Revisiting UML Class Relationship Recovery for Online Education

Dionysis Athanasopoulos
School of Electronics, Electrical Engineering, and Computer Science
Queen's University of Belfast, Northern Ireland, UK
D.Athanasopoulos@qub.ac.uk

Abstract

UML recovery has been a long-standing challenge for the software-engineering community. The complete recovery of UML class relationships needs the employment of both static and dynamic code analyses. However, the dynamic-code analysis is not usually applicable at the design time of programs and especially for incomplete programs in online education. To overcome this restriction, we propose a formally defined set of mappings between UML relationships and object-oriented relationships that are based on static-code analysis exclusively. We evaluate the precision and the recall of our mappings on student projects against ground-truth UML diagrams and against diagrams recovered by existing UML class recovery tools.

1 Introduction

The motivation of our research come from a real story. It all started a few weeks ago in the labs of a computer-science school. Amelia[1], an undergraduate student, wanted in the context of a software-design module to take online feedback on her UML class diagrams[2] that visualize the design of her Object-Oriented (OO) programs. Class diagrams describe the *static structure* of OO programs by showing the program's classes, fields, methods, and class relationships.

Amelia generally feels confident to build up a UML diagram only if she can map it to the source-code elements that implement the diagram. In other words, she prefers first writing (a skeleton of) her OO programs and then mapping them to UML diagrams via using her favorite integrated development environment, IDE (e.g., Eclipse[3]). The program that she has started developing today contains classes that are related to each other in various ways. Amelia found it difficult to build the diagram on her own and especially, to differentiate the usage of the various kinds of arrows that UML provides. In particular, she was confused while she was mapping the implementation-level relationships of her program to UML class arrows.

Thus, Amelia needed an online tool that takes as input her OO programs and outputs a visual medium for her programs. Such a tool should be quite precise with respect to the usage of the UML arrows. Moreover, the tool should be able to work on incomplete programs that cannot necessarily be executed. In other words, the tool should be based on the static-code analysis of OO programs. Amelia thought such a tool is a necessary classroom assistant in the era of online education that has recently stressed.

Luckily for Amelia, her module owner, Bob, suggested to her to use a freeware (e.g., `ObjectAid`[4]) that can be integrated with her IDE and recover UML diagrams from incomplete Java programs. Amelia was happy to see that the tool can draw UML diagrams by just dragging and dropping Java classes, providing a visual medium for Java programs. However, when she used the tool for her programs, she was concerned about the arrows used by the tool used for some Java class relationships in the recovered diagram. To double check the diagram, Amelia discussed her concerns with the module owner. Bob drew his own diagram and verified Amelia's concerns about the precision of the tool, as analysed in a next section of the current paper.

Overall, existing UML recovery tools that use static-code analysis are not precise enough for online learning purposes. Moreover, the state-of-the-art research approaches that could be adopted for overcoming this limitation are not completely based on static-code analysis (e.g., [1, 2]) or they do not satisfy the lifetime and the share-ability object properties required for recovering the UML composition relationship (e.g., [3]).

We contribute an initial version of an automatic approach that takes as input an OO program and outputs the expected UML class relationships. To this end, we formally define the concepts of OO classifier, OO relationship, and UML relationship via using static-code syntactic analysis exclu-

DOI reference number: 10.18293/SEKE2021-170

[1]Please note that the persona names in our story are fake.
[2]https://www.uml.org
[3]https://www.eclipse.org
[4]https://www.objectaid.com/home

sively. We further propose a formally defined set of mappings between OO and UML relationships that satisfy the required lifetime and the share-ability properties[5]. We finally evaluate the precision and the recall of our mappings on existing student projects against ground-truth UML diagrams and against diagrams recovered by existing professional UML class recovery tools.

The rest of the paper is structured as follows. Section 2 presents the related research approaches. Section 3 defines the concept of OO relationship. Section 4 maps OO relationships to UML relationships. Section 5 evaluates the effectiveness of our approach. Finally, Section 6 summarizes our contribution and proposes future research directions.

2 Related Work

UML class diagrams represent OO classifiers (e.g., class, interface), fields, methods, and classifier relationships. The UML standard[6] defines the following kinds of relationships between classifiers: dependency, inheritance, realization, association, aggregation, and composition. The association can be a directed or a unidirectional relationship.

We organize the existing approaches of the round-trip engineering between UML diagrams and OO programs into three categories. The first-category approaches generate source code from UML diagrams based on UML to OO mappings [4]. Other approaches recover business processes from UML sequence diagrams by using a set of heuristics [5]. The second-category approaches define consistency links between UML diagrams and source code [6, 7, 8, 9].

The third category includes reverse-engineering approaches that recover (parts of) UML diagrams from OO source code. [10] recovers UML use-case diagrams by using trace-ability links between use-case elements and classifiers. [11] recovers UML behaviour diagrams from source code by identifying patterns in the source code. [1, 2, 12] recover UML relationships by identifying mappings between UML and OO relationships. [13] apply heuristics to static and semantic analysis of Java classes.

Our approach belongs to the third category and is related to [1, 2, 3]. [1, 2] recover UML relationships via checking the following set of properties for objects: multiplicity, exclusivity, and lifetime. However, static and dynamic code analyses are used to confirm the properties.

[14, 3] recover composition relationships via checking the non-accessibility property for objects. To this end, [14, 3] check whether a reference to an object is exported by its owner object to a third-party object. However, [15] states that the definition of composition based on the non-accessibility property is not consistent with the UML specification. [15] further states that the lifetime and the share-ability properties are the properties that should be used for recognizing composition relationships. [15] specifies an OCL formalization of the above properties. However, the complete verification of the above properties needs both static and dynamic code analyses.

[5]We have left as future work the possible consideration of semantic code analysis (e.g., lexical analysis).

[6]https://www.omg.org/spec/UML

Program 1 OO Skeleton of the Flight-Booking Program

```
class BOOKING (ABSTRACT)
    int id; ▷ Built-in field.
    String name;
    double price;
    function BOOKING(int id, String n, double p, double e)
        this.id := id;
        this.name := n;
        this.price := p;
        this.extraPrice := e;
class ECONOMY EXTENDS BOOKING
    int seat;
    function ECONOMY(int id, String n, double p,int s,double e)
        super( id, n, p, e );
        this.seat := s;
class BUSINESS EXTENDS BOOKING
    String menu;
    function BUSINESS(int id,Stringn,doublep,Stringm,double e)
        super( id, n, p, e );
        this.menu := m;
class PRINTING
    function PRINTBUSINESSPRICE(Business b) ▷ Reference.
        print( ... );
    function PRINTECONOMYPRICE(Economy e)
        print( ... );
class FLIGHT
    Printing c := new Printing(); ▷ Developer-defined field.
    List<Business> bList; ▷ Owned object(s).
    List<Economy> eList;
    function ADDB(int id, String n, double p,String m,double e)
        Business b := new Business( id, n, p, m, e );
        c.printBusinessPrice( b );
        if bList = null then bList := new ArrayList<Business>
        bList.add( b );
    function ADDE(int id, String n, double p, int s, double e)
        Economy e := new Economy( id, n, p, s, e );
        c.printEconomyPrice( e );
        if eList = null then eList := new ArrayList<Economy>
        eList.add( e );
class BOOKINGSYSTEM
    function MAIN
        Flight f := new Flight(); ▷ Local variable.
        f.addB( 20, "Tom", 100, "Chicken", 1000 );
        f.addE( 5, "Sam", 100, 5, 10 );
```

3 Object-Oriented Relationships

We illustrate our definitions via using a running example. We take an example that corresponds to a small part of an OO flight-booking system. The program calculates the total price of a booking and prints out the overall booking information. The Java-like pseudo-code of the classes of the above program is provided in Prog. 1.

Classifier fields and methods. A classifier mainly consists of classifier-level fields (e.g., built-in data-types, objects of other classifiers) and/or methods.

Owned object reference. An `A` classifier can be associated to an object of a `B` classifier even if `A` has not created the

B object. In this case, A is associated with a *reference* to the B object (line 21 of Prog. 1). If A does not create a B object but A has a reference to the object that is kept in the fields of A, then A has an *owned reference* to the object. To distinguish the case of an object reference owned by the classifier that created the object, we further use the term *owned object*. If A creates a B object stored in the fields of A, then A has an *owned object* (line 27 in Prog. 1). The definitions of the concepts of owned object and reference are provided in the remainder of this section.

Object finalization. By default, all the references to an object are freed when a program finishes its execution. A classifier method may explicitly finalize an object via using a reference to the object before the termination of the program. If an A classifier has a reference to a B object and this reference has been finalised by another classifier, then A has lost/cannot refer to the B object.

Definition 1 (OO Classifier) *A classifier, c, includes (i) its name n whose prefix is its package path (this combination can uniquely identify the classifier in a program); (ii) its kind k (concrete class, abstract class, interface, enum); (iii) a (possibly empty) set of generic classifiers g_i that c extends/implements; (iv) a (possibly empty) set of classifier-level developer-defined f_i fields (owned objects), along with their maximum l_i multiplicity; (v) a (possibly empty) set of the d_i classifiers whose object references are explicitly finalized by c; (vi) a (possibly empty) set of the r_i object references that are owned by c; (vii) a (possibly empty) set of the methods of c; (viii) a (possibly empty) set of the u_i classifiers (along with their maximum l_i multiplicity) whose objects are created by c. If the object is created by using a combination of generic and concrete classifiers, then the u_i set includes both the generic and the concrete classifiers.*

$$c = \Big(n,\ k,\ \{g_i\},\ \{(f_i,\ l_i)\},\ \{d_i\},\ \{r_i\},\ \{m_i\},\ \{(u_i, l_i)\}\Big)$$

Definition 2 (Method) *A method is characterized by (i) its name n; (ii) a (possibly empty) set of arg_i arguments that are developer-defined classifiers (along with their maximum l_i multiplicity); (iii) its (possibly absent) ret developer-defined return type (along with its maximum l multiplicity).*

$$m = \Big(n,\ \{(arg_i,\ l_i)\},\ (ret, l)\Big)$$

Definition 3 (Owned Object) *A c_1 classifier owns an object of a c_2 classifier if the c_2 object belongs to the developer-defined fields of c_1 and the c_2 object has been created by the c_1 classifier. To put it formally, a c_2 object is owned by c_1 if the following condition is evaluated as true.*

$$ownedObj(c_2,\ c_1) := c_2 \in c_1.\{f_i\}\ \wedge\ c_2 \in c_1.\{u_j\}$$

Definition 4 (Owned Reference) *A c_1 classifier just owns a reference to an object of a c_2 classifier if the c_2 object reference belongs to the developer-defined fields of c_1 but the c_2 object has not been created by the c_1 classifier.*

$$ownedRef(c_2, c_1) := c_2 \in c_1.\{f_i\}\ \wedge\ c_2 \notin c_1.\{u_j\}$$

Definition 5 (Associated Reference) *A c_1 classifier is associated with a reference to an object of a c_2 classifier if the c_2 reference does not belong to the developer-defined fields of c_1, the c_2 object has not been created by the c_1 classifier, and the c_2 object is included in the arguments of a method of the c_1 classifier.*

$$assocRef(c_2, c_1) := c_2 \notin c_1.\{f_i\}\ \wedge$$
$$c_2 \notin c_1.\{u_j\}\ \wedge\ c_2 \in c_1.m_k.\{arg_l\}$$

4 OO and UML Relationship Mapping

According to [15], composition should be defined based on the lifetime and the share-ability properties. The share-ability property requires that an object of a classifier, along with the references to the object, must be owned by at most one composite classifier. The lifetime property requires that the object of a composite classifier cannot be outlived by its owned objects. In other words, when the object of a composite classifier is finalized, its owned objects and the references to the owned objects are finalized too.

Definition 6 (Object Share-ability) *A c_1 classifier shares a c_2 object with a c_3 classifier if there is a reference owned by c_3 to the c_2 object that is created and owned by c_1.*

$$share(c_1, c_2, c_3) := ownedObj(c_2, c_1) \wedge ownedRef(c_2, c_3)$$

To compare the lifetime between an object of a composite classifier and its owned objects via using OO relationships, we define and prove the following theorem that is based on the object share-ability.

Theorem 1 (Composite object lifetime) *The lifetime of an object of a c_1 composite classifier is longer than or the same to the lifetime of an object of a c_2 classifier that is owned by c_1 if there is no other c_3 classifier that explicitly finalizes the c_2 object and c_3 does not own a reference to the c_2 object. If c_3 owns a reference to the c_2 object, then c_1 should explicitly finalize c_2.*

$$life(c_1, c_2) := ownedObj(c_2, c_1)\ \wedge\ c_2 \notin c_3.\{d_i\}$$
$$(\ \nexists c_3 : ownedRef(c_2, c_3)\ \vee\ c_2 \in c_1.\{d_i\}\ )$$

Proof 1 *We assume that a c_2 object is owned by a c_1 object and we examine all the possible cases with respect to the ownership of the c_2 object/references and the finalization time of the objects.*

(a) If c_3 explicitly finalizes c_2, then c_1 cannot use its owned c_2 and consequently, the lifetime comparison of c_1 and c_2 is meaningless (the second condition is false).

(b) If c_3 does not explicitly finalize c_2, c_3 owns a reference to c_2, and

(i) c_1 is finalized without finalizing c_2 (swallow finalization), then c_2 has longer lifetime than c_1 because there is a live reference to c_2 in the c_3 object (both third and fourth conditions are false)

(ii) c_1 and c_2 are finalized together (deep finalization), then c_1 and c_2 have the same lifetime and c_3 cannot use c_2 because c_2 has been finalized (the first, second, and fourth conditions are true).

(c) If c_3 does not explicitly finalize c_2, if there is no c_3 object that owns reference(s) to c_2 and

(i) c_1 is finalized without finalized c_2 (swallow finalization), then there is no left object that uses c_2 and we consider that the lifetime of c_1 and c_2 is the same (the first, second, and third conditions are true)

(ii) c_1 and c_2 are finalized together (deep finalization), then c_1 and c_2 have the same lifetime (all conditions are true).

Illustrative example. The `Flight` object owns a `Business` object in Prog. 1, but there is no reference to the same `Business` object owned by another object. According to Theorem 1, the lifetime of the `Flight` object is longer or the same to the lifetime of the `Business` object.

Definition 7 (Composition) *A c_1 classifier is composed by a c_2 classifier if there is no c_3 classifier that shares with the c_1 classifier the same c_2 object and the lifetime of the c_1 object is longer or the same to the lifetime of the c_2 object.*

$$comp(c_1, c_2) := \nexists\, c_3 : share(c_1, c_2, c_3) \;\wedge\; life(c_1, c_2)$$

Illustrative example. The `Flight` object in Prog. 1 owns a `Business` object, there is no reference to the same `Business` object that is owned by another object, and the `Flight` and the `Business` objects have the same lifetime. In this case, the `Flight` and the `Business` classes have a UML composition relationship.

Aggregation relates a composite classifier and its owned objects/references. To capture this relationship, we use the owned object and reference relationships (Def. 3 and Def. 4), without the composite and the owned objects/references satisfying the lifetime and the share-ability properties.

Definition 8 (Aggregation) *A c_1 classifier aggregates a c_2 classifier if c_1 owns c_2 object(s)/reference(s) but c_1 does not have a composition relationship with c_2.*

$$aggr(c_1, c_2) := (ownedObj(c_1, c_2) \;\vee\; ownedRef(c_1, c_2)) \wedge !\, comp(c_1, c_2)$$

According to the UML standard, association exists when a classifier is associated with references to object(s) of another classifier. In other words, the association can be defined by using Def. 4. But if the former classifier is composite that owns the object(s)/reference(s) of the latter classifier, then the classifiers may have a composition/aggregation relationship.

Definition 9 (Association) *A c_1 classifier is associated with a c_2 classifier if a c_1 object does not own a c_2 object/reference and the c_1 object is associated with a reference to a c_2 object:* $assoc(c_1, c_2) := \; assocRef(c_1, c_2)$

Please note a set of binary associations can be combined to form N-ary associations that may exist. However, the current work focuses on the recovery of binary associations, leaving as future work the recovery of N-ary associations.

Definition 10 (Realization) *A c_1 classifier realizes a c_2 classifier if c_2 is an interface and c_1 implements c_2.*

$$impl(c_1, c_2) := c_2 = c_1.g_i \;\wedge\; c_2.k = \text{“}interface''$$

Definition 11 (Inheritance) *A c_1 classifier inherits from a c_2 classifier if c_1 extends c_2 and c_2 is concrete/abstract class:* $inher(c_1, c_2) := \; c_2 = c_1.g_i \;\wedge\; c_2.k = \text{“}class''$

The dependency generally indicates that a source classifier uses an object of a target classifier. But if the former is a composite classifier that owns the object(s)/reference(s) of the latter, then the classifiers have a composition/aggregation relationship. Otherwise, if the former uses a reference to an object of the latter, then the classifiers have an association relationship.

Definition 12 (Dependency) *A c_1 classifier depends on a c_2 classifier if a c_1 object does not own a c_2 object/reference, the c_1 object is not associated with a c_2 object reference, and c_2 is the return type of a c_1 method or c_1 has created the c_2 object.*

$$dep(c_1, c_2) := !\, ownedObj(c_1, c_2) \wedge !\, ownedRef(c_1, c_2) \;\wedge\; !\, assocRef(c_1, c_2) \;\wedge\; (c_2 \in c_1.\{u_i\} \;\vee\; c_2 = c_1.m_j.ret)$$

Overall example. Applying our definitions on Prog. 1, we took as output the UML class diagram of Fig. 2. On the contrary, the diagram generated by the professional `ObjectAid` UML recovery tool is presented in Fig. 1. Comparing the two diagrams, we observe that the diagrams differ in five out of the seven UML relationship arrows.

Table 1. The dataset that we used for the effectiveness evaluation of the `UML Recoverer`.

ID	Num. of Classifiers					Num. of Fields		Num. of	Num. of Method Arguments		Num. of Method Return-Types	
	Total	Concrete	Abstract	Interface	Enum.	Total	Dev. defined	Methods	Total	Dev. defined	Total	Dev. defined
1	15	15	0	0	0	27	6	33	62	3	11	0
2	23	23	0	0	0	79	40	112	64	36	68	19
3	23	13	1	9	0	66	16	92	90	12	22	0
4	30	29	1	0	0	69	38	102	60	17	48	6
5	31	21	7	3	0	54	16	154	123	43	97	12
6	36	30	2	4	0	48	20	168	124	36	128	20
7	34	23	2	9	0	26	8	89	81	36	33	6
8	44	35	3	6	0	99	58	216	107	40	112	25

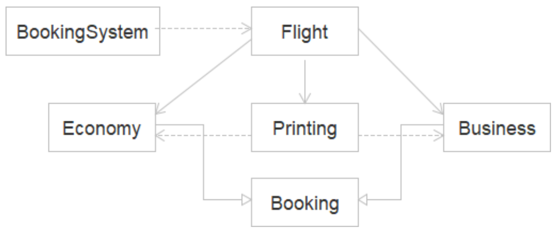

Figure 1. The diagram recovered by `ObjectAid` for Prog. 1.

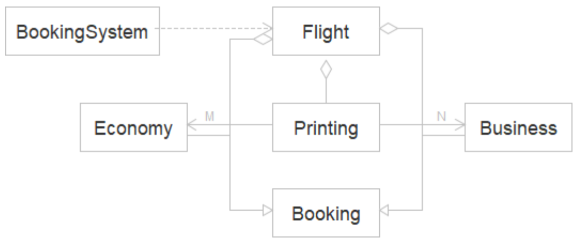

Figure 2. The diagram recovered based on our definitions for Prog. 1.

5 Experimental Evaluation

We implemented in Java the `UML Recoverer` research-prototype of our approach. We evaluate the effectiveness of the `UML Recoverer` on anonymized student projects against ground-truth UML diagrams and diagrams recovered by existing professional UML class recovery tools. The number of the classifiers of the projects ranges from 15 to 44 (Table 1) and the number of their UML relationships ranges from 20 to 170 relationships. Searching for existing (free to use) UML class recovery Eclipse plug-ins in the Eclipse Marketplace, we found that the most widely used tools currently are the `ObjectAid`[7] and the `UML Lab`[8]. To assess the effectiveness of the recovered binary UML relationships, we compare them against manually extracted relationships via using the precision and recall metrics [16].

The precision results are depicted in the first chart of Fig. 3. We observe the precision of the `UML Recoverer` steadily equals 1.0 in all projects (independently of the project cases). On the contrary, the precision of the other tools ranges from 0.37 to 0.86 and from 0.04 to 0.53, respectively. The recall results are depicted in the second chart of Fig. 3. We observe the recall of the `UML Recoverer` ranges from 0.79 to 1.0. In particular, the lower the number of the abstract classes and the interfaces a project includes, the higher the recall of the `UML Recoverer` is. This is due to the fact that the `UML Recoverer` does not capture association relationships to late-binding cases. The recall of the other tools ranges from 0.43 to 0.85 and from 0.05 to 0.53, respectively.

To explain why the precision and the recall values of the two tools is very low in some cases, we inspected the numbers of the UML relationships recovered by the tools and we made the following observations. The two tools do not recover the aggregation and the composition relationships at all. In particular, the `ObjectAid` considers as dependencies/associations the relationships that are aggregations or compositions. The `UML Lab` considers as associations the relationships that are dependencies, aggregations or compositions. The number of the associations recovered by the `UML Recoverer` is slightly lower than the ground-truth number. The reason is the late binding to objects. In particular, there are methods in the student projects that accept as input objects of abstract classes/interfaces and the `UML Recoverer` identifies the association to abstract classes/interfaces but not to concrete classes.

6 Conclusion and Future Work

We formally defined a set of mappings between UML relationships and OO relationships via using static-code analysis exclusively. A future direction of our work is the comparison of our algorithm against UML class recovery approaches that apply dynamic-code analysis. Another inter-

[7] `https://www.objectaid.com/class-diagram`
[8] `https://www.uml-lab.com/en/uml-lab/videos/reverse-egnineering`

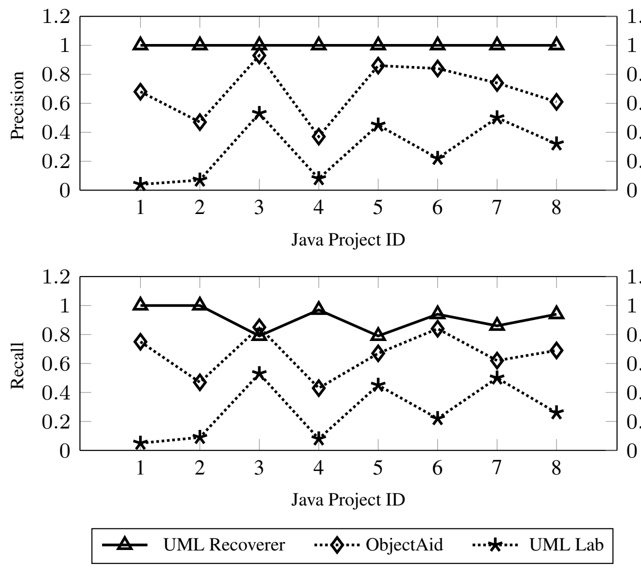

Figure 3. The precision and the recall results for the three recovery tools.

esting future direction would be the recovery of N-ary associations. Finally, the employment of semantic code analysis could further enrich the effectiveness of our approach.

References

[1] Y. Guéhéneuc and H. Albin-Amiot, "Recovering binary class relationships: Putting icing on the uml cake," in *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004, pp. 301–314.

[2] Y. Guéhéneuc, "A reverse engineering tool for precise class diagrams," in *Conference of the Centre for Advanced Studies on Collaborative research.* IBM, 2004, pp. 28–41.

[3] A. Milanova, "Composition inference for UML class diagrams," *Automated Software Engineering*, vol. 14, no. 2, pp. 179–213, 2007.

[4] W. Harrison, C. Barton, and M. Raghavachari, "Mapping UML designs to java," in *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2000, pp. 178–187.

[5] M. C. Leonardi, M. V. Mauco, L. Felice, G. Montejano, D. Riesco, and N. C. Debnath, "Recovering business process diagrams from UML diagrams," in *IEEE International Conference on Computer Systems and Applications*, 2010, pp. 1–6.

[6] H. M. Chavez, W. Shen, R. B. France, B. A. Mechling, and G. Li, "An approach to checking consistency between UML class model and its java implementation," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 322–344, 2016.

[7] D. Torre, Y. Labiche, M. Genero, and M. Elaasar, "A systematic identification of consistency rules for UML diagrams," *Journal of Systems and Software*, vol. 144, pp. 121–142, 2018.

[8] D. Torre, Y. Labiche, M. Genero, M. T. Baldassarre, and M. Elaasar, "UML diagram synthesis techniques: a systematic mapping study," in *ACM International Workshop on Modelling in Software Engineering, MiSE@ICSE*, 2018, pp. 33–40.

[9] D. Torre, Y. Labiche, M. Genero, M. Elaasar, and C. Menghi, "UML consistency rules: a case study with open-source UML models," in *ACM International Conference on Formal Methods in Software Engineering*, 2020, pp. 130–140.

[10] M. Grechanik, K. S. McKinley, and D. E. Perry, "Recovering and using use-case-diagram-to-source-code traceability links," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2007, pp. 95–104.

[11] J. Niere, "Recovering uml diagrams from java code using patterns," in *Workshop on Soft Computing Applied to Software Engineering*, 2001, pp. 1–9.

[12] M. J. Decker, K. Swartz, M. L. Collard, and J. I. Maletic, "A tool for efficiently reverse engineering accurate UML class diagrams," in *IEEE International Conference on Software Maintenance and Evolution*, 2016, pp. 607–609.

[13] A. M. Sutton and J. I. Maletic, "Mappings for accurately reverse engineering UML class models from C++," in *IEEE Working Conference on Reverse Engineering*, 2005, pp. 175–184.

[14] A. Milanova, "Precise identification of composition relationships for UML class diagrams," in *IEEE/ACM International Conference on Automated Software Engineering*, 2005, pp. 76–85.

[15] H. M. Chavez and W. Shen, "Formalization of UML composition in OCL," in *IEEE International Conference on Computer and Information Science*, 2012, pp. 675–680.

[16] R. A. Baeza-Yates and B. A. Ribeiro-Neto, *Modern Information Retrieval.* ACM Press/Addison-Wesley, 1999.