This paper has been accepted at ASE 2003 as short paper.

A Pragmatic Study of Binary Class Relationships

Yann-Gaël Guéhéneuc[*], **Hervé Albin-Amiot** [†]
École des Mines de Nantes
4, rue Alfred Kastler – BP 20 823
44 307 Nantes Cedex 3
France
{guehene,albin}@emn.fr

Abstract

A discontinuity exists between modeling 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 software implementation and design and hampers software maintenance. This discontinuity is particularly acute for binary class relationships, such as the association, aggregation, and composition relationships. We present a solution to bridge the discontinuity between implementation and design for the binary class relationships: We propose consensual definitions of the binary class relationships in terms of four properties (exclusivity, invocation site, lifetime, multiplicity). We describe algorithms to detect these properties in Java source code.

1 Introduction

A recurrent problem in the object-oriented software (re)engineering community is the automated transition between software implementation and design during maintenance.

Modeling and programming languages possess similar notions, such as class and inheritance, guaranteeing the continuity between implementation and design.

However, modeling languages aspire to provide higher-level abstractions and thus include notions not-existing in programming languages, in particular binary class relationships.

[*]This work has been partly funded by IBM OTI Labs – 2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada

[†]This work has been partly funded by Soft-Maint S.A. – 4, rue du Château de l'Éraudière – 44 324 Nantes – France.

The existence of binary class relationships in modeling languages brings a discontinuity in the transition between a software implementation and its design.

This discontinuity hinders the understanding of the software implementation, limits the capabilities and efficiency of reverse engineering tools, and impedes the communication among software maintainers.

In this paper, we consider only the association, aggregation, and composition relationships because they exist in common modeling languages, such as the UML, but are not explicit in standard programming languages, such as Java.

Lot of work exist on the definitions of binary class relationships. However, none of this work tackles the discontinuity between implementation and design[1].

Some researchers study binary class relationships at an abstract level, without linking their results to implementation level constructs, such as [5, 12].

Some other researchers formalize object-oriented programming languages without focusing on implementation issues [1, 4, 15].

Yet some other researchers consider only subsets of the binary class relationships and of their definitions at implementation level, such as [7, 13].

Industrial and open-source CASE tools, such as Rational Rose and ArgoUML, distinguish only graphically the association, aggregation, and composition relationships. The results of their reverse engineering algorithms contain erroneous relationships.

We think essential to bridge the discontinuity between programming and modeling languages for the binary class relationships to provide maintainers with tools to reverse engineer accurately software and to offer consistent round-tripping [3, page 517].

[1]A more detailed state of the art is available in [9].

A first solution to bridge the discontinuity is designing a new programming language (or an extension to an existing one) including the definitions of binary class relationships, for examples [6, 11, 14]; However, this solution uses a non-standard programming language and thus eliminates the benefits of standardization (debuggers, efficient compilers…).

A second solution is defining binary class relationships at implementation level, in terms of constructs of an existing programming language; This solution relies on definitions at design and implementation levels and on detection algorithms, which bring continuity between implementation and design.

We apply the second solution to the Java programming language with a pragmatic study of UML-like association, aggregation and composition relationships.

In section 2, we propose consensual definitions of the binary class relationships and discuss their properties at implementation level. In section 3, we focus on four common properties: Exclusivity, invocation site, lifetime, and multiplicity. In section 4, we refine the definitions with these properties. In section 5, we propose algorithms to detect binary class relationships using their properties. In section 6, we conclude and present some future work.

2 Definitions

We propose definitions of the association, aggregation, and composition relationships. We advocate that these definitions are as consensual as possible with the state of the art [9]. Indeed, our approach requires that one accepts our definitions; However, more than definitions, our approach shows that it is possible to bridge software implementation and design.

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

At implementation level, most authors agree that a binary class relationship involves the instances of two classes, an origin and a target, respectively `A` and `B`. It is oriented, irreflexive, anti-symmetric at instance and class level, and asymmetric at instance level [12].

Thus, we propose that an association between `A` and `B` is the ability of an instance of `A` to send a message to an instance of `B`.

Nothing prevents other relationships to link classes `B` and `A`: An association, an aggregation, or a composition relationship may exist between `B` and `A`.

Aggregation relationship *At design level*, an aggregation relationship is an association between two classes, respectively the *whole* and the *part*. Conceptually, a part has no sense without a whole.

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

The whole must define a field (or an array field, or field of type collection) of the type of its part. Instances of the whole send messages to the instances of the part.

Subclasses inherit the aggregation relationship between `A` and `B`, because subclasses inherit the structure and behavior of their superclasses[2].

Composition relationship *At design level*, a composition relationship is an aggregation relationship where the parts held by the whole are destroyed when the whole is destroyed.

At implementation level, we define a composition as an aggregation with a constraint between the lifetimes of the whole and of the parts and a constraint on the ownership of the parts by the whole. The instances of the part are exclusive to the instance of the whole.

The definition of the composition relationship only allows an association relationship between the part and the whole, to ensure the lifetime and ownership properties between the whole and its part.

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

3 Properties

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

Exclusivity An instance of a class involved in a relationship can, or cannot, be in another relationship at a given time.

$$EX(\mathtt{A},\mathtt{B}) \in \{true, false\}$$

[2]Inheritance of structure and behavior is subject to access-control limitations

We name $\mathbb{B}$ the set $\{true, false\}$. The value *true* states that an instance of `B` can take part in another relationship with another instance of `A` or of another class. The exclusivity property only holds at a given time: It does not prevent possible exchanges.

Invocation site Instances of `A`, involved in a relationship, send messages to instances of `B`.

$$IS(\texttt{A, B}) \subset \emptyset \cup \{\texttt{field}, \texttt{array field}, \texttt{collection}, \texttt{parameter}, \texttt{local variable}\}$$

The values of the IS property summarize the invocation sites for messages sent from instances of `A` to instances of `B`. There can be no message sent from `A` to `B`: $IS(\texttt{A, B}) = \emptyset$, or messages can be sent from `A` through a $\{\texttt{field}\}$ of type `B`, an $\{\texttt{array field}\}$, a field of type $\{\texttt{collection}\}$, a method $\{\texttt{parameter}\}$, a method $\{\texttt{local variable}\}$. We name *yes* the set $\{\texttt{field}, \texttt{array field}, \texttt{collection}, \texttt{parameter}, \texttt{local variable}\}$.

Lifetime This property constrains the lifetime of instances of `B` with respect to the lifetime of instances of `A`. It corresponds to the time elapsed between the times of destruction LT_d of two instances of `A` and `B` [5].

$$\begin{array}{rcl} LT(\texttt{A, B}) & = & LT_d(A) - LT_d(B) \\ & \in & \{-, +\} \end{array}$$

We name $\|$ the set $\{-, +\}$. $LT(\texttt{A,B}) = +$ if instances of `B` are destroyed before the corresponding instances of `A`, $LT(\texttt{A,B}) = -$ if destroyed after, and $LT(\texttt{A,B}) \in \|$ if their times of destruction are unrelated.

Multiplicity The number of instances of `B` allowed in a relationship with `A`.

$$MU(\texttt{A,B}) \subset \mathbb{N} \cup \{+\infty\}$$

We use an interval of the minimum and maximum numbers to represent the multiplicity.

Discussions The four properties we propose to define binary class relationships are orthogonal. However, the exclusivity and the multiplicity properties are closely related with one another.

4 Formalization

We formalize the binary class relationships at implementation level as three conjunctions of the four properties: AS, AG, and CO.

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

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

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

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

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

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

Discussion The definitions of the binary class relationships decompose into two fundamental parts: A static part corresponding to the MU and IS properties; A dynamic part corresponding to the EX and LT properties. This dichotomy between static and dynamic parts is important for the detection.

5 Detection

We briefly describe algorithms to check the four properties of binary class relationships and thus bring continuity between design and implementation[3].

[3]The interested reader may refer to [9, 10] for detailed explanations.

The detection of the static part of the binary class relationships is simple to perform using Java introspection capabilities and using a byte-code analysis framework: IBM CFParse v1.21 [8].

Detection of MU The detection of the values of the MU property corresponds to the fields and their multiplicities. In particular, we determine the types of collections using Java programming idioms [13].

Detection of IS We iterate through the byte-codes of each class, looking for invocation sites, to assign the corresponding values to the IS property.

The detection of the dynamic part uses a trace-analysis technique [10] to model a program execution as a sequence of execution events. There are three kinds of events, represented as Prolog terms: Assignment events emitted every time a field of an instance of a class `A` is assigned with an instance of a class `B`; Finalize events emitted when the Java virtual machine garbage-collects an instance; A program-end event that is emitted when the program terminates.

Detection of LT We check the lifetime property of the composition relationship with the help of a Prolog predicate [9]. This predicate builds a list of terms abstracting sequences of events in the execution trace depending on the order in which assignation, finalization, and program-end happen.

Detection of EX Following the same principle, we define a predicate to check the exclusivity property of the composition relationship.

The terms in these lists represent values of the lifetime and exclusivity properties between instances of two classes. Then, we can infer the value of the properties of the two classes by conjunctions.

6 Future work

In this paper, we propose definitions and algorithms to bridge the discontinuity between modeling and programming languages for binary class relationships.

We included our algorithms in two software reverse engineering tools, Ptidej and Caffeine, which bring consistency during maintenance and offer an improvement over existing industrial and academic tools.

We currently develop and use our reverse engineering tools to identify micro-architectures similar to design-patterns in Java programs [2].

Future work includes:

- To develop our approach with more *flavors* of binary class relationships, such as the *use*, shared-aggregation, and container relationships.
- To verify that our definitions are *really* consensual and that our set of properties is minimal with respect to existing definitions.
- To apply our reverse engineering tools on real-life programs and to validate their results with the developers of the programs.

References

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

[2] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *proceedings of the 16th conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press, November 2001.

[3] G. Booch. *Object-Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 2nd edition, September 1993.

[4] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the unifed modeling language. In *proceedings of the 11th European Conference for Object-Oriented Programming*, pages 344–366. Springer-Verlag, June 1997.

[5] F. Civello. Roles for composite objects in object-oriented analysis and design. In *proceedings of the 8th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 376–393. ACM Press, September 1993.

[6] S. Ducasse, M. Blay-Fornarino, and A.-M. Pinna-Dery. A reflective model for first class dependencies. In *proceedings of 10th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 265–280. ACM Press, October 1995.

[7] H. Eichelberger and J. W. von Gudenberg. On the visualization of Java programs. In *proceedings of the 1st international seminar on Software Visualization*, pages 295–306. Springer-Verlag, May 2002.

[8] M. Greenwood. *CFParse Distribution*. IBM AlphaWorks, September 2000.

[9] Y.-G. Guéhéneuc, H. Albin-Amiot, R. Douence, and P. Cointe. Bridging the gap between modeling and programming languages. Technical Report 02/09/INFO, Computer Science Department, École des Mines de Nantes, July 2002.

[10] Y.-G. Guéhéneuc, R. Douence, and N. Jussien. No Java without Caffeine – A tool for dynamic analysis of Java programs. In *proceedings of the 17th conference on Automated Software Engineering*, pages 117–126. IEEE Computer Society Press, September 2002.

[11] T. Hartmann, R. Jungclaus, and G. Saake. Aggregation in a behavior oriented object model. In *proceedings of 6th European Conference for Object-Oriented Programming*, pages 57–77. Springer-Verlag, June–July 1992.

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

[13] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In *proceedings of the 21st International Conference on Software Engineering*, pages 194–202. ACM Press, May 1999.

[14] B. B. Kristensen. Complex associations: Abstractions in object-oriented modeling. In *proceedings of the 9th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 272–283. ACM Press, October 1994.

[15] J. Noble and J. Grundy. Explicit relationships in object-oriented development. In *proceedings of the 18th conference on the Technology of Object-Oriented Languages and Systems*, pages 211–226. Prentice-Hall, November 1995.