32] is an open source reporting engine, used for business intelligence (reporting, OLAP) in Web and desktop applications. It can produce its output in various formats, such as PDF, XML or HTML.

#### Overall Program Metrics

| | Total |
|---|---|
| Classes | 1850 |
| Methods | 18930 |
| Lines of bytecode | 659006 |

### Collaborations

As in the previous example, we will provide some core results only, instead of covering all collaboration classes.

| Total collaborations | | 1242 |
|---|---|---|
| Binary collaborations | | 1198 |
| | Controlled by A | 792 |
| | Controlled by AB | 0 |
| | Direct | 284 |
| | Indirect | 122 |
| Ternary collaborations | | 44 |
| | Indirect | 44 |

In addition, there are 628 classes that contain collaboration–related control code. This includes, for example, instantiations of collaboration classes and calls to collaboration–related methods.

### Relationships

1897 of the total 18930 methods are moved from classes to the relationships, which is equivalent to 10% of the total number of methods. The transformation renders 17374 lines of bytecode obsolete, which corresponds to 6.0% of the total lines of bytecode.

### Assessment

Table 5.1.5 summarises the results for the Jasper Reports example.

| | Total | % |
|---|---|---|
| Classes | 1850 | |
| Collaborations | 1242 | |
| Methods moved | 1897 | 10% |
| Collaboration lines | 75205 | 12% |
| Obsolete lines | 17374 | 6% |

Table 5.5: Relationship detector results for the Jasper Reports example

The Jasper Report is another example of a framework. 12% of the bytecode lines are collaboration–related, with 6% becoming obsolete after transformation. With the dimensions of the example, these two properties amount to multiples of ten thousands of lines. Interestingly, there are no binary collaborations controlled by both classes in over a thousand collaborations. All structural invariants are either partial relations or partial functions, both of which are not explicitly enforced by the programmer. The partial relation is the default structural invariant, while the partial function is implicitly enforced by a single–valued collaboration field. Value-based invariants however occur frequently.

## 5.2 Concept Evaluation

We now put our results into a broader perspective and show how they can be generalised. In particular, we are interested in how introducing relationships as first-class concepts would change the structure and size of programs.

A first conclusion from our results is that collaborations are a substantial part of the applications. Collaboration-related methods and lines of code can take up to a fourth of the total methods and lines. In our examples, the collaboration implementations options occur with varying frequency. Table 5.6 shows the percentage of the options, compared to the total number of collaborations.

| Coll. Implementation option | % |
|---|---|
| Binary collaboration controlled by A | 72% |
| Binary direct collaboration | 17% |
| Binary indirect collaboration | 8% |
| Ternary indirect collaboration | 3% |
| Binary collaboration controlled by AB | 0.2% |

Table 5.6: Occurrence of collaboration implementation options

By far the most prevalent option is the binary collaboration controlled by A, the most simple of all options. In our example, it constitutes almost 3/4 of the total amount of collaborations. The second most frequent is the binary direct collaboration, with 17% of the total number in our examples. It is then followed by the binary indirect collaboration option, with 8%. Interestingly, the ternary collaboration implementation option occurs more frequently than expected – 3% for our examples. It even appears more often than the binary collaboration controlled by AB implementation option, which was practically non-existent outside the university example. From these results we can gather that programmers mostly choose the simplest collaboration option for their implementation. The option associates the collaboration participants through references, with no additional indirection supplied. This means that the collaboration can be accessed through one of the participants only. For instance, if `class A` is associated with a `class B` in a binary collaboration controlled by A, then for a participant `A` the associated participant `B` can be determined. However, as the `B` does not have a reference to `A`, a participant `B` cannot determine its associated participant `A`. Furthermore, accessing all participants of a collaboration when instantiated is another issue. Users need to write their own code to achieve an iteration over the set of participants.

Another key element are the invariants. Value-based invariants occur frequently in the programs we analysed, with the same patterns repeating themselves, often in various methods throughout a class. For example, the Jasper Report example frequently checks if values are greater than zero, such as `srcCategorySeries >= 0`. We believe that offering the mechanism of value–based invariants in the concept of relationship provides more security and consistency for a implementation. Classes in class–based implementations often have multiple methods adding participants to the collaboration. The value–based checks need to be the same in each of these methods. Not only is this code redundant, but there is also the problem of possibly lacking consistency. The programmers themselves are responsible to ensure that the value–based checks happen consistently. The relationship–based approach allows the programmer to define the value–based invariants in one location. Structural invariants are almost never enforced. Even in examples as large as the Jasper Report with approximately 2000 classes, the structural invariants are either partial relation or partial function. As previously mentioned, partial relation is the default for the structural invariants of a collaboration. The partial function is implicitly enforced by representing the collaboration field as a single–valued reference. The invariant detection recognises the patterns that point to structural invariants, as in the university example. However, it seems that programmers avoid the additional code required to enforce an invariant. Again, we believe that the relationship–based approach provides valuable support in this case: Through a simple invariant clause, programmers can specify the structure of their relationships. There is no code necessary on the programmer's side to enforce it. The structural invariant checks improve the security of an application: "wrong" user input, such as trying to add two participants to the relationship `Marriage` that are already married to other participants, can be handled correctly (by not adding the participants to the relationship in this example).

A further key aspect is the distribution of collaboration–related code among the classes. Various program elements are connected to the collaborations, from the classes themselves, to fields, methods and invariant code patterns. Often, these elements are also distributed and duplicated within the classes. As the amount of collaboration–related elements is fairly substantial, it makes sense to separate the classes from the collaborations. With the relationship–based approach, classes contain the elements that are relevant to themselves. The relationships then provide a central location for the elements that are concerned with the relationship. Arguably, this separation of relationships from classes makes the design and maintenance of applications more consistent.

A final argument is the lines of bytecode that become obsolete in a relationship–based approach. The numbers are conservative, which means that the actual number of lines no longer needed may exceed that amount. This is due to the collaboration–related methods. For our metrics, we include only additions to a collaboration, retrieval from a collaboration and structural invariant code. Many collaboration–related methods however may become completely obsolete, if they contain such elements only. From a bytecode point of view we cannot determine this with certainty. Our approach therefore is to move the collaboration–related methods in their entirety to the relationship and remove the obsolete lines. A user can then remove the methods that no longer contain relevant instructions.

# Chapter 6

# Conclusion

## 6.1 Summary

In this thesis we investigated how relationships occur in the modelling of software systems and their object-oriented implementation. We first offered a short overview of the research into relationships, from the database and object-oriented programming communities. With this background information, we then analysed how relationships are modelled and implemented in class–based implementations. For this purpose, we introduced a set of simple examples that covered all types of relationships and implemented them first with classes, then with relationships. As a next step, we generalised the results from the examples and defined the *collaboration implementation options*, the patterns of how classes form relationships in class–based programs. Special focus was also put on *structural* and *value–based invariants* and their relevance to relationships. We then designed and implemented the relationship detector, a tool that uncovers hidden relationships in Java bytecode programs. The analysis was centered around the *collaboration detection algorithm* that determines the collaboration implementation options from the bytecode input. Finally, we tested the relationship detector with a set of increasingly complex examples and provided the results with our predefined metrics. These results were then put into a broader perspective and we discussed the benefits of adding relationships as first-class concepts to object-oriented programming languages.

## 6.2 Future Work

The relationship detector could be improved and extended in the following areas

- **Improvements to implementation strategies and heuristics**
  The relationship detector operates with a series of strategies and heuristics, as defined in section 4.4.2. These strategies and heuristics could be refined and extended.
- **Transformation to a relationship–based implementation**
  Our tool extracts all necessary information to transform the class–based Java implementation to a relationship–based implementation: the participants, invariants and methods to be moved are all known and available. However, currently the relationship detector does not actually generate the relationship–based (byte-) code.
- **Collaboration visualisation**
  The relationship detector has text-based outputs for the statistics, collaborations, inheritance structures, and relationships. A graphical user interface and graphical output (e.g. in terms of charts) could be added.
- **Eclipse plugin**
  Another nice feature would be to create an eclipse plugin for the relationship detector.

# Bibliography

[1] Understanding tradeoffs among different architectural modeling approaches. In *WICSA '04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[2] Java SDK 1.5. http://java.sun.com/j2se/1.5.0/, 2006.

[3] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 565–575. Morgan Kaufmann, 1991.

[4] Paolo Atzeni, Wesley W. Chu, Hongjun Lu, Shuigeng Zhou, and Tok Wang Ling, editors. *Conceptual Modeling - ER 2004, 23rd International Conference on Conceptual Modeling, Shanghai, China, November 2004, Proceedings*, volume 3288 of *Lecture Notes in Computer Science*. Springer, 2004.

[5] Stephanie Balzer, Patrick Eugster, and Thomas R. Gross. Internal report: Value-based and structural invariants for object relationships, 2006.

[6] Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual database design: an Entity-relationship approach*. Benjamin-Cummings Publishing Co., Inc., 1992.

[7] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *OOPSLA*, pages 1–6, 1989.

[8] Gavin M. Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *ECOOP*, pages 262–286, 2005.

[9] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. The unified modeling language user guide. *J. Database Manag.*, 10(4):51–52, 1999.

[10] Éric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, November 2002.

[11] Peter Pin-Shan Chen. The entity-relationship modeltoward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[12] Ramez A. Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[13] Gregor Engels, Roland Huecking, Stefan Sauer, and Annika Wagner. Uml collaboration diagrams and their transformation to java. In *UML*, pages 473–488, 1999.

[14] The JHotDraw Graphics Framework. http://www.jhotdraw.org/, 2006.

[15] William Harrison, Charles Barton, and Mukund Raghavachari. Mapping uml designs to java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 178–187, New York, NY, USA, 2000. ACM Press.

[16] Compiler Design I. http://www.lst.inf.ethz.ch/teaching/lectures/ss06/222/index.html, 2006.

[17] Ivar Jacobson, Grady Booch, and James E. Rumbaugh. Excerpt from "the unified software development process": The unified process. *IEEE Software*, 16(3):82–90, 1999.

[18] Tetsuro Katayama. Proposal of a supporting method for diagrams generation with the transformation rules in uml. In *APSEC*, pages 475–484, 2002.

[19] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[20] Bent Bruun Kristensen. Complex associations: abstractions in object-oriented modeling. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 272–286, New York, NY, USA, 1994. ACM Press.

[21] Gary T. Leavens. Introduction to the literature on object-oriented design, programming, and languages. *OOPS Messenger*, 2(4):40–53, 1991.

[22] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[23] Bertrand Meyer. *Object-oriented software construction (2nd ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[24] Bertrand Meyer. *The Dependent Delegate Dilemma, in Engineering Theories of Software Intensive Systems, Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems, Marktoberdorf, Germany, from 3 to 15 August 2004, eds. Manfred Broy, J Gruenbauer, David Harel, C.A.R. Hoare, NATO Science Series II: Mathematics, Physics and Chemistry, vol. 195.* Springer-Verlag, 2005.

[25] Badri Narasimhan, Shamkant B. Navathe, and Sundaresan Jayaraman. On mapping er models into oo schemas. In *ER*, pages 402–413, 1993.

[26] J. Noble. Basic relationship patterns, 1997.

[27] J. Noble and J. Grundy. Explicit relationships in object-oriented development, 1995.

[28] Wikipedia: Relations over a set. http://en.wikipedia.org/wiki/binary_relation, 2006.

[29] ASM Home Page. http://asm.objectweb.org/index.html, 2006.

[30] BCEL Home Page. http://jakarta.apache.org/bcel/index.html, 2006.

[31] SOOT Home Page. http://www.sable.mcgill.ca/soot/, 2006.

[32] Jasper Reports Project. http://jasperforge.org/sf/projects/jasperreports, 2006.

[33] The JBoss Application Server Project. http://labs.jboss.com/portal/jbossas, 2006.

[34] The Jetty Project. http://jetty.mortbay.org/jetty/index.html, 2006.

[35] The JFox Project. http://labs.huihoo.com/jfox/index.html, 2006.

[36] James E. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *OOPSLA*, pages 466–481, 1987.

[37] Mathupayas Thongmak and Pornsiri Muenchaisri. Design of rules for transforming uml sequence diagrams into java code. In *APSEC*, pages 485–, 2002.

[38] Java Generics Tutorial. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, 2006.

[39] R. J. Wieringa. *Requirements engineering: frameworks for understanding*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[40] Rebecca Wirfs-Brock and Alan McKean. *Object Design: Roles, Responsibilities, and Collaborations*. Pearson Education, 2002. Foreword By-Ivar Jacobson and Foreword By-John Vlissides.

# Appendix A

# User's Guide

## A.1 Overview

The relationship detector is shipped in the file `relationDetector.tar.gz`, and has the following structure.

- **collaboration_classes**
  Contains the data structures for the collaboration classes.
- **Config**
  Contains the configuration file for the relationship detector.
- **containers**
  Provides the various data structures for the relationship detection.
- **doc**
  Contains the javadoc documentation for the relationship detector project.
- **invariants**
  Provides the data structures for the invariants.
- **main**
  Contains the main control class and the relationship detection classes.
- **statistics**
  Provides the structures to gather statistical information.
- **Test**
  Contains some test classes.

## A.2 Setup instructions

### A.2.1 Eclipse project setup

1. Extract the contents
   `tar xzf relationshipDetector.tar.gz`
2. Start eclipse
3. Create a new project
   Choose `File --> New --> Project`
   Create project from the relationship detector source

4. **Download ASM**
   Get the current `asm_jar` file from
   `http://forge.objectweb.org/projects/asm/`

5. **Add the `asm_jar` to the java build path**
   Choose `Project --> Properties --> Java build path`
   Choose `Add external jars` and specify the `asm_jar` location.

### A.2.2 Setup without eclipse

1. **Extract the contents**
   `tar xzf relationshipDetector.tar.gz`

2. **Download ASM**
   Get the current `asm_jar` file from
   `http://forge.objectweb.org/projects/asm/`

3. **Add the `asm_jar` file to the java build path**
   `export PATH=\$PATH:\<path to the jar file\>`

# Appendix B

# Implementation Details

## B.1 Configuration Settings

- **Collection item classes**
  - java/awt/list
  - java/util/AbstractCollection
  - java/util/AbstractList
  - java/util/AbstractMap
  - java/util/AbstractSequentialList
  - java/util/AbstractSet
  - java/util/ArrayList
  - java/util/HashMap
  - java/util/Hashtable
  - java/util/HashSet
  - java/util/LinkedHashMap
  - java/util/LinkedHashSet
  - java/util/LinkedList
  - java/util/TreeMap
  - java/util/TreeSet
  - java/util/Vector
  - java/util/WeakHashMap
- **One-dimensional collection item clases**
  - java/awt/list
  - java/util/AbstractCollection
  - java/util/AbstractList
  - java/util/AbstractSequentialList
  - java/util/AbstractSet
  - java/util/ArrayList
  - java/util/HashSet
  - java/util/LinkedHashSet

  - java/util/LinkedList
  - java/util/TreeSet
  - java/util/Vector

- **Two-dimensional collection item classes**
  - java/util/AbstractMap
  - java/util/HashMap
  - java/util/Hashtable
  - java/util/LinkedHashMap
  - java/util/TreeMap
  - java/util/WeakHashMap

## B.2 Bytecode Instructions

This section gives the list of bytecode instructions that are analysed and the rules that apply for each instruction for the operand stack. The following notation is used for the operand stack rules:

```
x, y, ... --> v, w, ...
```

The left side of the arrow shows the state of the operand stack *before* the bytecode instruction is executed. The right side shows the state *after* execution. The stack grows from left to right, i.e. the rightmost element is the top element.

- **Field instructions**
  - GETFIELD
    `objectref --> value`
  - PUTFIELD
    `objectref, value --> ..`
  - GETSTATIC
    `.. --> value`
  - PUTSTATIC
    `value --> ..`

- **Method instructions**
  - INVOKEVIRTUAL
    `objectref, [arg1, [arg2 ...]] --> ..`
  - INVOKEINTERFACE
    `objectref, [arg1, [arg2 ...]] --> ..`
  - INVOKESPECIAL
    `objectref, [arg1, [arg2 ...]] --> ..`
  - INVOKESTATIC
    `arg1, [arg2 ...] --> ..`

- **Variable instructions**
  - LLOAD
    `.. --> value`
  - FLOAD
    `.. --> value`

  - DLOAD
    `.. --> value`
  - ALOAD
    `.. --> value`
  - ILOAD
    `.. --> value`
  - LSTORE
    `value --> ..`
  - FSTORE
    `value --> ..`
  - DSTORE
    `value --> ..`
  - ASTORE
    `value --> ..`
  - ISTORE
    `value --> ..`
  - RET
    `.. --> ..`

- **Type instructions**
  - CHECKCAST
    `objectref --> objectref`
  - NEW
    `.. --> objectref`
  - ANEWARRAY
    `count --> arrayref`
  - INSTANCEOF

- **IInc instruction**
  - IINC
    `.. --> ..`
- **Int instructions**
  - BIPUSH
    `.. --> value`
  - SIPUSH
    `.. --> value`
  - NEWARRAY
    `count --> arrayref`
- **Jump instructions**
  - IFEQ
    `value(int) --> ..`
  - IFNE
    `value(int) --> ..`

- IFLT
  `value(int) --> ..`
- IFGE
  `value(int) --> ..`
- IFLE
  `value(int) --> ..`
- IF_ICMPEQ
  `value2, value1 --> ..`
- IF_ICMPNE
  `value2, value1 --> ..`
- IF_ICMPLT
  `value2, value1 --> ..`
- IF_ICMPGE
  `value2, value1 --> ..`
- IF_ICMPGT
  `value2, value1 --> ..`
- IF_ICMPLE
  `value2, value1 --> ..`
- IF_ACMPEQ
  `ref2, ref1 --> ..`
- IF_ACMPNE
  `ref2, ref1 --> ..`
- GOTO
  `.. --> ..`
- JSR
  `.. --> address`
- IFNULL
  `value --> ..`
- IFNONNULL
  `value --> ..`

- **Other instructions**
  For further bytecode instructions, please refer to [22].

# Appendix C

# Examples

## C.1 University

The university example is an extended version of the example given in section 3.2.

Sep 08, 06 18:08 **Building.java** Page 1/1

```
public class Building {

  private String name;

  public Building(String name){
    this.name = name;
  }

  public String getName(){
    return name;
  }
}
```

May 15, 06 17:01 **Room.java** Page 1/1

```
public class Room {

  private String name;

  public Room(String name){
    this.name = name;
  }

  public String getName(){
    return name;
  }
}
```

Tuesday September 26, 2006 Building.java, Room.java 1/1

May 16, 06 14:32 **Location.java** Page 1/1

```
public class Location {

  private Building building;
  private Room room;

  public Location(Building building, Room room){
    this.building = building;
    this.room = room;
  }
}
```

Sep 06, 06 16:27 **Student.java** Page 1/1

```
import java.util.*;
public class Student extends Person {


  private String studentNumber;
  private LinkedList courses;
  private int maxCourses;

  public Student(String studentNumber, String name, String address){
    super(name, address);
    this.studentNumber = studentNumber;
    this.courses = new LinkedList();
    this.maxCourses = 5;
  }

  public boolean enrolInCourse(Course course){
    if (courses.size() +1 <= maxCourses) {
      courses.add(course);
        if (course.enrolInCourse(this)) {
          return true;
        }
        return false;
    }
    else
      return false;
  }

  public String getStudentNumber(){
    return studentNumber;
  }
}
```

Tuesday September 26, 2006 Location.java, Student.java 1/1

Sep 12, 08 11:31 **Lecturer.java** Page 1/1

```java
import java.util.*;
public class Lecturer extends Person {

  private Location office;
  private LinkedList courses;
  private LinkedList colleagues;

  public Lecturer(Location office, String name, String address){
    super(name, address);
    this.office = office;
  }

  public void addCourse(Course course){
    courses.add(course);
  }

  public void getCourses(){
    for (int i=0; i<courses.size(); i++){
      Course course = (Course)courses.get(i);
      System.out.println(course);
    }
  }

  public void testReflexive(Lecturer colleague){
    colleagues = new LinkedList();
    if (colleague != this){
      colleagues.add(colleague);
    }
  }
}
```

Sep 12, 08 11:31 **Person.java** Page 1/1

```java

public class Person {

  private String name;
  private String address;

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

Tuesday September 26, 2006 Lecturer.java, Person.java 1/1

Sep 07, 08 15:45 **Course.java** Page 1/1

```
import java.util.*;
public class Course {

  private Lecturer lecturer;
  private Location location;
  private LinkedList participants;
  private int maxStudents;
  private String name;

  public Course(Lecturer lecturer, Location location){
    this.lecturer = lecturer;
    this.location = location;
    this.participants = new LinkedList();
    this.maxStudents = 30;
  }

  public boolean enrolInCourse(Student student){
    if (participants.size() + 1 <= maxStudents){
      participants.add(student);
      return true;
    }
    else
      return false;
  }

  public void getParticipants(){
    for (int i=0; i<participants.size(); i++){
      Student student = (Student)participants.get(i);
      System.out.println(student+" "+student.getStudentNumber());
    }
  }
}
```

Sep 13, 08 16:19 **University.java** Page 1/1

```
import java.util.LinkedList;

public class University {

  LinkedList courses;

  public void addCourse(Course course){
    courses.add(course);
  }

  public static void main(String[] args) {

    Building building = new Building("CAB");
    Room room = new Room("E22");
    Location location = new Location(building, room);
    Lecturer profMiller = new Lecturer(location, "Prof. Miller", "Haldeneggsteig 4, 8000 Zu
erich");

    Student studentMiller = new Student("03-333-555", "Henry Miller", "Bahnhofstrasse 14,
8037 Zuerich");
    Course course = new Course(profMiller, location);
    studentMiller.enrolInCourse(course);

  }

}
```

Tuesday September 28, 2008 Course.java, University.java 1/1

## C.2 Calendar

The calendar example is taken from the course material of “Introduction to programming in the large” at the ETH in Zurich.

Jun 08, 06 14:37 **Appointment.java** Page 1/3

```
/*
 * Appointment.java ---
 *     Sourcefile for class Appointment
 *     of the simple calendar manager.
 *     It implements the data container for a single appointment.
 */

/*
 * $Id: Appointment.java,v 1.1 2002/06/14 06:57:17 wwwlst Exp $
 */


import java.io.Serializable;
import java.util.Observable;

/**
 * A class for appointments in the simple calendar.
 */
public class Appointment
extends Observable
implements Serializable {

  /*
   * Appointment objects are observed by AppointmentView objects and
   * Day objects.
   */


  protected int startTime;
  protected int endTime;
  protected String description;
  protected String note;
  protected boolean destroyed;


  public Appointment(int startTime, int endTime, String description,
         String note) {
    this.startTime = startTime;
    this.endTime = endTime;
    this.description = description;
    this.note = note;
    this.destroyed = true;
  }

  /**
   * set all fields of this appointment. If any field changes, all
   * registered Observers are notified.
   */
  public void setAll(int startTime, int endTime, String description,
         String note) {
    if ((this.startTime != startTime) || (this.endTime != endTime)
  || (this.description != description) || (this.note != note)) {
      // Something has changed.
      setChanged();
    }
    this.startTime = startTime;
    this.endTime = endTime;
    this.description = description;
    this.note = note;
    this.destroyed = false;
    notifyObservers();
  }
```

Jun 08, 06 14:37 **Appointment.java** Page 2/3

```
  /**
   * Marks this appoint as being invalid.
   * All Observers will be notified and will realized that
   * they have to remove this appointment from their
   * display / data structure.
   */

  public void destroy() {
    destroyed = true;
    setChanged();
    notifyObservers();
  }

  /**
   * Called by Observers who check if this appointment
   * is still valid.
   */
  public boolean isValid() {
    return !destroyed;
  }

  /**
   * Get field startTime of this appointment.
   */
  public int getStartTime() {
    return startTime;
  }


  /**
   * Get field endTime of this appointment.
   */
  public int getEndTime() {
    return endTime;
  }


  /**
   * Get field description of this appointment.
   */
  public String getDescription() {
    return description;
  }


  /**
   * Get field note of this appointment.
   */
  public String getNote() {
    return note;
  }


  /**
   * Create a textual representation of this appointment.
   */
  public String toString() {
    StringBuffer s = new StringBuffer();
    if (startTime < 10) {
      s.append('0');
    }
    s.append(startTime);
    s.append(":00-");
    if (endTime < 10) {
```

Jun 08, 06 14:37 **Appointment.java** Page 3/3

```
      s.append('0');
    }
    s.append(endTime);
    s.append(":00 ");
    s.append(description);
    return s.toString();
  }

}
```

Sep 08, 06 16:00 **AppointmentView.java** Page 1/3

```java
/*
 * AppointmentView.java ---
 *    Sourcefile for class AppointmentView
 *    of the simple calendar manager.
 *    It is the user interface for a single Appointment.
 */

/*
 * $Id: AppointmentView.java,v 1.1 2002/06/14 06:57:40 wwwlst Exp $
 */


import java.awt.Button;
import java.awt.Choice;
import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.Panel;
import java.awt.TextArea;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Observable;
import java.util.Observer;


/**
 * Class that displays an appointment and allows editing its contents.
 */
public class AppointmentView extends Frame implements Observer, ActionListener{

  protected Appointment app;

  protected Choice startChoice;
  protected Choice endChoice;
  protected TextField descriptionField;
  protected TextArea noteArea;
  protected Button okButton;
  protected Button cancelButton;


  /**
   * Create a new AppointmentView for the given Appointment.
   */
  public AppointmentView(Appointment app) {
    int i;
    Panel p;

    this.app = app;

    startChoice = new Choice();
    endChoice = new Choice();
    for (i = 0; i < 24; i++) {
      startChoice.add(Integer.toString(i));
      endChoice.add(Integer.toString(i));
    }
    try {
      startChoice.select(app.getStartTime());
    } catch (IllegalArgumentException e) {
      startChoice.select(0);
    }
    try {
      endChoice.select(app.getEndTime());
```

Sep 08, 06 16:00 **AppointmentView.java** Page 2/3

```java
    } catch (IllegalArgumentException e) {
      endChoice.select(0);
    }
    descriptionField = new TextField(app.getDescription(), 20);
    noteArea = new TextArea(app.getNote(), 4, 20);
    okButton = new Button("OK");
    cancelButton = new Button("Cancel");
    okButton.addActionListener(this);
    cancelButton.addActionListener(this);
    // Define the layout.
    setLayout(new GridLayout(0,1));
    p = new Panel();
    p.add(new Label("Start Time:"));
    p.add(startChoice);
    add(p);
    p = new Panel();
    p.add(new Label("End Time:"));
    p.add(endChoice);
    add(p);
    p = new Panel();
    p.add(new Label("Description:"));
    p.add(descriptionField);
    add(p);
    p = new Panel();
    p.add(new Label("Note:"));
    p.add(noteArea);
    add(p);
    p = new Panel();
    p.add(okButton);
    p.add(cancelButton);
    add(p);
    pack();
    app.addObserver(this);
  }

  /**
   * main method to test class AppointmentView.
   */
  public static void main(String[] args) {
    // Just for testing this class.
    System.err.println("Testing class AppointmentView.");

    Appointment app =
      new Appointment(8,12,"Kurs", "Der erste Teil des Kurses");
    AppointmentView av1 = new AppointmentView(app);
    av1.show();
    AppointmentView av2 = new AppointmentView(app);
    av2.show();
  }


  /**
   * This method is called whenever the displayed appointment changes.
   */
  public void update(Observable o, Object arg) {
    if (o == app) {
      if (!app.isValid())
  this.dispose();
      else {
  try {
    startChoice.select(app.getStartTime());
  } catch (IllegalArgumentException e) {
    startChoice.select(0);
  }
```

Sep 08, 06 16:00 **AppointmentView.java** Page 3/3

```
  try {
    endChoice.select(app.getEndTime());
  } catch (IllegalArgumentException e) {
    endChoice.select(0);
  }
  descriptionField.setText(app.getDescription());
  noteArea.setText(app.getNote());
  pack();
      }
    }
  }


  /**
   * This method is called if the user clicks a button.
   */
  public void actionPerformed(ActionEvent e) {
    Object source = e.getSource();
    if (source == okButton) {
      app.setAll(startChoice.getSelectedIndex(),
     endChoice.getSelectedIndex(),
     descriptionField.getText(),
     noteArea.getText());
    }
    if ((source == okButton) || (source == cancelButton)) {
      if (!app.isValid())
  app.destroy();
      app.deleteObserver(this);
      this.dispose();
    }
  }

}
```

Jun 08, 06 14:37 **Calendar.java** Page 1/2

```
/*
 * Calendar.java ---
 *     Sourcefile for class Calendar
 *     of the simple calendar manager.
 *     It contains the main program.
 */

/*
 * $Id: Calendar.java,v 1.1 2002/06/14 06:58:01 wwwlst Exp $
 */


import java.io.FileInputStream;
import java.io.ObjectInputStream;

/**
 * This class is only used to hold the main method for the SimpleCalendar.
 */
public class Calendar {

  /**
   * The name of the file where the calendar is stored.
   */
  public static final String filename = "calendar.dat";


  /**
   * Read a month or create a new one and display it.
   */
  public static void main(String[] args) {
    if (args.length != 1) {
      usage();
      System.exit(1);
    }

    Month mon = null;
    Day day;
    if (args[0].equals("new")) {
      mon = new Month(31);
      int i;
      for (i = 0; i < mon.getNumDays(); i++) {
  day = new Day();
  mon.setDayAt(i, day);
      }
    } else if (args[0].equals("file")) {
      try {
  FileInputStream fis = new FileInputStream(filename);
  ObjectInputStream in = new ObjectInputStream(fis);
  // Read in an object. It should be an instance of class Month.
  mon = (Month) in.readObject();
  in.close();
      } catch (Exception ex) {
  System.err.println("Could NOT read calendar: " + ex);
  System.exit(1);
      }
    } else {
      usage();
      System.exit(1);
    }
    MonthView mv = new MonthView(mon, filename);
    mv.show();
  }
```

Jun 08, 06 14:37 **Calendar.java** Page 2/2

```
  /**
   * Explain how to invoke the program.
   */
  public static void usage() {
    System.err.println("usage: java Calendar new");
    System.err.println("       java Calendar file");
  }

}
```

Wednesday September 13, 2006 Calendar.java 1/1

Jun 08, 06 14:37 **Day.java** Page 1/3

```
/*
 * Day.java ---
 *    Sourcefile for class Day
 *    of the simple calendar manager.
 *    It implements the data container for days.
 */

/*
 * $Id: Day.java,v 1.1 2002/06/14 06:58:25 wwwlst Exp $
 */


import java.io.ObjectInputStream;
import java.io.IOException;
import java.io.Serializable;
import java.util.Observable;
import java.util.Observer;
import java.util.Enumeration;
import java.util.Vector;


/**
 * A class for days in the simple calendar.
 */
public class Day
extends Observable
implements Observer, Serializable {
  /*
   * Day objects are observed by DayView objects and observe
   * Appointment objects.
   */

  /**
   * All appointments of this day. The appointments are kept in
   * ascending order of their startTime.
   */
  protected Vector appointments;


  /**
   * Defines custom deserialization behaviour - calls method
   * initAfterDeserialization.
   */
  private void readObject(ObjectInputStream in)
      throws IOException, ClassNotFoundException
  {
    in.defaultReadObject();
    Enumeration e = appointments.elements();
    while (e.hasMoreElements()) {
      ((Observable) e.nextElement()).addObserver(this);
    }
  }


  /**
   * Create a new day object.
   */
  public Day() {
    appointments = new Vector();
  }


  /**
   * Add an appointment to this day. The appointment is inserted in the
```

Jun 08, 06 14:37 **Day.java** Page 2/3

```
   * appointment vector at the correct position and this day is added
   * to the observers of the appointment. Afterwards all observers of
   * this day have to be notified.
   */
  public void addAppointment(Appointment app) {
    int i;

    if (appointments.contains(app)) {
      // Avoid adding an appointment twice.
      return;
    }
    insertAppointmentAtCorrectPosition(app);
    // Tell me whenever this Appointment changes.
    app.addObserver(this);
    notifyObservers();
  }


  /**
   * Insert an appointment at the correct position into the vector
   * of the appointments of this day.
   */
  private void insertAppointmentAtCorrectPosition(Appointment app) {
    int i = 0;
    while ((i < appointments.size())
     && (((Appointment) appointments.elementAt(i)).getStartTime()
         < app.getStartTime())) {
      i++;
    }
    if (i == appointments.size()) {
      // End of vector reached - add at end.
      appointments.addElement(app);
    } else {
      // Insert in the middle of the vector.
      appointments.insertElementAt(app,i);
    }
    setChanged();
  }


  /**
   * Remove an appointment from this day. After it is removed, this day
   * no longer has to be an observer for the appointment. The observers
   * of this day also have to be notified that this day has changed.
   */
  public void removeAppointment(Appointment app) {
    if (appointments.removeElement(app)) {
      // app was removed.
      // We do no longer want to be an observer for app.
      app.deleteObserver(this);
      setChanged();
    }
    notifyObservers();
  }


  /**
   * Return an enumeration (iterator) for the appointments of this day.
   */
  public Enumeration elements() {
    return appointments.elements();
  }
```

Wednesday September 13, 2006 Day.java 1/2

Jun 08, 06 14:37 **Day.java** Page 3/3

```
  /**
   * Return the appointment at position i in this day.
   */
  public Appointment appointmentAt(int i) {
    return (Appointment) appointments.elementAt(i);
  }


  /**
   * This method is called whenever one of the observed Appointments
   * is changed. If the position of the appointment in the vector of
   * appointments is not correct anymore, it has to be adjusted. The
   * observers of this day have to be notified, too.
   */
  public void update(Observable o, Object arg) {
    // Check whether the changed appointment has to be inserted somewhere
    // else in the vector to keep the appointments in ascending order.
    int i = appointments.indexOf(o);
    if (i != -1) {
      Appointment app = (Appointment) o;
      if (app.isValid()) {
  boolean reinsert_required = false;
  if (i > 0) {
    // Compare with element i-1.
    if (app.getStartTime()
        < ((Appointment)appointments.elementAt(i-1)).getStartTime()) {
      reinsert_required = true;
    }
  }
  if (i+1 < appointments.size()) {
    // Compare with element i+1.
    if (app.getStartTime()
        > ((Appointment)appointments.elementAt(i+1)).getStartTime()) {
      reinsert_required = true;
    }
  }
  if (reinsert_required) {
    appointments.removeElement(app);
    insertAppointmentAtCorrectPosition(app);
  }
      } else
  appointments.removeElement(app);
    }
    // Now tell all my observers that I have changed.
    setChanged();
    notifyObservers();
  }

}
```

Jun 08, 06 14:38 **DayView.java** Page 1/3

```
/*
 * DayView.java ---
 *    Sourcefile for class DayView
 *    of the simple calendar manager.
 *    It is the user interface for a Day.
 */

/*
 * $Id: DayView.java,v 1.1 2002/06/14 06:58:44 wwwlst Exp $
 */


import java.awt.Button;
import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.List;
import java.awt.Panel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Enumeration;
import java.util.Observable;
import java.util.Observer;


/**
 * Class that displays a day and allows editing its contents.
 */
public class DayView
extends Frame
implements Observer, ActionListener {

  /**
   * The day that is displayed by this DayView.
   */
  protected Day day;

  protected List appList;
  protected Button editButton;
  protected Button addButton;
  protected Button deleteButton;
  protected Button closeButton;


  /**
   * Create a new DayView for the given Day.
   */
  public DayView(Day day) {
    Panel p;

    this.day = day;

    appList = new List(7);
    fillAppointmentList();
    editButton = new Button("Edit");
    editButton.addActionListener(this);
    addButton = new Button("Add");
    addButton.addActionListener(this);
    deleteButton = new Button("Delete");
    deleteButton.addActionListener(this);
    closeButton = new Button("Close");
    closeButton.addActionListener(this);
    // Define the layout.
    setLayout(new GridLayout(0,1));
    add(appList);
```

Jun 08, 06 14:38 **DayView.java** Page 2/3

```
    p = new Panel();
    p.add(editButton);
    p.add(addButton);
    p.add(deleteButton);
    p.add(closeButton);
    add(p);
    pack();
    day.addObserver(this);
  }


  /**
   * Fill the displayed list of appointments with the textual
   * representation of the respective appointments.
   */
  private void fillAppointmentList() {
    appList.removeAll();
    Enumeration en = day.elements();
    while (en.hasMoreElements()) {
      appList.add(en.nextElement().toString());
    }
  }


  /**
   * main method to test class DayView.
   */
  public static void main(String[] args) {
    // Just for testing this class.
    System.err.println("Testing class DayView.");

    Appointment app1 =
      new Appointment(8,12,"Kurs 1", "Der erste Teil des Kurses");
    Appointment app2 =
      new Appointment(13,17,"Kurs 2", "Der zweite Teil des Kurses");
    Day day = new Day();
    day.addAppointment(app2);
    day.addAppointment(app1);

    DayView dv1 = new DayView(day);
    dv1.show();
    DayView dv2 = new DayView(day);
    dv2.show();
  }


  /**
   * This method is called whenever the displayed day changes.
   */
  public void update(Observable o, Object arg) {
    fillAppointmentList();
  }


  /**
   * This method is called if the user clicks a button.
   */
  public void actionPerformed(ActionEvent e) {
    Object source = e.getSource();
    if (source == editButton) {
      int i = appList.getSelectedIndex();
      if (i != -1) {
  Appointment app = day.appointmentAt(i);
  AppointmentView av = new AppointmentView(app);
```

Wednesday September 13, 2006 DayView.java 1/2

Jun 08, 06 14:38 **DayView.java** Page 3/3

```java
  av.show();
      }
    } else if (source == addButton) {
      Appointment app = new Appointment(8, 9, "NEW", "");
      day.addAppointment(app);
      AppointmentView av = new AppointmentView(app);
      av.show();
    } else if (source == deleteButton) {
      int i = appList.getSelectedIndex();
      if (i != -1) {
  Appointment app = day.appointmentAt(i);
  app.destroy();
  // day.removeAppointment(app);
  // not necessary: day-object will be notified
  // by a canceled Appointement and will remove it
  // itself from its schedule.
      }
    } else if (source == closeButton) {
      day.deleteObserver(this);
      dispose();
    }
  }
}
```

Jun 08, 06 14:38 **Month.java** Page 1/1

```java
/*
 * Month.java ---
 *     Sourcefile for class Month
 *     of the simple calendar manager.
 *     It implements the data container for months.
 */

/*
 * $Id: Month.java,v 1.1 2002/06/14 07:01:16 wwwlst Exp $
 */


import java.io.Serializable;


/**
 * A class for months in the simple calendar.
 */
public class Month
implements Serializable
{

  /**
   * The days of this month.
   */
  protected Day[] days;


  /**
   * Create a new month with the given number of days.
   */
  public Month(int numDays) {
    days = new Day[numDays];
  }


  /**
   * Return the number of days of this month.
   */
  public int getNumDays() {
    return days.length;
  }


  /**
   * Set the day with the given number in this month.
   */
  public void setDayAt(int i, Day day) {
    days[i] = day;
  }


  /**
   * Return the day with the given number in this month.
   */
  public Day getDayAt(int i) {
    return days[i];
  }

}
```

Wednesday September 13, 2006 Month.java 1/1

```java
/*
 * MonthView.java ---
 *     Sourcefile for class MonthView
 *     of the simple calendar manager.
 *     It is the user interface for a Month.
 */

/*
 * $Id: MonthView.java,v 1.1 2002/06/14 07:01:51 wwwlst Exp $
 */


import java.awt.Button;
import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.Panel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;


/**
 * Class that displays a month and allows viewing its days.
 */
public class MonthView
extends Frame
implements ActionListener {


  /**
   * The month that is displayed by this MonthView.
   */
  protected Month mon;
  /**
   * The name of the file where to save the month.
   */
  protected String filename;

  protected Button saveButton;
  protected Button quitButton;
  protected Button[] dayButtons;


  /**
   * Create a new MonthView for the given Month.
   */
  public MonthView(Month mon, String filename) {
    Panel p;
    int i;

    this.mon = mon;
    this.filename = filename;
    p = new Panel();
    p.setLayout(new GridLayout(0,7));
    dayButtons = new Button[mon.getNumDays()];
    for (i = 0; i < mon.getNumDays(); i++) {
      dayButtons[i] = new Button(Integer.toString(i+1));
      dayButtons[i].addActionListener(this);
      p.add(dayButtons[i]);
    }
    // Define the layout.
```

```java
    setLayout(new GridLayout(0,1));
    add(p);
    p = new Panel();
    p.setLayout(new GridLayout(0,2));
    saveButton = new Button("Save");
    saveButton.addActionListener(this);
    p.add(saveButton);
    quitButton = new Button("Exit");
    quitButton.addActionListener(this);
    p.add(quitButton);
    add(p);
    pack();
  }




  /**
   * This method is called if the user clicks a button.
   */
  public void actionPerformed(ActionEvent e) {
    Object source = e.getSource();
    int i;
    if (source == quitButton) {
      this.dispose();
      System.exit(0);
    } else if (source == saveButton) {
      try {
  FileOutputStream fos = new FileOutputStream(filename);
  ObjectOutputStream out = new ObjectOutputStream(fos);
  out.writeObject(mon);
  out.flush();
  out.close();
      } catch (IOException ex) {
  System.err.println("Save NOT successful: " + ex);
      }
    } else {
      // one of the day buttons.
      i = 0;
      while ((i < mon.getNumDays()) && source != dayButtons[i]) {
  i++;
      }
      if (i < mon.getNumDays()) {
  Day day = mon.getDayAt(i);
  if (day != null) {
    DayView dv = new DayView(day);
    dv.show();
  }
      }
    }
  }

}
```

# List of Tables

# List of Figures