Recovering Binary Class Relationships: Putting Icing on the UML Cake

Yann-Gaël Guéhéneuc*
Université de Montréal
Montréal, Québec, Canada
guehene@iro.umontreal.ca

Hervé Albin-Amiot†
École des Mines de Nantes
Nantes, France
albin@emn.fr

ABSTRACT

A discontinuity exists between object-oriented modeling and programming languages. This discontinuity arises from ambiguous concepts in modeling languages and a lack of corresponding concepts in programming languages. It is particularly acute for binary class relationships—association, aggregation, and composition. It hinders the traceability between software implementation and design, thus hampering software analysis. We propose consensual definitions of the binary class relationships with four minimal properties—exclusivity, invocation site, lifetime, and multiplicity. We describe algorithms to detect automatically these properties in source code and apply these on several frameworks. Thus, we bridge the gap between implementation and design for the binary class relationships, easing software analysis.

Categories and Subject Descriptors

D.1.5 [**Programming techniques**]: Object-oriented programming; D.2.1 [**Software engineering**]: Requirements and–or specifications—*Languages*; D.2.7 [**Software engineering**]: Distribution, maintenance, and enhancement—*Restructuring, reverse engineering, and reengineering*;

General Terms

Design, standardisation, languages, theory.

Keywords

Design–implementation discontinuity, binary class relationships, consensual definitions, formalisations, minimal properties, detection algorithms, UML, Java.

*This work has been partly funded by IBM OTI Labs – 2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada.
†This work is partly funded by Soft-Maint – 4, rue du Chateau de l'Éraudière – 44 324 Nantes – France.

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

1. INTRODUCTION

Design models, such as class diagrams, describe the architecture of programs at a higher level of abstraction than source code. These models are an invaluable help for software engineers—both developers and maintainers—to analyse programs architectures, design choices, behaviours, and implementations.

However, design models produced during the design phase are often forgotten during the implementation phase—under time pressure usually—and thus present major discrepancies with their actual implementation frequently. Such divergent models are of little help to software engineers who must understand or analyse a program during its development and after its release. A recurrent problem in the object-oriented software (re)engineering community is the automated recovery of design models from a program implementation.

Modeling and programming languages possess similar concepts with similar semantics, such as class and inheritance, guaranteeing a continuity between implementation and design. However, modeling languages aspire to provide higher-level abstractions and thus include concepts absent from programming languages, in particular binary class relationships. The very existence of binary class relationships in modeling languages brings a discontinuity in the transition between implementation and design.

This discontinuity hinders the understanding of a program implementation, limits the capabilities and efficiency of automated reverse engineering tools, and impedes the communication among software engineers. Definitions of binary class relationships proposed by modeling languages should answer the problem of discontinuity. However, most definitions present ambiguities, as stated for example by Henderson-Sellers and Barbier [26], and offer no operational semantics, *i.e.*, no implementation choices [11, 23]. It is essential to bridge the gap between programming and modeling languages for binary class relationships to build sound foundations for understanding, round-tripping [9, page 517], design pattern identification...

A first solution to bridge the gap is to design a new programming language (or an extension to an existing one) including binary class relationships, for examples [16, 24, 30]. However, this solution uses a non-standard programming language and thus eliminates the benefits of standardisation (debuggers, development environments, efficient compilers). A second solution is to define binary class relationships at implementation level, in terms of constructs of an existing programming language. This solution relies on definitions

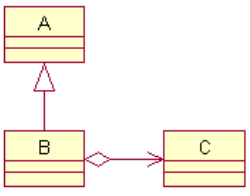
of the binary class relationships at design and implementation levels and on detection algorithms, which bring continuity between implementation and design. We apply this second solution to the Java programming language with a pragmatic study of UML association, aggregation, and composition relationships.

In section 2, we detail the problem and summarise previous work. In section 3, we propose consensual definitions of the binary class relationships and discuss their properties at implementation level. In section 4, we focus on four common properties: Exclusivity, invocation site, lifetime, and multiplicity. In section 5, we formalise the definitions with the properties and show that the properties form a minimal set. In section 6, we propose algorithms to detect binary class relationships using their properties. These algorithms bring continuity between programming and modeling languages. In sections 7, we introduce and validate implementations of our algorithms on several frameworks. In section 8, we conclude and present future work.

2. QUESTION AND PREVIOUS ANSWERS

We study definitions of binary class relationships at design and implementation levels to bridge the gap between implementation and design. We consider only unidirectional association, aggregation, and composition relationships because these relationships exist in common modeling languages, such as the UML, but are not explicit in standard programming languages, such as Java. Moreover, bidirectional relationships may be expressed as two unidirectional relationships, as we further discuss in section 5.4, after the formalisations of the relationships.

The UML class diagram in Figure 1(a) defines three classes `A`, `B`, and `C`, an inheritance relationship from `B` to `A`, and an aggregation relationship between `B` and `C`. Figure 1(b) shows Java source code corresponding to the classes `A`, `B`, and `C`. While we can identify easily the inheritance relationship in Java (`extends` keyword), how would the aggregation relationship between class `B` and class `C` be expressed? As a field? As a collection? How would the implementation reflect an aggregation or a composition relationship? More generally: **How to define binary class relationships so we can detect them in implementation?**



(a) A UML class diagram

```
public class A {
}
public class B extends A {
    ...
}
public class C {
}
```

(b) Possible implementation for the UML class diagram in Figure 1(a).

Figure 1: Illustration of the question.

A lot of work exists on binary class relationships. However, none of this work tackles the discontinuity between implementation and design and answers the previous question. We only present here four lines of work typifying mainstream researches on binary class relationships. We further detail work related to definitions of binary class relationships when discussing their formalisations in section 5. The interested reader may report to a previous work [21] for a more complete depiction of the state of the art.

Some researchers studied binary class relationships without linking their results to implementation-level constructs, such as [10, 11, 14, 25, 26, 34]. For example, in their survey of the UML relationships [26], Henderson-Sellers and Barbier proposed a set of characteristics for whole–part relationships. They showed that definitions of the aggregation and composition relationships are incomplete, overlapping, and contradictory, and proposed revised definitions. However, they did not link their definitions with implementation.

Other researchers formalised object-oriented programming languages without focusing on implementation issues [1, 4, 10, 32]. For example, Abadi and Cardelli "create a simple model that is flexible enough to represent more complex notions [of object classification and inheritance], but that does not directly embody any particular one" [1, page 51]. They developed a type calculus for object-oriented programming languages, focusing on objects, classes, inheritance, and delegation. They did not consider more complex relationships.

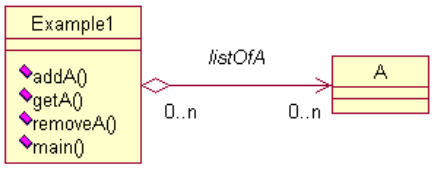
Yet, some other researchers only considered subsets of the binary class relationships and of their definitions at implementation level, such as [12, 27, 29, 31]. For example, Jackson and Waingold developed a tool, Womble, for lightweight extraction of object-models from Java byte-codes [27]. They proposed algorithms for detecting association relationships between classes and heuristics to infer their multiplicity and mutability properties and to deal with container classes. The authors limited their research to the association relationship: They only mentioned aggregation relationships and did not discuss detection of composition relationships.

Currently, industrial and open-source CASE tools, such as Rational Rose and ArgoUML, do not clearly define binary class relationships. They distinguish graphically the association, aggregation, and composition relationships, but their reverse engineering algorithms produce erroneous or inconsistent relationships. For example, the UML diagram in Figure 2(a) represents two classes `Example1` and `A` that an aggregation relationship links together. We could implement this diagram with the source code in Figure 2(b).

If we reverse-engineer with Rose v2002.05.00 the source code in Figure 2(b), the original aggregation relationship is *not* detected. The class diagram recovered from the source code displays an association relationship between classes `Example1` and `A`, as shown in Figure 3.

We could replace the array by a collection from the Java collection hierarchy, for example the `List` interface and its `ArrayList` implementation, as in Figure 4(a). Figure 4(b) shows the class diagram recovered using Rose for the alternate implementation.

The aggregation relationship still does not exist, an association relationship links the `Example2` class and the collection class, which is inconsistent with the expected class diagram. An implementation detail—the use of a collection—erroneously changes the class diagram representing the program design because of the lack of continuity between implementation and design for binary class relationships.



(a) UML class diagram.

```
public class Example1 {
    private A[] listOfAs = new A[10];
    private int numberOfAs = 0;

    public void addA(final A a) {
        // Complete implementation not shown here.
        this.listOfAs[numberOfAs++] = a;
    }
    public A getA(final int index) {
        return this.listOfAs[index];
    }
    public void removeA(final A a) {
        // Complete implementation not shown here.
    }
    public static void main(final String[] args) {
        final Example1 example1 = new Example1();
        example1.addA(new A());
        // ...
    }
}
```

(b) Implementation of the diagram in Figure 2(a).

Figure 2: A simple case study.

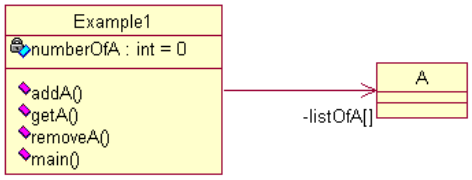


Figure 3: Class diagram recovered from the source code in Figure 2(b) using Rational Rose.

Thus, to our best knowledge, previous work focused on different problems related to binary class relationships: Definitions and properties at design level, formalisations and detections of a subset of the relationships. We build on this work and further study the association, aggregation, and composition relationships to bridge the gap between implementation and design.

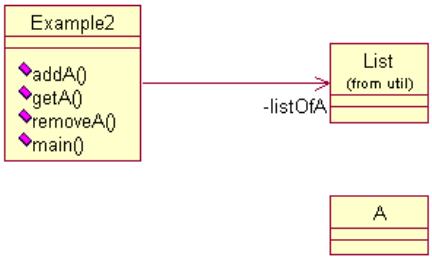
3. DEFINITIONS OF THE BINARY CLASS RELATIONSHIPS

We propose definitions of the association, aggregation, and composition relationships at design and implementation level. We advocate that these definitions are as consensual as possible with the literature. Our approach requires that one accepts our definitions, however, beyond the definitions, our approach shows that it is possible to bridge software implementation and design.

```
public class Example2 {
    private List listOfAs = new ArrayList();

    public void addA(final A a) {
        this.listOfAs.add(a);
    }
    public A getA(final int index) {
        return (A) this.listOfAs.get(index);
    }
    public void removeA(final A a) {
        this.listOfAs.remove(a);
    }
    public static void main(final String[] args) {
        final Example2 example2 = new Example2();
        example2.addA(new A());
        // ...
    }
}
```

(a) Alternate implementation of the case study.



(b) Class diagram recovered from the source code in Figure 4(a) using Rational Rose.

Figure 4: Other implementation of the case study.

3.1 Association Relationship

At design level, an association relationship is a conceptual link between two classes. Each class can have multiple instances involved in the relationship.

At implementation level, most authors agree that an association relationship involves instances of two classes, an origin and a target. An association relationship is oriented, irreflexive, anti-symmetric at instance and class level, and asymmetric at instance level [26].

Thus, we propose that an association relationship between classes `A` and `B` be the ability of an instance of `A` to send a message to an instance of `B`. Nothing prevents other relationships to link classes `A` and `B`: An association, an aggregation, or a composition relationship may exist between `A` and `B` (as well as between `B` and `A`).

The following source code represents two classes `A` and `B` linked by an association relationship, through the `void operation1(B)` method.

```
public class AssociationExample1 {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.operation1(b);
        b.operation2();
        ...
    }
}
public class A {
    public void operation1(B b) {
        b.operation2();
    }
}
public class B {
    public void operation2() {
    }
}
```

An alternate implementation of the association relationship between classes `A` and `B` could use a declaration of a local variable in a method of class `A`, for example:

```
void operation3() {
    B anotherB = new B();
    anotherB.operation2();
}
```

3.2 Aggregation Relationship

At design level, an aggregation relationship is an association between two classes, respectively *whole* and *part*.

At implementation level, we say that an aggregation relationship exists between classes `A` and `B` when the definition of `A`, the whole, contains instances of `B`, its part.

The whole must define a field ("simple", array, or collection) of the type of its part. Instances of the whole send messages to instances of its part. Subclasses inherit aggregation relationships, because subclasses inherit the structure and behaviour of their superclasses, with respect to access-control limitations [5].

The following source code represents two classes `A` and `B` linked by an aggregation relationship. The aggregation relationship exists through the field `B b` and the `void operation1()` method body.

```
public class AggregationExample1 {
    public static void main(String[] args) {
        B b = new B();
        A a = new A(b);
        a.operation1();
        b.operation2();
        ...
    }
}
public class A {
    private B b;
    public A(B b) {
        this.b = b;
    }
    public void operation1() {
        this.b.operation2();
    }
}
public class B {
    public void operation2() {
    }
}
```

The aggregation relationship could be implemented also using a collection of instances of class `B`, for example:

```
private List listOfBs;
public void operation3() {
    ((B) listOfBs.get(0)).operation2();
}
```

3.3 Composition Relationship

At design level, a composition relationship is an aggregation relationship which parts are destroyed when the whole is destroyed. Parts might be instantiated before the whole is instantiated and they can be exchanged during the lifetime of the whole. All parts owned by a whole at the moment of its destruction are also destroyed.

At implementation level, we define a composition relationship as an aggregation relationship with a constraint between the lifetimes of the whole and of its part and a constraint on the ownership of the part by the whole. Instances of the whole own the instances of its part. Instances of the part might be instantiated before the whole is instantiated, but they must not belong to any other whole. They are exclusive to the instance of the whole. The definition of the composition relationship allows only an association relationship between part and whole, to ensure the lifetime and ownership properties between whole and part.

The following code represents two classes `A` and `B` linked by a composition relationship. The composition relationship exists through the private field `B b`, the `void operation1()` method, and the Java virtual machine garbage-collector. The instance of class `B` is garbage-collected[1] along with the instance of class `A`. The privateness of field `B b` and the absence of methods returning a reference on this field participate in the lifetime dependency and in the exclusive ownership between instances of `A` and `B`.

```
public class CompositionExample1 {
    public static void main(String[] args) {
        A a = new A();
        a.operation1();
        ...
    }
}
public class A {
    private B b = new B();
    public void operation1() {
        this.b.operation2();
    }
}
public class B {
    public void operation2() {
    }
}
```

As with an aggregation relationship, the composition relationship could be implemented using a collection also.

3.4 Discussion of the Definitions

The definitions at implementation level of the binary class relationships use four language-independent properties, subset of all existing properties of the relationships. Association and aggregation relationships allow *multiple* instances of `A` and `B` to take part in the relationships, while the composition relationship allows multiple instances of `B` to be in a relationship with one instance of `A` at a time. In an aggregation relationship, instances of `A` access instances of `B` through a particular *invocation site*: A field. In a composition relationship, instances of `B` are *exclusive* to their corresponding instances of `A` and instances of `A` and `B` have related *lifetimes*.

[1] Garbage-collection of instances of class `B` happens actually before *or* after the garbage-collection of instances of class `A`.

4. PROPERTIES OF THE BINARY CLASS RELATIONSHIPS

We detail and discuss now the four properties of the binary class relationships at implementation level.

4.1 Exclusivity Property

An instance of class `B` involved at a given time in a relationship with an instance of class `A` can, or cannot, be in another relationship at the same time. We name $\mathbb{B}$ the set $\{true, false\}$. We define the exclusivity property as:

$$EX : Class \times Class \rightarrow \mathbb{B}$$

Given `A` and `B`, two classes: $EX(\texttt{A}, \texttt{B}) \in \{true, false\}$. Value *true* states that an instance of class `B` can take part in another relationship with another instance of class `A` or of another class. Value $false$ indicates that it cannot. The exclusivity property only holds at a given time and it does not prevent possible transferals.

For example, in a `Country`–`Language` relationship, a `Language` may be part of more than one `Country`: $EX($`Country`, `Language`$) = false$. In a `Computer`–`Keyboard` relationship, a `Keyboard` is part of one and only one `Computer` at a given time: $EX($`Computer`, `Keyboard`$) = true$.

4.2 Invocation-Site Property

Instances of class `A`, involved in a relationship, send messages to instances of class `B`. We name *any* the set of all possible invocation sites.

$$any = \{\texttt{field}, \texttt{array field}, \texttt{collection field}, \texttt{parameter}, \texttt{array parameter}, \texttt{collection parameter}, \texttt{local variable}, \texttt{local array}, \texttt{local collection}\}$$

We distinguish three levels of invocation sites: Fields, parameters, and local variables. Also, we distinguish "simple" invocation sites, arrays, and collections because they imply different sets of programming idioms for their declarations and for their uses, which we need to individualise when detecting the relationships, as we further discuss in section 6. The set *any* of invocation sites is language-independent and its elements map to source code constructs in class-based programming languages, such as C++ (whether using pointers or not, as detailed in section 6.4), Java, and Smalltalk. We define the invocation-site property as:

$$IS : Class \times Class \subseteq any$$

Given `A` and `B`, two classes: $IS(\texttt{A}, \texttt{B}) \subseteq any$. Values of the IS property describe the invocation sites for messages sent from instances of class `A` to instances of class `B`. There can be no message sent from class `A` to class `B`: $IS(\texttt{A}, \texttt{B}) = \emptyset$, or messages can be sent from `A` through a `field` (respectively a `parameter`, a `local variable`) of type `B`, an `array field`, or a field of type `collection`.

For example, in a `Figure`–`Line` relationship, the `Figure` propagates `draw()` messages to its composing lines using a collection of instances of `Line`: $IS($`Figure`, `Line`$) =$ {`collection field`}. A `Line` should not know its enclosing `Figure`: $IS($`Line`, `Figure`$) = \emptyset$ [8]. If class `Figure` declares a `contains(Line[])` method (to check whether a figure encloses a set of lines), then $IS($`Line`, `Figure`$) =$ {`array parameter`, `collection field`}.

4.3 Lifetime Property

The lifetime property constrains the lifetimes of all instances of class `B` with respect to the lifetimes of all instances of class `A`. It relates to the difference between the times of destruction LT_d of two instances of classes `A` and `B` [14]. The time is in any convenient unit, in seconds, in CPU ticks.

$$LT_d : Instance \rightarrow \mathbb{N}$$

In programming languages with garbage collection, LT_d matches the moment where an instance is ready to be collected for garbage. We infer from LT_d a relation between the lifetimes of all instances of two classes `A` and `B`. We name $\|$ the set $\{-, +\}$.

$$LT : Class \times Class \rightarrow \|$$

Given `A` and `B`, two classes: $LT(\texttt{A}, \texttt{B}) \in \{-, +\}$. $LT(\texttt{A}, \texttt{B}) = +$ if *all* instances of class `B` are destroyed before the corresponding instances of class `A`, $LT(\texttt{A}, \texttt{B}) = -$ if destroyed after. We shall write $LT(\texttt{A}, \texttt{B}) \in \|$ if the times of destruction of instances of classes `A` and `B` are unspecified: The property does not state if instances of class `B` are destroyed before *or* after instances of class `A`.

For example, in a `Window`–`Button` relationship, when the `Window` closes and is ready for garbage collection, the `Button` must also be ready for garbage collection: $LT($`Window`, `Button`$) = +$.

4.4 Multiplicity Property

The multiplicity property specifies the number of instances of class `B` allowed in a relationship with class `A`. We express this property as:

$$MU : Class \times Class \subset \mathbb{N} \cup \{+\infty\}$$

Given `A` and `B`, two classes: $MU(\texttt{A}, \texttt{B}) \subset \mathbb{N} \cup \{+\infty\}$. For the sake of simplicity, we use an interval of the minimum and maximum numbers to represent multiplicity. We only consider multiplicity at the *target* end of a relationship. The interested reader may refer to Jackson and Waingold [27] for a discussion on multiplicity at both ends of a relationship.

For example, in a `Cell`–`DNACode` relationship, a `Cell` possesses one and only one `DNACode`: $MU($`Cell`, `DNACode`$) = [1, 1]$. In a `Car`–`Wheel` relationship, a `Car` usually possesses four `Wheel`s but may have a fifth spare `Wheel`: $MU($`Car`, `Wheel`$) = [4, 5]$.

4.5 Discussion of the Properties

The four properties, which we propose to define binary class relationships, are orthogonal. However, the exclusivity and multiplicity properties are closely related with each other. For example, in the `Country`–`Language` relationship:

- The multiplicity property states the number of instances of class `Language` that each instance of the `Country` class possesses: $MU($`Country`, `Language`$) = [1, +\infty]$. (Canada possesses two official languages, English and French, and several spoken languages: Inuktitut; Punjabi; Portuguese...)
- The exclusivity property states if an instance of class `Language` is shared among instances of class `Country` or of other classes: $EX($`Country`, `Language`$) = false$. (French is spoken in France, in Canada...)

Also, we show that $LT(\texttt{A}, \texttt{B}) = + \Leftrightarrow LT(\texttt{B}, \texttt{A}) = -$, indeed $LT(\texttt{A}, \texttt{B}) = + \Leftrightarrow LT_d(\texttt{A}) > LT_d(\texttt{B}) \Leftrightarrow LT(\texttt{B}, \texttt{A}) = -$.

5. FORMALISATIONS OF THE BINARY CLASS RELATIONSHIPS

We formalise the binary class relationships at implementation level as three conjunctions of the four properties, respectively AS, AG, and CO. We show examples and discuss characteristics of these formalisations, in particular the minimality of the set of properties.

Formalisations of the binary class relationships are important in two respects: First, they provide formal language-independent definitions of the relationships for understanding and communication among software engineers; Second, they are the basis of the detection algorithms needed to bridge the gap between implementation and design, as presented in section 6.

5.1 Association Relationship

We define an association relationship between classes `A` and `B`, $AS(\mathtt{A},\mathtt{B})$, as:

$$\begin{array}{rcll} AS(\mathtt{A},\mathtt{B}) = & & & \\ (EX(\mathtt{A},\mathtt{B}) \in \mathbb{B}) & \wedge & (EX(\mathtt{B},\mathtt{A}) \in \mathbb{B}) & \wedge \\ (IS(\mathtt{A},\mathtt{B}) = any) & \wedge & (IS(\mathtt{B},\mathtt{A}) = \emptyset) & \wedge \\ (LT(\mathtt{A},\mathtt{B}) \in \|) & \wedge & (LT(\mathtt{B},\mathtt{A}) \in \|) & \wedge \\ (MU(\mathtt{A},\mathtt{B}) = [0,+\infty]) & \wedge & (MU(\mathtt{B},\mathtt{A}) = [0,+\infty]) & \end{array}$$

We forbid message invocation from instances of class `B` to instances of class `A` because we consider unidirectional relationships only. Bidirectional relationships may be expressed as two unidirectional relationships, as we show in section 5.4.

For example, the following source code, incomplete as it is, verifies the values of the properties of the association relationships. The naming of instances `a` and `b` precludes their mutual exclusivity: They can be used later in the `main()` method body. Possible uses of instances `a` and `b` also suggests unrelated lifetimes.

```
public class AssociationExample2 {
    public static void main(String[] args) {
        A a = new A(); // EX(A, B) = EX(B, A) ∈ 𝔹
        // because of the possible later use of a and b.
        B b = new B(); // LT(A, B) = LT(B, A) ∈ ‖
        a.operation1(b); // MU(A, B) = MU(B, A) = [0, +∞]
        b.operation2();
        ...
        // Possible uses of a and b.
    }
}
public class A {
    public void operation1(B b) {
        b.operation2(); // IS(A, B) ⊂ any
    }
}
public class B {
    public void operation2() { // IS(B, A) = ∅
    }
}
```

5.2 Aggregation Relationship

We define the aggregation relationship between classes `A` and `B`, $AG(\mathtt{A},\mathtt{B})$, as:

$$\begin{array}{rcll} AG(\mathtt{A},\mathtt{B}) = & & & \\ (EX(\mathtt{A},\mathtt{B}) \in \mathbb{B}) & \wedge & (EX(\mathtt{B},\mathtt{A}) \in \mathbb{B}) & \wedge \\ (IS(\mathtt{A},\mathtt{B}) & \subseteq & \{\mathtt{field}, \mathtt{array\ field}, & \\ & & \mathtt{collection\ field}\}) & \wedge \\ (IS(\mathtt{B},\mathtt{A}) & = & \emptyset) & \wedge \\ (LT(\mathtt{A},\mathtt{B}) \in \|) & \wedge & (LT(\mathtt{B},\mathtt{A}) \in \|) & \wedge \\ (MU(\mathtt{A},\mathtt{B}) = [0,+\infty]) & \wedge & (MU(\mathtt{B},\mathtt{A}) = [1,+\infty]) & \end{array}$$

The following source code illustrates an aggregation relationship between classes `A` and `B`. For example, the field `B b` sets the lower bound of $MU(\mathtt{A},\mathtt{B})$ to 1, possibility of other field declarations sets the upper bound to $+\infty$.

```
public class AggregationExample2 {
    public static void main(String[] args) {
        B b = new B(); // EX(A, B) ∈ 𝔹
        A a = new A(b); //EX(B, A) ∈ 𝔹
        a.operation1(); // LT(A, B) ∈ ‖
        b.operation2(); // LT(B, A) ∈ ‖
        ...
        // Possible uses and creations of instances of B.
    }
}
public class A {
    private B b; // MU(A, B) = [1, +∞] ⊂ [0, +∞]
    public A(B b) {
        this.b = b; // MU(B, A) = [1, 1] ⊂ [1, +∞]
    }
    public void operation1() {
        this.b.operation2(); // IS(A, B) = {field}
        // ⊂ {field, array field, collection field}
    }
    ...
    // Possible declarations and uses of instances of B.
}
public class B {
    public void operation2() { // IS(B, A) = ∅
    }
}
```

5.3 Composition Relationship

We define the composition relationship between classes `A` and `B`, $CO(\mathtt{A},\mathtt{B})$, as:

$$\begin{array}{rcll} CO(\mathtt{A},\mathtt{B}) = & & & \\ (EX(\mathtt{A},\mathtt{B}) = true) & \wedge & (EX(\mathtt{B},\mathtt{A}) = false) & \wedge \\ (IS(\mathtt{A},\mathtt{B}) & \subseteq & \{\mathtt{field}, \mathtt{array\ field}, & \\ & & \mathtt{collection\ field}\}) & \wedge \\ (IS(\mathtt{B},\mathtt{A}) & = & \emptyset) & \wedge \\ (LT(\mathtt{A},\mathtt{B}) = +) & \wedge & (LT(\mathtt{B},\mathtt{A}) = -) & \wedge \\ (MU(\mathtt{A},\mathtt{B}) = [1,+\infty]) & \wedge & (MU(\mathtt{B},\mathtt{A}) = [1,1]) & \end{array}$$

The following code represents a composition relationship between classes `A` and `B`. For example, the value of $MU(\mathtt{A},\mathtt{B})$ is $[1,1]$ because the instance of class `B` can belong to one and only one instance of class `A`.

```
public class CompositionExample2 {
    public static void main(String[] args) {
        A a = new A(); // EX(B, A) = false
        // because of the possible later creation
        // and use of other instances of A.
        a.operation1(); // (LT(A, B) = +) ∧ (LT(B, A) = −)
        ...
        // Possible uses and creations of instances of A.
    }
}
public class A {
    private B b = new B(); // MU(A, B) = [1, 1] ⊂ [1, +∞]
    public void operation1() { // MU(B, A) = [1, 1]
        this.b.operation2(); // EX(A, B) = true
    } // IS(A, B) = {field} ⊂ {field, array field,
    // collection field}
}
public class B {
    public void operation2() { // IS(B, A) = ∅
    }
}
```

5.4 Discussion of the Formalisations

Interesting characteristics of the binary class relationships stem from their formalisations: Minimality of the set of properties; Order among the relationships; Decomposition of the relationships in static and dynamic parts. Also, our formalisations support common usages of binary class relationships: Acquaintance and optimisations; Aggregation and shared aggregation; Navigability; Qualification; Symmetrical relationships and backpointers.

Minimality of the Set of Properties. The four properties form a subset of all properties proposed in the literature, for examples in [14, 26]. We prove that this subset $\mathcal{P} = \{EX, IS, LT, MU\}$ is minimal with respect to our formalisations and to other definitions in two steps.

First, we show that this subset is minimal with respect to our formalisations: If we remove any property from $\mathcal{P}$, then some of our formalisations are undistinguishable.

Let us assume a composition relationship $CO(\mathtt{A}, \mathtt{B})$ between classes A and B. If we remove the EX property from $\mathcal{P}$, then we can no longer distinguish $CO(\mathtt{A}, \mathtt{B})$ from an aggregation relationship, $AG(\mathtt{A}, \mathtt{B})$. Indeed, the values of the properties for $CO(\mathtt{A}, \mathtt{B})$ minus EX satisfy the values of the properties for $AG(\mathtt{A}, \mathtt{B})$:

$$\begin{aligned}
(IS(\mathtt{A},\mathtt{B}) \subseteq \{\ldots\})_{CO} &= (IS(\mathtt{A},\mathtt{B}) \subseteq \{\ldots\})_{AG} \\
(IS(\mathtt{B},\mathtt{A}) = \emptyset)_{CO} &= (IS(\mathtt{B},\mathtt{A}) = \emptyset)_{AG} \\
(LT(\mathtt{A},\mathtt{B}) = +)_{CO} &\in (LT(\mathtt{A},\mathtt{B}) \in \|)_{AG} \\
(LT(\mathtt{B},\mathtt{A}) = -)_{CO} &\in (LT(\mathtt{B},\mathtt{A}) \in \|)_{AG} \\
(MU(\mathtt{A},\mathtt{B}) = [1,+\infty])_{CO} &\subset (MU(\mathtt{A},\mathtt{B}) = [0,+\infty])_{AG} \\
(MU(\mathtt{B},\mathtt{A}) = [1,1])_{CO} &\subset (MU(\mathtt{B},\mathtt{A}) = [1,+\infty])_{AG}
\end{aligned}$$

Now, let us assume an aggregation relationship $AG(\mathtt{A}, \mathtt{B})$ between classes A and B. If we remove the IS property from $\mathcal{P}$, then we can no longer distinguish $AG(\mathtt{A}, \mathtt{B})$ from an association relationship, $AS(\mathtt{A}, \mathtt{B})$. Indeed, the values of the properties for $AG(\mathtt{A}, \mathtt{B})$ minus IS satisfy the values of the properties for $AS(\mathtt{A}, \mathtt{B})$:

$$\begin{aligned}
(EX(\mathtt{A},\mathtt{B}) \in \mathbb{B})_{AG} &= (EX(\mathtt{A},\mathtt{B}) \in \mathbb{B})_{AS} \\
(EX(\mathtt{B},\mathtt{A}) \in \mathbb{B})_{AG} &= (EX(\mathtt{B},\mathtt{A}) \in \mathbb{B})_{AS} \\
(LT(\mathtt{A},\mathtt{B}) \in \|)_{AG} &= (LT(\mathtt{A},\mathtt{B}) \in \|)_{AS} \\
(LT(\mathtt{B},\mathtt{A}) \in \|)_{AG} &= (LT(\mathtt{B},\mathtt{A}) \in \|)_{AS} \\
(MU(\mathtt{A},\mathtt{B}) = [0,+\infty])_{AG} &= (MU(\mathtt{A},\mathtt{B}) = [0,+\infty])_{AS} \\
(MU(\mathtt{B},\mathtt{A}) = [1,+\infty])_{AG} &\subset (MU(\mathtt{B},\mathtt{A}) = [0,+\infty])_{AS}
\end{aligned}$$

Likewise, let us assume a composition relationship between classes A and B,$CO(\mathtt{A}, \mathtt{B})$. If we remove the LT property from $\mathcal{P}$, then we can no longer distinguish $CO(\mathtt{A}, \mathtt{B})$ from an aggregation relationship, $AG(\mathtt{A}, \mathtt{B})$. The values of the IS and MU properties for $CO(\mathtt{A}, \mathtt{B})$ satisfy the values of these properties for $AG(\mathtt{A}, \mathtt{B})$, as do the values of the EX property:

$$\begin{aligned}
(EX(\mathtt{A},\mathtt{B}) = true)_{CO} &\in (EX(\mathtt{A},\mathtt{B}) \in \mathbb{B})_{AG} \\
(EX(\mathtt{B},\mathtt{A}) = false)_{CO} &\in (EX(\mathtt{B},\mathtt{A}) \in \mathbb{B})_{AG}
\end{aligned}$$

Finally, let us assume an aggregation relationship $AG(\mathtt{A}, \mathtt{B})$ between classes A and B. If we remove the MU property from $\mathcal{P}$, then we can no longer distinguish $AG(\mathtt{A}, \mathtt{B})$ from an association relationship, $AS(\mathtt{A}, \mathtt{B})$. The values of the EX and LT properties for $AG(\mathtt{A}, \mathtt{B})$ satisfy the values of these properties for $AS(\mathtt{A}, \mathtt{B})$, as do the values of the IS property:

$$\begin{aligned}
(IS(\mathtt{A},\mathtt{B}) \subseteq \{\ldots\})_{AG} &\subset (IS(\mathtt{A},\mathtt{B}) = any)_{AS} \\
(IS(\mathtt{B},\mathtt{A}) = \emptyset)_{AG} &= (IS(\mathtt{B},\mathtt{A}) = \emptyset)_{AS}
\end{aligned}$$

Thus, the four properties in $\mathcal{P}$ are all required to distinguish our formalisations of the association, aggregation, and composition relationships: $\mathcal{P}$ forms a minimal subset of all properties of binary class relationships with respect to our definitions. We explain why the association and aggregation relationships are distinguished by the IS and MU properties and the aggregation and composition relationships by the EX and LT properties in a following discussion on the static and dynamic parts of the relationships.

Second, we show that all properties in $\mathcal{P}$ appear in all definitions of binary class relationships in the literature. However, for lack of space, we cannot detail all definitions of binary class relationships in the literature. Indeed, we found in a previous survey 26 definitions distributed in 5 categories [21]. Thus, we only present one example per category:

- Definitions in natural languages: 5 definitions. For example, definitions of the association and aggregation relationships in the precursor book by Rumbaugh *et al.* [33, sections 3.2, 4.1, and 15.6] states that: It is sometimes necessary to check whether a part already belongs to a whole, which relates to *exclusivity*, and the *multiplicity* of instances of classes involved; Some operations of a whole apply to its part, using attributes containing pointers to the part, which relates to *invocation site*. The *lifetime* property is not explicitly mentioned, but memory management—object deletion, dangling pointer, memory leak—is cited as concern for implementation of the relationships.

- Formal definitions: 5 definitions. For example, interpretation of association relationships in the Syntropy methodology [7] explicitly mentions cardinality constraints, related to *multiplicity* and *lifetime*, as well as lifetime constraints, related to *exclusivity*. Also, data encapsulation (value attribute or reference attribute) is described as a mean to implement association and aggregation relationships, which relates to *invocation site*, although the formalisation interprets "the association without any knowledge of the structure of the object it associates" [7, page 8].

- Definitions with properties: 5 definitions. For example, definitions of the aggregation and composition relationships by Henderson-Sellers and Barbier [26, table 4, page 356] use several characteristics, such as: C1. Propagation of one or more operations and C5. Propagation of destruction operation, related to *invocation site* and *lifetime*; C2. Ownership, related to *exclusivity*; P1. Whole–part, related to *multiplicity*.

- Definitions embodied in programming languages: 3 definitions. For example, in the Troll programming language [24], definitions of static aggregations, dynamic aggregations, and disjointed aggregations distinguish two kinds of complex objects: Non-disjoint

and disjoint, which relates to *exclusivity*. Complex objects may belong to one or several aggregations, which relates to *multiplicity*. They possess attribute values which they alter by local events (events are abstractions of methods), related to *invocation site*. They have explicit life cycles, which relates to *lifetime*.

- Definitions at the implementation level: 8 definitions. For example, in the tool WOMBLE [27] for lightweight extraction of object-models from Java byte-codes, an association relationship may be annotated to show its *multiplicity* and its mutability, which relates to *exclusivity*. Heuristics infer multiplicity and mutability properties and deal with container classes, which relate to *invocation site*. However, the tool does not distinguish aggregation from association relationships and does not consider composition relationships, although the authors mention that *lifetime*-dependency inference would be required.

Thus, with the two previous steps, we show that $\mathcal{P}$ is a minimal subset of all properties of binary class relationships.

Order among the Binary Class Relationships. The formalisations of the relationships with their properties show that an order exists among the association, aggregation, and composition relationships. The values of the aggregation relationship properties are more constraining than those of the association relationship:

$$\begin{aligned} AG(\texttt{A},\texttt{B}) = \\ AS(\texttt{A},\texttt{B}) \quad & \wedge \\ (IS(\texttt{A},\texttt{B}) \quad = \quad & \{\texttt{field},\texttt{array field}, \\ & \texttt{collection field}\}) \quad \wedge \\ (MU(\texttt{B},\texttt{A}) \quad = \quad & [1,+\infty]) \end{aligned}$$

The values of the composition relationship properties are more constraining than those of the aggregation relationship. In particular, the exclusivity property is stronger for the part in a composition relationship than in an aggregation relationship, because, in a composition relationship, the part must not belong to any other aggregation or composition relationship:

$$\begin{aligned} CO(\texttt{A},\texttt{B}) = \\ AG(\texttt{A},\texttt{B}) \quad & \wedge \\ (EX(\texttt{A},\texttt{B}) = true) \quad & \wedge \quad (EX(\texttt{B},\texttt{A}) = false) \quad \wedge \\ (LT(\texttt{A},\texttt{B}) = +) \quad & \wedge \quad (LT(\texttt{B},\texttt{A}) = -) \quad \wedge \\ (MU(\texttt{A},\texttt{B}) = [1,+\infty]) \quad & \wedge \quad (MU(\texttt{B},\texttt{A}) = [1,1]) \end{aligned}$$

Static and Dynamic Parts. The formalisations of the binary class relationships decompose in two fundamental parts: A static part corresponding to the MU and IS properties; A dynamic part corresponding to the EX and LT properties. Association and aggregation relationships are inherently static, thus their static parts are important for their distinction and their detection. A composition relationship is an aggregation relationship with additional constraints on the behaviour of composed instances, thus its dynamic parts are important for its distinction from an aggregation relationship and its detection.

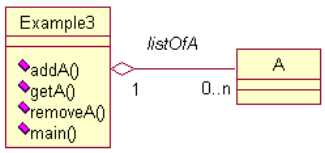
Acquaintance and Optimisation. For optimisation purposes, a class `A` associated with a class `B` may declare a field of type `B` to record a particular instance of `B`. According to our definitions, the relationship between `A` and `B` then evolves from an association to an aggregation relationship. This is not a problem because we aim at bringing continuity between design and implementation. We try to expose consistently, not to interpret, software engineers' intent.

For example, software designers specified an association relationship between classes `Person` and `Course`. Software engineers implemented this association relationship by declaring a field of type `Course` in class `Person`. If the recovered relationship is an aggregation relationship, it provides important data to software maintainers: First, maintainers know that an aggregation relationship *actually* exists between classes `Person` and `Course`; Second, maintainers may act about the discrepancy between the *actual* implementation and the *desired* design as appropriate.

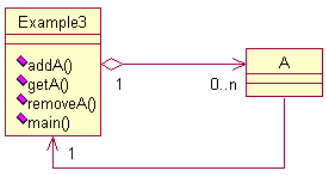
Composition and Shared Composition. We only consider *non-shared* composition relationships. The multiplicity of the part for the composition relationship, $MU(\texttt{B},\texttt{A}) = [1,1]$, prevents a part to belong to more than one whole. This multiplicity ensures the lifetime property of a part with respect to its whole for the composition relationship and consistency between the definitions of the aggregation and composition relationships.

Navigability. In the UML notation, navigability is specified with arrows on association ends (respectively aggregation and composition ends). We choose to study only unidirectional relationships because:

- A bidirectional relationship can be expressed as two opposite unidirectional relationships with constraints between them [28], if appropriate. For example, we could express the UML class diagram in Figure (a) using the class diagram in Figure (b). (In the UML notation, no arrow is displayed when a relationship is bidirectional.)



(a) Bidirectional aggregation relationship



(b) Unidirectional aggregation and association relationships

- In standard programming languages, bidirectional relationships are actually implemented as two opposite unidirectional relationships.

Thus, we think that it is adequate to express bidirectional relationships as two opposite unidirectional relationships. Software engineers may decide to *fusion* semantically-related opposite unidirectional relationships at design level *after* reverse-engineering.

Qualification. We do not qualify the formalisations of the relationships with labels identifying the particular relationships being specified. Indeed, the formalisations define the association, aggregation, and composition relationships and are characteristics that particular relationships may have.

For example, there may exist two associations relationships, `teaches` and `takes`, between the classes `Person` and `Course`. We would say that:

`teaches`	verifies	$AS(\texttt{Person}, \texttt{Course})$
`takes`	verifies	$AS(\texttt{Person}, \texttt{Course})$

Symmetrical Relationships and Backpointers. The presence of a binary class relationship between classes `A` and `B` does not preclude the existence of other relationships. Thus, relationships between classes `A` and `B` may induce a cycle. Typically, there is a cycle when a backpointer records the owner of an instance or when two classes possess fields of each other's type. In such cases, two symmetrical aggregation relationships would exist: $AG(\texttt{A}, \texttt{B})$ and $AG(\texttt{B}, \texttt{A})$.

6. DETECTION ALGORITHMS FOR THE BINARY CLASS RELATIONSHIPS

We bring continuity between implementation and design by defining detection algorithms for the previous properties and formalisations of the binary class relationships. We introduce algorithms to compute the values of the four properties that we use to infer the existence of association, aggregation, and composition relationships in Java source code. For the sake of brevity, we describe the principles of the algorithms only, the interested reader may refer to previous work for detailed explanations on static and dynamic analyses for Java [21, 22]. The algorithms are independent of the target programming language, because they rely on language-independent properties only. However, the algorithms take advantage of language-dependent programming idioms as we show for the Java programming language.

6.1 Principles of the Detection Algorithms

Detection of association relationships requires collecting the value of the IS property only. Detection of aggregation relationships requires inferring the values of the IS and MU properties. Detection of composition relationships requires collecting the value of the IS and MU properties and of the EX and LT properties. We collect values of the invocation site, IS, and multiplicity, MU, properties using static analyses. We infer values of the exclusivity, EX, and lifetime, LT, properties using dynamic analyses.

6.2 Association, Aggregation Relationships

We use static analyses to compute values of the static part—the MU and IS properties—of the association and aggregation relationships. The Java programming language uses an intermediate language made of byte-codes. We perform static analyses on this intermediate language, using a byte-code analysis framework: IBM CFParse v1.21. We choose to work with the intermediate language because class files are always available, while source code may be unavailable when proprietary [35].

Computation of IS. We iterate through the byte-codes of each class, looking for byte-codes corresponding to method invocations: `InvokeInterface`, `InvokeStatic`, `InvokeSpecial`, and `InvokeVirtual`. Then, depending on invocation sites (`field`, `array field`, `collection field`, `parameter`, `array parameter`, `collection parameter`, `local variable`, `local array`, `local collection`), we assign values to the IS property and identify the target of the relationships.

A difficulty arises when invocation sites are collections, `collection field`, `collection parameter`, `local collection`, because collections are typically un-typed. If we assume that these kinds of collections are homogeneous (containing instances with a common superclass different from `Object`), it is possible to determine their types using well-known Java programming idioms, such as pairs of `add()`–`remove()` accessors [27].

Also, the algorithm are parameterised specifically to recognise user-defined collections in addition to collections from the standard Java class libraries, such as `Map`, `List`, and `Set` and their implementations.

Computation of MU. Detection of the values of the MU property uses invocation sites also: We assign value $[0, 1]$ to the MU property if the invocation site is `field`, `parameter`, `local variable`, value $[0, +\infty]$ if the invocation site is `array field`, `array parameter`, `local array`, `collection field`, `collection parameter`, or `local collection`.

6.3 Composition Relationships

The dynamic part—the EX and LT properties—of the composition relationship is difficult to detect. We propose a solution using a trace-analysis technique [22]. A composition relationship is an aggregation relationship with specific values for the exclusivity and lifetime properties. Thus, we use the trace-analysis technique to compute, for each aggregation relationship, values of the exclusivity and lifetime properties and, if the values match, to change it to a composition relationship.

Model. We model a program execution as a trace: A sequence of execution events. There are three kinds of events, represented as Prolog terms: Assignment events emitted every time a field of an instance of class `A` is assigned with an instance of class `B`; Finalise events emitted when the Java virtual machine garbage-collects an instance[2]; A program-end event that is emitted when the program terminates. We generate and analyse on the fly the trace of a program execution using Prolog predicates. Then, we infer the existence of composition relationships from computed values of the exclusivity and lifetime properties.

[2]We assume our tool emits a finalise event as soon as an instance becomes ready for garbage collection.

Computation of LT. We assess the lifetime property of aggregation relationships using a Prolog predicate, `checkLTProperty/3` [21]. This predicate builds a list of terms abstracting sequences of events in the execution trace depending on the order in which assignations, finalisations, and program-end happen. The `checkLTProperty/3` builds and maintains a list of `pendingAssignation` and of `lifetimeProperty` terms:

- `pendingAssignation` terms are added to the list upon assignation of an instance of class `B` to a `field` (respectively, `array field`, `collection field`) of an instance of class `A`.
- `pendingAssignation` terms are converted into `lifetimeProperty` terms upon finalisation of instances of classes `A` and `B` if appropriate, *i.e.*, finalisations of instances of class `A` occur after finalisations of all corresponding instances of class `B` ($LT_d(\texttt{A}) > LT_d(\texttt{B})$).
- `pendingAssignation` terms are converted into `lifetimeProperty` terms when the program terminates.

Computation of EX. Following the same principle, we define a predicate to check the exclusivity property of aggregation relationships, `checkEXProperty/3` [21], which builds and maintains a list of `exclusivityProperty` terms, according to assignations and program-termination events.

Finally, the terms in the lists represent values of the exclusivity and lifetime properties between instances of classes `A` and `B`. We infer values of the exclusivity and lifetime properties of the two classes by conjunctions (*and* operator) of the values between their corresponding instances.

6.4 Discussion of the Detection Algorithms

The set of invocation sites map to Java source code directly. With other programming languages such as C++, it is necessary to adapt the static analyses to handle pointers and structures, for example. However, the set of invocation sites does not change with different class-based programming languages, it is language independent.

Our detection algorithms suffer from two main limitations which impedes their correctness and completeness.

- We base the detection of aggregation relationships on static analyses and on heuristics expressing common programming idioms, *i.e.*, a collection is *generally* access through specific accessors, to infer the type of stored instances. However, developers may not have followed the common programming idioms, which leads to a decreased precision of the algorithms, as shown in the following validations, in section 7.1.
- We base the detection of composition relationships on dynamic analyses, which have well-known limitations, such as code coverage, statistical significance of the runs, and dependency of the overall results on the *and* operator used to combine results from runs. These limitations decrease the precision and usability of the detection algorithms, as shown in section 7.2.

We could replace our heuristics and dynamic analyses with alias analyses but their benefits are yet unclear with respect to simplicity and performance of the algorithms.

7. VALIDATIONS OF THE DETECTION ALGORITHMS

We have adapted the static analysis algorithms from our previous tool PatternsBox [3] in Ptidej, a tool to reverse-engineer Java programs, and we implemented the dynamic analysis algorithms using Caffeine [22], a tool for the dynamic analysis of Java programs. We have tested our implementations on: Java AWT v1.2.2 [37], JHotDraw v5.1 [18], and JUnit v3.7 [17].

7.1 Association, Aggregation Relationships

The association relationship is the simplest relationship to detect and provides an overwhelming number of hits: There are respectively $2,784+1,505+636 = 4,925$ association relationships for the 583 classes of the three frameworks. Thus, aggregation relationships are the most informative with respect to static analyses. Static analyses of the three frameworks take respectively 18.66, 7.81, and 7.11 seconds on a Pentium II 600 Mhz.

Table 1 shows the results of detection of aggregation relationships for the different frameworks. It details the number: Of *classes* per framework; Of relationships (multiplicity $+\infty$) *found*; Of relationships (multiplicity $+\infty$) detected by a *manual* analysis; And, of *false hits*. The results do not include aggregation relationships of multiplicity 1, because detection of these relationships is not an issue and would inaccurately increase the precision[3]. Their detection requires only static analyses of classes to find "simple" fields, such as `private B b;`, which multiplicity and target are obvious. Also, the results involve only aggregation relationships that are homogeneous[4], and that do neither involve primitive types, nor wrapper types, nor the `String` type, nor listeners[5]. Such homogeneous aggregation relationships are typically described using collection classes implementing the `List` interface, such as the `Vector` class.

We performed a manual analysis to check the accuracy of our algorithms. We do not obtain a precision of 100% because developers of the three frameworks did not respect some of the Java idioms assumed by our detection algorithms. Developers implemented `add()` methods without the corresponding `remove()` or `get()` methods, or pairs of `add()`–`remove()` methods with different argument types. For example, the `Component` class of Java AWT declares methods `add(PopupMenu)` and `remove(MenuComponent)` to add and to remove popup menus from a component.

However, this lack of precision actually helps in understanding programs and in identifying potential problems through the enforcement of programming standards. Thus, the absence of aggregation relationship between classes `Component` and `PopupMenu` highlights the discrepancy between methods `add(PopupMenu)` and `remove(MenuComponent)`.

[3]For example, on JHotDraw v5.1, the precision is 98% for 151 existing aggregation relationships, multiplicity 1 and $+\infty$, to compare with a precision of 75% for 8 existing aggregation relationships, multiplicity $+\infty$ only.

[4]We say that an aggregation relationship is homogeneous if it involves instances of classes with a common superclass *different* from `Object`.

[5]The detection of aggregation relationships with primitive types, wrappers, `String`, or listeners does not provide helpful information to bridge the gap between implementation and design, because primitive types are not instances of classes while wrappers, `String`, and listeners are not application-specific classes.

Aggregation relationships with multiplicity $+\infty$				
Frameworks	Classes	Found	Manual	False hits
JAVA AWT v1.2.2	367	17	20	1
JHOTDRAW v5.1	171	6	8	0
JUNIT v3.7	45	1	4	0
Total	583	24	32	1
Precision: 0.75 ($\frac{Found}{Manual}$)		Recall: 0.96 ($\frac{Found}{Found+False\ hits}$)		

Table 1: Results of static analyses for aggregation relationships with multiplicity $+\infty$.

7.2 Composition Relationships

Table 2 presents the results of detecting composition relationships in the JUNIT v3.7 framework, using dynamic analyses on the `junit.samples.money.MoneyTest` test case with three different user interfaces (UI). We did not perform dynamic analyses on the JHOTDRAW v5.1 framework, because it requires user-interaction, nor on the JAVA AWT v1.2.2 framework because it is not runnable.

We perform the analyses based on previously found aggregation relationships, because composition relationships are aggregation relationships with stronger dynamic properties. A complete analysis of the program would be too costly in time and would not provide more results.

Some aggregation relationships reveal themselves as composition relationships upon dynamic analyses of the `MoneyTest` test case with the different UI: Properties of lifetime and of exclusivity hold. We performed a manual analysis to check the accuracy of our dynamic analyses and we obtain a precision of 100% and a recall of 100% for the given conditions of execution (multiple runs of the `MoneyTest` test case with the different UI).

However, the results are subject to caution because they correspond to a subset of all possible execution paths: Depending on the execution path, the inferred relationships may vary. For example, there is no composition relationship between classes `TestSuite` and `Test` [21] with the AWT-based UI while there is one with the text-based UI. Code coverage analyses [6, 15] are being considered to increase confidence on the dynamic analyses.

7.3 Discussion of the Validations

Quantitative and Qualitative. Preceding results are interesting in two respects:

- Quantitatively, we can compute metrics based on detected relationships. These metrics further enhance the understanding that software engineers have of a program using other metrics, such as Chidamber and Kemerer's metrics [13].
- Qualitatively, we can assess the quality of a design using the number of detected relationships, the presence of opposite relationships. We can also compare detected relationships and relationships defined during design, identify design anomalies, highlight implementation discrepancies and bugs [27].

Practicality. Results of the static analyses have a precision of 75% and a recall of 96%. They show that our algorithms, although neither complete nor correct, are good enough in practice to detect aggregation relationships and, by extension, composition relationships correctly. It is possible to take in consideration more Java idioms, however, the current versions of our algorithms represent acceptable compromises between computation time, complexity, and precision. The dynamic part of the composition relationship is subject to caution because of the limitations of dynamic analyses between executions. Also, dynamic analyses are time consuming: Our dynamic analysis tool, CAFFEINE, slows down program executions up to a factor of 5,000 [22]. Thus, detection of composition relationships should only be performed to answer specific questions from software engineers [36].

Comparisons. Validations of our definitions and of results from our algorithms against third-party definitions and tools is a very difficult task. We would like to apply our algorithms on documented frameworks and to compare results with the documentation of these frameworks or analyses of other tools. However, developers mainly use modeling language with little operational semantics when designing their software, such as the UML, and thus do not consider differences among the association, aggregation, and composition relationships. Also, we were unable to find a framework both publicly available and of reference in which the developers explicitly distinguished the three binary class relationships. For example, the authors of JHOTDRAW v5.1 use the notation proposed by Gamma *et al.* [19]. This notation offers the association relationship, called *acquaintance* relationship, and a generic aggregation relationship, called the *part-of* relationship. The notation defines the implementation of acquaintance and of part-of relationships only loosely and does not mention the composition relationship.

Consequences. Detection of the binary class relationships in the JAVA AWT, JHOTDRAW, and JUNIT frameworks reduces the gap between implementation and design.

Indeed, software maintainers have a better understanding of both the inner working and of the design of the frameworks through the knowledge of the relationships among classes. They can check desired designs against actual implementations and identify possible design defects or erroneous relationship implementations. They can also identify design patterns more easily, thus developing their understanding of the frameworks one step further [2].

Finally, software engineers could use the knowledge of the relationships to translate the frameworks to other class-based object-oriented programming languages. Indeed, they can translate the relationships among classes from one programming language to another accurately using their language-independent properties.

8. CONCLUSION AND FUTURE WORK

We tackled the problem of discontinuity between programming and modeling languages in the specific case of three binary class relationships: Association, aggregation, and composition. We answered the question: "How to define binary class relationships so we can detect them in implementation?" We proposed and formalised definitions of the relationships with a minimal set of four language-independent properties at implementation level. We presented detection algorithms based on the properties and sketched their implementations. We validated the algorithms on well-known frameworks with respect to our definitions. These definitions and algorithms bring continuity in the transition between a program implementation and its design. We integrated implementations of our algorithms in two reverse-

Composition relationships with multiplicity $+\infty$				
User interfaces	Found	Manual	False hits	Classes in the relationships
Text	3	3	0	`TestResult`–`TestFailure` ($\times 2$), `TestSuite`–`Test`
Java AWT	3	3	0	`TestResult`–`TestFailure` ($\times 2$), `junit.awtui.TestRunner`–`Throwable`
Java Swing	3	3	0	`TestResult`–`TestFailure` ($\times 2$), `junit.swingui.TestRunner`–`TestRunView`
Total	3	3	0	
Precision: 1.00 ($\frac{Found}{Manual}$)			Recall: 1.00 ($\frac{Found}{Found+False\ hits}$)	

Table 2: Results of dynamic analyses for composition relationships with multiplicity $+\infty$ in the JUnit v3.7 framework (`MoneyTest` class).

engineering tools for Java programs, Ptidej and Caffeine. These tools ensure consistent continuity and traceability between implementation and design, which is a major improvement over existing industrial and academic software and–or reverse engineering tools.

Currently, we enhance and use our reverse engineering tools to study on firm foundations the identification of micro-architectures similar to solutions of design-patterns in Java programs and of detect design defects [2, 20]. We also work on improving static and dynamic analyses for the C++ and Smalltalk programming languages.

Future work includes:

- To assess the benefits of full-fledge alias analyses and their implementations to improve the detection of the relationships.
- To implement code coverage analyses to increase the confidence in the dynamic analyses.
- To develop a set of criteria to assess the quality of reverse-engineered class diagrams, to compare reverse-engineering tools objectively.
- To verify that our definitions are *really* consensual with respect to developers' intent when designing software.
- To apply our reverse engineering tools, Ptidej and Caffeine, on real-life programs and to validate their results with the programs developers.
- To apply our approach on other programming languages. For instance, we could use an abstract-syntax tree for the static analyses of C++ programs, and destructor-related events for dynamic analyses.
- To develop our approach with more *flavours* of binary class relationships, such as the use, shared-aggregation, and container relationships.
- To express our formal definitions with OCL, the constraint language supplied with UML, to allow seamless integration with UML tools.
- To improve the precision of the detection by considering alternate implementations of the algorithms and alternate definitions of the relationships.

ACKNOWLEDGMENTS

We gratefully thank Rémi Douence, Pierre Cointe, Houari Sahraoui, and the anonymous reviewers for their valuable help and comments on previous versions of this paper.

REFERENCES

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, second edition, 1998.

[2] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In Debra Richardson, Martin Feather, and Michael Goedicke, editors, *proceedings of the 16th conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press, November 2001.

[3] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In Bedir Tekinerdogan, Pim Van Den Broek, Motoshi Saeki, Pavel Hruby, and Gerson Sunyé, editors, *proceedings of the 1st ECOOP workshop on Automating Object-Oriented Software Development Methods*. Centre for Telematics and Information Technology, University of Twente, October 2001. TR-CTIT-01-35.

[4] Pascal André, Annya Romanczuk, Jean-Claude Royer, and Aline Vasconcelos. An algebraic view of UML class diagrams. In Christophe Dony and Houari Sahraoui, editors, *proceedings of the 6th colloquium on Languages and Models with Objects*, pages 261–276. Hermès Science Publications, January 2000.

[5] Gilles Ardourel and Marriane Huchard. AGATE: Access Graph bAsed Tools for handling Encapsulation. In *proceedings of the 16th conference on Automated Software Engineering*, pages 311–314. IEEE Computer Society Press, November 2001. Short paper.

[6] Boris Bezier. *Software Testing Techniques*. Van Nostrand Rheinhold Company, New York, 1990.

[7] Juan C. Bicarregui, Kevin C. Lano, and Tom S. E. Maibuam. Objects, associations and subsystems: A hierarchical approach to encapsulation. In Mehmet Aksit and Satoshi Matsuoka, editors, *proceedings of 11th European Conference on Object-Oriented Programming*, pages 324–343. Springer-Verlag, June 1997.

[8] Edwin Blake and Steve Cook. On including part hierarchies in object-oriented languages, with an implementation in Smalltalk. In Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, *proceedings of the 1st European Conference on Object-Oriented Programming*, pages 41–50. Springer-Verlag, June 1987.

[9] Grady Booch. *Object-Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 2^{nd} edition, September 1993.

[10] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a formalization of the unifed modeling language. In Mehmet Aksit and Satoshi Matsuoka, editors, *proceedings of the 11^{th} European Conference for Object-Oriented Programming*, pages 344–366. Springer-Verlag, June 1997.

[11] Jean-Michel Bruel, Brian Henderson-Sellers, Franck Barbier, Annig Le Parc, and Robert B. France. Improving the UML metamodel to rigorously specify aggregation and composition. In Shushma Patel, Yingxu Wang, and Ronald H. Johnston, editors, *proceedings of the 7^{th} international conference on Object-Oriented Information Systems*, pages 5–14. Springer-Verlag, August 2001.

[12] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller, François Lustman, and Guy Saint-Denis. Design properties and object-oriented software changeability. In Jürgen Ebert and Chris Verhoef, editors, *proceedings of the 4^{th} Conference on Software Maintenance and Reengineering*, pages 45–54. IEEE Computer Society Press, February 2000.

[13] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object-oriented design. Technical Report E53-315, MIT Sloan School of Management, December 1993.

[14] Franco Civello. Roles for composite objects in object-oriented analysis and design. In Andreas Paepcke, editor, *proceedings of the 8^{th} conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 376–393. ACM Press, September 1993.

[15] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transaction on Software Engineering*, 15(11):1318–1332, November 1989.

[16] Stéphane Ducasse, Mireille Blay-Fornarino, and Anne-Marie Pinna-Dery. A reflective model for first class dependencies. In Frank Manola, editor, *proceedings of 10^{th} conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 265–280. ACM Press, October 1995.

[17] Erich Gamma and Kent Beck. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, July 1998.

[18] Erich Gamma and Thomas Eggenschwiler. JHotDraw. Web site, 1998.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[20] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Quioyun Li, Richard Riehle, Gilda Pour, and Bertrand Meyer, editors, *proceedings of the 39^{th} conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press, July 2001.

[21] Yann-Gaël Guéhéneuc, Hervé Albin-Amiot, Rémi Douence, and Pierre Cointe. Bridging the gap between modeling and programming languages. Technical Report 02/09/INFO, Computer Science Department, École des Mines de Nantes, July 2002.

[22] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caffeine – A tool for dynamic analysis of Java programs. In Wolfgang Emmerich and Dave Wile, editors, *proceedings of the 17^{th} conference on Automated Software Engineering*, pages 117–126. IEEE Computer Society Press, September 2002.

[23] William Harrison, Charles Barton, and Mukund Raghavachari. Mapping UML designs to Java. In Doug Lea, editor, *proceedings of the 15^{th} conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 178–188. ACM Press, October 2000.

[24] Thorsten Hartmann, Ralf Jungclaus, and Gunter Saake. Aggregation in a behavior oriented object model. In Ole Lehrmann Madsen, editor, *proceedings of 6^{th} European Conference for Object-Oriented Programming*, pages 57–77. Springer-Verlag, June–July 1992.

[25] Brian Henderson-Sellers. Some problems with the UML v1.3 metamodel. In Ralph H. Sprague Jr., editor, *proceedings of the 34^{th} annual Hawaii International Conference on System Sciences*, pages 3052–3064. IEEE Computer Society Press, January 2001.

[26] Brian Henderson-Sellers and Franck Barbier. A survey of the UML's aggregation and composition relationships. *L'objet : Logiciel, Base de données, Réseaux*, 5(3/4):339–366, December 1999.

[27] Daniel Jackson and Allison Waingold. Lightweight extraction of object models from bytecode. In David Garlan and Jeff Kramer, editors, *proceedings of the 21^{st} International Conference on Software Engineering*, pages 194–202. ACM Press, May 1999.

[28] Ralf Kollmann and Martin Gogolla. Application of UML associations and their adornments in design recovery. In Elizabeth Burd and Peter Aiken, editors, *proceedings of the 8^{th} Working Conference on Reverse Engineering*, pages 81–91. IEEE Computer Society Press, October 2001.

[29] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In Linda M. Wills and Ira Baxter, editors, *proceedings of the 3^{rd} Working Conference on Reverse Engineering*, pages 208–215. IEEE Computer Society Press, November 1996.

[30] Bent Bruun Kristensen. Complex associations: Abstractions in object-oriented modeling. In J. Eliot B. Moss, editor, *proceedings of the 9^{th} conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 272–283. ACM Press, October 1994.

[31] Esperanza Marcos, Belen Vela, José M. Cavero, and Paloma Cáceres. Aggregation and composition in object-Relational database design. In Albertas Caplinskas and Johann Eder, editors, *proceedings of the 5^{th} east-european conference on Advances in*

Databases and Information Systems, pages 195–209. Springer-Verlag, September 2001.

[32] James Noble and John Grundy. Explicit relationships in object-oriented development. In Bertrand Meyer, editor, *proceedings of the 18^{th} conference on the Technology of Object-Oriented Languages and Systems*, pages 211–226. Prentice-Hall, November 1995.

[33] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Inc., 1^{st} edition, October 1991.

[34] Monika Saksena, Robert B. France, and Maria M. Larrondo-Petrie. A characterization of aggregation. In Colette Rolland, editor, *proceedings of the 5^{th} international conference on Object-Oriented Information Systems*, pages 363–372. Springer-Verlag, September 1998.

[35] Pamela Samuelson. Reverse engineering under siege. *Communications of the ACM*, 45(10):15–20, October 2002.

[36] Elliot Soloway, Jeannine Pinto, Stanley Letovsky, David Littman, and Robin Lampert. Designing documentation to compensate for delocalized plans. *Communication of the ACM*, 31(11):1259–1267, November 1988.

[37] Sun Microsystems, Inc. *Java Abstract Window Toolkit*, May 2000.