

Bridging the Gap between Modeling and Programming Languages

Yann-Gaël Guéhéneuc*, **Hervé Albin-Amiot†**,
Rémi Douence, and **Pierre Cointe**

École des Mines de Nantes
4, rue Alfred Kastler – BP 20823
44307 Nantes Cedex 3
France

{guehene|albin|douence|cointe}@emn.fr

July 11, 2002

Abstract

A discontinuity exists between modeling languages and object-oriented programming languages. This discontinuity is a consequence of ambiguous notions in modeling languages and lack of corresponding notions in object-oriented programming languages. It hinders the transition between a software design and its implementation, and vice versa. Thus, it hampers the implementation and the maintenance processes. This discontinuity is particularly acute for binary class relationships, which describe, at the design level, notions such as association, aggregation, and composition. From the current state of the art, we propose synthetic definitions for the binary class relationships at the design level and corresponding definitions at the implementation level. We express the latter definitions in terms of common properties. We present algorithms to synthesize code for these properties and to detect these properties in code. These algorithms allow us to generate and to detect binary class relationships. We verify the detection algorithms on several well-known frameworks. The definitions and algorithms bring continuity between modeling languages and object-oriented programming languages.

*This work is partly funded by Object Technology International, Inc. – 2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada

†This work is partly funded by Soft-Maint – 4, rue du Château de l'Éraudière – 44 324 Nantes – France.

Contents

1	Gap between Design and Implementation	4
2	State of the Art	9
2.1	Definitions in Natural Language	9
2.2	Formal Definitions	11
2.3	Properties of the Binary Class Relationships	12
2.4	New Programming Languages or Extensions	13
2.5	Definitions at the Implementation Level	14
3	Definitions of the Binary Class Relationships	16
3.1	Association relationship	16
3.2	Aggregation relationship	17
3.3	Composition relationship	17
4	Properties of the Binary Class Relationships	19
4.1	Definitions	19
4.2	Discussion	20
4.3	Implementation Examples	20
5	Redefinitions of the Binary Class Relationships	23
5.1	Redefinitions	23
5.2	Discussion	24
5.2.1	Order among binary class relationships	24
5.2.2	Symmetrical relationships and backpointers	25
5.2.3	Acquaintance and optimizations	25
5.3	Implementation Examples	26
6	Code Synthesis Algorithms	35
7	Detection Algorithms	37
7.1	Principles	37
7.2	Static Detection Algorithms	37
7.2.1	Implementation Concerns	37
7.2.2	Detection of <i>IMS(Class1, Class2)</i>	37
7.2.3	Detection of <i>MU(Class)</i>	38
7.2.4	Static Detection Example	39
7.3	Dynamic Detection Algorithms	39
7.3.1	Model	39
7.3.2	Detection of <i>LT(Class)</i>	40
7.3.3	Detection of <i>EX(Class)</i>	43
7.3.4	Dynamic Detection Example of the Composition Relationship	44
7.3.5	Implementation Concerns	45

8	Validation	46
8.1	Algorithms	46
8.2	Definitions	50
9	Conclusion and Future Work	51
A	Composition and Finalization	52

1 Gap between Design and Implementation

A recurrent problem in the object-oriented software engineering community is the transition between a software design and its implementation, and vice versa, during the implementation and the maintenance phases.

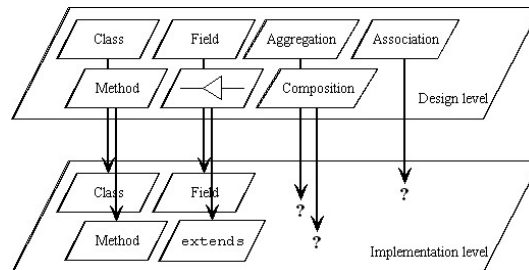
A software design is described using a modeling language, such as the UML; a software implementation is written using a standard class-based programming language, such as Java.

The UML and Java possess common notions, such as the notion of class and the notion of inheritance. However, the UML includes extra notions that do not exist in Java, for example the notion of stereotype, or the notions of binary class relationships: Association, aggregation, and composition.

In this paper, we only focus on the binary class relationships. The very existence of the notions of binary class relationships in the UML brings a discontinuity in the transition between the software design and its implementation, because no such notions exist in Java. This discontinuity hinders the implementation of software designs and the understanding of software implementations. It also impedes the communication among software engineers and makes it difficult teaching software designing.

Example of problem: Transition and discontinuity

The following figure shows that some notions exist both at the design level (UML) and at the implementation level (Java), such as the notions of class or inheritance.

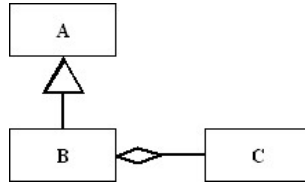


The notions of association, aggregation, and composition relationships (UML) do not exist at the implementation level (Java).

The definitions of the binary class relationships proposed by the UML should answer this problem of discontinuity. Unfortunately, the definitions are given in natural language and present several ambiguities; moreover, they offer no hint on possible implementation choices (see for instance [8] and [24]).

Example of problem: Implementation choices

The following UML class diagram defines three classes, an inheritance relationship, and an aggregation relationship:



The following Java source code corresponds to the three classes A, B, and C:

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

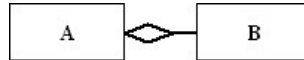
How should we implement the aggregation relationship between class B and class C? As a field? As a collection? With a pair of getter and setter? Or with a pair of `add()`–`remove()` methods?

Even world-class industrial CASE tools, such as RATIONAL ROSE¹, TOGETHERSOFT TOGETHER², or BORLAND JBUILDER³, or open-source CASE tools, such as ARGOUML⁴, do not have clear definitions of the binary class relationships. They graphically distinguish the association, aggregation, and composition relationships; but the code synthesized for the different binary class relationships is the same. The code reengineering tools produce erroneous or inconsistent relationships.

¹<http://www.rational.com>
²<http://www.togethersoft.com/>
³<http://www.borland.com/>
⁴<http://argouml.tigris.org/>

Example of problem: Limitations of industrial tools

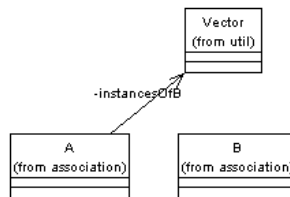
The following UML diagram represents two classes A and B, which an aggregation relationship links together.



The source code generated by RATIONAL ROSE v2001.03.00 is the following:

```
public class A {                public class B {
    private B instancesOfB[];
    public A()
    {
    }
}                                public B()
                                {
                                }
}
```

This source code is equivalent for the association, the aggregation, and the composition relationship, even though they represent different notions and have different semantics. Now, if we replace the array by a collection, instance of class `java.util.Vector` for example, as it is generally implemented (see Section 8), the diagram obtained by reengineering the modified source code with RATIONAL ROSE is:



The aggregation relationship disappears, and an association relationship now links the class A and the class `Vector`, which is inconsistent with the original class diagram.

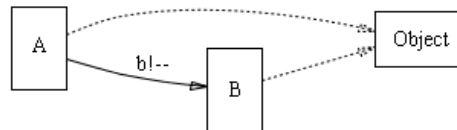
Academic tools, such as WOMBLE [30] or CHAVA [31], extract object-model from Java source code or byte-codes. CHAVA only proposes delegation and field access as inter-class relationships. WOMBLE goes further by describing an inference system for association relationships but does not consider aggregation and composition relationships.

Example of problem: Limitations of academic tools

The following Java source code presents two classes A and B linked by a composition relationship (cf. Section 3). An instance *a* of class A strongly owns an instance *b* of class B because *b* is encapsulated within *a* and because the life-time of *b* depends on the lifetime of *a*:

```
public class A {
    private B b;
    public void attach(final B b) {
        this.b = b;
    }
    public void operation() {
        this.b.operation();
    }
}
public class B {
    public void operation() {
    }
}
```

Using WOMBLE, we extract the object-model associated with classes A and B. The following figure presents this object-model:



The relationship between the A and B classes is defined as an association named *b*, with a multiplicity of exactly one (the ! symbol) and a static mutability (the -- symbol) stating that the instance of B does not change during the lifetime of the instance of A.

WOMBLE does not infer other relevant information, such as the lifetime dependency or the exclusivity property between instances of A and B, required to distinguish between association, aggregation, and composition relationships.

A first solution to the problem of discontinuity between modelling and implementation for the binary class relationship consists in building a new programming language or in extending an existing one with the appropriate notions of binary class relationships. However, this solution goes against the very purpose of general-purpose modelling languages, such as the UML. As stated in the UML Preface [26], “*the UML is used as a general-purpose modeling language, with potentially many implementation targets [...] The semantics are defined in an implementation language-independent way*” (pp. 6–7 and 11). This solution also implies using a non-standard programming language and thus eliminates the benefits of standardization: Class libraries; Development environments; Re-engineering tools; Efficient compilers and virtual machines.

A second solution consists in defining the notions of binary class relationships in terms of constructs of an existing programming language. We choose this

approach in this paper, with the Java programming language. We take a very pragmatic viewpoint and do not propose yet-another-debate on the whole-part relationship or a study of the properties of the binary class relationships. We propose simple definitions of the binary class relationships at the implementation level and algorithms that bring continuity between the UML, at the design level, and Java, at the implementation level. We apply these algorithms to the design and analysis of several frameworks. We show that, despite their apparent simplicity, our definitions and algorithms are accurate.

First, we survey the literature in a quest for precise definitions of the binary class relationships at the design level and definitions at the implementation level (Section 2). From this state of the art and in face of the ambiguities and lack of coherence of the definitions, we propose synthetic definitions of the binary class relationships, at the design level. Then, we present definitions of the relationships as well as examples at the implementation level (Section 3). Definitions at the implementation level are not alone sufficient to bring continuity between the design level and the implementation level. We need algorithms to synthesize code and to detect the binary class relationships. These algorithms require precise definitions of the binary class relationships. Using their common properties (Section 4), we refine the definitions at the implementation level in terms of their properties (Section 5). Then, we present our algorithms (Section 6 and 7) and validate them on several well-know frameworks, such as JHOTDRAW (Section 8). Finally, we conclude by discussing our approach, some of its immediate applications, and its generalization.

2 State of the Art

There is a large body of work on binary class relationships, several authors already mentioned the ambiguities of the existing definitions and the lack of mapping between the design level and the implementation level. We do not pretend to present a complete survey of the research work on binary class relationships; Rather, we propose work representing the main approaches to tackle the ambiguities and the lack of mapping.

The reader already convinced of these ambiguities and of the lack of definitions at the implementation level may skip this section and proceed to Section 3.

The ambiguities are particularly visible for the definitions in natural language, even if these definitions are *de facto* standards, such as in the UML or in the CASE tool RATIONAL ROSE (Section 2.1).

Some authors investigate the use of formal methods to disambiguate the definitions (Section 2.2), or propose extensive studies of the properties of the binary class relationships (Section 2.3). However, they place their studies at the design level and do not consider the mapping with the implementation level.

At the implementation level, authors define new programming languages or extend existing ones (Section 2.4). However, as mentioned in the introduction, these solutions depart from the UML search for language independence and they remove the benefits of standardization.

Few authors tackle the problem of discontinuity between the design level and the implementation level (Section 2.5). Their works only partially address the mapping between the design level and the implementation level, with a standard object-oriented programming language.

2.1 Definitions in Natural Language

In the OMT methodology, Rumbaugh [42] considers only two kinds of binary class relationships. A *link* is a physical or conceptual connection between two instances. Mathematically, a link is a tuple, an ordered list of instances. A link is an instance of an association. An association describes a group of links with a common structure and semantics. An association describes a set of potential links, as a class describes a set of potential instances. Associations are fundamentally bi-directional. Aggregation is an association. The most significant property of an aggregation is its transitivity. The aggregation is not symmetrical.

For Skogan [45], users should manipulate an ordinary association when they want to represent general relationship between two classes. They should apply the aggregation relationship when the objects representing the parts of a container object can exist without the container object. A composition association is a strong aggregation, in the sense that if a container object is deleted all its contained objects are deleted as well. Users should exploit the composition relationship when the objects representing the parts of a container object *cannot exist without* the container object.

RATIONAL, with ROSE, proposes the most widely used software engineering tool. RATIONAL claims that its tool is 100% UML-compliant. In [39], RATIONAL proposes the following definitions for the association, aggregation, and composition relationships: A binary association is an association among exactly two classes (including the possibility of a reflexive association from a class to itself). Aggregation is entirely conceptual and does nothing more than distinguish a *whole* from a *part*. Simple aggregation does not change the meaning of navigation across the association between the whole and its parts, nor does it link the lifetimes of the whole and its parts. Composition is a form of aggregation with strong ownership and coincident lifetime as parts of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it. Such parts can also be removed before the death of the composite.

For Wien [47], if one object is logically related to one or more other objects there exists an association among those objects. Associations with higher degree than binary have to be implemented as objects. Aggregations are used to build composite objects. Aggregations need a special construct called *containment constructs* [48]. A *dependent* object is an object whose existence depends on the existence of another object. An aggregate object is an object with references to dependent objects and to independent objects. A composite object is a special kind of aggregate object. *Only one composite link* can point to a dependent object, but many other references to this object can exist. Composite links are necessary to realize *cascade delete*, *value propagation*, or *method propagation*. *Exclusive components* are parts of only one aggregate and *shared components* are parts of more than one aggregate [35]. There is a possibility for *recursion* with composite objects [35].

Every year, the OMG releases a new and revised version of UML specifications. In 1999 [37], the OMG gave the following definitions: A binary association is an association among exactly two classes⁵ (including the possibility of an association from a class to itself); An aggregation is a special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part; A composition is a form of aggregation with strong ownership and coincident lifetime of part with the whole. The part is strongly owned by the composite and may not be part of any other composite. Two years later, in 2001 [38], the OMG provided new and considerably longer definitions. In the new definitions, the OMG distinguishes several properties for the binary class relationships: *Multiplicity*; *Navigability*; *Replaceability*; *Shareability*; *Transitivity*; *Anti-symmetricalness*; *Lifetime*. However, as noted by other authors [28], the definitions manifest many ambiguities. The UML v1.4 is less ambiguous, but still does not mention implementation details.

⁵The real definitions involve *classifiers*, not classes, in the UML terminology.

2.2 Formal Definitions

For Noble *et al.* [36], association relationships indicate that one object makes use of another object in some way. Aggregation relationships specify the aggregation of objects to form larger object models. They do not give any definition of the composition relationship and do not define a mapping between their definitions and the implementation level.

Biccaregui *et al.*, for their formalization of the methodology for object-oriented analysis named Syntropy [6] with the object calculus, define an association as a many-many relation between objects of two classes. They propose two constraints: Cardinality and lifetime. The cardinality constraint indicates if the association is optional, compulsory, or one-to-one. The lifetime constraint is interpreted independently of multiplicities and indicates that an aggregate object only exists if linked with some set of other objects and that this set must be constant throughout its lifetime.

In their work [7], Breu *et al.* describe the use of streams and stream processing functions for the formalization of the UML. They define an association as either a set of data links, a separate class, or a communication link. However, they leave the choice of the appropriate semantics to the developers, without further discussing the criteria for this choice. They distinguish composition and aggregation relationships according to the shareability and lifetime properties of the parts; but, they do not further define those properties.

In his paper [18], France illustrates the use of formal specification techniques to develop a precise semantics for UML requirement class diagrams. Requirement class diagrams represent problem-oriented structures; i.e., problem domain concepts. Class diagrams decompose into two types of constructs: Classes and associations. Associations may be general associations, aggregations, compositions, or generalization structures. For associations, France only considers the *multiplicity* and *changeability* properties; the *navigability* and *visibility* properties being irrelevant to requirement class diagrams. Associations may also possess a weak or a strong *frozen* property related to the deletion of associated objects. Aggregation are associations with an additional *ownership* property. Composition are aggregation with a deletion property: If a whole is deleted, then all the parts that are *currently* associated with the whole are deleted. Parts may be removed from the whole before its deletion. However, France does not discuss the implementation of the different associations and of their properties.

For André *et al.* [3], an association defines a relation among classes as a set of links. Each link corresponds to a tuple of object, one per class in the relation. Each association can be enriched by several notations: *name*, *reading direction*, *roles*, *multiplicity*, *constraints*, *properties*, *qualification*, and *derivation*. For the aggregation, they do not give a specific definition, considering the aggregation carries the same semantics as plain association. Composition is a form of aggregation with strong ownership and coincident lifetime between the parts and the whole. The authors define strong ownership and coincident lifetime with respect to the shareability of the parts.

2.3 Properties of the Binary Class Relationships

In his precursory work, Civello [10] divides whole–part association (WPA) in two categories: Functional and non-functional WPA. In a functional WPA, also named *assembly-component*, the part is included in the whole because the part contributes to the function of the whole; while in a non-functional WPA, the connection between the whole and the part is looser. Non-functional WPAs divide in two categories: *Group-member*, which models an association relationship; *Tuple-element*, which models an aggregation relationship. Groups are sets of objects brought together because they share some common properties. Tuples model a relation between two entities, entities which normally exist independently from each other. The author proposes the first classification of WPA properties: *Spatial or temporal inclusion*; *Attribution*; *Visibility*; *Encapsulation*; *Sharing*; *Part-Whole inseparability*; *Whole–Part inseparability*; *Immutability*; *Ownership*; and, *Collaborations*. The author stays at the design level and concludes that current object-oriented methods and programming languages lack expressiveness to represent the richness in semantics of WPAs.

In their work [44], Saskena *et al.* defines the aggregation relationship in terms of three primary characteristics and three secondary characteristics. The primary characteristics are: *Structural*; *Lifetime binding*; and, *Ownership*. The secondary characteristics are: *Sharing of parts*; *Homeomerousity*⁶; and, *Property propagation*. In particular, they give an extensive study of the possible lifetime bindings between a part and a whole, and they mention the need for an aggregate class to have methods that call methods of the parts with regard to the ownership characteristic.

In their survey of the UML aggregation and composition relationships and other studies, Henderson-Sellers and Barbier [4, 27, 28] propose a set of characteristics for the whole–part relationship. Based on their primary, secondary, and derived characteristics, the authors show that the definitions of the aggregation and composition relationships in UML are incomplete, overlapping, and contradictory. They propose revised definitions of the aggregation and composition relationships, with four different options. According to the first option, *separability*, which is compatible with the current UML documentation and books, the aggregation relationship represents a whole–part relationship that is irreflexive at the instance level, antisymmetric at the class and instance level, and asymmetric at the instance level. The whole and the part are separable. The aggregation relationship induces the propagation of one or more operations from the whole to the part and the ownership of the part by the whole. However, there is no existential dependency between the whole and the part and no propagation of destruction operations. The composition relationship is an aggregation with existential dependency and propagation of the destruction operations. However, they do not tackle the problem of implementation with standard object-oriented programming languages.

In their work, Vauttier *et al.* [46] study the behavior of composite objects. In contrast with other works, which studied the composition relationship, they

⁶Sameness of part and whole

focus on the *behavior* of composite objects. They distinguish between the local behavior of a composite object and its global behavior. The local behavior corresponds to the functionalities particular to the class of the composite object and the classes of its components. The global behavior corresponds to the functionalities between a composite object and its components and between a component and the other components. The purpose of the dichotomy between local and global behavior is to improve the reusability of the components, the separation between the interface particular to the composite object and its interface as a composition front-end, and the behavior of the composite objects.

In their work, Bruel [8] propose improvements to the notions of aggregation and composition relationships without breaking any significant dislocation between the UML v1.x and UML v2.0. They base their work on the previous work of Henderson-Sellers, who studied the characteristic of whole-part relationships and their combinations, see above [28]. They describe UMLTRANZ, an automated transformation tool of UML class diagram into the Z formal description language.

2.4 New Programming Languages or Extensions

In their paper [25], Hartmann *et al.* present a new programming language: Troll. The Troll programming language features events and explicit object aggregation. In particular, complex objects describe aggregation of sub-objects into structured objects. In a complex object, sub-objects may be altered only by events local to the complex object, but their attributes are visible outside the complex object. There are three kinds of complex objects: *Static aggregation of objects*, where the composition is determined at compile-time; *Dynamic complex objects*, where the composition can change at run-time; and, *Disjoint complex objects*, where sub-objects cannot exist outside the complex object. Object inclusion is the semantic foundation of complex objects in Troll: Safe object import; Communication via event calling and sharing. The model of complex objects proposed in their paper is interesting because mathematically sound, however it requires a specific and non-standard programming language.

In his work [32], Kristensen introduces language mechanisms to support *implicit* and *complex* associations. Enclosing classes describe implicit associations. Association classes describe complex associations. For instance, the relationship between a **Customer** and a (banking) **Consortium** is a complex association. A **Customer** decomposes into an **Account** and a **Cash-Card**, thus defining an implicit association between **Customer** on the one hand and **Account** and **Cash-Card** on the other hand. A specific programming language supports both implicit and complex associations. However, the approach presented by the author does not treat neither of the composition relationship nor of the implementation issue.

Ducasse *et al.* [16] propose a reflective model to express and to automatically manage dependencies among objects. Their language, FLO, is an extension to Smalltalk and also exists for C++. In his thesis [15], Ducasse describes the use of his dependency mechanism to define the composition relationship. However, he does not mention about the association and the aggregation relationship.

2.5 Definitions at the Implementation Level

Rumbaugh, in his article [41], proposes an explicit representation of the relations among objects. The author argues that the “*use of relations as a semantic construct can have a major impact on the formulation and elucidation of a design, but only if they are considered as semantic constructs of similar weight to classes and generalization.*” The author defines a relation as associating instances of n classes, to state that these instances are associated in some way. However, the author does not distinguish among association, aggregation, and composition relationships.

In the Object Programming Laboratory project, Rousseau *et al.* [40] designed a mapping to include abstractions from object-oriented methodologies, such as OMT, into object-oriented programming languages, such as C++. However, they only address the problem of code synthesis with object-oriented programming languages, they do not tackle the problem of detection, and their project has been terminated.

For Martin [34], an association represents the ability of one instance to send a message to another instance. An aggregation is a typical whole-part relationship. This is exactly the same as an association with the exception that instances cannot have *cyclic aggregation* relationships. A composition is exactly like an aggregation, except that the lifetime of the part is controlled by the whole. The whole may take direct responsibility for *creating* or *destroying* the part or it may accept an already created part, and later *pass it on* to some other whole that assumes responsibility of it.

Jackson and Waingold develop a tool, WOMBLE, for the lightweight extraction of object-models from Java byte-codes [30]. They propose an inference system that distinguishes subclass and association relationships between classes. An association relationship may be annotated to show the multiplicity (zero or more, zero or one, exactly one) and the mutability (static or not) of the instances of the target class with respect to the instances of the origin class. They also propose heuristics to infer multiplicity and mutability properties and to deal with container classes. Jackson and Waingold’s research work is closely related to our own study and provides us with interesting ideas. However, they limit their research to the association relationship: They only mention the aggregation relationship and they do not discuss the composition relationship.

For Chaumon *et al.* [9], an association exists when one class references variables of another class. An aggregation exists when the definition of one class involves objects of the other class. These definitions are close to the implementation; however, they do not give a definition of the composition relationship.

In their paper [24], Harrison *et al.* propose a method to map UML designs

to the Java programming language. They introduce the notion of *cursor*, which encapsulates the complexity and the implementation of associations. However, they do not discuss the specific implementation details of cursors and associations.

In their research work [33], Marcos *et al.* propose some guidelines to translate an UML model into a model for object-relational databases. They summarize the definitions of the aggregation and composition relationships as given in [43]: An aggregation is a special form of association between classes, which represents the concept of *whole* and *part*. In a simple aggregation, several wholes can share a same part and the part has no lifetime restriction with regard to its wholes. A composition is a special kind of aggregation: A part belongs to one and only one whole. The part lives and dies with its whole. A part can be explicitly removed from its whole. The authors present an interesting mapping between the relationship definitions and their implementation in SQL 1999. However, they do not consider message sends and they use SQL 1999-specific language construct to constrain the life-cycle of the tables.

Definitions of the binary class relationships also exist within the software visualization community. For example, Eichelberger *et al.* present a graph drawing framework and a compiler to display Java programs using the UML notation [17]. However, researchers focus on visualization techniques, such as graph layout algorithms, rather than on information retrieval techniques and definitions. The interested reader may report to [12] for more information on visualization techniques.

3 Definitions of the Binary Class Relationships

From the previous state of the art, we now summarize the definitions of the binary class relationships at the design level in a general and as synthetic as possible way.

Then, we propose simple definitions of the binary class relationships at the implementation level. These definitions are closely related to the definitions in subsection 2.5, which address the implementation of the binary class relationships. With these definitions, we make explicit our choices for the implementation of the binary class relationships. Despite their simplicity, these definitions model with a great accuracy the relationships in real frameworks, see Section 8.

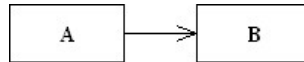
This section also presents examples of the binary class relationships implemented in Java. This section is *not* about proposing real-life examples. The reader might come up with more interesting and more complex examples. Yet, this is not essential for the purpose of this section. We aim at giving a *flavor* of the possible implementations of the binary class relationships.

3.1 Association relationship

Definition 1: At the design level

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

The following figure represents an association relationship between two classes A and B, using the UML notation.



Definition 2: At the implementation level

A binary class relationship involves the instances of two classes. A binary class relationship is oriented, irreflexive, anti-symmetric at the instance and class level, and asymmetric at the instance level [28]. An association between two classes A and B is the ability of an instance of A to send a message to an instance of B, with the possibility of mutual associations between the instances. The subclasses inherit the association relationship between classes A and B, because in class-based programming languages, subclasses inherit the structure and behavior of their superclasses. For example, if an association relationship exists between two classes A and B, an association relationship also exists between A and SubB, where SubB is a subclass of B.

The following code represents an association relationship between two classes A and B.

```
public class A {
    public void operation(B b) {
        b.operation();
    }
}
public class B {
    public void operation() {
    }
}
```

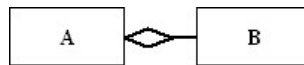
The association relationship between classes A and B exists through the method `void operation(B)`.

3.2 Aggregation relationship

Definition 3: At the design level

A binary aggregation relationship at the design level involves two classes, respectively the *whole* and the *part*. Conceptually, a part has no sense without a whole.

The following figure represents an aggregation relationship between two classes A and B, using the UML notation.



Definition 4: At the implementation level

An aggregation relationship exists when the definition of one class contains instances of the other class. It distinguishes a *whole* from a *part*. The aggregate class must define a field (or an array field, or a collection) of the type of the aggregated class. Instances of the aggregate class send messages to the referenced instance of the aggregated class.

The following code represents a aggregation relationship between two classes A and B.

```
public class A {
    public B b;
    public A(B b) {
        this.b = b;
    }
    public void operation() {
        this.b.operation();
    }
}
public class B {
    public void operation() {
    }
}
```

The aggregation relationship between classes A and B exists through the field B `b` and the method `void operation()`.

3.3 Composition relationship

Definition 5: At the design level

A composition relationship is an aggregation relationship where the parts held by the whole are destroyed when the whole is destroyed. These parts can be exchanged during the life-cycle of the whole, but all the parts owned by the whole at the moment of its deletion are destroyed in cascade.

The following figure represents a composition relationship between two classes A and B, using the UML notation.



Definition 6: At the implementation level

A composition is an aggregation between two classes, with a constraint between the lifetime of the instance of the whole and the lifetime of the instances of the part and a constraint on the ownership of the instance of the part by the instance of the whole.

The whole may take direct responsibility for creating the part or it may accept an already created part. A whole may pass a part to some other whole, which then assumes responsibility for it. When a whole is deleted, all its parts are deleted as well.

An instance of the whole owns the instance of its part. The instance of its part must not belong to any other whole (either through an aggregation relationship or a composition relationship). The instance of the part is exclusive to the instance of the whole.

The following code represents a composition relationship between two classes A and B.

```
public class A {
    private B b;
    public void attach(B b) {
        this.b = b;
    }
    public void operation() {
        this.b.operation();
    }
}
public class B {
    public void operation() {
    }
}
```

The composition relationship exists through the private field `B b`, the methods `void attach(B)` and `void operation()`, and the garbage collector: The Java Virtual Machine (JVM) collects for garbage the instance of class `B` stored in the instance of class `A` before collecting the instance of class `A`. The reader interested in the details of the garbage collector may report to Appendix A and to the counter-examples in Section 5.3. 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 `A` and `B`: The impossibility of obtaining, from outside of `A`, a reference on the field `B b` ensures the complete encapsulation of the instance of `B` in the instance of `A` and thus the lifetime dependency and the exclusive ownership.

4 Properties of the Binary Class Relationships

From Section 3, we identify a set of properties important to the definitions of binary class relationships at the implementation level.

This set of properties is not exhaustive. The interested reader may report to [10, 28] for discussions on all the properties of binary class relationships. Nevertheless, the properties we present here are necessary and sufficient to refine the previous definitions of the binary class relationships.

Then, we reformulate the definitions of the binary class relationships in terms of their properties. We illustrate the properties with examples in Java.

4.1 Definitions

Any binary class relationship possesses the four following properties:

Property 1: Exclusivity: $EX(\text{Class1}, \text{Class2})$

An instance of the class involved in the relationship can or cannot take part in another relationship at a given time.

$EX(\text{Class1}, \text{Class2}) \in \{true, false\}$.

In the `Country–Language` relationship, a `Language` may be part of more than one `Country`: $EX(\text{Country}, \text{Language}) = false$. In the `Computer–Keyboard` relationship, a `Keyboard` is part of one and only one `Computer`: $EX(\text{Computer}, \text{Keyboard}) = true$. The exclusivity property only holds at a given time: It does not prevent possible exchanges. In the `Computer–Keyboard` relationship, `Keyboard` is part of one and only one `Computer`, but it can be exchanged with another computer, for example in the binary class relationship `Laptop–Keyboard`.

Property 2: Message Send: $IMS(\text{Class1}, \text{Class2})$

Instances of `Class1`, involved in the relationship, send messages to instances of `Class2`.

$IMS(\text{Class1}, \text{Class2}) \subset \{indifferent, yes, no, field, array, field, collection, parameter, local\ variable\}$.

In the `Figure–Rectangle` relationship, the `Figure` propagates the `draw()` message to its `Rectangle`: $IMS(\text{Figure}, \text{Rectangle}) = \{yes\}$. But the `Rectangle` should not operate on its enclosing `Figure`: $IMS(\text{Rectangle}, \text{Figure}) = \{no\}$.

Property 3: Lifetime: $LT(\text{Class})$

This property constrains the lifetime of all the instances of class `Class`, involved in the relationships. The lifetime $LT(\text{Class})$ of an instance is the time elapsed between its instantiation $LF_i(\text{Class}) \in \mathbb{N} \cup \{indifferent\}$ and its destruction $LF_d(\text{Class}) \in \mathbb{N} \cup \{indifferent\}$, $LT(\text{Class}) = LF_d(\text{Class}) - LF_i(\text{Class}) \in \mathbb{N} \cup \{indifferent\}$ [10]. The time is given using any convenient unit, for example in seconds or in CPU ticks. In programming languages with garbage collection, $LF_d(\text{Class})$ matches with the moment where the instance is collected for garbage.

In the `Window–Button` relationship, when the `Window` closes and is ready for garbage collection (not accessible), the `Button` must also be ready for garbage collection (being not accessible): $LT_d(\text{Window}) \geq LT_d(\text{Button})$.

Property 4: Multiplicity: $MU(\text{Class1}, \text{Class2})$

The number of instances of a class `Class2` allowed in the relationship with class `Class1`. $MU(\text{Class1}, \text{Class2})$ is defined as $MU(\text{Class1}, \text{Class2}) \subset \mathbb{N}$. However, for the sake of simplicity of this paper, we use an interval of the minimum number and maximum number of instances to represent the multiplicity.⁷

In the `Cell-DNACode` relationship, a `Cell` possesses one and only one `DNACode`: $MU(\text{Cell}, \text{DNACode}) = [1, 1]$. On the contrary, in the `Car-Wheel` relationship, a `Car` usually possesses four `Wheels` but may have a fifth spare `Wheel`: $MU(\text{Car}, \text{Wheel}) = [4, 5]$.

4.2 Discussion

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

- The multiplicity property states the number of instance of class `Language` that possesses each instance of class `Country`: $MU(\text{Country}, \text{Language}) = [1, n]$. (Italy possesses one main language, Italian, and several secondary languages: Ladin; German; Turkish...)
- The exclusivity property states if an instance of class `Language` is shared among instances of class `Country` or of other classes: $EX(\text{Country}, \text{Language}) = \text{true}$. (We consider that Italy and Germany both speak German, possibly with different accents, vocabulary, and idioms.)

4.3 Implementation Examples

We now exemplify the properties using the Java programming language.

Example Exclusivity: $EX(\text{Class1}, \text{Class2})$

The exclusivity property corresponds to the following code:

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

In this example, the instance of class `A` is potentially shared with other: $EX(\text{B}, \text{A}) = \text{false}$, while the instance of class `B` is exclusive to the method `B operation()`: $EX(\text{A}, \text{B}) = \text{true}$.

Example Instance Message Send: IMS(Class1, Class2)

The instance message send property corresponds to the following code:

```
class A {
    private C c;
    public void operationB(B b) {
        b.operation();
    }
    public void operationC() {
        c.operation();
    }
}
public class B {
    public void operation() {
    }
}
public class C {
    public void operation() {
    }
}
```

In this example, an instance of class A may send a message to an instance of class B, through the parameter B b: $IMS(A,B) = \{\text{parameter}\}$. An instance of class A may also send a message to an instance of class C, through the field C c: $IMS(A,C) = \{\text{field}\}$.

Example Lifetime: LT(Class)

The following code illustrates the lifetime dependency between the instances of two classes A and B.

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

In this example, the instance of class B is exclusive to the instance of A. When the JVM collects the instance of class A for garbage, it collects first the instance of class B held by A: $LT_d(A) \geq LT_d(B)$.

Example Multiplicity: MU(Class)

The multiplicity property corresponds to the following code:

```
class A {  
    private B b;  
    private C c = new C();  
    private D[] d;  
    private E[] e = new E[] { new E() };  
}
```

In this example, an instance of class A possesses different relationships with classes B, C, D, and E. The multiplicities of the instances of these classes are: $MU(\mathbf{B}) = [0, 1]$; $MU(\mathbf{C}) = [1, 1]$; $MU(\mathbf{D}) = [0, n], n \in \mathbb{N}$; and, $MU(\mathbf{E}) = [1, n], n \in \mathbb{N}$.

5 Redefinitions of the Binary Class Relationships

5.1 Redefinitions

We now redefine the binary class relationships in terms of their properties.

Redefinition 1: Association relationship

An association relationship between two classes `Class1` and `Class2`, $AS(\text{Class1}, \text{Class2})$, has the following properties:

$$\begin{aligned}
 AS(\text{Class1}, \text{Class2}) = & \\
 & (IMS(\text{Class1}, \text{Class2}) = \{yes\}) \wedge \\
 & (IMS(\text{Class2}, \text{Class1}) = \{no\}) \wedge \\
 & (MU(\text{Class1}) = [0, n], n \in \mathbb{N}) \wedge \\
 & (MU(\text{Class2}) = [0, n], n \in \mathbb{N}) \wedge \\
 & (EX(\text{Class1}, \text{Class2}) = false) \wedge \\
 & (EX(\text{Class2}, \text{Class1}) = false) \wedge \\
 & (LT(\text{Class1}) = indifferent) \wedge \\
 & (LT(\text{Class2}) = indifferent)
 \end{aligned}$$

Nothing prevent other relationships to link classes `Class2` and `Class1`: An association, an aggregation, or a composition relationship may exist between `Class2` and `Class1`, see Section 3.1.

Redefinition 2: Aggregation relationship

Let assume two classes `Class1` and `Class2`, and an aggregation relationship between them: We call `Class1` and `Class2` respectively the `Whole` and the `Part`, and the aggregation relationship, $AG(\text{Class1}, \text{Class2})$, possesses the following properties:

$$\begin{aligned}
 AG(\text{Class1}, \text{Class2}) = & \\
 & (IMS(\text{Whole}, \text{Part}) = \{\text{field}, \text{array field}, \\
 & \quad \text{collection}\}) \wedge \\
 & (IMS(\text{Part}, \text{Whole}) = \{no\}) \wedge \\
 & (MU(\text{Whole}) = [1, 1]) \wedge \\
 & (MU(\text{Part}) = [0, n], n \in \mathbb{N}) \wedge \\
 & (EX(\text{Whole}, \text{Part}) = false) \wedge \\
 & (EX(\text{Part}, \text{Whole}) = false) \wedge \\
 & (LT(\text{Whole}) = indifferent) \wedge \\
 & (LT(\text{Part}) = indifferent)
 \end{aligned}$$

The definitions of the aggregation and the composition relationships, Sections 3.2 and 3.3, forbid the existence of a composition relationship between the `Part` and the `Whole`.

Redefinition 3: Composition relationship

When a composition relationship links two classes **Class1** and **Class2**, we call **Class1** and **Class2** respectively the **Whole** and the **Part**, and the composition relationship, noted $CO(\text{Class1}, \text{Class2})$, has the following properties:

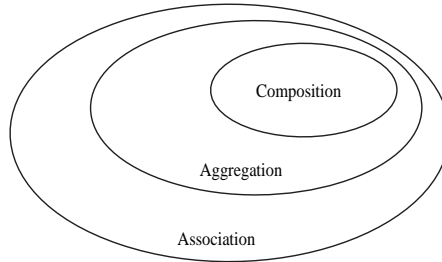
$$\begin{aligned} CO(\text{Class1}, \text{Class2}) = & \\ & (IMS(\text{Whole}, \text{Part}) = \{ \text{field}, \text{array field}, \\ & \quad \text{collection} \}) \wedge \\ & (IMS(\text{Part}, \text{Whole}) = \{ \text{no} \}) \wedge \\ & \quad (MU(\text{Whole}) = [1, 1]) \wedge \\ & \quad (MU(\text{Part}) = [1, n], n \in \mathbb{N}) \wedge \\ & (EX(\text{Whole}, \text{Part}) = \text{true}) \wedge \\ & (EX(\text{Part}, \text{Whole}) = \text{false}) \wedge \\ & (LT_d(\text{Whole}) \geq LT_d(\text{Part})) \end{aligned}$$

The definition of the composition relationship, Section 3.3, only allows an association relationship between the **Part** and the **Whole**.

5.2 Discussion

5.2.1 Order among binary class relationships

The properties show that there exists an order among the association, aggregation, and composition relationships, as seen of the figure below.



The properties of the aggregation relationship are more constraining than the properties of the association relationship:

	Association	Aggregation
$IMS(\text{Whole}, \text{Part})$	$\{ \text{yes} \}$	$\{ \text{field}, \text{array field}, \text{collection} \}$
$MU(\text{Class1})$	$[0, n], n \in \mathbb{N}$	$[1, 1]$

The properties of the composition relationship are more constraining than the properties 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 another aggregation or composition relationship.

	Aggregation	Composition
$MU(\mathbf{Part})$	$[0, n], n \in \mathbb{N}$	$[1, n], n \in \mathbb{N}$
$EX(\mathbf{Whole}, \mathbf{Part})$	<i>false</i>	<i>true</i>
$LT(\mathbf{Part})$	<i>indifferent</i>	$LT_d(\mathbf{Whole}) \geq$
$LT(\mathbf{Whole})$	<i>indifferent</i>	$LT_d(\mathbf{Part})$

This order also shows that the binary class relationships decomposes into two fundamental parts: A static part corresponding to the MU and IMS properties; A dynamic part corresponding to the EX and LT properties. This dichotomy between the static and dynamic parts of the binary class relationships is important for their code synthesis and their detection (see Section 6 and 7).

5.2.2 Symmetrical relationships and backpointers

The presence of a binary class relationship between two classes **A** and **B** does not preclude the existence of other relationships between the two classes or between **B** and **A**, except when specifically forbidden in the definition of the binary class relationship, as in the composition relationship.

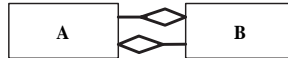
The relationships between two classes **A** and **B** can induce a cycle. Typically, there is a cycle when a backpointer records the owner of an instance or when two classes possess fields of each other's type:

```
public class A {
    private B component;
    ...
}

public class B {
    private A owner;
    ...
}
```

In such a case, two symmetrical aggregation relationships exist: $AG(\mathbf{A}, \mathbf{B})$ and $AG(\mathbf{B}, \mathbf{A})$.

The following figure shows the UML diagram corresponding to the existence of two symmetrical aggregation relationships between classes **A** and **B**.



5.2.3 Acquaintance and optimizations

For optimization reasons, an instance of a class **A** associated with an instance of a class **B** may declare a field of type **B**. According to our definitions, the relationship between **A** and **B** then evolves from an association relationship to an aggregation relationship. This is not a problem because we aim at bringing continuity between the design and the implementation levels. We do not try to interpret or to explain the developer's intent.

5.3 Implementation Examples

We now present several examples and counter-examples of the binary class relationships in Java and makes explicit their properties in the code. In this section, we do *not* propose real-life examples. Our goal is to give an *overview* of the possible implementations of the binary class relationships. We use these examples and counter-examples to illustrate our code synthesis and detection algorithms in Section 6 and 7.

Example 1: Association relationship

This example presents two classes A and B, which an association relationship links together. The association relationship exists through the method `void operation(B)`.

```
1 public class Association {
    // EX(A, B) = false,
    // EX(B, A) = false,
    // LT(A) = indifferent,
    // LT(B) = indifferent
5     public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.operation(b);
10        b.operation();
    }
}
public class A {
    // MU(A) = [0, n],
    public void operation(B b) { // MU(B) = [0, n],
15        b.operation(); // IMS(A, B) = yes
    }
}
public class B {
    // IMS(B, A) = no
20    public void operation() {
    }
21 }
```

The exclusivity properties $EX(A, B)$ and $EX(B, A)$ are *false* because of lines 7 and 8: The explicit naming of the instances of classes A and B allows other objects to reference and to use them.

The $IMS(Class1, Class2)$ is *{yes}* because of line 15: Instances of class A send messages to instances of class B. The $IMS(Class2, Class1)$ is *{no}* because class B does not send a message to class A.

The lifetime properties $LT(Class1)$ and $LT(Class2)$ are *indifferent*: No embedded references tie the lifetimes of instances of class A with instances of class B.

The multiplicity properties $MU(Class1)$ and $MU(Class2)$ are $[0, n], n \in \mathbb{N}$: Nothing prevents an instance of class A to reference and to use several instances of class B. Reciprocally, nothing prevents several instances of class A to reference and to use a same instance of class B.

Example 2: Another association relationship

This is another example of two classes A and B, which an association relationship links together.

```
1 public class Association {
    // EX(A, B) = false,
    // EX(B, A) = false,
    // LT(A) = indifferent,
5 // LT(B) = indifferent
    public static void main(String[] args) {
        A a = new A();
        B b = a.operation();
        b.operation();
10 }
    }
    public class A {
        public B operation() {           // MU(A) = [0, n],
            B b = new B();             // MU(B) = [0, n],
15         b.operation();              // IMS(A, B) = yes
            return b;
        }
    }
    public class B {
20     public void operation() {        // IMS(B, A) = no
    }
22 }
```

The exclusivity properties $EX(A, B)$ and $EX(B, A)$ are *false* because of lines 7 and 8: The explicit naming of the instances of classes A and B and the return type of method B `operation()` on line 16 allow other objects to reference and to use them.

The $IMS(Class1, Class2)$ is $\{yes\}$ because of line 15: Instances of class A send messages to instances of class B. The $IMS(Class2, Class1)$ is $\{no\}$ because class B does not send a message to class A.

The lifetime properties $LT(Class1)$ and $LT(Class2)$ are *indifferent*: No embedded references tie the lifetimes of instances of class A with instances of class B. The creation of an instance of class B in the method B `operation()` of class A does not tie their lifetimes: When the method B `operation()` returns, either the instance stays accessible through another reference (as in this example, on line 8), or no reference points on it anymore and it is ready for garbage collection, independently of the instance of class A.

The multiplicity properties $MU(Class1)$ and $MU(Class2)$ are $[0, n]$, $n \in \mathbb{N}$: Nothing prevents an instance of class A to reference and to use several instances of class B. Reciprocally, nothing prevents several instances of class A to reference and to use a same instance of B.

Example 3: Aggregation relationship

This example presents two classes A and B, which an aggregation relationship links together. Class A plays the role of whole, class B plays the role of part. The aggregation relationship exists through the field B `b` and the method `void operation()`.

```
1 public class Aggregation {
    // EX(A, B) = false,
    // EX(B, A) = false,
    // LT(A) = indifferent,
5   // LT(B) = indifferent
    public static void main(String[] args) {
        B b = new B();
        A a = new A(b);
        a.operation();
10        b.operation();
    }
    public class A {                // MU(A) = [1, 1]
        private B b;                // MU(B) = [1, 1]
15        public A(B b) {            // (in [0, n])
            this.b = b;
        }
        public void operation() {
            this.b.operation();      // IMS(A, B) = field
20    }
    }
    public class B {                // IMS(A, B) = no
        public void operation() {
25    }
    }
```

The exclusivity properties $EX(\text{Whole}, \text{Part})$ and $EX(\text{Part}, \text{Whole})$ are *false* because of lines 7 and 8: The explicit naming of the instances of classes A and B allows other objects to reference and to use them.

The $IMS(\text{Whole}, \text{Part})$ is *{field}* because of line 19: Instances of class A send messages to instances of class B through the field B `b`, line 14. The $IMS(\text{Part}, \text{Whole})$ is *{no}* because class B does not send messages to class A.

The lifetime properties $LT(\text{Whole})$ and $LT(\text{Part})$ are *indifferent*: The instance of class B may outlive the instance of class A. Reciprocally, the field B `b` of class A could be set to `null`: the instance of class A could outlive the instance of class B.

The multiplicity properties of $MU(\text{Whole})$ is $[1, 1]$ and of $MU(\text{Part})$ is $[1, 1]$: The instance of class A references one instance of class B; The instance of class B is not be part of another aggregation or composition relationship.

Example 4: Composition relationship

This example presents two classes **A** and **B**, which a composition relationship links together. Class **A** plays the role of whole, class **B** plays the role of part. The composition relationship exists through the private field **B b**, the methods `void attach(B)` and `void operation()`, and the garbage collector: The JVM collects for garbage the instance of class **B** stored in the instance of class **A** before collecting the instance of class **A**. The reader interested in the details of the garbage collector may report to Appendix A and to the counter-examples below. The privateness of field **B b** participates in the lifetime dependency between **A** and **B**: The privateness ensures the encapsulation of the instance of **B** in the instance of **A** and thus the lifetime dependency.

```
1 public class Composition {
    // EX(A, B) = true,
    // EX(B, A) = false,
    // LT(A) = indifferent,
5    // LTd(A) > LTd(B)
    public static void main(String[] args) {
        A a = new A();
        a.attach(new B());
        a.operation();
10    }
    public class A {
        private B b;
        public void attach(B b) {
15         this.b = b;
        }
        public void operation() {
            this.b.operation();
20        }
    }
    public class B {
        public void operation() {
24        }
    }
}
```

The exclusivity property of $EX(\mathbf{Whole}, \mathbf{Part})$ is *true* because no other object can reference the instance of class **B** created on line 8 beside the instance **a** of class **A**. The exclusivity property $EX(\mathbf{Part}, \mathbf{Whole})$ is *false* because of lines 7: The explicit naming of the instance of class **A** allows other objects to reference and to use it.

The $IMS(\mathbf{Whole}, \mathbf{Part})$ is *{field}* because of line 18: Instances of class **A** send messages to instances of class **B** through the field **B b**, line 15. The $IMS(\mathbf{Part}, \mathbf{Whole})$ is *{no}* because class **B** does not reference class **A**.

The lifetime properties of classes **Whole** and **Part** are:

$LT_d(\mathbf{Whole}) \geq LT(\mathbf{Part})$. The JVM collects the instance of class **B** before the instance of class **A**.

The multiplicity properties of $MU(\mathbf{Whole})$ is $[1, 1]$ and of $MU(\mathbf{Part})$ is $[1, 1]$: The instance of class **A** references one instance of class **B**; The instance of class **B** is not be part of another aggregation or composition relationship.

Example 5: Another composition relationship

This example proposes two classes A and B, linked by a composition relationship. The instance of class B is strongly owned by the instance of class A: The instance of the A class is responsible for creating the instance of class B.

```
1 public class Composition {
    // EX(A, B) = false,
    // LT(A) = indifferent,
    // LTd(A) > LTd(B)
5 public static void main(String[] args) {
    A a1 = new A();
    a1.operation();
}
10 public class A {
    private B b;
    public A() {
        b = new B();
15 }
    public void operation() {
        this.b.operation();
    }
}
20 public class B {
    public operation() {
    }
}
23 }
```

The exclusivity property of $EX(\text{Whole}, \text{Part})$ is *true* because no other object can reference the instance of class B created on line 14 beside the instance of class A. The exclusivity property $EX(\text{Part}, \text{Whole})$ is *false* because of line 6: The explicit naming of the instance of class A allows other objects to reference and to use it.

The $IMS(\text{Whole}, \text{Part})$ is *{field}* because of line 17: Instances of class A send messages to instances of class B through the field B b, line 11. The $IMS(\text{Part}, \text{Whole})$ is *{no}* because class B does not send messages to class A.

The lifetime properties of classes **Whole** and **Part** are:

$LT_d(\text{Whole}) \geq LT(\text{Part})$. The JVM collects the instances of class B before the instances of class A.

The multiplicity properties of $MU(\text{Whole})$ is [1, 1] and of $MU(\text{Part})$ is [1, 1]: The instances of class A reference one instance of class B, respectively; The instances of class B are not part of another aggregation or composition relationship.

Example 6: Interlaced composition relationship

This example presents two classes A and B, which two interlaced composition relationships link together. The composition relationships exist through the field B `b`, the methods `void attach(B)` and `void operation()`, and the garbage collector (see counter-example below).

```
1 public class Composition {
    // EX(A, B) = false,
    // LT(A) = indifferent,
    // LTd(A) > LTd(B)
5 public static void main(String[] args) {
    A a1 = new A();
    A a2 = new A();

    a1.attach(new B()); // EX(B, A) = true,
10 a1.attach(null);

    a2.attach(new B()); // EX(B, A) = true
    a2.operation();
    }
15 }
    public class A { // MU(A) = [1, 1]
        private B b; // MU(B) = [1, 1]
        public void attach(B b) { // (in [0, n])
            this.b = b;
20 }
        public void operation() {
            this.b.operation(); // IMS(A, B) = field
        }
    }
25 public class B { // IMS(A, B) = no
        public operation() {
        }
28 }
```

The exclusivity property of $EX(\text{Whole}, \text{Part})$ is *true* because no other object can reference the instances of class B created on lines 9 and 12 beside the instances `a1` and `a2` of class A, respectively. The exclusivity property $EX(\text{Part}, \text{Whole})$ is *false* because of lines 6 and 7: The explicit naming of the instances of class A allows other objects to reference and to use them.

The $IMS(\text{Whole}, \text{Part})$ is *{field}* because of line 22: Instances of class A send messages to instances of class B through the field B `b`, line 17. The $IMS(\text{Part}, \text{Whole})$ is *{no}* because class B does not send messages to class A.

The lifetime properties of classes **Whole** and **Part** are:

$LT_d(\text{Whole}) \geq LT(\text{Part})$. The JVM collects the instances of class B before the instances of class A.

The multiplicity properties of $MU(\text{Whole})$ is $[1, 1]$ and of $MU(\text{Part})$ is $[1, 1]$: The instances of class A reference one instance of class B, respectively; The instances of class B are not part of another aggregation or composition relationship.

Example 7: Counter-example of composition relationship

The following code is a counter-example of the composition relationship. The binary class relationship between classes A and B is not a composition relationship. The composition relationship does not exist because the instance of class B may potentially be used “outside” of A. Thus, the exclusivity and lifetime properties do not hold: $\neg EX(A, B) \wedge \neg(LT_d(A) \geq LT_d(B))$.

```
public class Composition {
    public static void main(String[] args) {
        A a = new A();           // EX(B, A) = false
        B b = new B();           // not(EX(A, B) = true)
        a.attach(b);
        a.operation();           // LT(A) = indifferent
    }
}

public class A {                // MU(A) = [1, 1]
    private B b;                 // MU(B) = [1, 1]
    public attach(B b) {         // (in [0, n])
        this.b = b;
    }
    public void operation() {    // not(LTd(A) > LTd(B))
        this.b.operation();     // IMS(A, B) = field
    }
}

public class B {                // IMS(B, A) = no
    public operation() {
    }
}
```

Example 8: Counter-example of interlaced composition relationship

The following code is a counter-example of interlaced composition relationships. The binary class relationship between classes A and B is not a composition relationship. The composition relationship does not exist because the instance of class B may potentially be used “outside” of A. Thus, the exclusivity and lifetime properties do not hold: $\neg EX(A, B) \wedge \neg(LT_d(A) \geq LT_d(B))$.

```

public class Composition {
    // EX(B, A) = false,
    // LT(A) = indifferent
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new A();
        a1.attach(new B()); // EX(A, B) = true
        a1.attach(null);
        a2.attach(new B());
        a2.operation();
        B b = a2.getB(); // EX(A, B) = false
        a2 = null;
        a1 = null;
        b.operation();
    }
}

public class A { // MU(A) = [1, 1]
    private B b; // MU(B) = [1, 1]
    public attach(B b) { // (in [0, n])
        this.b = b;
    }
    public B getB() { // not(LTd(A) > LTd(B))
        return this.b;
    }
    public void operation() {
        this.b.operation(); // IMS(A, B) = field
    }
}

public class B { // IMS(B, A) = no
    public operation() {
    }
}

```

Example 9: Another counter-example of interlaced composition relationship

The following code is a counter-example of interlaced composition relationships. The binary class relationship between classes A and B is not a composition relationship. The composition relationship does not exist because the instances of class B may potentially be used “outside” of A. Thus, the exclusivity does not hold: $\neg EX(A, B)$, even though the lifetime property holds: $LT_d(A) \geq LT_d(B)$.

```
public class Composition {
    public static void main(String[] args) {
        A a1 = new A();           // EX(B, A) = false
        A a2 = new A();           // EX(B, A) = false

        a1.attach(new B());
        a2.attach(a1.getB());     // EX(A, B) = false

        a1.operation();
        a2.operation();

        // LTd(B) < LTd(A)
    }
}

public class A {                 // MU(A) = [1, 1]
    private B b;                 // MU(B) = [1, 1]
    public attach(B b) {        // (in [0, n])
        this.b = b;
    }
    public B getB() {
        return this.b;
    }
    public void operation() {
        this.b.operation();    // IMS(A, B) = field
    }
}

public class B {                 // IMS(B, A) = no
    public operation() {
    }
}
```

6 Code Synthesis Algorithms

In the previous sections, we gave definitions of the binary class relationships at the design level and at the implementation level. We express the definitions at the implementation level in terms of common properties.

This is not yet sufficient to bring continuity between the design level and the implementation level. We need to define code synthesis and detection algorithms for the binary class relationships according to their definitions and to their properties.

We base the code synthesis on a meta-model [2]. We use it to describe constructions of the Java programming language: This meta-model describes entities such as class or interface, and elements such as field, method, and the association, aggregation, and composition relationships. We build models of software architecture using these entities and elements. Each entities and elements know how to synthesize its corresponding code, according to our definitions.

Each entity and element embeds one and only one possible choice of implementation, the choices exemplified in Section 5.3. One choice is enough because we are not interested in giving a full range of possible implementations but only in providing one implementation that respects the desired properties.

Thus, the code synthesis related to the binary class relationships does not present major difficulties. We propose here an overview of our code synthesis mechanisms, the interested reader may refer to [1] for more details.

Example Code synthesis of an aggregation relationship

The following code describes an aggregation relationship between two classes A and B. We define the two classes and their aggregation using our meta-model (classes `PClass`, `PAssociation`). The declaration of the aggregation relationship, line 5, specifies the name of the relationship (`aggregation`), the target class (`aClassB`), and the cardinality (2 implies of cardinality of n).

```
1 PClass aClassA = new PClass("A");
  PClass aClassB = new PClass("B");
  // Aggregation "aggregation".
  PAggregation anAggregation =
5   new PAggregation("aggregation", aClassB, 2);
6 aClassA.addElement(anAggregation);
```

Then, we ask the model to synthesize its corresponding code. We obtain the following result:

```
1 public class A {
  // Aggregation: aggregation
  private java.util.Vector aggregation =
    new java.util.Vector();
5   public void addB(B aB) {
      aggregation.addElement(aB);
    }
  public void removeB(B aB) {
      aggregation.removeElement(aB);
10  }
  }
  public class B {
13 }
```

This piece of code satisfies the definition and the properties we present in the previous sections.

Similarly, we produce results for the association and the composition relationship that satisfy the desired properties. The pieces of code we generate satisfy *per se* the desired properties, but it is the developer's duty to modify them while maintaining the desired properties. The interested reader may refer to [16, 24, 41] for discussions on mechanisms to preserve the desired properties.

7 Detection Algorithms

7.1 Principles

The association, aggregation, and composition relationships decompose into four properties: Exclusivity, message send, lifetime, and multiplicity. The detection of the binary class relationships implies inferring the values for the different properties and comparing these inferred values with the desired values.

The detection of the association relationship only requires the value of the $IMS(\text{Class1}, \text{Class2})$ property. The detection of the aggregation relationship requires inferring the values of the $IMS(\text{Class1}, \text{Class2})$ and $MU(\text{Class})$ properties. The detection of the composition relationship requires the detection of the value of the $IMS(\text{Class1}, \text{Class2})$ and the $MU(\text{Class})$ properties as for the aggregation relationship, and of the values of the $EX(\text{Whole}, \text{Part})$, $LT_d(\text{Whole})$, and $LT_d(\text{Part})$ properties.

We realize the detection of the values of the message send and multiplicity properties using static analysis. We infer the values of the exclusivity and lifetime properties using dynamic analysis.

7.2 Static Detection Algorithms

The detection of the static part of the binary class relationships is simple to perform using Java capabilities and specificities.

7.2.1 Implementation Concerns

The detection algorithms we present in this section are specifically tailored for Java. The Java programming language is a class-based programming language with introspection capabilities, which we use for part of the static detection.

The Java programming language uses an intermediate language, made of byte-codes, and a virtual machine (JVM). The JVM interprets the byte-codes contained in the class files, obtained by compiling Java source code.

We perform other part of the static analysis on the intermediate language, using a byte-code analysis framework: IBM CFPARSE v1.21. We chose to operate on the intermediate language because the class files are always available, while the Java source code may be proprietary.

7.2.2 Detection of $IMS(\text{Class1}, \text{Class2})$

We iterate through the byte-codes of each class, looking for the byte-codes corresponding to a message invocation: `InvokeInterface`, `InvokeStatic`, `InvokeSpecial`, and `InvokeVirtual`.

```
public List getIMS(ClassFile aClassFile) {
    final List ims = new ArrayList();
    final MethodInfoList methodInfoList =
        aClassFile.getMethods();
```

```

for (int j = 0; j < methodInfoList.length(); j++) {
    final MethodInfo methodInfo = methodInfoList.get(j);
    final AttrInfoList attributeInfoList =
        methodInfo.getAttrs();
    final CodeAttrInfo codeAttributeInfo =
        (CodeAttrInfo) attributeInfoList.get("Code");
    final byte[] rawCode = codeAttributeInfo.getCode();
    final ImmutableCodeSegment immutableCodeSegment =
        new ImmutableCodeSegment(rawCode);

    for (int j = 0;
         j < immutableCodeSegment.getNumInstructions();
         j++) {

        int offset =
            immutableCodeSegment.getOffset(j);
        int opCode =
            ByteArray.getByteAtOffset(rawCode, offset);

        switch (opCode) {
            case JVMConstants.INVOKEINTERFACE :
            case JVMConstants.INVOKESTATIC :
            case JVMConstants.INVOGESPECIAL :
            case JVMConstants.INVOKEVIRTUAL :
                // Add the message send to the list.
        }
    }
}
return ims;
}

```

7.2.3 Detection of *MU(Class)*

The detection of the values of the *MU(Class)* property corresponds to the fields and their multiplicities. We find typed fields, based on array (i.e., with cardinality 1 and n) using the standard Java reflection API: From the classes, we extract the existing fields and their corresponding types.

```

public List getMU(Class aClass) {
    final List mufields = new ArrayList();
    int nbUntypedCollections = 0;

    // Looking for any collection and distinguishing between
    // arrays (typed collections) and other collections
    // (un-typed collections).
    Field[] fields = aClass.getDeclaredFields();
    for (int i = 0; i < fields.length; i++) {
        Class type = fields[i].getType();
        if (type.isArray()) {
            mufields.add(
                new MUField(
                    type.getComponentType(),
                    MUField.ARRAY_BASED));
        }
        else if (isCollection(type)) {
            nbUntypedCollections++;
        }
    }
}

```

The difficulty arises when fields are typed as Java collections (*Map*, *List*, or *Set*), because these collections are un-typed. If we assume that these kinds of collections are homogeneous (with elements with a common superclass different

from `Object`) [30], it is possible to determine their types using well-known Java programming idioms, such as pairs of `add()`–`remove()` accessors. The following algorithm describes a technique to find accessors and `add()`–`remove()` pairs.

```

// Looking for accessors to type the un-typed collections.
if (nbUntypedCollections > 0) {
    Set removeMethods = new HashSet();
    Map adds = new HashMap();
    Method[] methods = aClass.getDeclaredMethods();
    for (int i = 0; i < methods.length; i++) {
        Method currentMethod = methods[i];
        String key = makeKey(currentMethod);
        if (isAddIdiom(currentMethod)) {
            adds.put(key, currentMethod);
        }
        else if (isRemoveIdiom(currentMethod)) {
            removeMethods.add(key);
        }
    }
}

// looking for matching add--remove pairs.
for (Iterator i = adds.keySet().iterator();
     i.hasNext();) {
    String addKey = (String) i.next();
    Method addMethod = (Method) adds.get(addKey);
    if (removeMethods.contains(addKey)) {
        mufields.add(
            new MUFIELD(
                addMethod.getParameters()
                [MUFIELD.ARGUMENT_NUMBER],
                MUFIELD.IDIOM_BASED));
    }
}

return mufields;
}

```

7.2.4 Static Detection Example

We develop a prototype tool, PATTERNSBOX [1], which includes the algorithms for static detection. We apply our algorithms on the examples of Section 5.3, we find the correct association and aggregation relationships.

We applied PATTERNSBOX on realistic applications, Section 8 details some of the results.

7.3 Dynamic Detection Algorithms

The dynamic part of the composition relationship is the most difficult to detect. We propose a solution to this problem using a trace-analysis technique.

7.3.1 Model

We perform the dynamic analysis of a trace with the help of predicates written with a logic programming language, Prolog. We use Prolog because it possesses high-level pattern-matching capabilities and because it is expressive enough, through its backtrack mechanism. Prolog already showed its adequacy to query traces in different works [14, 13].

We model a program execution as a trace: A sequence of execution events. There are three kinds of events, represented as Prolog compound terms (for the sake of readability, we associate each event with a unique chronological event number):

- Assignment events generated every time a field of an instance of a class A, numbered 1000, is assigned with an instance of class B, numbered 1001:

```
assignment(<Event number>, A, 1000, B, 1001)
```

- Finalize events generated when the JVM garbage-collects an instance. In the following, we assume our tool generates a finalize event as soon as an instance becomes useless (i.e., is ready for garbage collection):

```
finalization(<Event number>, A, 1000)
```

- A program-end event that is generated when the program terminates. We need this event to manage pending finalizations:

```
programEnd(<Event number>)
```

7.3.2 Detection of $LT(\text{Class})$

We represent the execution of the Java program in Example 4, Section 5.3, by the following list, TRACE4:

```
TRACE4 = [
  assignment(1, A, 1000, B, 1001),
  finalization(2, B, 1001),
  finalization(3, A, 1000),
  programEnd(4)
].
```

This trace verifies the lifetime property of the composition relationship ($LT_d(\text{Whole}) \geq LT_d(\text{Part})$). Indeed, first a field of the instance numbered 1000 of the A class is assigned with the instance 1001 of the B class. Second, the instance 1001 of B is finalized before the instance 1000 of A is finalized. Third, the program ends.

The following trace, also from Example 4 but with different memory settings, verifies the lifetime property of the composition relationship as well:

```
[
  assignment(1, A, 1000, B, 1001),
  programEnd(2),
  finalization(3, A, 1000),
  finalization(4, B, 1001)
]
```

Indeed, the finalizations only occur *after* the end of the program, as a benevolent effort from the JVM to free resources in a clean way (i.e., before it actually exits and the operating system frees any remaining allocated-resources). We consider these finalizations only as reminiscences of the program execution

and thus neglect the conflict between their chronological order and the lifetime property of the composition relationship.

We propose to formally define and check the lifetime property of the composition relationship with the help of a Prolog predicate, `checkLTProperty/3`. This predicate builds a list of compound terms representing sequences of events in the execution trace. Sequences of events are of three kinds:

- `pendingAssignment(EID, A, AID, B, BID, [])`
When the predicate encounters an assignment.
- `pendingAssignment(EID, A, AID, B, BID, [finalization(B, BID)])`
When the predicate has encountered an assignment and then the finalization of the second instance, variable B.
- `lifetimeProperty(A, AID, B, BID, true)`
When the instances numbered AID and BID of class A and B verify the lifetime property, `lifetimeProperty(A, AID, B, BID, false)` when they do not.

We define the `checkLTProperty/3` predicate as follows:

```

checkLTProperty(
    assignment(_, A, AID, B, BID),
    LIST,
    NLIST) :-
    append(LIST, [pendingAssignment(A, AID, B, BID, [])], NLIST).
checkLTProperty(
    finalization(EID, A, AID),
    LIST,
    NLIST) :-
    doesLTPropertyHold(finalization(EID, A, AID), LIST, NLIST).
checkLTProperty(
    programEnd(_),
    LIST,
    NLIST) :-
    convertPendingAssignations(LIST, NLIST).
checkLTProperty(_, LIST, LIST).

```

Depending on the event that the JVM emits, we record a compound term with the encountered assignment-event, or we check all pending assignments with the encountered finalization-event, or we convert all pending assignments when the program ends.

We check all pending assignments on receiving a finalization-event using the following `doesLTPropertyHold/3` predicate.

```

doesLTPropertyHold(finalization(EID, A, AID), [], []).
doesLTPropertyHold(
    finalization(_, B, BID),
    [pendingAssignment(A, AID, B, BID, []) | REST],
    [pendingAssignment(A, AID, B, BID, [finalization(B, BID)]) | NLIST]) :-
    doesLTPropertyHold(Y, LIST, NLIST).
doesLTPropertyHold(
    finalization(_, A, AID),
    [pendingAssignment(A, AID, B, BID, [finalization(B, BID)]) | REST],
    [lifetimeProperty(A, AID, B, BID, true) | REST]).

```

```

doesLTPropertyHold(
    finalization(_, A, AID),
    [pendingAssignment(A, AID, B, BID, []) | REST],
    [lifetimeProperty(A, AID, B, BID, false) | REST]).
doesLTPropertyHold(
    finalization(EID, A, AID),
    [CAR | CDR],
    [CAR | NCDR]) :-
    doesLTPropertyHold(finalization(EID, A, AID), CDR, NCDR).

```

When receiving a finalization or a program-end event, we iterate through the list of event-sequences and convert pending assignments into `lifetimeProperty` compound terms as required, using the `convertPendingAssignations/2` predicate.

```

convertPendingAssignations([], []).
convertPendingAssignations(
    [pendingAssignment(A, AID, B, BID, [_]) | REST],
    [lifetimeProperty(A, AID, B, BID, true) | NLIST]) :-
    convertPendingAssignations(REST, NLIST).
convertPendingAssignations(
    [pendingAssignment(A, AID, B, BID, []) | REST],
    [lifetimeProperty(A, AID, B, BID, true) | NLIST]) :-
    convertPendingAssignations(REST, NLIST).
convertPendingAssignations(
    [A | REST],
    [A | NLIST]) :-
    convertPendingAssignations(REST, NLIST).

```

The evaluation of the `checkLTProperty/3` predicate on trace `TRACE4` return the following list:

```
[lifetimeProperty(A, 1000, B, 1001, true)]
```

This list expresses that the lifetime property of the composition relationship holds for the instances 1000 and 1001 of classes A and B, respectively.

Let us now consider the trace `TRACE6`, obtained with Example 6, Section 5.3.

```

TRACE6 = [
    assignation(1, A, 1000, B, 1001),
    finalization(2, A, 1000),
    finalization(3, B, 1001)
    programEnd(4),
].

```

In this case, the evaluation of the `checkLTProperty/3` predicate does not recognize the lifetime property of the composition relationship, because the instance of B is finalized *after* the instance of A:

```
[lifetimeProperty(A, 1000, B, 1001, false)]
```

The trace `TRACE5`, obtained with Example 5, Section 5.3, contains two interlaced occurrences of the lifetime-property pattern of events.

```

TRACE5 = [
    assignation(1, A, 1000, B, 1001),
    finalization(2, B, 1001),
    assignation(3, A, 1002, B, 1003),
    finalization(4, A, 1000),
    finalization(6, B, 1003),
    finalization(5, A, 1002),
    programEnd(7),
].

```

The result of the evaluation of `checkLTProperty/3` on `TRACE5` is the list:

```
[lifetimeProperty(A, 1000, B, 1001, true),
 lifetimeProperty(A, 1002, B, 1003, true)]
```

The trace `trace7`, obtained with Example 7, Section 5.3, contains only one occurrence of the lifetime-property pattern of events (the second pattern is not correct).

```
TRACE7 = [
  assignation(1, A, 1000, B, 1001),
  finalization(2, B, 1001),
  assignation(3, A, 1002, B, 1003),
  finalization(4, A, 1000),
  finalization(5, A, 1002),
  finalization(6, B, 1003),
  programEnd(7),
].
```

The `checkLTProperty/3` predicate returns the list:

```
[lifetimeProperty(A, 1000, B, 1002, true)
 lifetimeProperty(A, 1001, B, 1003, false)]
```

7.3.3 Detection of *EX*(Class)

Following the same principle, we define a predicate to check the exclusivity property of the composition relationship, `checkEXProperty/3`.

```
checkEXProperty(
  assignation(_, A, AID, B, BID),
  LIST,
  NLIST) :-
  updateEXProperties(A, AID, B, BID, LIST, NLIST, true).
checkEXProperty(_, LIST, LIST).
```

We store in a list all assignations as `exclusivityProperty` compound term and update accordingly previous terms:

```
updateEXProperties(
  A, AID, B, BID,
  [],
  [exclusivityProperty(A, AID, B, BID, true)],
  true).
updateEXProperties(
  A, AID, B, BID,
  [],
  [exclusivityProperty(A, AID, B, BID, false)],
  false).
updateEXProperties(
  A, AID, B, BID,
  [exclusivityProperty(A0, AID0, B, BID, true) | REST],
  [exclusivityProperty(A0, AID0, B, BID, false) | NREST],
  EX) :-
  updateEXProperties(A, AID, B, BID, REST, NREST, false).
updateEXProperties(
  A, AID, B, BID,
  [exclusivityProperty(A0, AID0, B0, BIDO, true) | REST],
  [exclusivityProperty(A0, AID0, B0, BIDO, true) | NREST],
  EX) :-
  updateEXProperties(A, AID, B, BID, REST, NREST, EX).
```

```

updateEXProperties(
  A, AID, B, BID,
  [CAR | CDR],
  [CAR | NCDR],
  EX) :-
  updateEXProperties(A, AID, B, BID, CDR, NCDR, false).

```

We iterate through the list of exclusivity properties and update them according to the new assignation event emitted by the JVM. Then, we add a new term with the appropriate values.

7.3.4 Dynamic Detection Example of the Composition Relationship

Now, we define a predicate that verifies the composition relationship on a trace. We specify the `compositions/1` predicate using the `checkEXProperty/3` and `checkLTProperty/3` predicates, as follows:

```

compositions(LIST) :-
  checkProperties([], LEXP, [], LLTP),
  checkComposition(LEXP, LLTP, LIST).

```

The `compositions/1` predicate checks the exclusivity and lifetime properties on a trace generated by the JVM. The `checkProperties/4` predicate drives the remote JVM, receives events from the remote JVM, and calls the two predicates `checkEXProperty/3` and `checkLTProperty/3`.

```

checkProperties(LEXP, NNLEXP, LLTP, NLLTP) :-
  nextEvent(
    [generateConstructorEntryEvent,
     generateFieldModificationEvent,
     generateFinalizerExitEvent,
     generateProgramEndEvent],
    E),
  interpretEvent(E, IE),
  checkEXProperty(IE, LEXP, NLEXP),
  checkLTProperty(IE, LLTP, NLLTP),
  !,
  (
    (IE = programEnd,
     NNLEXP = NLEXP,
     NLLTP = NLLTP)
  ;
    checkProperties(NLEXP, NNLEXP, NLLTP, NLLTP)
  ).
checkProperties(LEXP, LEXP, LLTP, LLTP).

```

When the `checkProperties/4` predicate receives a program-end event, it returns two lists, respectively of exclusivity properties and of lifetime properties. Then, the `checkComposition/3` predicate fuses the two lists into a unique list of composition properties. A compound term `composition(A, AID, B, BID, OKAY)` describes each composition property.

```

checkComposition([], [], []).
checkComposition(
  [exclusivityProperty(A, AID, B, BID, OKAY) | EXPREST],
  [lifetimeProperty(A, AID, B, BID, OKAY) | LTPREST],
  [composition(A, AID, B, BID, OKAY) | NLIST]) :-
  checkComposition(EXPREST, LTPREST, NLIST).

```

```
checkComposition(  
    [exclusivityProperty(A, AID, B, BID, _) | EXPREST],  
    [lifetimeProperty(A, AID, B, BID, _) | LTPREST],  
    [composition(A, AID, B, BID, false) | NLIST]) :-  
    checkComposition(EXPREST, LTPREST, NLIST).
```

7.3.5 Implementation Concerns

The JVM provides an architecture to obtain dynamic information at run-time: The JAVA PLATFORM DEBUG ARCHITECTURE (JPDA). The JPDA provides an interface, the JAVA DEBUG INTERFACE (JDI), to get events from the JVM at runtime: The JVM fires events when a field is accessed or when a method is called.

We develop a prototype tool, CAFFEINE [23]. This tool instruments the JVM to generate a trace of the program execution, through the JDI interface.

Our tool generate a finalize event as soon as an instance becomes useless by calling the garbage collection at every step of the execution⁸ (for the sake of efficiency, incremental garbage collection should be considered).

⁸The interested reader may report to our example in Appendix A, where we explicitly call the garbage collector and the finalization methods

8 Validation

In this section, we validate our algorithms to detect association, aggregation, and composition relationships on three well-known frameworks, with respect to our definitions (supplemented by a manual analysis when required). Then, we validate our definitions by comparing the results of our algorithms against class diagrams and object-models documenting different frameworks.

8.1 Algorithms

We have tested implementations of the static detection algorithms, using our tool PATTERNSBOX, on several well known frameworks:

- JAVA AWT v1.2.2 [29]
- JHOTDRAW v5.2 [19]
- JUNIT v3.7 [20]

The association relationship is the weakest of the three binary class relationships, and provides an overwhelming number of hits (there are $2784 + 1505 + 636 = 4925$ association relationships for the 583 classes of the three frameworks). The aggregation relationship is the most informative with respect to static analysis.

The following table shows the results of the detection of the aggregation for the different frameworks. It details the number: Of *classes* per framework; Of relationships (multiplicity n) *found*; Of relationships (multiplicity n) detected by a *manual* analysis; Of *false hits*; And, of relationships *missed* by our algorithms.

Framework	Classes	Found	Manual	False hits	Missed
JAVA AWT v1.2.2	367	17	20	1	3
JHOTDRAW v5.2	171	6	8	0	2
JUNIT v3.7	45	1	4	0	3
Total	583	24	32	1	8
Precision: $0.75 \left(\frac{Found}{Manual} \right)$		Recall: $0.96 \left(\frac{Found}{Found+False\ hits} \right)$			

The results do not include the aggregation relationships of multiplicity 1, because the detection of these relationships is not an issue and would inaccurately increase the precision⁹. They only involve the aggregation relationships that are homogeneous, and that do neither involve primitive types, nor wrapper types, nor the `String` type. Such homogeneous aggregation relationships are typically implemented using the `Vector` class.

We perform the manual analysis ourselves to make sure our algorithms are accurate. The following tables summarize the existing aggregation relationships

⁹For example, on JHOTDRAW v5.2, the precision is of 98% for 151 existing aggregation relationships, multiplicity 1 and n , to compare with a precision of 75% for 8 existing aggregation relationships, multiplicity n .

we found according to our definitions in the three frameworks and highlight the aggregation relationships found by our algorithms.

For the sake of clarity, we do not include package names when there are no ambiguities. We do not mention relationships with listeners, because listeners are a separate concern with respect to the relationships among classes.

JAVA AWT v1.2.2		
Class	In aggregation relationship with	Class(es)
CardLayout	1	Component
Component	1	PopupMenu
Container	1	Component
Window	1	WeakReference
EventQueue	1	Queue
GridBagLayout	1	Component
MenuBar	1	Menu
Menu	1	MenuItem
StringSelection	1	DataFlavor
DragGestureRecognizer	1	InputEvent
TextJustifier	1	GlyphJustificationInfo
TextLine	1	TextLineComponent
TextMeasurer	1	TextLineComponent
Area	1	Curve
ColorConvertOp	2	ICC_Profile, ColorSpace
IndexColorModel	1	IndexColorModel
FilteredImageSource	1	ImageConsumer
MemoryImageSource	1	ImageConsumer
RenderableImageProducer	1	ImageConsumer
Book	1	Book\$BookPage
Total	21	

JHOTDRAW v5.2		
Class	In aggregation relationship with	Class(es)
NodeFigure	1	Connector
CompositeFigure	1	Figure
StandardDrawingView	4	Painter ($\times 2$), Figure ($\times 2$)
CommandMenu	1	Command
CommandChoice	1	Command
Iconkit	1	Image
StorableInput	1	Storable
StorableOutput	1	Storable
StorageFormatManager	1	StorageFormat
CustomToolBar	2	Component ($\times 2$)
PertFigure	2	PertFigure ($\times 2$)
PolyLineFigure	1	Point
Total	17	

JUNIT v3.7		
Class	In aggregation relationship with	Class(es)
TestResult	2	TestFailure ($\times 2$)
TestSuite	1	Test
junit.awtui.TestRunner	2	Test, Throwable
junit.swingui.TestRunner	1	TestRunView
TestTreeModel	3	Test ($\times 3$)
Total	9	

We do not obtain a precision of 100%, because the developers of the three frameworks did not respect some of the Java idioms required by our detection algorithms (for examples, an `add()` method without the corresponding `remove()` method or `get()` method, or a pair of `add()`–`remove()` methods with different argument types).

The following tables present the results of the detection of the composition relationship for the JUNIT v3.7 framework, using dynamic analysis on the `junit.samples.money.MoneyTest` class with text-based UI, AWT-based UI, and Swing-based UI, respectively. We did neither perform dynamic analysis on the JHOTDRAW v5.2 framework because it requires user-interaction nor on the JAVA AWT v1.2.2 framework because it is not a runnable program in itself.

We perform the analyses starting from the results of the aggregation relationship detection, because the composition relationship is an aggregation relationship with strong dynamic properties and because complete analyses of the frameworks would be too costly in time and would not provide more results.

JUnit v3.7		
Class	In composition relationship with	Class(es)
<code>TestResult</code>	2	<code>TestFailure</code> ($\times 2$)
<code>TestSuite</code>	1	<code>Test</code>
<code>junit.awtui.TestRunner</code>	N/A	<code>Test</code> , <code>Throwable</code>
<code>junit.swingui.TestRunner</code>	N/A	<code>TestRunView</code>
<code>TestTreeModel</code>	N/A	<code>Test</code> ($\times 3$)
Total	3	

When running the `junit.samples.money.MoneyTest` class with the text-based UI, the aggregation relationships reveal themselves as composition relationships: The properties of lifetime and of exclusivity hold.

When running the `junit.samples.money.MoneyTest` class with the AWT-based UI, the dynamic analysis also exposes the aggregation relationships as composition relationships. However, these results are subject to caution because they correspond to a subset of all possible execution paths. In case of failures or errors in tests, aggregation relationships remain what they are:

JUnit v3.7		
Class	In composition relationship with	Class(es)
<code>TestResult</code>	2	<code>TestFailure</code> ($\times 2$)
<code>TestSuite</code>	0	<code>Test</code>
<code>junit.awtui.TestRunner</code>	1 (<code>Throwable</code>)	<code>Test</code> , <code>Throwable</code>
<code>junit.swingui.TestRunner</code>	N/A	<code>TestRunView</code>
<code>TestTreeModel</code>	N/A	<code>Test</code> ($\times 3$)
Total	3	

Indeed, in case of failure or error, the property of exclusivity does not hold between classes `TestSuite` and `junit.awtui.TestRunner`, and the class `Test`. The `TestResult` class gives away the defective instance of class `Test` to an instance of class `junit.awtui.TestRunner`, which stores it in a collection.

When running the `junit.samples.money.MoneyTest` class with the Swing-based UI, the dynamic analysis displays results similar to the ones obtained with an AWT-based UI:

- Composition relationships exist between classes `TestResult` and `TestFailure`, and classes `junit.swingui.TestRunner` and `TestRunView`.
- Composition relationships do not exist between the `TestSuite` and `TestTreeModel` classes, and the `Test` class, because instances of class `Test` are shared among those two classes in case of failure or error.

JUnit v3.7		
Class	In composition relationship with	Class(es)
<code>TestResult</code>	2	<code>TestFailure</code> ($\times 2$)
<code>TestSuite</code>	0	<code>Test</code>
<code>junit.awtui.TestRunner</code>	N/A	<code>Test</code> , <code>Throwable</code>
<code>junit.swingui.TestRunner</code>	1	<code>TestRunView</code>
<code>TestTreeModel</code>	0	<code>Test</code> ($\times 3$)
Total	3	

These results demonstrate the interest of dynamic analysis and also its main limitation:

- The detection of strong relationships (with respect to their properties) may uncover subtle inter-classes dependencies and thus possible inter-classes design defects [22].
- Depending on the execution path, the relationships among classes may vary in a way unexpected by the developers.

Code coverage analyses [5, 11] should be considered to increase confidence on the analysis and therefore the exactness of the relationships among classes.

8.2 Definitions

The validation of our definitions and of results from our algorithms against third-party definitions and tools is a difficult task. We would like to apply our algorithms on documented frameworks and to compare the results with the documentation of these frameworks or the analyses of other tools. However, developers mainly use modelling language such as the UML, when designing their software, and thus they do not make a clear distinction among the association, aggregation, and composition relationships.

The authors were unable to find a framework both publicly available and of reference (such as JHOTDRAW or JUNIT) in which the developers explicitly distinguished the three binary class relationships.

For example, the authors of JHOTDRAW v5.2 use the notation proposed in [21]. This notation offers the association relationship, called *acquaintance* relationship, and a generic aggregation relationship, called the *part-of* relationship. The notation does not mention the composition relationship.

9 Conclusion and Future Work

In this paper, we tackle the implementation of three binary class relationships implementation in standard class-based programming languages. At the design level, we propose synthetic definitions of the common binary class relationships: Association, aggregation and composition. At the implementation level, we propose unambiguous definitions, code synthesis and detection algorithms, along with code examples. We implement the algorithms and we validate them on several well-known frameworks, with respect to our definitions and the developers' intent. This work is a first step towards bridging the gap between modeling and programming languages.

We currently apply our approach to design patterns. Our work focuses on relationships among classes, and thus directly applies to the detection and the code synthesis of design patterns, because design patterns are “*descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*” ([21], p. 3).

Further work includes:

- Completing our prototypes, with respect to the dynamic detection, using for example a reference-counter garbage-collector to generate the finalize events.
- Making our prototypes scalable, and testing them on bigger frameworks, such as the OTI ECLIPSE development environment.
- Investigating other UML notions, such as the interaction diagram, which we can check with trace-analysis algorithms.
- Applying our approach to other class-based programming languages. For instance, in C++, we could use an abstract-syntax tree instead of our meta-model for the static detection, and the destructor-related event to generate finalize events.
- Integrating our prototypes with the OTI ECLIPSE development environment.

A Composition and Finalization

Example Composition

This example presents a composition relationship. It illustrates the lifetime dependency between instances of class A and B. Because of the composition relationship between A and B, the instance of class B is ready for garbage collection before the instance of class A: The `finalize()` method of class B is called by the JVM before the `finalize()` method of class A.

```
public class Composition {
    public static void main(String[] args) {
        A a = new A(new B());
        a.operation();
        System.runFinalizersOnExit(true);
        System.gc();
    }
}

public class A {
    private B b;
    public A(B b) {
        super();
        this.b = b;
    }
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("Instance of class A finalized.");
    }
    public void operation() {
        b.operation();
        System.out.println("Method operation of class A.");
    }
}

public class B {
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("Instance of class B finalized.");
    }
    public void operation() {
        System.out.println("Method operation of class B.");
    }
}
```

The output produced when executing the code above is:

```
Method operation of class B.
Method operation of class A.
Instance of class B finalized.
Instance of class A finalized.
```

References

- [1] 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 ASE*, pages 166–173. IEEE Computer Society Press, November 2001.
- [2] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In Bedir Tekinerdogan, editor, *Proceedings of the ECOOP Workshop on Automating Object-Oriented Software Development Methods*, June 2001.
- [3] 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 LMO*, pages 261–276. Hermès Science Publications, January 2000.
- [4] Franck Barbier. The whole-part relationship in object modeling: A definition in cOIoR. *Information and Software Technology*, 43:19–39, 2001.
- [5] Boris Bezier. *Software Testing Techniques*. Van Nostrand Reinhold Company, New York, 1990.
- [6] Juan C. Bicarregui, Kevin C. Lano, and Tom S. E. Maibum. Objects, associations and subsystems: A hierarchical approach to encapsulation. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of ECOOP*, pages 324–343. Springer-Verlag, 1997.
- [7] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a formalization of the unified modeling language. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of ECOOP*, pages 344–366. Springer-Verlag, 1997.
- [8] 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 OOIS*, pages 5–14. Springer-Verlag, August 2001.
- [9] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller, François Lustman, and Guy Saint-Denis. Design properties and object-oriented software changeability. In *Proceedings of CSMR*, pages 45–54. IEEE Computer Society Press, 2000.
- [10] Franco Civello. Roles for composite objects in object-oriented analysis and design. In *Proceedings of OOPSLA*, pages 376–393. ACM Press, 1993.

- [11] 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.
- [12] Stephan Diehl, editor. *Software Visualization*. Springer-Verlag Berlin Heidelberg, May 2002. ISBN 3-540-43323-6.
- [13] Mireille Ducassé. Coca: A debugger for c based on fine grained control flow and data events. In *Proceedings of ICSE*, May 1999.
- [14] Mireille Ducassé. OPIUM: An extendable trace analyser for prolog. *The Journal of Logic Programming, special issue on Synthesis, Transformation and Analysis of Logic Programs*, 1999.
- [15] Stéphane Ducasse. *Intégration Réflexive de Dépendances Dans un Modèle À Classes*. PhD thesis, University of Nice – Sophia Antipolis – I3S, January 1997.
- [16] Stéphane Ducasse, Mireille Blay-Fornarino, and Anne-Marie Pinna-Dery. A reflective model for first class dependencies. In *Proceedings of OOPSLA*, pages 265–280. ACM Press, 1995.
- [17] Holger Eichelberger and Jürgen Wolff Von Gudenberg. On the visualization of java programs. In Stephan Diehl, editor, *Software Visualization*, pages 295–306. Springer-Verlag Berlin Heidelberg, 2002.
- [18] Robert B. France. A problem-oriented analysis of basic UML static requirements modeling concepts. In *Proceedings of OOPSLA*, pages 57–69. ACM Press, 1999.
- [19] Erich Gamma. JHotDraw, 1998. Available at <http://members.pingnet.ch/gamma/JHD-5.1.zip>.
- [20] Erich Gamma and Kent Beck. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.
- [21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [22] 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 TOOLS USA*, pages 296–305. IEEE Computer Society Press, July 2001.
- [23] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caffeine – a tool for dynamic analysis of Java programs. In *Proceedings of ASE*, 2002.

- [24] William Harrison, Charles Barton, and Mukund Raghavachari. Mapping UML designs to java. In *Proceedings of OOPSLA*, pages 178–188. ACM Press, 2000.
- [25] Thorsten Hartmann, Ralf Jungclaus, and Gunter Saake. Aggregation in a behavior oriented object model. In O. Lehrmann Madsen, editor, *Proceedings of ECOOP*, pages 57–77, Berlin, 1992. Springer-Verlag.
- [26] Linda Heaton. *Preface to the UML v1.4*, chapter 0.4, pages XXI–XXXVIII. Formal/01-09-71. OMG, September 2001.
- [27] Brian Henderson-Sellers. Some problems with the UML v1.3 metamodel. In *Proceedings of ICSS*. IEEE Computer Society Press, 2001.
- [28] Brian Henderson-Sellers and Franck Barbier. A suvery of the UML’s aggregation and composition relationships. In Jacques Malenfant and Roger Rousseau, editors, *Actes de LMO*, pages 339–366. Hermès, 1999.
- [29] Sun Microsystems Inc. Java abstract window toolkit. Technical report, Sun Microsystems Inc., 2000.
- [30] Daniel Jackson and Allison Waingold. Lightweight extraction of object models from bytecode. In *Proceedings of ICSE*, pages 194–202. ACM Press, May 1999.
- [31] Jeffrey Korn, Yih-Farn Chen, and Eleftherios Koutsofios. Chava: Reverse engineering and tracking of java applets. In *Proceedings of the Working Conference on Reverse Engineering*, pages 314–325, 1999.
- [32] Bent Bruun Kristensen. Complex associations: Abstractions in object-oriented modeling. In *Proceedings of OOPSLA*, pages 272–283. ACM Press, 1994.
- [33] Esperanza Marcos, Belen Vela, José M. Caverro, and Paloma Cáceres. Aggregation and composition in object - relational database design. In A. Caplinskas and J. Eder, editors, *Proceedings of the Fifth East-European Conference on Advances in Databases and Information Systems*, pages 195–209. Springer, 2001.
- [34] Robert C. Martin. Association, aggregation, and composition, 1998. ootips.org/uml-hasa.html.
- [35] Nelson M. Mattos, Klaus Meyer-Wegener, and Bernhard Mitschang. Grand tour of concepts for object-orientation from a database point of view. *Data and Knowledge Engineering*, 9:321–352, 1993.
- [36] James Noble and John Grundy. Explicit relationships in object-oriented development. In *Proceedings of TOOLS 18*. Prentice-Hall, 1995.
- [37] Object Management Group, Inc. *UML 1.3 Specification*, 1999.

- [38] Object Management Group, Inc. *UML 1.4 Specification*, 2001.
- [39] Rational. What is the difference between aggregation and composition and how are they represented in rose?, 2000. www.rational.com/technotes/rose_html/Rose_html/technote_10793.html.
- [40] B. Rousseau, A. Aimar, A. Khodabandeh, and P. Palazzi. Filling the gap between OO methodologies and programming languages. Programming Techniques Group, CERN ECP Division, 1211 Geneva 23, Switzerland, 1995.
- [41] James Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proceedings of OOPSLA*, pages 466–481. ACM Press, 1987.
- [42] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall Inc., Englewood Cliffs, 1991.
- [43] James Rumbaugh, Robert Jacobson, and Grady Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.
- [44] Monika Saksena, Robert B. France, and Maria M. Larrondo-Petrie. A characterization of aggregation. In *Proceedings of the 5th International Conference on Object-Oriented Information Systems, OOIS'98*. Springer-Verlag, September 1998.
- [45] David Skogan. Application schema specific data structure, 1999. www.informatics.sintef.no/UML2XML/std/Clause8.htm.
- [46] Sylvain Vauttier. Une nouvelle approche de la spécification du comportement des objets composites en UML. In *Proceedings of LMO*, pages 277–292. Hermès, 1999.
- [47] Universität Wien. Aggregation and composite objects., 2000. www.ifs.univie.ac.at/ISOO/OOCONCEPT/aggregation.html.
- [48] X3/SPARC/DBSSG/OOBTG. Final report 25. Technical Report 25, Xerox, August 1991.