Softw Syst Model
DOI 10.1007/s10270-013-0390-0

REGULAR PAPER

A pattern-based approach for improving model quality

Mira Balaban · Azzam Maraee · Arnon Sturm · Pavel Jelnov

Received: 11 April 2013 / Revised: 1 September 2013 / Accepted: 1 November 2013
© Springer-Verlag Berlin Heidelberg 2013

Abstract UML class diagrams play a central role in modeling activities, and it is essential that class diagrams keep their high quality all along a product life cycle. Correctness problems in class diagrams are mainly caused by complex interactions among class-diagram constraints. Detection, identification, and repair of such problems require background training. In order to improve modelers' capabilities in these directions, we have constructed a catalog of anti-patterns of correctness and quality problems in class diagrams, where an anti-pattern analyzes a typical constraint interaction that causes a correctness or a quality problem and suggests possible repairs. This paper argues that exposure to correctness anti-patterns improves modeling capabilities. The paper introduces the catalog and its pattern language, and describes experiments that test the impact of awareness to modeling problems in class diagrams (via concrete examples and anti-patterns) on the analysis capabilities of modelers. The experiments show that increased awareness implies increased identification. The improvement is remarkably noticed when the awareness is stimulated by anti-patterns, rather than by concrete examples.

Keywords Anti-patterns · Pattern languages · Pattern awareness · Experiments · Modeling problems · Analysis capabilities · Software engineering education · Correctness · Quality

Communicated by Dr. Jordi Cabot.

M. Balaban · A. Maraee (✉)
Computer Science Department, Ben-Gurion University of the Negev, Beer-Sheva, Israel
e-mail: mari@cs.bgu.ac.il

M. Balaban
e-mail: mira@cs.bgu.ac.il

A. Maraee
Deutsche Telekom Laboratories, Ben-Gurion University of the Negev, Beer-Sheva, Israel

A. Sturm
Information Systems Engineering, Ben-Gurion University of the Negev, Beer-Sheva, Israel
e-mail: sturm@bgu.ac.il

P. Jelnov
Eitan Berglas School of Economics, Tel Aviv University, Tel Aviv, Israel
e-mail: pavlikjel@gmail.com

1 Introduction

Models are the backbone of the emerging model-driven engineering (MDE) approach, whose paradigm is the development of software via repeated model transformations. The quality of models used in such a process affects not only the final result, but also the development process itself. Erroneous or low-quality models can hide key abstractions while highlighting false inter-relationships that can lead the project evolution in a wrong direction.

We believe that good modeling can be achieved by modeling education and experience. Indeed, the quality of a model can be improved via successive transformations (refactoring), but state-of-the-art software tools do not enforce semantic correctness, e.g., identifying contradictory constraints and offering repair advices, and do not provide automatic model refactoring. In any case, such methods cannot replace good modeling qualities, i.e., no automation support can replace a good modeler. Therefore, this paper deals with problems that are highly relevant to MDE.

Modeling is a design activity that requires creativity and adaptation to ever changing contexts and new requirements [1]. Good modeling cannot follow a protocol of

technical procedures. Consequently, teaching modeling (like teaching other design activities) is hard and requires a variety of approaches and methods. One such method that seems to meaningfully improve design capabilities involves the study of design patterns, which are expert advices for solving typical problems that might occur in multiple contexts. This approach assigns an educational role to design patterns: Awareness to design patterns yields better solutions. Moreover, providing anti-patterns that improve modeling education is essential, regardless of future automation capabilities.

The origin of the design patterns approach lies in the field of architecture, where the architect Alexander [2] introduced a new approach that involved characterization of typical problems, for which he suggested architectural solutions, subject to context analysis. Since then, patterns have been used in a diversity of fields, including patterns in education [3, 4], software communication [5], human–computer interfaces [6], and analysis patterns in business modeling [7].

The design patterns trend in software construction gained increasing popularity since the appearance of the design patterns book of the "GoF" [8], and software design patterns are available today in almost every programming language. A dual approach in software design involves *anti-patterns* [9], which present popular bad solutions for software design problems, followed by desirable good solutions.

This paper focuses on correctness and quality problems in UML class diagrams, and the educational role of anti-patterns in improving their identification. We concentrate on class-diagram modeling, due to its widespread usage, central role in UML modeling, and the difficulty of producing high-quality models. Since both industry and academia usually neglect the semantic correctness of class diagram, the problems and corresponding patterns identified in this paper originate from our own analysis of class-diagram semantics.

Correctness and good quality are highly significant for models, since they serve as an abstract interface to applications, and are especially important in early stages of software design. Problems of correctness and quality frequently occur in large models, in particular when distributed among several developers [10]. They are caused by complex interactions among class-diagram constraints, and their identification and repair require deep understanding of modeling design problems.

We argue that the exposure to modeling problems and in particular to anti-patterns that analyze such problems, provide identification means and suggest solutions, will improve modeling skills [11]. We describe experiments that test the impact of awareness to modeling problems, on model analysis capabilities. The experiments check the model analysis skills of modelers, before and after teaching correctness problems in class diagram, via concrete examples and via anti-patterns. The results of the experiments assess our hypothesis, and remarkably so when anti-patterns are introduced.

The rest of the paper is organized as follows. Section 2 presents a variety of causes for incorrect and low-quality class-diagram modeling. In Sect. 3, we discuss various issues in pattern specification and the rationale for selecting anti-patterns for addressing correctness problems in modeling. Sect. 4 describes the anti-pattern catalog. It informally introduces the *pattern class diagram* (*PCD*) language used in our catalog and demonstrates how concrete modeling problems are abstracted by anti-patterns. Sect. 5 describes the experiments that evaluate the impact of awareness to correctness and quality problems on class-diagram analysis skills, Sect. 6 surveys related work, and Sect. 7 concludes the paper.

2 Incorrectness and low quality in class-diagram modeling

The class-diagram language is the backbone of UML [12]. It consists of *classes, associations, class descriptors—attributes, operations* and *constraints* that are imposed on them. Figure 1 presents a class diagram without attributes, that demonstrates most of the constraints in the language. In the rest of this paper, we use symbolic notations for discussing some constraints. An association with association ends (also termed *properties*) a, b is denoted $[a - b]$, and an association r between classes A, B with properties a, b, and multiplicity constraints $[m_a, n_a]$, $[m_b, n_b]$, respectively, is denoted $r(a : A[m_a, n_a],\ b : B[m_b, n_b])$. Class hierarchy between classes A, B, class A being the subclass, is denoted $A \prec B$. This notation is overloaded for property subsetting constraints, where $p_1 \prec p_2$ stands for p_1 subsets p_2. A *generalization set* (*GS*) constraint between a superclass C to subclasses $C_1, \ldots, C_n$ with constraint $Cstr$ is denoted $GS(C, C_1, \ldots, C_n; Cstr)$.

The meaning of a class diagram is given by its *instances*. A class-diagram instance has a *domain*, and a *denotation mapping*, that maps elements of the class diagram to elements in or over the domain. Classes are mapped to sets of objects in the domain, and associations are mapped to relationships between these sets. The denotation of classes and associations are called *extensions*. Attributes are mapped to mappings defined on class extensions. Constraints denote restrictions over the extensions of the class-diagram elements. An instance of a class diagram is *legal* if it satisfies all the constraints; it is empty if all class extensions are empty and is infinite if some class extension is not finite. A constraint c is *entailed* by a class diagram CD, denoted $CD \models c$ if it holds in every legal instance of CD. Formal definition of the language, including abstract syntax and set-based semantics, appears in [13, 14].

Figure 2 demonstrates a class diagram (a) and a non-empty legal instance of it (b). The class diagram includes multi-

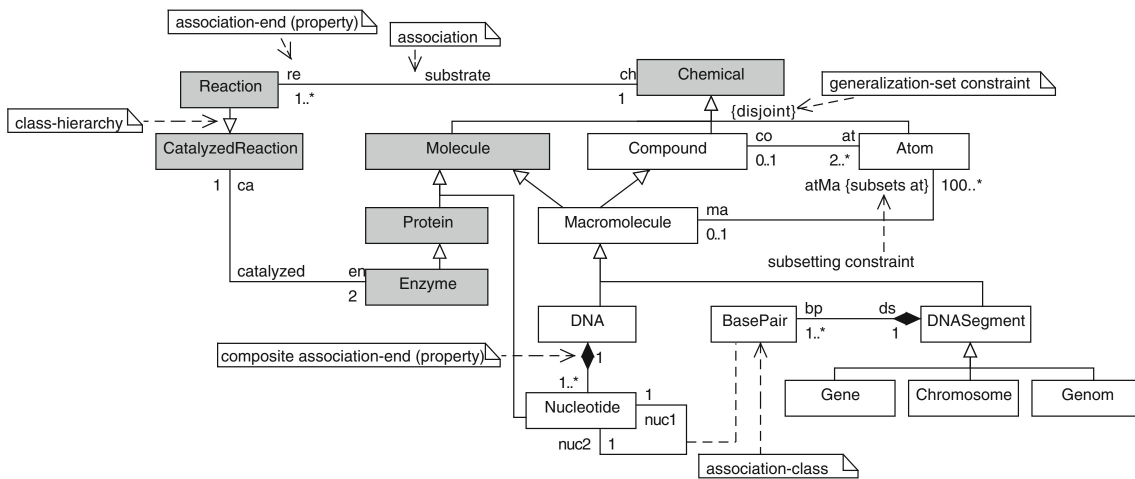


Fig. 1 A class diagram admitting correctness and quality problems

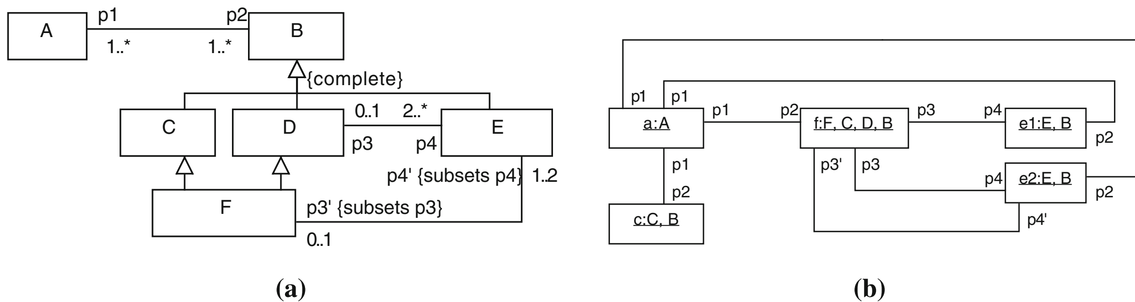


Fig. 2 A class diagram and one of its legal instances

plicity constraints, class hierarchy with multiple inheritance, a *complete GS* constraint, and a subsetting constraint. The instance is legal since it satisfies model constraints:

1. *Class-hierarchy constraints*: All objects belong also to all superclasses of their direct class. For example, the object f of class F also belongs to all superclasses of F; the objects $e1, e2$ of class E also belong to all superclasses of E.
2. *Multiplicity constraints*:
   - Property $p1$: The objects c, f, $e1$, $e2$, as B objects, are all linked to a single A object, a.
   - Property $p2$: a, which is the only A object in the diagram, is related to c, f, $e1$, $e2$.
   - Property $p3$: $e1$, $e2$ are $p3$ linked, each, to exactly one D object, f.
   - Property $p4$: f, as a D object, is linked to two E objects.
   - Property $p3'$: $e1$, $e2$ are not $p3'$ linked to more than a single F objects.
   - Property $p4'$: The only F object, f, is $p4'$ linked to a single E object.
3. *The complete constraint*: All B objects belong to subclasses of B, i.e., B is covered by the extensions of its subclasses.
4. *The subsetting constraints*: A subsetting constraint $p \prec q$ means that all p linked objects are also q linked. Indeed, $e2$ is $p3'$ linked to f and also $p3$ linked to f; f is $p4'$ linked to $e2$, and also $p4$ linked to it.

The two main correctness problems in UML class diagrams are *consistency* [15] and *finite satisfiability* [16]. Both are demonstrated in the class diagram in Fig. 1. Consistency deals with necessarily empty classes, i.e., with class diagrams that include contradictory constraints that cannot be satisfied by any instance. Such class diagrams are termed *inconsistent*, since they have classes with empty extensions in every legal instance. Figure 3a shows the subdiagram of Fig. 1, that causes a consistency problem. It includes the classes that participate in the *disjoint* generalization set *GS(Chemical, Molecule, Compound, Atom; Disjoint)*, and the two class-hierarchy constraints *MacroMolecule* $\prec$ *Molecule* and *MacroMolecule* $\prec$ *Compound*. The *disjoint* semantics enforces the extensions of *Molecule* and *Compound* to be disjoint sets in every legal instance. Therefore,

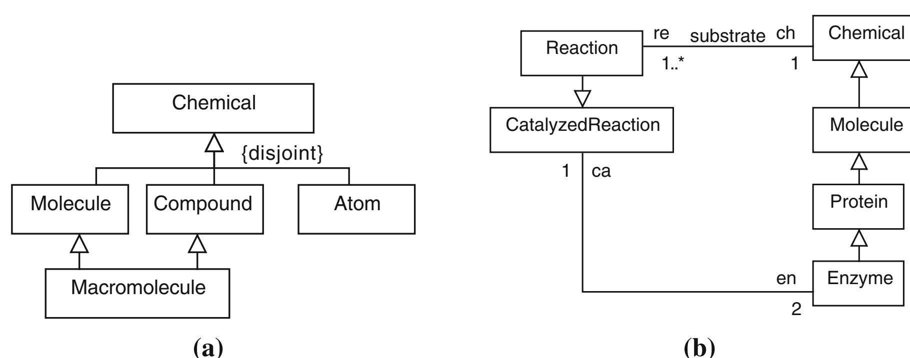


Fig. 3 Two class diagrams with consistency and finite satisfiability problems

the extension of *MacroMolecule*, which is a subset of both *Molecule* and *Compound* extensions, must be an empty set in every legal instance. Consequently, the class diagram is inconsistent.

Finite satisfiability deals with necessarily empty or infinite classes, i.e., with class diagrams involving multiplicity constraints that cannot be satisfied if the associated classes have finite extensions. That is, if the associated classes have non-empty extensions, then they must be infinite. Such class diagrams are termed not *finitely satisfiable* since they have classes with either empty or infinite extensions in every legal instance. Figure 3b shows the subdiagram of Fig. 1, that causes a finite satisfiability problem. It involves classes *Chemical, Reaction, CatalyzedReaction, Enzyme, Protein*, and *Molecule*, and the associations and class hierarchies among them. The multiplicity and class-hierarchy constraints imply that in every legal instance of this subdiagram, the numbers Ch, R, Cr, E, P, M of instances of classes *Chemical, Reaction, CatalyzedReaction, Enzyme, Protein, Molecule*, respectively, must satisfy the relationships $2R \leq 2Cr = E \leq P \leq M \leq Ch \leq R$, which can be satisfied only by empty or infinite extensions of these classes. Consequently, the class diagram is not finitely satisfiable.

Quality of class diagrams is a less quantitative criterion. Low quality can be caused by a variety of reasons, such as *redundancy* of information, *incomplete* information, too low or too high *coupling* of classes, *in-cohesive* classes, and excessive usage of *static* or *key attributes*. In this paper, we refer only to the first two reasons: redundant and incomplete specification. Redundancy refers to over or imprecise specification, where the redundant or missing information can be syntactic[1] or semantic. Incompleteness refers to essentially missing information. A deep semantic meaning of incompleteness involves model validation (e.g., as performed by the USE system [17]), which aims at finding unwanted model instances. Here, we refer to a more shallow meaning of incompleteness, i.e., models that do not display existing constraints.

There is a delicate dialectical relationship between the two quality problems, as providing too much information can yield redundancy, while omitting information might result in incompleteness. In Fig. 1, in addition to the correctness problems, there are also problems of incomplete information: The subsetting constraint $atMA \prec at$ on the *atMa* property entails an unmarked subsetting constraint $ma \prec co$ on the *ma* property; the *GS* constraint *GS(Chemical, Molecule, Compound, Atom; disjoint)* implies many unmarked *GS* constraints, such as *GS(Chemical, Gene, Atom; disjoint)*.

Correctness and quality problems in class diagrams are caused by undesirable interactions of constraints. Due to the diversity of constraints in class diagrams, analysis of problematic constraint interaction is complex and requires study of typical interactions. In the following, we present several cases of constraint interactions that cause correctness or quality problems. In the following sections, we show that such cases can be abstracted by *anti-patterns*, that group together problematic interactions admitting a similar structure, while separating interactions that have diverging causes. This analysis forms the basis for the study of *class-diagram anti-patterns*.

Inconsistency problems: Figure 4 presents three cases of inconsistency in class diagrams, that are caused by different kinds of constraint interactions:

1. Figure 4a presents an inconsistency problem caused by the interaction of the *disjoint* GS constraint and the multiple inheritance of class C_3: The disjoint constraint forces class C_3 to be empty in every legal instance.
2. Figure 4b is inconsistent due to the interaction of the *disjoint* and the *complete* constraints: It forces the extension of class C_3 to be empty, since an instance of C_4 must be an instance of C_1 or C_2, which are disjoint from C_3.
3. Figure 4c presents an inconsistency problem that is caused by the interaction of the composition and the subsetting constraints. An instance c_3 of C_3 must be related

[1] Some missing syntactic information is detected by tools.

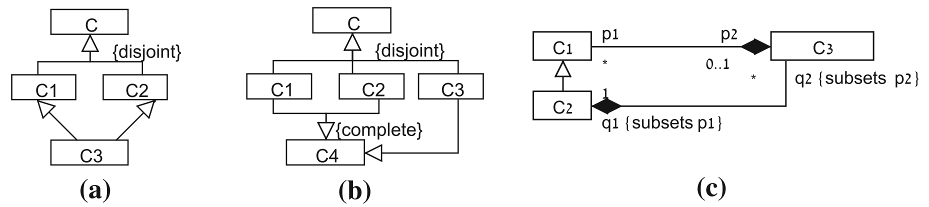


Fig. 4 Consistency problems that are caused by different kinds of constraint interactions. **a** Interaction of *disjoint* and multiple inheritance. **b** Interaction of *disjoint* and *complete*. **c** Interaction of composition and subsetting

to an instance c_2 of C_2 via the *composition* association-end q_1. Due to the subsetting constraint $q_1 \prec p_1, c_2$ must be related to c_3 via the *composition* property p_2, causing a composition cycle.

Finite satisfiability problems: Figure 5 presents six class diagrams that are not finitely satisfiable due to five kinds of problematic constraint interactions:

1. In Fig. 5a, each instance of C has a single successor and at least two predecessors. Therefore, if the number of C-s in a legal instance is c, and the number of predecessor–successor links is d, then d must satisfy $d = c \cdot 1$ and also $d \geq c \cdot 2$, implying the inequality $c \geq c \cdot 2$, that can be satisfied only by an empty or infinite extension of class C. The problematic constraint interaction in this case involves the multiplicity constraints on the *predecessor–successor* association.

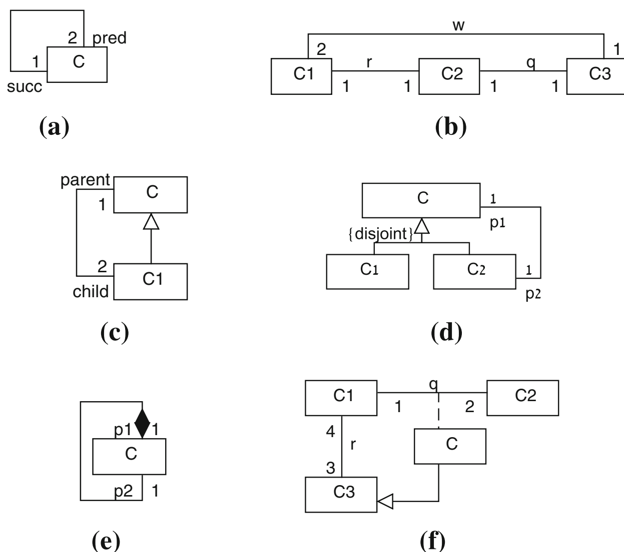


Fig. 5 Finite satisfiability problems that are caused by different kinds of constraint interactions: (**a**, **b**) Interaction of multiplicity constraints in an association cycle. (**c**) Interaction of multiplicity constraints in a cycle of associations and class-hierarchy constraints. (**d**) Interaction of multiplicity and GS constraints in a cycle of associations and class-hierarchy constraints. (**e**) Interaction of multiplicity and composition constraints in an association cycle. (**f**) Interaction of multiplicity constraints on an association cycle via an association-class

2. In Fig. 5b, the problematic constraint interaction involves the multiplicity constraints in the cycle of associations w, q, r, similarly to the first case.
3. In Fig. 5c, the interaction between the class hierarchy $C_1 \prec C$ and the multiplicity constraints on the $[parent-child]$ association causes a finite satisfiability problem.
4. In Fig. 5d, the problematic interaction involves the *disjoint* constraint and the multiplicity constraints on the $[p_1 - p_2]$ association.
5. In Fig. 5e, the problematic interaction involves the multiplicity constraints on $[p_1 - p_2]$ and the composition constraint on the p_1 property. In this class diagram, if the extension of C is finite and non-empty, the extension of the $[p_1 - p_2]$ association must be cyclic (i.e., includes a cycle of composition linked objects, which contradicts the semantic constraints of composition). Therefore, C cannot be finite, i.e., it can have only empty or infinite extensions.
6. Finally, in Fig. 5f, the interaction between the multiplicity constraints on the associations q, the class hierarchy $C \prec C_3$ and multiplicity constraints on the associations r causes a finite satisfiability problem.

Redundancy and incompleteness problems: Figure 6 presents redundant and incomplete class diagrams due to different kinds of constraint interactions:

1. In Fig. 6a redundancy is caused by the multiplicity constraints on properties p_1, p_2, p_3, p_4. For a legal finite and non-empty instance, if the number of C_1-s is c_1 and the number of C_2-s is c_2, then the multiplicity constraints on properties $p_2,\ p_3$ imply $c_1 = c_2$. But the multiplicity constraint *1..** on property p_1 enables $c_1 \geq c_2$. Therefore, the $*$ maximum multiplicity constraint is not sufficiently tight, i.e., redundant.
2. In Fig. 6b, the association–class hierarchy (AHC) $C_1 \prec C$ entails the visualized association specialization $c_1 \prec c$ and the subsetting constraints $b_1 \prec b,\ a_1 \prec a$ [14]. We think that in this case, all entailed constraints should not be visualized since they involve the elements that directly participate in the AHC, and are usually naturally understood by modelers. Therefore, the explicit visualization of the association-hierarchy constraint is redundant.

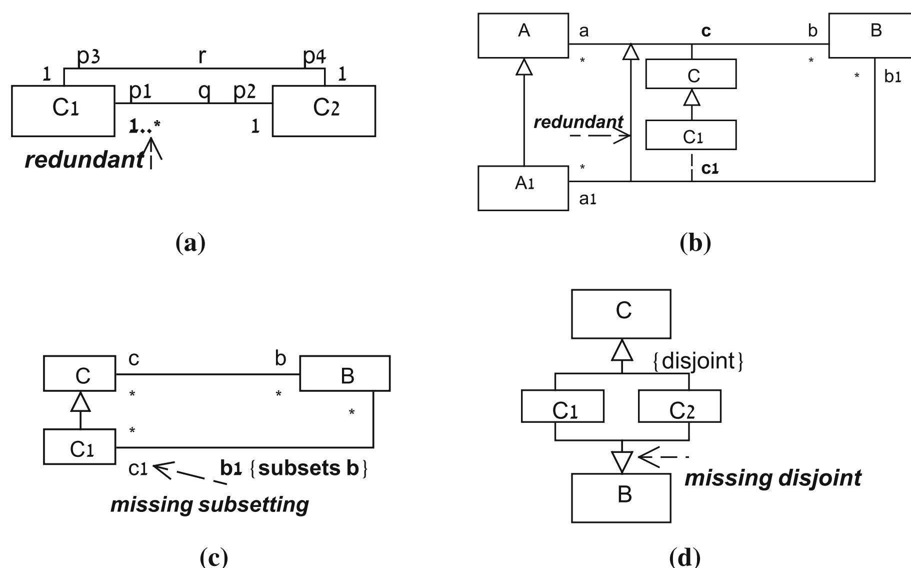


Fig. 6 Class diagrams admitting redundancy and incomplete information problems

3. In Fig. 6c, the subsetting constraint $b_1 \prec b$ entails the subsetting constraint $c_1 \prec c$ on the c_1 property. We think that this is a case of incomplete information since the missing symmetric constraint is not easily inferred by modelers.
4. In Fig. 6d, the GS constraint $GS(C, C_1, C_2; disjoint)$ entails a non visualized GS constraint $GS(B, C_1, C_2; disjoint)$. The missing information is meaningful as it involves different constructs of the class diagram and cannot be easily inferred by modelers (think of large class diagrams), and it affects automatic verification tools[2].

Having demonstrated the diversity of causes for incorrectness and low-quality problems (by concrete examples), we argue that in real class diagrams, understanding the various interactions among constraints, and their impact on correctness or quality, is not straightforward and requires training. In order to support awareness for modeling problems, we have developed a catalog [20] of anti-patterns for correctness and quality problems. The patterns in the catalog are sorted by kinds of modeling problems. Each anti-pattern characterizes a design problem that can be realized in multiple (usually infinity of) cases, suggests an identification structure and solutions.

3 Anti-patterns for correctness and quality problems

Patterns provide solutions to characteristic problems that occur in multiple concrete cases, within various contexts. The formulation of patterns requires abstraction over multiple concrete cases. For patterns whose aim is principally educational, their formulation raises several controversial issues concerning the nature of patterns and their specification. In this section, we shortly discuss the issues that we have dealt with, while developing the anti-pattern catalog.

Anti-patterns as an educational instrument: Anti-patterns present corrections to wrongly solved problems. They are based on the assumption that learning from commonly occurring mistakes has an educational value. The issue of using positive or negative examples in software and modeling education is controversial (see Sect. 6). Since our catalog deals with correctness and quality problems, its patterns present (negative) examples of bad interactions of constraints.

Using concrete examples or abstractions: Concrete examples are certainly needed for demonstrating multiple occurrences of a pattern problem. Sometimes patterns are specified simply by presenting multiple concrete examples [8, 21, 22] that explain more general verbal specifications. Yet, since a pattern handles a typical case that occurs in multiple situations, concrete examples might fall short for characterizing the common problem that is realized in these examples. There might be essential conceptual structures hidden within the concrete examples, which can be highlighted only by their abstraction. Indeed, the examples in Sect. 2 show that a single correctness problem can be caused by different kinds of constraint interactions, that might be hard to detect in concrete examples.

Using abstract structures instead of concrete examples enables accurate specification of problems and solutions, since multiple concrete examples can be replaced by a single abstraction. The benefit of abstract structures is twofold: They can identify similar instances by ignoring marginal differences and single out different instances by emphasizing

[2] The missing information causes failure of the ***FiniteSat*** algorithm [13, 18]. Indeed, this algorithm can be strengthened by pre-processing of propagation of *disjoint, incomplete* constraints [13, 19].

different relevant abstractions. The anti-patterns in our catalog [20] use abstract structures for describing class diagrams (as suggested in [23]). The abstract structures are defined as *PCDs*. Every anti-pattern in our catalog provides a thorough analysis of the relevant constraint interaction problem.

Modeling level of pattern specification: A class-diagram pattern describes multiple class diagrams. Therefore, its specification must involve meta-model constructs. One approach, taken by [24], and described in some detail in Sect. 6, defines modeling patterns as a domain-specific modeling language (DSML), that resides on the meta-model level. We adopt a different approach, that extends the class-diagram language with abstraction constructs like *variable* and *collection* (e.g., an *association sequence*, which is an ordered collection of associations)[3]. These extensions are captured in the *PCD* language, which is formally defined in [25] (called *non-ground class diagram*). A pattern class diagram that includes abstract elements can be instantiated by multiple (concrete) class diagrams.

Cognitive effectiveness of modeling patterns: Modeling patterns have an educational value. Therefore, patterns should be described in an intuitively appealing manner. We use the criterion of *cognitive effectiveness* of [26], which refers to the clarity of translations between cognitive and visual concepts. Moody [27] suggests some principles for achieving cognitive effectiveness:

1. *Semiotic Clarity*—a one-to-one correspondence between semantic constructs and representation symbols.
2. *Semantic Transparency*—easy association between symbols and their corresponding concepts.
3. *Perceptual discriminability*—clear distinction between symbols that represent different constructs.

For modeling patterns, it means that pattern structures should have visualization (concrete syntax) that directly reflects the intended model. This criterion dictates our decision to define the *PCD* language as an extension of the class-diagram language, rather than as another modeling layer. The implications are as follows:

1. The visualization of a pattern class diagram naturally extends the visualization of a class diagram.
2. It is easy to distinguish the abstract constructs from the concrete in a pattern class diagram.
3. The instantiation of a pattern class diagram into (concrete) class diagrams is an intuitive relation that directly reflects its meaning.

[3] Our approach is inspired from logic in the sense of having both constant and variable symbols that can be quantified.

4 The anti-pattern catalog and its usage

The catalog [20] includes patterns for solving problems of *correctness* or of *quality* of class diagrams. The correctness problems refer to the two formal correctness problems of consistency and finite satisfiability. Quality problems refer to formally correct design problems that do not meet criteria of desirable design. The quality problems are further classified into *incomplete design*, *redundancy problems*, and *comprehension problems*. Within categories, patterns are classified by kinds of constraint interactions that cause different problems. This problem-based classification is contrasted with the approach of [28], which is syntax-semantics-pragmatics based.

Based on the above classification, the catalog currently includes a total of 43 patterns: 15 patterns for finite satisfiability problems, 11 patterns for *consistency problems* and 17 patterns for *quality* problems. Each anti-pattern provides an identification structure for its problem, associated with concrete examples, proof of the problem, and suggested repair options. Note that the catalog is continuously under development, and new patterns are being added.

In order to describe patterns, we need a means to specify their components. For class-diagram modeling, the need is for a language for describing multiple class diagrams that realize the pattern's problem. Section 4.1 below shortly introduces the *PCD* language, and Sect. 4.2 presents the main parts of the patterns that were used in the identification experiments described in Sect. 5.

4.1 The pattern class diagram (PCD) language

The abstractions needed for pattern specification include *variables* ranging over class-diagram elements and *compound structures* of class-diagram elements. Variables are needed in order to enable instantiation by concrete elements. Compound structures are needed for capturing structures like association sequences, hierarchy sequences, interleaved association-hierarchy sequences, and class collections.

We extend the UML class-diagram meta-model with new classes that capture the new abstractions and provide their concrete syntax as new visual notations in class diagrams. Figure 7 presents the main part of the meta-model extension for the *PCD* specification. Elements that belong to the UML class-diagram meta-model appear as highlighted rectangles.

The enhancement covers insertion of variables and insertion of compound structures. In order to account for variables, the relevant classes are duplicated. For example, the `Class` meta-class turns into an abstract class with three subclasses `VariableClass, ConcreteClass` and `NonOrderedClassCollection`. Compound structures are specified via the `Collection` meta-class and its subclasses. Figure 8 presents the solution suggested in the

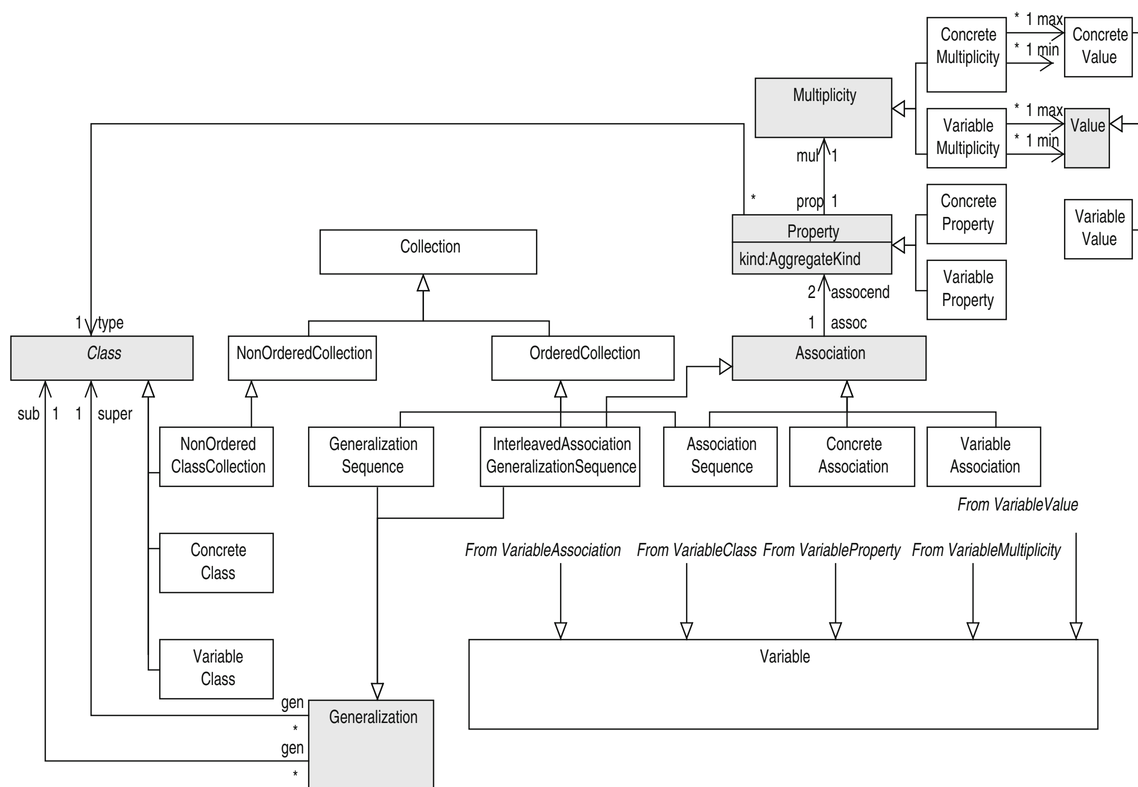


Fig. 7 The Meta-model of the *PCD* language

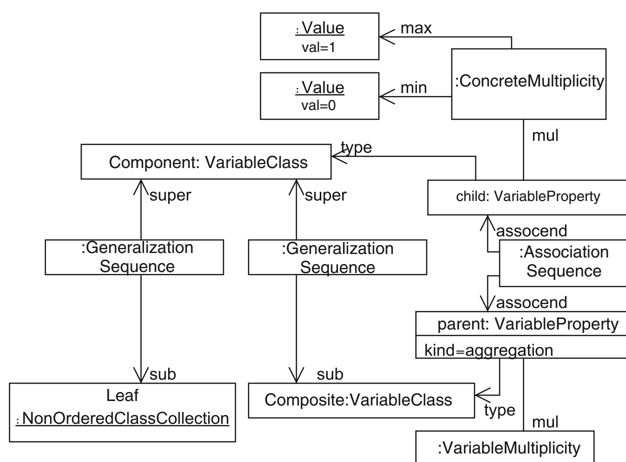


Fig. 8 The identifying structure of *Composite* pattern, as an explicit instance of the *PCD* meta-model

GoF *Composite* design pattern [8], as an explicit instance of this meta-model.

Specification of patterns as explicit instances of the meta-model of the language is precise, and essential for automation. Yet, it is useless for education, since the explicit instance representation is cognitively ineffective. It is hard telling the correspondence between the explicit instance in Fig. 8 and the intended structure in Fig. 9a. The conventional approach in the modeling community is to adopt a visual *concrete syntax* that represents meta-model instances in a cognitively effective way.

The concrete syntax for pattern class diagrams extends the standard visualization of class diagrams, with visualization for variables and for collections.

1. **Variable visualization:** Variable elements are visualized in the same way as their non-variable counterparts, but their label is prefixed by the "?" symbol. For example, variable classes are visualized as classes in a class diagram, but have a "?" prefix in their names; variable multiplicities are visualized as regular multiplicities but with a "?" prefix label.
2. **Compound element visualization:** Compound elements have distinct visualizations that remind their origins. Sequences of class-diagram relation elements (associations, hierarchy, aggregation, composition) are visualized by lines or arrows that remind their non-sequence counterparts, with distinct labels that specify the sequence kind. The label "*" is used for denoting sequences of any length (including the empty sequence), and the label "+" is used for non-empty sequences. For example, a class hierarchy (generalization) possibly empty sequence is represented by a generalization line, labeled by "*". The use of ***,+** labeled lines and arrows is motivated by the traditional meaning of these operators as denoting ≥ 0 and >0 repe-

Fig. 9 The composite design pattern. **a** The solution to the *Composite* design pattern of the GoF (without specification of operations). **b** An instance of the solution of the *Composite* design pattern in the graphics domain (following the GoF)

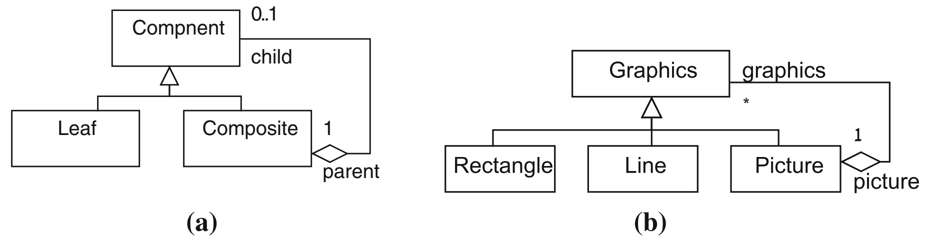


Table 1 Visualization of compound elements in *PCD*

Graphic notation	Meaning
+ ; ≥n, ≤ n, =n	A Non-Ordered-Class-Collection with size ≥ 0 A Non-Ordered-Class-Collection with size > 0 A constrained Non-Ordered-Class-Collection
* +	An Association-sequence with size ≥ 0 (or ≥ 1)
* *	An Aggregation-sequence (Composition-sequence) with size ≥ 0
* *	An Interleaved-Association-Aggregation-sequence with size ≥ 0
* +	A Generalization-sequence with size ≥ 0
*	An Interleaved-Association-Generalization with size ≥ 0
*	An Interleaved-Composition-Generalization-sequence with size ≥ 0
*	An Interleaved-Association-Aggregation-Generalization-sequence with size ≥ 0

Table 2 Examples of *PCD* compound elements and their possible class-diagram denotations

PCD compound element	A denoted class diagram
?C1 + ?C2	C1 p1 * p2 * C p3 * p4 * C2
?C1 * ?C2	C
?C1 + ?C2	C1 C p1 0..1 p2 * C2
?C + Classes	C p1 * q1 0..1 C1 p2 * q2 0..1 C2

titions, respectively. Table 1 shows the concrete syntax of some compound elements, and Table 2 presents examples of compound elements of *PCD*, and possible class diagrams that are denoted. Figure 10 describes the concrete syntax for the solution of the *Composite* design pattern (the precise meta-model instance appears in Fig. 8).

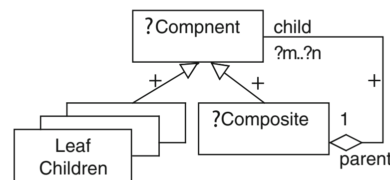


Fig. 10 Visualization in *PCD* of the solution of the *Composite* design pattern

The visualization of pattern class diagrams is cognitively effective due to the following features:

1. The concrete syntax follows the conventional rules of class-diagram visualization. That is, the rules for visualization of the new elements of *PCD* are either identical or very similar to their class-diagram counterparts.
2. The new *PCD* elements are clearly singled out from their class-diagram counterparts.
3. Instantiation (replacement) of compound elements in a pattern class diagram follows intuitive rules of zooming into an abstraction.

A pattern class diagram denotes a set of class diagrams, termed its *instances*. This semantics is given by a mapping *pcd*→*cd*, that substitutes concrete elements for variable elements of the same type, and elaborates compound abstractions. For example, a variable class is substituted by a class, an association sequence is replaced by a sequence of associations, and a class collection is replaced by a non-empty set of classes. The precise definition is given in [25]. For example, the class diagram in Fig. 9b is an instance of the pattern class diagram in Fig. 10. Figure 11 details a possible stepwise replacement procedure that computes a *pcd*→*cd* mapping. Figure 11b shows the result after replacing a variable

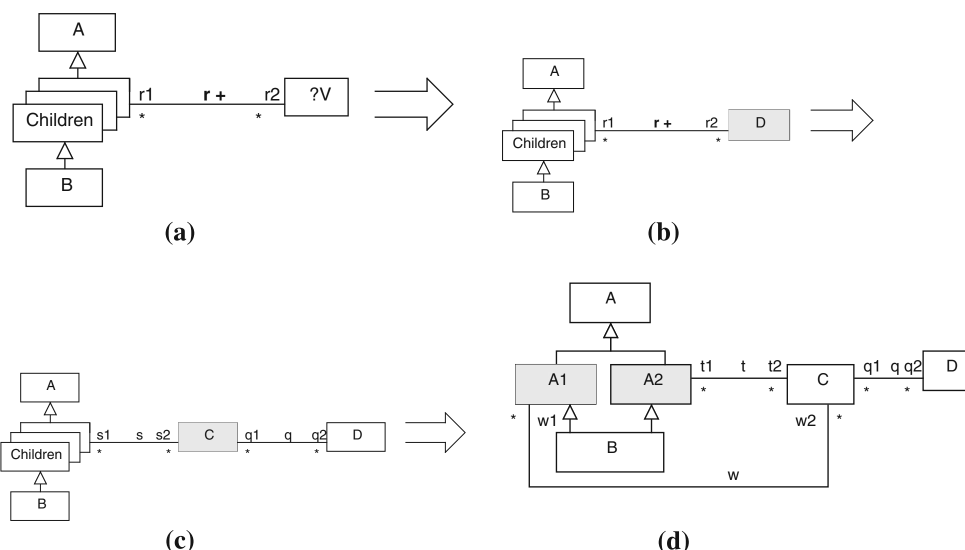


Fig. 11 Instantiation of a pattern class diagram into a class diagram. **a** A pattern class diagram. **b** After variable replacement. **c** After *associationsequence* replacement. **d** After class collection replacement

class; Fig. 11c shows the result after replacing an association sequence; Fig. 11d shows the result following the replacement of a class collection.

Evaluation of the PCD language: The *PCD* language enables abstraction over class diagrams. Its main characteristic is the built-in support for the *variable* and the *compound* concepts, which are smoothly added to the Class-diagram language. This characterization provides for the main advantages of the language:

Abstraction-based: The language accounts for all patterns that require the notions of *Variable* or of *Aggregation*. We say that it is *abstraction-concept-based* rather than *pattern-based*, since it is not affected by the introduction of new patterns.
Formal definition: Patterns are plain instances of the language meta-model, using the regular class-diagram instantiation mapping.
Cognitively effective concrete syntax: The visualization is an intuitive extension of the class-diagram visualization since it uses standard class-diagram visualizations, and added annotations like $*$, $+$ have their common meaning. There are no pattern-specific rules for visualization.
Class-Diagram intuitive semantics: The $pcd \rightarrow cd$ mapping uses the intuitive meaning of the abstraction elements. The mapping accounts for instantiation of all patterns.

4.2 Pattern examples

A pattern specification consists of nine entries: (1) Name; (2) Pattern problem (a textual description of the problem handled by the pattern); (3) Concrete examples; (4) Pattern identification structure in the form of a pattern class diagram; (5) Listing of the involved meta-model elements; (6) Pattern verification constraint; (7) Repair advice (refactoring); (8) Listing of related patterns; (9) Pattern justification—a correctness proof for the claims of the pattern identification, verification, and advice.

Example 1 [Pure Multiplicity Cycle (PMC) Pattern]

1. **Pattern name**: *Pure Multiplicity Cycle* (*PMC*).
2. **Problem**: A cycle of associations with multiplicity constraints might introduce a finite satisfiability problem.
3. **Concrete example**: See Fig. 12a.
4. **Pattern identification structure**: See Fig. 12b.
5. **Involved meta-model elements**: The meta-classes *Association* and *Class*.
6. **Pattern verification**: The pattern identification structure characterizes a necessary but not sufficient condition for existence of a finite satisfiability problem caused by the multiplicity constraints in an association cycle. The verification condition below provides the sufficient condition. Its specification refers to a schematic description of an instance class diagram of the pattern identification structure, as in Fig. 13.
   The cycle of associations causes a finite satisfiability problem if and only if one of the following conditions holds:

   - If for all $1 \leq i \leq k$, $n_i \neq *$: $\prod_{i=1}^{k} m'_i < \prod_{i=1}^{k} n_i$.
   - If for all $1 \leq i \leq k$, $n'_i \neq *$: $\prod_{i=1}^{k} m_i < \prod_{i=1}^{k} n'_i$.

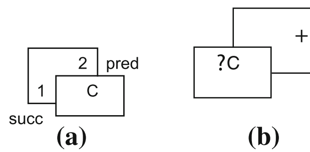


Fig. 12 Concrete example and identification structure of the *PMC* pattern

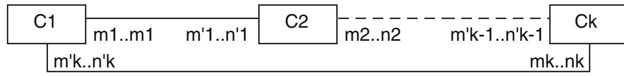


Fig. 13 An instance of the pattern identification structure

7. **Repair advice**: Consider relaxation of the multiplicity constraints: Decrease a minimal multiplicity value or increase a maximal multiplicity value.
8. **Related patterns**: The *Multiplicity-Hierarchy-Cycle (HMC)* pattern.
9. **Pattern justification**: A proof that an occurrence of the *pattern identification* structure causes a finite satisfiability problem if and only if it satisfies the *pattern verification* condition.

Below, we shortly describe the problems, identification structures, and some repair advices of several anti-patterns that were used in the experiments described in Sect. 5.

1. ***The Multiplicity-Hierarchy-Cycle (HMC) pattern***: This anti-pattern characterizes finite satisfiability problems due to interaction of multiplicity constraints on a cycle of associations and class-hierarchy constraints. Figure 14 presents its identification structure.
2. ***The Diamond pattern***: This anti-pattern characterizes consistency problems due to interaction between multiple class-hierarchy and disjoint constraints. Concrete examples appear in Figs. 4a and 15b. The identification structure specification is presented in Figs. 15a, b presents an instance class diagram. Classes *Academic*, *GraduateStudent* and *FacultyMember* instantiate variable classes $?C$, $?C_1$, $?C_2$ receptively, class *VistorLecturer* instantiates the *Children Collection-class*, and the class hierarchies *ExternalPhDLecturer* $\prec$ *PhDStudent* $\prec$ *GraduateStudent* and *ExternalPhDLecturer* $\prec$ *Faculty Member* instantiate the generalization sequences from $?D$ to $?C_1$ and to $?C_2$, respectively. A possible repair advice in this pattern is to remove the disjoint constraint.
3. ***The Disjoint Paths (DisPaths) pattern***: This anti-pattern characterizes finite satisfiability problems due to interaction of a disjoint GS constraint with multiplicity constraints on the association sequences from the GS-subclasses to the GS-superclass. Concrete examples appear in Figs. 5d and 16b. The pattern identification structure appears in Fig. 16a.
4. ***The Disjoint Complete (DisCom) pattern***: This anti-pattern characterizes inconsistency problems caused by the interaction of intersecting disjoint and complete GSs, with class-hierarchy sequences from a subclass of the disjoint GS to the superclass of the complete GS. Figure 17a presents the identification structure and Fig. 17b presents an instance.

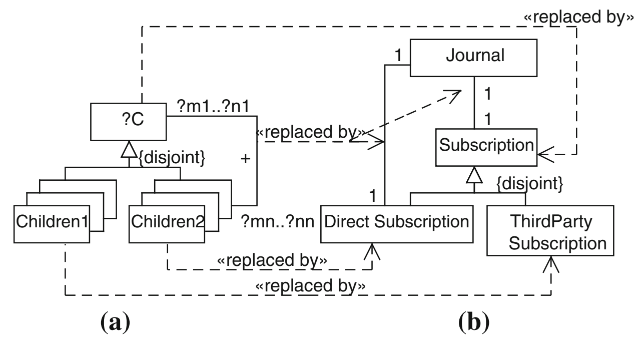


Fig. 16 The DisPath pattern: The identification structure (**a**) and an instance class diagram (**b**)

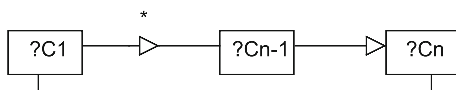
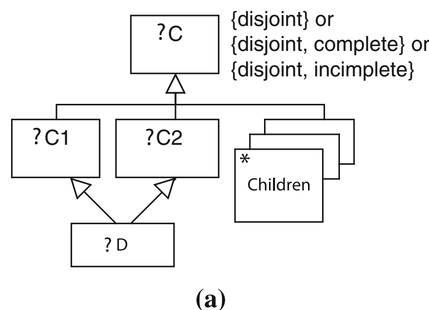
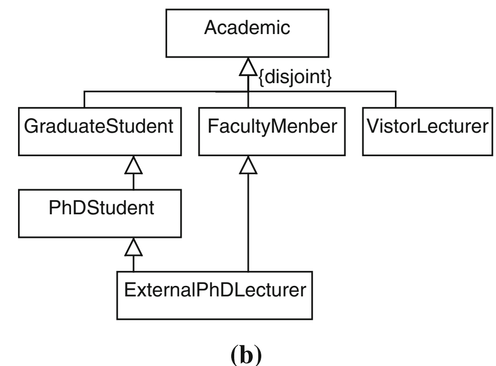


Fig. 14 The identification structure of the HMC anti-pattern

Fig. 15 The Diamond pattern: The identification structure and an instance class diagram. **a** The diamond identification structure. **b** An instance of the diamond pattern





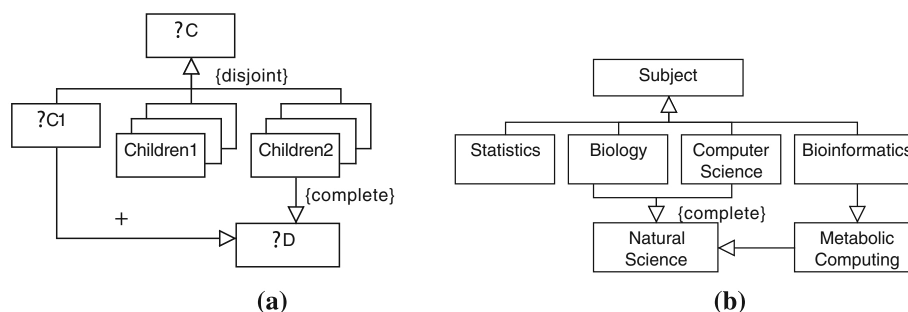


Fig. 17 The DisCom pattern identification structure. **a** Identification structure. **b** An instance class diagram

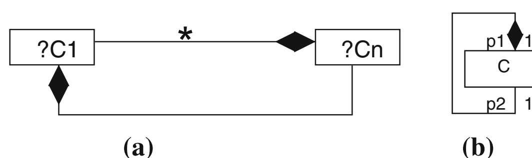


Fig. 18 The identification structure and an instance of the PCC pattern

5. ***Pure Composition Cycle (PCC) pattern***: This anti-pattern characterizes finite satisfiability problems due to interaction of composition constraints, multiplicity constraints, and a cycle of associations. Figure 18a presents the identification structure and Fig. 18b presents an instance.
6. ***Association–Class-Hierarchy (AHC) Pattern***: This anti-pattern characterizes finite satisfiability problems due to interaction between an association-class, the multiplicity constraints on the associated association, and multiplicity constraints on a cycle of associations and class-hierarchy constraints between the association-class and one of the classes of the associated association. Figure 19b presents an example, and Fig. 19a presents the identification structure.
7. ***XOR Subsetting (XORSub) pattern***: This anti-pattern characterizes consistency problems due to interaction of a *XOR* and a subsetting constraints. Figure 20a presents the identification structure and Fig. 20b presents an instance.

5 Evaluating the impact of anti-pattern awareness on class-diagram analysis skills

In order to examine the extent to which awareness to modeling problems helps in identifying erroneous models, we conducted three independent experiments that check the effectiveness of introducing modeling problems on their identification. The first experiment examines the effect of introducing modeling problems via examples, the second experiment tests the effect of introducing such problems via anti-patterns, and the third one verifies that using a catalog of anti-patterns leads to accurate identification of modeling problems. In the following, we elaborate on the goals and hypotheses, the participants, the procedure, and the results. Then, we discuss the results of the experiments, followed by the threats to validity.

5.1 Goals, hypotheses, and variables

In order to examine the questions aforementioned, we set the goals and hypotheses for three experiments as shown in Table 3.

For the first and the second experiments, the independent variables are the level of awareness of modeling problems, i.e., general comments (for the control groups) and concrete examples and anti-patterns (for the other group). The dependent variable is the number of identified modeling problems.

5.2 Participants

The participants were third-year software engineering students with background in computer science and background in information systems engineering. The first and the second experiments were conducted in class as part of a course on Object-Oriented Analysis and Design. Student participation was voluntary. Nevertheless, as an incentive to actively and properly participate in the experiments, the students were told that their participation would add bonus points to the final grades, in proportion to their achievement. The third experiment was part of an exam at the end of the course.

In the first experiment, the students were divided randomly into groups of 27 (general introduction) and 18 (examples). To verify that the division was not a factor in the experiment success, we tested the similarity of the groups based on their mean grade point average (GPA) using the t test statistical analysis and found no statistical differences ($t = -0.747,\ p = 0.459$). In the second experiment, we had 43 students who did both parts before introducing the anti-patterns and after that introduction. In the third experiment, we had 61 students identifying modeling problems after they were familiar with the anti-patterns and the catalog. The first

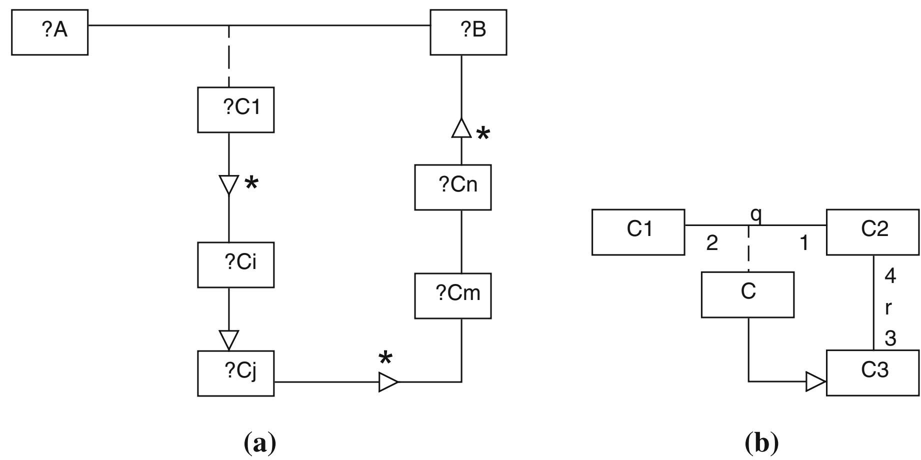


Fig. 19 The identification structure and an instance of the AHC pattern

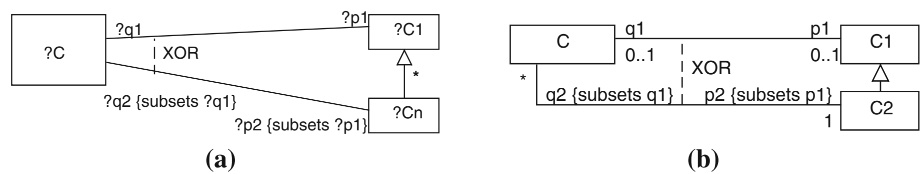


Fig. 20 The identification structure and an instance of the XORSub pattern

Table 3 Experiment goals and hypotheses

Exp. number	Goal	Null hypothesis
1	Test whether the ability of non-experts to identify problems in a given model improves after being exposed to concrete examples of modeling problems	H_0^1: no significant difference exists between the number of problems identified by non-experts when modeling problems are (1) generally explained; (2) introduced by examples
2	Test whether the ability of non-experts to identify problems in a given model improves after being exposed to anti-patterns for identifying modeling problems	H_0^2: no significant difference exists between the number of problems identified by non-experts when modeling problems are (1) generally explained; (2) introduced by anti-patterns
3	Test whether non-experts efficiently identify problems by following the notion of anti-patterns for identifying modeling problems	

experiment was performed in one class, and the second and the third experiment were performed in another class[4].

5.3 Experiment procedure

During the course (related to all experiment), the participants studied the syntax and semantics of the class-diagram language. The experiments were executed in several stages as described in Table 4.

The modeling problems selected for identification for the experiments were those that represent a variety of constraints interactions and were given special emphasis in class. The anti-patterns that address these problems were elaborated in the previous section. The chosen problems cover most constraints studied in the course, at various levels of difficulty. They involve the multiplicity, composition, association-class, class hierarchy, and disjoint-GS constraints. The difficulty level is assessed by the number of constraints involved, types of constraint (for example, multiplicity is simpler than class hierarchy), and the diagram structure (existence of cycles and sequences of associations etc.). The *Diamond, PMC*, and *PCC* anti-patterns have simple identification structures, each involving only two kinds of constraints. The *HMC* anti-pattern is similar to the *PMC* in its identification structure but involves class hierarchy. The *AHC* anti-pattern includes the association-class, multiplicity and class-hierarchy constraints, and its identification structure involves a sequence of associations.

[4] The numbers of participants in the second and third experiments are different since the second experiment was voluntary while the third was part of an exam.

Table 4 Stages of the experiments

Exp. number	Stage 1	Stage 2	Stage 3	Stage 4
1	All students were taught the syntax and semantics of class diagrams, including the introduction of consistency and finite satisfiability modeling problems	The first group received a class diagram (topic: university management systems) and had to identify modeling problems	The students were shown concrete examples with modeling problems as in Fig. 5. The students were not exposed to the anti-pattern catalog	The second group received the same class diagram (topic: university management systems) and had to identify modeling problems
2		The students received a class diagram (topic: university management system) and had to identify modeling problems	The students were taught the notion of anti-patterns for model correctness, the pattern specification language, and were exposed to the pattern catalog	The students received another class diagram (topic: library management system) with the same modeling problems and had to identify modeling problems
3		The students were taught the notion of anti-patterns for model correctness, the pattern specification language, and were exposed to the pattern catalog. They also had home assignment regarding the identification of modeling problems		The students received a synthetic class diagram (domain agnostic) and had to identify modeling problems

The most complex identification structure is that of the *DisPath* anti-pattern, which includes class-hierarchy, disjoint-GS constraints, and sequences of associations between subclasses to their superclass. In all experiments, we had a pre-defined solution. We checked whether students identified the problems, and whether they provided proper explanations.

5.4 Results

In the following, we present the results of the three experiments. Table 5 presents the results of the first experiment. The numbers in the first two rows indicate the average identification percentage of the students within each group, and the last row presents the statistical significance level. It clearly shows that the second group achieved better results. Applying the Mann–Whitney test[5], we found out that most of the differences were statistically significant (these are marked in bold). The results indicate that by increasing the awareness of students to modeling problems via concrete examples, they were able to better identify modeling problems. Thus, we reject the null hypothesis and state that indeed there is a difference between the number of problems identified by non-experts when introducing modeling problems by (1) general explanations or (2) concrete problem examples, in favor of the latter option.

Table 6 presents the results of the second experiment. Here, again the numbers in the first two rows indicate the average identification percentage of the students within each of the groups (in that case rounds), and the last row presents the statistical significance level. Following the Wilcoxon rank test[6], the results indicate a significant difference (these are marked in bold) between the number of problems that non-experts identify when general comments on modeling problems and anti-patterns are introduced (in favor of the anti-patterns). Thus, we reject the null hypothesis and state that indeed there is a difference between the number of problems identified by non-experts when provided with general explanation of modeling problems and when introducing anti-patterns, in favor of introducing the anti-patterns. Therefore, it is obvious that an increased awareness of modeling problems by means of introducing anti-patterns results in an improved ability to identify problems in erroneous models.

Observing the results of the second experiment, we were interested in checking the extent to which anti-patterns help in identifying only modeling problems that occur in the given class diagrams. To reach this goal, we set up the third experiment in which we compared the number of identified prob-

[5] The Mann–Whitney test is a nonparametric statistical hypothesis test for assessing whether one of two sets of independent observations tends to have larger values than the other [29].

[6] The Wilcoxon signed-rank test is a nonparametric statistical hypothesis test used when comparing two related samples (in our case, each student had two comparable observations) [30].

Table 5 Results of the first experiment

	Diamond	AHC	PMC	HMC	DisPath	PCC	Total
Group 1—before	38.89	57.41	57.41	22.22	24.07	61.11	43.52
Group 2—after	66.66	61.11	94.44	61.11	27.78	88.89	66.67
Sig.	0.056	0.788	**0.006**	**0.009**	0.868	**0.037**	**0.002**

Table 6 Results of the second experiment

	XORSub	Diamond	DisCom	PCC	DisPath	Total
Before	80	35	20	47.5	15	39.5
After	90.5	90	90	77.5	60	80
Sig.	0.059	**0.00**	**0.00**	0.07	**0.00**	**0.00**

lems to the overall actual number of problems in the given diagram. We further used standard information retrieval measurements, namely *precision, recall*, and *F-measure* [31], for measuring the identification accuracy. Precision measured the fraction of the number of correct problems identified with respect to the number of identified problems; recall measured the fraction of the number of correct problems identified with respect to the number of modeling problems in the diagram; F-measure, which is a standard derived metric defined as the harmonic mean of precision and recall, explains the trade-off between the precision and recall. It was calculated as follows:

$$F\text{-}measure = \frac{2 \cdot precision \cdot recall}{precision + recall}.$$

The average recall was 85.58 %, which means that the students identified the existing problems to a large extent. The average precision was 66.39 %, which means that they also found non-existing problems, but to a smaller extent. The F-measure was 73 %.

5.5 Discussion

Summarizing the results, we found out that an increased awareness to modeling problems improves the identification rate of modeling problems in class diagrams. Also, the identification rate improved remarkably when anti-patterns were introduced. Introducing concrete examples improves the problem identification by 50 % (from 43.52 to 66.67 %), whereas introducing anti-pattern improves the problem identification by 100 %. (from 39.5 to 80 %). Concrete examples become non-effective for problems with complex interactions and structures. For example, in the case of the *DisPath* problem (shown in Fig. 5d), introducing an example did not improve the students' results (see Table 5). We assume that this example fails to capture the pattern's complex structure: A GS with a *collection* of subclasses (the example presents only two subclasses) and *association paths* from the superclass to its subclasses, that would enable the problem identification. Indeed, after introducing the patterns in the second experiment, the students show significant improvement, as illustrated in Table 6. The identification structure of *DisPath* (Fig. 17a) captures the abstract structures of the patterns using *association paths* and *class collection*.

We believe that providing anti-patterns improves the students' ability to identify modeling problems as these consist of abstract and accurate specification of the problems. Yet, due to different settings of the two experiments, we were unable to statistically justify the results. Only further experiments will allow checking the impact of introducing anti-patterns instead of concrete examples.

5.6 Threats to validity

The results of the experiments need to be considered in view of several threats to validity:

Construct validity threats concerns the relationships between theory and observation, which are mainly due to the method used to assess the outcomes of tasks. In this study, we examined a specific approach and a set of modeling problems. Thus, the results may be influenced from these selections. However, reviewing other anti-patterns, we found that we addresses representative portion of modeling problems.

Internal validity threats concerns external factors that may affect the dependent variables, which may be due to individual factors such as familiarity with the domain, the degree of commitment of the subjects, and the training level that the subjects underwent (e.g., the examples used). All subjects had similar background of the domains, they share similar commitments, and although training include examples, their selection was of low importance as we explicate their generalization. The various factors mentioned are mitigated by the experiment designs that we chose. That is, we used one-factor experiment design with two treatments and random assignments that should eliminate the possible threats.

Conclusion validity threats concerns the relationship between the treatment (the additional education) and the outcome. We followed the various assumptions of the statistical tests when analyzing the results. For example, when data normality could not be assumed, we performed a sta-

tistical analysis using nonparametric tests. Also, subjectivity was reduced as we had a pre-defined solution that consists of the modeling problems. Nevertheless, human judgment was required when analyzing the students' explanations.

External validity threats concerns the ability to generalize the results. The main threat in this area stems from the choice of subjects and from using simple tasks in the experiment. The subjects were bachelor students with little experience in modeling but they represent a population of modelers. We believe that the students were much more model oriented than practitioners as they learned in-depth the class-diagram syntax and semantics. More generally, [32] argue that using students as subjects instead of software engineers is not a major issue, as long as the research questions are not specifically focused on experts, as is the case in this study. The tasks used in the first experiments were limited in their size (i.e., only one class diagrams) to allow sufficient time for identifying the modeling problems. However, these were complex enough and consisted of several problems. Thus, we believe that this kind of threat is eliminated. We think that further studies are required to further generalize the results and further refined their causes.

6 Related work

Patterns and anti-pattern research and collections span a wide range of paradigms, like human–computer interfaces [6], Web site design [33] pedagogy [3, 4], and software design [8, 34–36]. The latter has gained much attention as stressed by [37]. Addressing patterns and anti-patterns requires the definition of suitable languages, the classification of patterns (mostly in catalogs), and the training (and educating) with these. A variety of approaches for pattern formalization (some of which are surveyed below) is included in [38].

In Sect. 6.1, we discuss approaches to the design of languages of modeling patterns, in Sect. 6.2, we review existing anti-pattern catalogs, and finally, in Sect. 6.3, we survey empirical studies that check issues related to teaching design patterns.

6.1 Languages for specification of modeling patterns

In this subsection, we analyze languages of modeling patterns in light of several properties that are defined below. We divide the properties into *language* and *pragmatic* categories. The language properties are the following:

1. The **approach** property refers to the mathematical theory or modeling approach.
2. The **supported abstractions** property refers to the kinds of concepts that the pattern language supports.
3. The **pattern sensitivity** property refers to whether the modeling language is changed upon addition or modification of patterns.
4. The **model/domain dependency** property examines whether pattern specification relies on model or domain-specific terminology, i.e., relies on an associated meta-model or a domain model.
5. The **pattern instantiation** property refers to the rules for deriving concrete cases from patterns, whether these rules are language specific or standard instantiation rules.

The pragmatic properties are the following:

1. The **visualization** property checks whether pattern visualization (assuming that there is one) introduces new visual syntax with respect to the modeling language or domain, and to what extent.
2. The **applications** property refers to whether any indication for using the language is reported, e.g., case studies or experiments.
3. The **support** property looks at the various means provided in order to use the language, e.g., examples, a public catalog and a publicly available detection tool.

We demonstrate our analysis on two notable approaches *RBML* [24], and *VPML* [39], and shortly describe other approaches we reviewed. The analysis is classified into languages that are based on a *Pattern* model, and languages that are based on other abstraction means, such as *Variable, Function*, and *Collection* abstractions. We summarize with an inclusive property comparison of all surveyed approaches.

6.1.1 Languages that are based on a pattern meta-model

The RBML specification [24, 40, 41]: France et al. [24] introduce a role-based modeling language for patterns, based on the UML meta-model. For each pattern, the UML meta-model is extended with pattern-specific meta-classes that correspond to roles in the pattern. For example, the *Composite* pattern of [8], whose identifying structure is described in Fig. 9a, would be specified by the meta-model in Fig. 21. This meta-model specializes the highlighted UML meta-classes by adding three meta-classes for the *Composite*, *Leaf*, and *Component* roles, three meta-classes *ComponentGeneralizationSet*, *LeafGeneralization*, and *CompositeGeneralization* for the generalization roles *Composite* $\prec$ *Component* and *Leaf* $\prec$ *Component*, and three meta-classes *ComposedOf*, *Child*, and *Parent* for the association between *Composite* and *Component*. The pattern-specific meta-classes can be further constrained using OCL [42, 43]. The addition of new classes

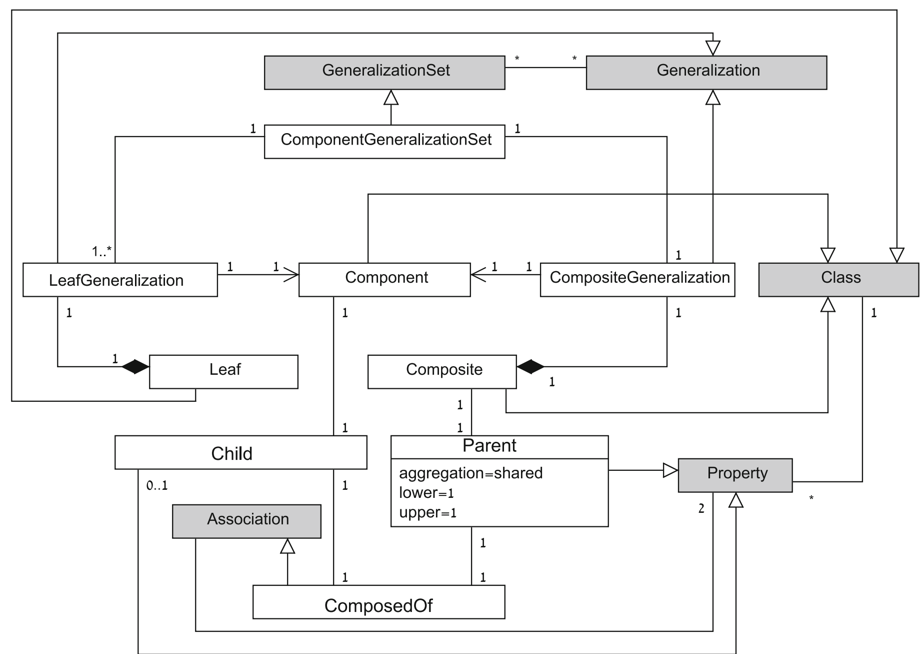


Fig. 21 The meta-model specification of [24] for the identification structure of the *Composite* design pattern

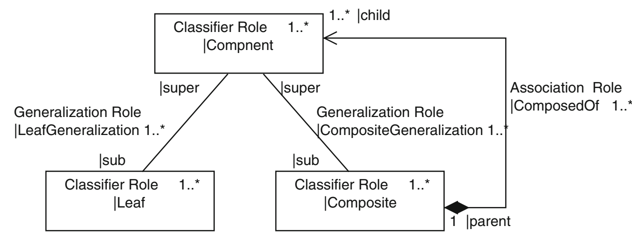


Fig. 22 The Composite pattern in the *RBML* language (extracted and simplified from [41])

to the meta-model when introducing new patterns makes the approach pattern sensitive.

The precise meta-model specification is not cognitively effective, as it hides the intended *Composite* design pattern structure behind the details of the syntax specification. Thus, in [40], the authors proposed a visual concrete syntax representation, entitled *RBML*, that uses a class diagram like notation. Figure 22 presents the *RBML* version for the solution structure of the *Composite* design pattern.

Instantiation of *RBML* patterns into concrete cases requires acquaintance with its meta-model. For example, specialization in the graphics domain, as in Fig. 9b, requires *role-based* instantiation for the associations in the *RBML* pattern: The associations with the *Generalization role* are instantiated by class-hierarchy constraints, while the association with the *Association role* is instantiated by a binary association. *RBML* uses multiplicity constraints to restrict the number of elements playing the role in a denoted (conforming) class diagram. The *1..1* multiplicity on the *parent* property specifies that an instance of *Component* (i.e., a class) can have only one property with the *ComposedOf* association. This interpretation is different from the UML semantics of multiplicity constraints, which refers to the number of objects that can be related.

The VPML specification [44]: *VPML* is a visual language for specification of design patterns in a given domain model[7]. It is introduced by a MOF-based generic *Pattern* model, associated with concrete visual syntax for instances. *VPML* enables pattern abstraction using *contextual properties* written as OCL derived expressions, *pattern composition* and *pattern variability*. The latter supports pattern *conditions, corollaries, variants*, and *parametrization* of role attributes whose type is primitive. The *VPML* visualization is similar to a class diagram, with specific visualization for external or internal roles in patterns, for pattern composition, and for the various variability means. Figure 23 presents the *VPML* specification of the *Composite* design pattern. It uses pattern composition with an *ObjectRecursion* pattern, which defines the general structure of a recursive class hierarchy.

The rules for instantiation of patterns into concrete cases are not explicitly discussed in *VPML*, but are implemented within their detection tool, using its *PResults* DSML for reporting detected pattern occurrences[8].

[7] *Epattern* [39] is an earlier MOF-based visual language of these authors.

[8] *VPML* is used in an implemented pattern detection tool (done by mapping to a QVTr model transformation). Two case studies of pattern detection, involving the GoF patterns and control flow patterns in BPMN, are reported.

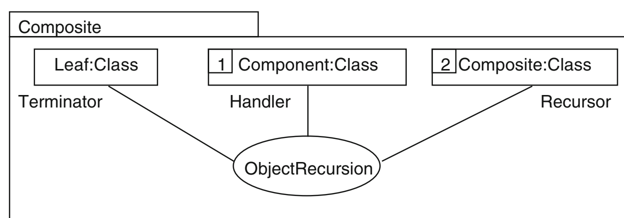


Fig. 23 The *VPML* specification of the *Composite* design pattern

The specification of patterns in *RBML* and in *VPML* is role-based. But, while *RBML* extends the pattern model with pattern-specific roles, the generic *Pattern* model of *VPML* is not affected by specific patterns. *VPML* patterns are specified as instances of the pattern language and rely on an associated meta-model or domain specific. That is, *RBML* is pattern sensitive, while *VPML* is model or domain dependent.

Bottoni et al. [45–47] introduce a Category theory based pattern specification language that models multiple aspects of patterns. They capture variable components in patterns, role interaction, pattern expansion (including multiple instantiations), addition of positive and negative invariants, pattern composition, and conflict analysis. Formulation of the solutions of all GoF patterns is demonstrated, as well as, support for completion of models, based on given patterns. Their approach is similar to that of *VPML* in having a generic Pattern model, indicating that it is not sensitive to pattern modification, and like *VPML*, it is model dependent, since it relies on an associated meta-model. For pattern visualization, the approach uses the conventions of the models to which the patterns apply, with few additional notations.

Kim and Carrington [48] formulate in *Object-Z* a generic *Pattern* model, that includes behavior aspects. For each pattern, the generic model is extended with information about the pattern-specific roles. Therefore, like *RBML*, this approach is pattern sensitive. They demonstrate the formulation of the solutions to some creational, structural, and behavioral GoF patterns. Pattern instantiation (binding) in class diagrams, is also formulated, including constraints for its validity. This Object-Z-based language is not associated with a visual concrete syntax language.

Guennec et al. [49, 50] suggest to extend the UML meta-model with pattern-oriented stereotypes, so that patterns and their occurrences can be simultaneously specified in UML. They point to limitations of UML constructs such as *parametrized collaborations* and *collaboration usage* to account for design patterns, and demonstrate how using the suggested stereotypes some solutions to GoF patterns can be specified and instantiated. They also suggest that full account to behavioral aspects of patterns requires using temporal operators. Their visualization uses standard UML graphics.

Mens and Tourwe [51] suggest using a declarative meta-programming approach for pattern management, including specification and evolution control. They use the logic programming language SOUL for formulating a simple *pattern* model that supports *roles* and *constraints* (predicates *pattern, role, patternInstance* and *patternConstraint*). Instances of patterns are created using the assertion mechanism of logic programming (the *createPattern* predicate). This framework is further developed for supporting pattern transformation, including refactoring and pattern merge, that are associated with conflict detection and resolution.

Maplesden et al. [52, 53] present the visual *DPML* language which is based on a class-diagram-oriented *pattern* model, that is based on the concepts of *participant, dimension, binary relation* and *constraint*. The *dimension* concept accounts for participant repetition. The pattern model is extended to account for pattern instantiation concepts. The language provides visualization for the main participant types, and for the *dimension* concept. The associated tool supports pattern instantiation.

6.1.2 Languages that are based on general mathematical abstractions

Eden et al. [54–56] present the *Codecharts* visual language, designed to support software design round trip, i.e., modeling, analysis, and conformance activities. *Codecharts* has a small set of primitives that represent the notions of *Class, Binary-relationship, Signature, Hierarchy* and *Isomorphism*. Its abstraction means consist of the *Variable* and *Dimension* concepts. The latter captures high-order sets of *Codecharts* elements. The formal foundation of *Codecharts* relies on translation to first-order predicate logic. *Codecharts* is supported by a fully automated design verification tool. It has been used for pattern formulation, e.g., the solutions to GoF patterns [55, 57], using its *Variable* and *Dimension* abstraction means.

Bayley and Zhu [58] formulate the 23 GoF patterns using a structured text protocol that is based on first-order predicate logic. The constraints in pattern specifications refer to a formal specification of a simplified portion of UML in a grammatical model (*GEBNF*) that is associated with constraints written in first-order predicate logic. Their pattern formulation includes specification of static and dynamic conditions, and enables specification of pattern variants. No visualization is provided. They report on a detection tool and a case study that shows the importance of behavioral conditions in the GoF patterns.

Ballis et al. [59, 60] present a visual pattern notation that is based on three visual pattern constructors: *Replication, hier-*

archy, and *concatenation*. The intention is to express possible repetition in pattern instantiation, multiple subclasses in a hierarchy, and combination of relations. The visual notation is translated into a functional language, for which an instance detection algorithm is provided. In terms of supported abstractions, this approach supports variable abstraction and some restricted forms of collections. The language is used for representing the GoF patterns and some anti-patterns.

The *Pattern Class Diagram* (*PCD*) language, which is used in our catalog, supports abstraction over class diagrams. It is *abstraction-based* rather than *role-based*. It does not assume a *Pattern* model, since the correctness anti-patterns in our catalog hardly have a *role* notion. The supported abstractions include *variable, set* and *sequence*. The *variable* abstraction can apply to any type in the class-diagram model. The *set* abstraction (unordered collection) applies to classes, and the *sequence* abstraction applies to associations, aggregations, and hierarchies. *PCD* sequences can consist of mixed type elements, an abstraction feature not supported by any of the above languages. The language is oblivious to domain models, and to addition, removal, or modification of patterns. Pattern visualization is clearly distinguished from pattern instantiation, and the instantiation rules do not require acquaintance with the language meta-model. *PCD* is a class-level language. Therefore, object-level constraints, which are needed for pattern verifications in the catalog, require object-level languages, like OCL [43], F-OML [61] or first-order predicate logic.

6.1.3 Property-based comparison of the reviewed languages

The comparison is summarized in two tables below. Table 7 summarizes the *language* properties of the pattern specification languages that we have reviewed, and Table 8 summarizes the *pragmatic* properties. Note that among the languages that are *Pattern* model based, the *Pattern sensitivity* and the *model/domain dependency* are complementary features. This is not surprising, since the information about the underlying model/domain must be provided somewhere—either added to the generic model within pattern-specific roles, or as an associated model to patterns. Also, the role abstraction for the ones that are pattern sensitive is model-tailored. All languages that are not *Pattern* model based, include various kinds of *Variable* abstractions.

6.2 Catalogs of modeling patterns

The experiments described in the previous section show that awareness of developers to modeling problems meaningfully improves their analytic skills. Therefore, having support for pattern specification and usage is an important aspect in modeling training. Yet, in spite of the multiple suggestions for languages for modeling patterns, there are very few catalogs for supporting modeling activities.

Tanriover and Bilgen [21, 62] present a set of anti-patterns for semantic and quality problems in the structural and behavioral aspects of UML. Anti-patterns are conceived as an integral part of what they term the *modeling inspection* process[9], an approach for improving the quality of models. The specification of anti-patterns uses concrete examples and textual descriptions. No pattern specification language is used. The catalog includes anti-patterns for some class diagram consistency problems like the *Diamond* anti-pattern, and for the finite satisfiability problem that results from the interaction of multiplicity constraints on a unary association (as in Fig. 3a). The anti-patterns refer only to a subset of the class-diagram constraints.

Elaasar et al. [28, 63] introduce a catalog of modeling anti-patterns. It refers to the UML and the MOF modeling languages. The anti-patterns in the catalog are grouped into four categories:

1. **MOF and UML syntax anti-patterns**: The syntactic problems are based on the MOF and UML *well-formedness rules* [12]. For example *Classifier Has Generalization Cycle* is a UML anti-pattern that characterizes illegal generalization cycles in UML.
2. **Semantic anti-patterns**: The anti-patterns in this category capture syntactic quality problems, i.e., redundant or incomplete design problems. Correctness problems, i.e., consistency and finite satisfiability problems, are not handled.
3. **Convention anti-patterns**: This category includes anti-patterns that capture common violations of conventions, such as naming (e.g., a non-alphabetic name of an *Element*) and documentation conventions.
4. **Notational problems**: The anti-patterns in this category capture quality problems, such as completeness or clarity of visual representations with respect to their metamodel.

The anti-patterns are written in the *QVTr* language of the MOF [64] and are intended to support automatic detection of modeling problems in models, based on their specifications in terms of the meta-model. The *VPML* visual language of these authors [65, 66] is intended to turn the catalog more accessible, using the translation from *VPML* to *QVTr*.

The anti-pattern catalog described in this paper includes anti-patterns for the major correctness problems in class diagrams. These patterns are not provided in either of the above

9 Similarly to *code inspection*.

Table 7 *Language* properties of pattern specification languages

	Approach	Supported abstractions	Pattern sensitivity	Model/domain dependency	Pattern instantiation
Pattern model based					
RBML	Meta-modeling	Model-tailored role, multiplicity, role binding	Yes	No	Meta-model instantiation
VPML	Meta-modeling—MOF-based	Pattern-role, role inter-relationship, pattern dependency, role binding	No	Yes	No explicit rules (embedded in the detection tool)
Bottoni, Guerra and Lara	Category theory	Pattern-role, variable, repetition	No	Yes	Variable substitution with repetition
Kim and Carrington	Object-Z	Model-tailored roles, role inter-relationship	Yes	No	Role binding (text)
Guennec, Sunye and Jezequel	Meta-modeling		No	Yes	$<< meta >>$, $<< instance >>$ stereotypes
Mens and Tourwe	Declarative meta-programming using SOUL	Pattern-role, variable	No	Yes	Variable substitution
DPML	Meta-modeling, UML	Pattern-participant, repetition, participant binding	No	Yes	Pattern instantiation meta-model
Mathematical abstractions					
Codecharts (LePUS)	Logic	Variable, high-order sets	No	Yes	Variable substitution
Ballis, Baruzzo and Comini	Functional	Variable, subclass-collection, repetition	No	Yes	Variable substitution with repetition, subclass-collection expansion
PCD	Set-theoretic abstract syntax	Typed variable, collection	No	Yes	Variable substitution, collection expansion

Table 8 *Pragmatic* properties of pattern specification languages

	Visualization	Applications	Support
Pattern model based			
RBML	Model-tailored; special semantics		Editor, examples, pattern detection tool
VPML	UML notation for pattern collaboration, special visualization for pattern-roles and role inter-relationships	Detection tool applied to GoF and to control flow patterns in BPMN	Online catalog, examples, all GoF patterns, translation to QVTr, pattern detection tool
Bottoni, Guerra and Lara	Model-tailored	–	Examples in structural and interaction domains, all GoF patterns
Kim and Carrington	No visualization support is provided	–	Few GoF examples
Guennec, Sunye and Jezequel	UML-based	–	GoF examples, Pattern instantiation, Pattern detection
Mens and Tourwe	No visualization support is provided	–	Few GoF examples
DPML	New pattern visualization language	Usage experiments	Gof examples, pattern instantiation tool with consistency analysis
Mathematical abstractions			
Codecharts (LePUS)	Codecharts new notation	Software design and accuracy experiments	All GoF patterns, fully automated modeling, code-generation and correctness tool
Bayley and Zhu	No visualization support is provided	–	All GoF patterns
Ballis, Baruzzo and Comini	New notation	–	GoF examples
PCD	Extended class diagram	Education-targeted experiments	Online catalog, examples

catalogs. Pattern presentation is cognitively effective, due to the *PCD* visual language that is used for writing identification structures and suggested repairs. The purpose of the catalog is twofold. First, it has an educational role, and thus all patterns are introduced with all pattern related information as described in Sect. 4.2. Second, *PCD* specification is used as input for the detection tool that is currently under construction.

6.3 Empirical studies about teaching design patterns

Design patterns do not form an organized structured theory. Therefore, teaching and learning design patterns is a non-trivial task. Their abstract, somewhat scattered nature, poses difficulties for students and educators. In particular, understanding the context of patterns (when to apply a pattern) is a major problem. Indeed, it is well known that the learning curve for properly **using** design patterns is slow.

Multiple studies deal with issues concerning teaching design patterns and its impact [67–74]. Chatzigeorgiou et al. [74] present an observational study of student ability to understand and apply design patterns before and after patterns are introduced. The conclusion is that patterns that are easy to comprehend and apply were frequently used, though students found it difficult to document exactly which problems had been solved by each pattern.

Patterns versus anti-patterns in education: Using positive examples (like design patterns) or negative examples (like anti-patterns) is a well-known effective educational tool [75–77], but there is a debate concerning the effectiveness of the two approaches. The use of anti-patterns in teaching in general and in teaching human–computer interaction concepts in particular is the subject of [78, 79]. This empirical study claims that teaching positive examples (patterns) is significantly more effective than teaching negative examples (anti-patterns). The authors suggest that negative examples are incompatible with the internal process of acquiring and representing knowledge. Stamelos et al. [80] argue against these results and demonstrate the effectiveness of anti-patterns by two laboratory exercises (in the area of project management). They further stress that knowledge represented by anti-patterns is effectively transferred to stu-

dents, and users were able to understand the symptoms, causes, and consequences arising from the problems in each anti-pattern.

Bolloju et al. [77] also examine the effectiveness of patterns and anti-patterns. They conducted a controlled experiment using undergraduate students to study the usefulness of negative and positive examples in teaching object modeling skills. Their results indicate that both positive and negative examples are useful, but for different tasks. Positive examples are recommended for modeling syntactic quality, while negative examples are appropriate for understanding semantic equality. This result indeed supports our choice for using anti-patterns, as the main theme of our catalog is semantic correctness.

7 Conclusion

This paper addresses the educational role of anti-patterns in improving modeling skills in UML class diagrams. We presented a catalog of correctness and quality anti-patterns for class diagrams and discussed its role as an educational tool. The catalog organizes anti-patterns by correctness or low-quality problems. Each pattern provides an identification structure, proof of the problem, and repair options. The educational role of the catalog is to increase the awareness of designers to problematic inter-relationships among modeling elements. To the best of our knowledge, this is the first catalog that provides an in-depth analysis of causes of correctness and quality problems, together with repair advices. The catalog is intended to play an educational role in teaching object modeling.

We argue that for educational purposes, patterns should be visually formulated using model-level concepts. To do this, we extend the UML class-diagram meta-model with new classes that capture new abstractions needed for pattern specification such as *compound structures* of class-diagram elements and *variables* that range over class-diagram elements. The enhancement introduces new notations such as collections and association paths. The advantage of this approach lies in its generality and simplicity. The new abstractions enjoy visualizations that are directly associated with their intended meanings.

In order to examine the extent to which an awareness of modeling problems helps identify erroneous models, we conducted two experiments: The first examined the impact of anti-patterns when they appeared as concrete examples; the other checked the impact of anti-patterns when presented as pattern class diagrams. Both experiments showed that anti-patterns had a significant positive impact: Students were able to identify modeling problems after being exposed to anti-patterns. Moreover, the results showed that the identification rate improved significantly when anti-patterns were presented using pattern class diagrams.

Following the results of this research, we plan to weave the catalog in the class material of our object modeling courses. We also plan to extend the catalog with patterns of typical OCL idioms, and possibly add design patterns for class-diagram modeling. We are in the process of integrating the catalog with our reasoning ***FiniteSatUSE*** tool [81], in order to produce a human understandable explanation and repair advice for the identified problems[10].

A different future direction involves full implementation of the *PCD* language, including an associated constraints language (either OCL [43] or FOML [61] based). We intend to develop a tool that will use the *PCD* language for class-diagram querying, testing, and identification of structural patterns. In particular, this tool will be used for identification of patterns that are not identified by the ***FiniteSatUSE*** tool.

Acknowledgments We thank Adiel Ashrov for his help in the construction of the online catalog. We are also indebted to the referees for providing detailed comments that helped in improving the paper.

Appendix: the forms in the experiments

In this Appendix, we provide the material of the three experiments. For each experiment, we first provide the used class diagrams, followed by tables that indicate pattern occurrences in those class diagrams.

The first experiment: This experiment examines the effect of introducing **concrete examples** of modeling problems to non-experts, on their ability to identify problems in class diagrams. The experiment form includes the class diagram in Fig. 24, for a university management system. Table 9 lists the occurrences of the anti-patterns *Diamond*, *AHC*, *PMC*, *HMC*, *DisPath*, and *PCC* in this class diagram.

The second experiment: This experiment examines the effect of introducing **anti-patterns** of modeling problems to non-experts, on their ability to identify problems in class diagram. In this experiment, we used two class diagrams:

1. The class diagram in Fig. 25, which is an extended version of the university management system from the first experiment, was used before introducing the anti-patterns.
2. The class diagram in Fig. 26, which describes a library management system, was used after introducing the anti-patterns.

[10] This integration is not intended to produce automatic repairs, but just to suggest repair options.

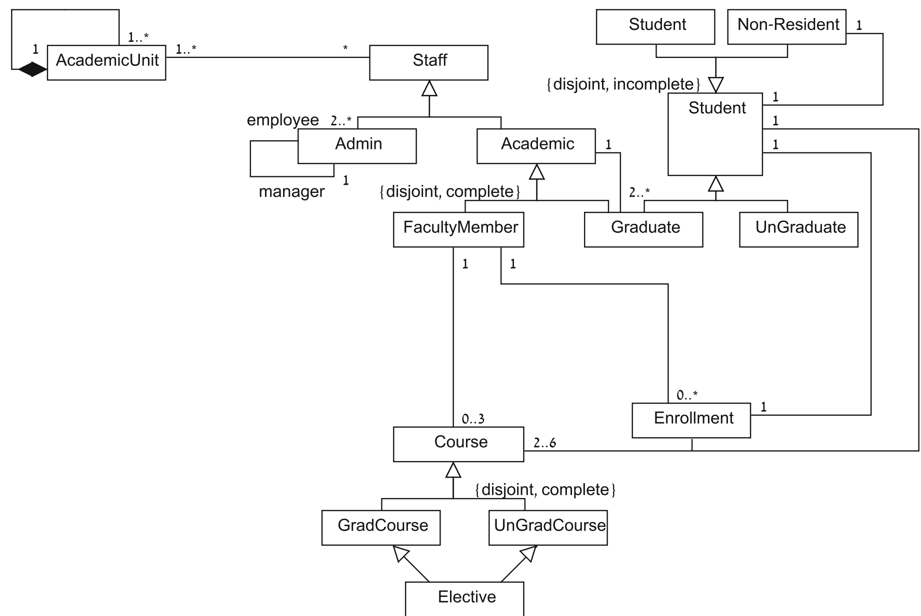


Fig. 24 The class diagram used in the first experiment (the university management system)

Table 9 Pattern occurrences the class diagram used in the first experiment

Pattern type	Pattern occurrences
Diamond	Course-GraduateCourse-UnderGraduateCourse-Elective
AHC	Course-Enrollment-Student-Graduate-Academic-FacultyMember
PMC	Admin
HMC	Student-Course-FacultyMember-Academic-Graduate
DisPath	Student-Resident-NonResident
PCC	AcademicUnit

Tables 10 and 11 list the occurrences of the anti-patterns *XORSub*, *Diamond*, *DisCom*, *PCC*, and *DisPath* in these class diagrams.

The third experiment: This experiment examines the efficiency of problem identification after the anti-patterns were introduced. In this experiment, we used the synthetic class diagram shown in Fig. 27. Table 12 lists the occurrences of the anti-patterns *PMC*, *HMC* and *AHC* in this class diagram.

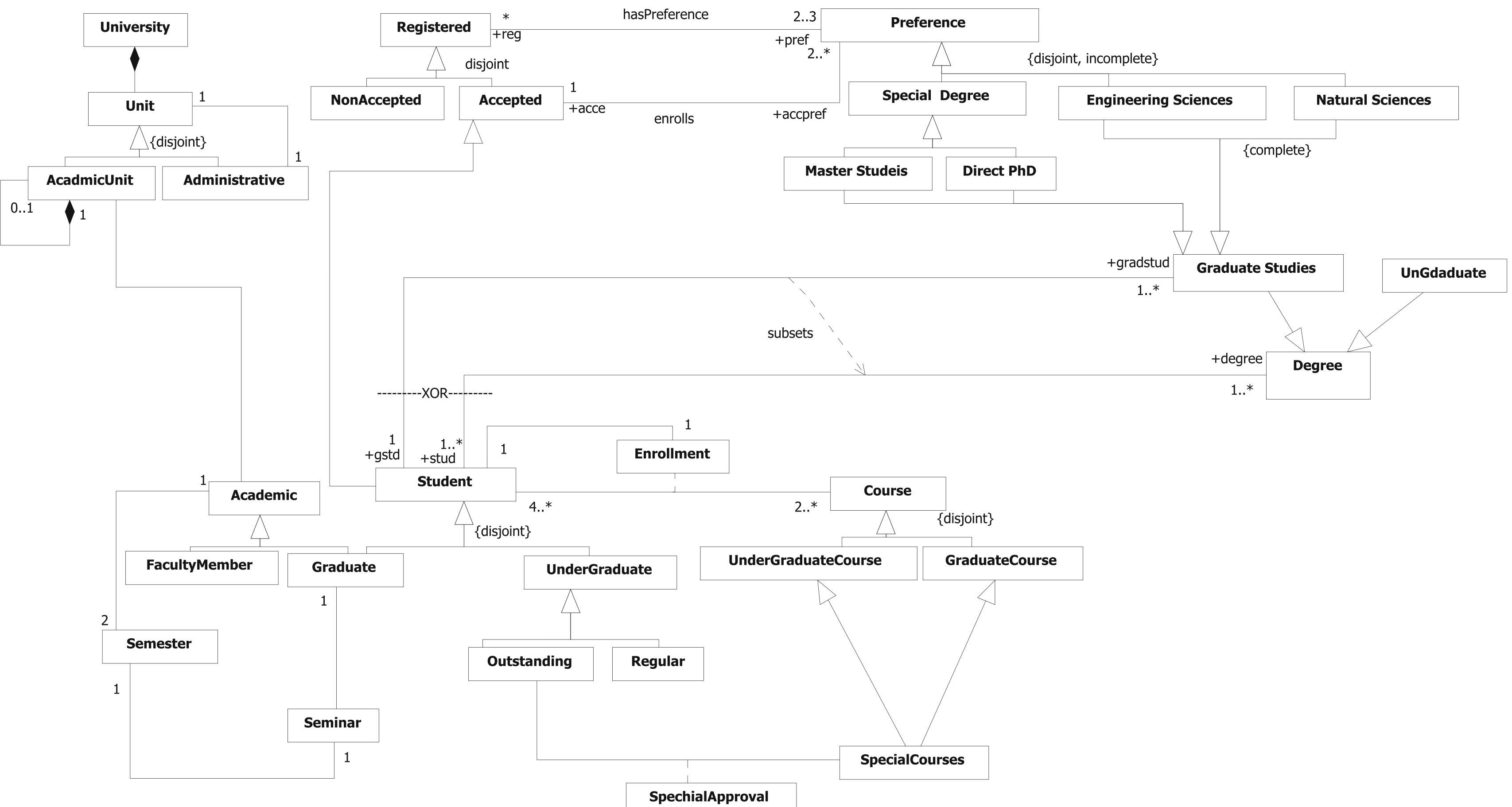


Fig. 25 The class diagram used in the second experiment before introducing the anti-patterns (the extended university management system)

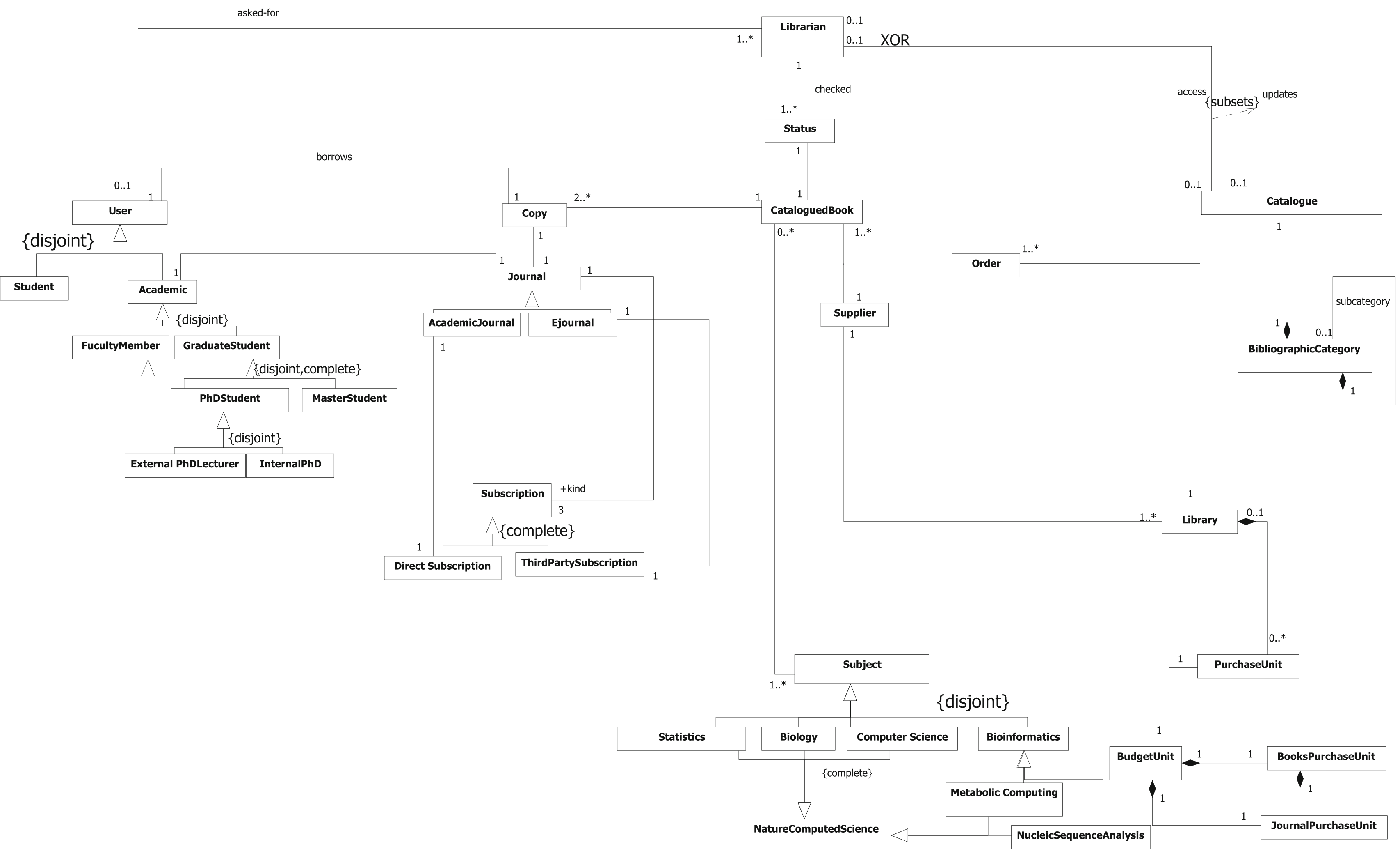


Fig. 26 The class diagram used in the second experiment after introducing the anti-patterns (the library management system)

Table 10 Pattern occurrences in the class diagram used in the second experiment before introducing the anti-patterns

Pattern type	Pattern occurrences
XORSub	Graduate Studies-Student-Degree
Diamond	Course-UnderGraduateCourse-GraduateCourse-SpecialCourse
DisCom	Preference-SpechialDegree-EngineeringSciences-NaturalSciences-DirectPhD-Master Studeis-GraduateStudies
PCC	AcademicUnit
DisPath	Unit-AcademicUnit-Administrative

Table 11 Pattern occurrences in the class diagram used in the second experiment after introducing the anti-patterns

Pattern type	Pattern occurrences
XORSub	Librarian-Catalog
Diamond	Academic-FacultyMember-GraduateStudent-PhdStudent-ExternalPhdStudent
DisCom	Subject-Biology-NatureComputerScience-MetabolicComputing-Bioinformatics
PCC	BibliographyCategory
DisPath	User-Student-Academic-Journal-Copy

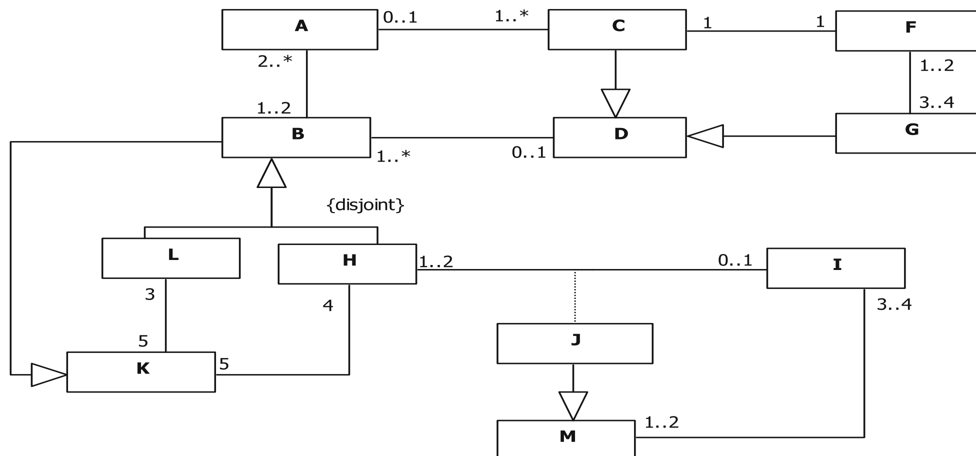


Fig. 27 The synthetic class diagram used in the third experiment

Table 12 Pattern occurrences in the class diagram used in the third experiment

Pattern type	Pattern occurrences
PMC	A-C-F-G-D-B
HMC	B-L-H-K
AHC	J-M-I

References

1. Lindland, O., Sindre, G., Sølvberg, A.: Understanding quality in conceptual modeling. IEEE Softw. **11**, 42–49 (1994)
2. Alexander, C.: The Timeless Way of Building, vol. 1. Oxford University Press, Oxford (1979)
3. The Pedagogical Patterns Project. http://www.pedagogicalpatterns.org (2010)
4. Commission, E.: The E-Len Project. http://www2.tisip.no/E-LEN/patterns_info.php (2008)
5. Rising, L. (ed.): Design Patterns in Communications Software. Cambridge University Press, New York, NY (2001)
6. Borchers, J.: A Pattern Approach to Interaction Design, 1st edn. Wiley, London (2001)
7. Fowler, M.: Analysis Patterns: Reusable Object Models. Addison-Wesley, Reading, MA (1997)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, vol. 206. Addison-Wesley, Reading, MA (1995)
9. Brown, W., Malveau, R., McCormick, H., Mowbray, T.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, New York (1998)
10. Makarenkov, V., Jelnov, P., Maraee, A., Balaban, M.: Finite satisfiability of class diagrams: practical occurrence and scalability of the finitesat algorithm. In: MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation, pp. 1–10. ACM (2009)
11. Balaban, M., Maraee, A., Sturm, A.: Management of correctness problems in UML class diagrams: towards a pattern-based approach. Int. J. Inf. Syst. Model. Des. **1**, 24–47 (2010)
12. OMG: UML 2.4 Superstructure Specification. Specification Version 2.4.1. Object Management Group (2011)
13. Balaban, M., Maraee, A.: Finite satisfiability of UML class diagrams with constrained class hierarchy. ACM Trans. Softw. Eng. Methodol. (TOSEM) **22**(3), Article 24, [42 pages] (2013). doi:10.1145/2491509.2491518
14. Maraee, A., Balaban, M.: Inter-association constraints in UML2: comparative analysis, usage recommendations, and modeling guidelines. In: MoDELS 2012 (2012)
15. Berardi, D., Calvanese, D., Giacomo, D.: Reasoning on UML class diagrams. Artif. Intell. **168**, 70–118 (2005)
16. Maraee, A., Makarenkov, V., Balaban, B.: Efficient recognition and detection of finite satisfiability problems in UML class diagrams: handling constrained generalization sets, qualifiers and association class constraints. In: MCCM08 (2008)
17. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. J. Softw. Syst. Model. **4**, 386–398 (2005)
18. BGU Modeling Group. http://www.cs.bgu.ac.il/~modeling/ (2010)
19. Maraee, A., Balaban, M.: Efficient recognition of finite satisfiability in UML class diagrams: strengthening by propagation of disjoint constraints. In: Proceedings International Conference Model-Based Systems Engineering MBSE '09, pp. 1–8 (2009)
20. BGU Modeling Group: A Catalog of UML Class Diagram Anti-Patterns. http://www.cs.bgu.ac.il/~cd-patterns/ (2010)
21. Tanriover, O., Bilgen, S.: A framework for reviewing domain specific conceptual models. Comput. Stand. Interfaces **33**, 448–464 (2011)
22. Llano, M., Pooley, R.: UML specification and correction of object-oriented anti-patterns. In: ICSEA '09, pp. 39–44 (2009)
23. Riehle, D.: Lessons learned from using design patterns in industry projects. In: Transactions on Pattern Languages of Programming II, vol. 6510 of LNCS, pp. 1–15 (2011)
24. France, R., Kim, D., Ghosh, S., Song, E.: A UML-based pattern specification technique. IEEE Trans. Softw. Eng. **30**, 193–206 (2004)
25. Maraee, A.: UML Class Diagrams-Semantics, Correctness and Quality. PhD thesis, Ben Gurion University of the Negev (2012)
26. Figl, K., Derntl, M., Caeiro Rodriguez, M., Botturi, L.: Cognitive effectiveness of visual instructional design languages. J. Vis. Lang. Comput. **21**, 359–373 (2010)
27. Moody, D.: The "physics" of notations: towards a scientific basis for constructing visual notations in software engineering. IEEE Trans. Softw. Eng. **35**, 756–779 (2009)
28. Elaasar, M., Briand, L., Labiche, Y.: Metamodeling Anti-Patterns. https://sites.google.com/site/metamodelingantipatterns (2010)
29. Mann, H., Whitney, D.: On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. **18**, 50–60 (1947)
30. Wilcoxon, F.: Individual comparisons by ranking methods. Biometr. Bull. **1**, 80–83 (1945)
31. Manning, C., Raghavan, P., Schtze, H.: Introduction to Information Retrieval, vol. 1. Cambridge University Press, Cambridge, MA (2008)
32. Kitchenham, B.A., Lawrence, S., Lesley, P., Pickard, M., Jones, P.W., Hoaglin, D.C., Emam, K.E.: Preliminary guidelines for empirical research. IEEE Trans. Softw. Eng. **28**(8), 721–734 (2002)
33. Toxboe, A.: User interface design pattern library. http://ui-patterns.com/patterns (2011)
34. The Hillside Group: Software Patterns. http://hillside.net/patterns/links (2010)
35. The AntiPatterns Group: The AntiPatterns Catalog. Cunningham & Cunningham, Inc. (2010)
36. El-Attar, M., Miller, J.: Improving the quality of use case models using antipatterns. Softw. Syst. Model. **9**, 141–160 (2008)
37. Henninger, S., Corrêa, V.: Software pattern communities: current practices and challenges. In: Proceedings of the 14th Conference on Pattern Languages of Programs, pp. 14:1–14:19. PLOP 07, New York, NY, USA, ACM (2007)
38. Taib, T.: Design Patterns Formalization Techniques. IGI Publishing, Hershey, PA (2007)
39. Elaasar, M., Briand, L., Labiche, Y.: A metamodeling approach to pattern specification. In: Model Driven Engineering Languages and Systems, vol. 4199 of LNCSpp, pp. 484–498. Springer, Berlin (2006)
40. Kim, D.: The role-based metamodeling language for specifying design patterns. In: Design Pattern Formalization Techniques. GI Global, pp. 183–205 (2007)
41. Kim, D., El Khawand, C.: An approach to precisely specifying the problem domain of design patterns. J. Vis. Lang. Comput. **18**, 560–591 (2007)
42. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, Reading, MA (2003)
43. OMG: OMG Object Constraint Language (OCL). Specification Version 2.3.1. Object Management Group (2012)
44. Elaasar, M., Briand, L., Labiche, Y.: VPML: An Approach to Detect Design Patterns in MOF-Based Modeling Languages. Technical Report SCE-10-02, Carleton University (2010)

45. Bottoni, P., Guerra, E., de Lara, J.: A language-independent and formal approach to pattern-based modelling with support for composition and analysis. Inf. Softw. Technol. **52**, 821–844 (2010)
46. Bottoni, P., Guerra, E., de Lara, J.: Towards a formal notion of interaction pattern. In: IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2010, pp. 235–239 (2010)
47. Bottoni, P., Guerra, E., De Lara, J.: An Algebraic Formalization of the GoF Design Patterns (2010). ArXiv, preprint arXiv:1003.3338.
48. Kim, S., Carrington, D.: A formalism to describe design patterns based on role concepts. Formal Aspects Comput. **21**, 397–420 (2009)
49. Ho, W., Jézéquel, J., Le Guennec, A., Pennaneac'h, F.: UMLAUT: an extendible UML transformation framework. In: 14th IEEE International Conference on Automated Software Engineering, 1999, pp. 275–278. IEEE (1999)
50. Le Guennec, A., Sunyé, G., Jézéquel, J.: Precise modeling of design patterns. In: Proceedings of UML 2000, vol. 482–496 of LNCS, pp. 482–496. Springer, Berlin (2004)
51. Mens, T., Tourwe, T.: A declarative evolution framework for object-oriented design patterns. In: Proceedings of the International Conference on Software, Maintenance, pp. 570–579 (2001)
52. Maplesden, D., Hosking, J., Grundy, J.: A visual language for design pattern modelling and instantiation. In: Proceedings of the IEEE 338 (2001)
53. Mapelsden, D., Hosking, J., Grundy, J.: Design pattern modelling and instantiation using DPML. In: Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications. Australian Computer Society, Inc., pp. 3–11 (2002)
54. Eden, A.H., Gasparis, A.H., Nicholson, J., Kazman, R.: Modeling and visualizing object-oriented programs with Codecharts. Form. Methods Syst. J. **43**(1), 1–28 (2013). Published online: http://link.springer.com/article/10.1007/s10703-012-0181-1/fulltext.html
55. Eden, A.H.: Codecharts: Roadmaps and Blueprints for Object-Oriented Programs. Wiley, London (2011)
56. Eden, A., Gasparis, E., Nicholson, J.: LePUS3 and Class-Z reference manual. Technical Report, University of Essex, CSM-474, ISSN, pp. 1744–8050 (2007)
57. Nicholson, J., Gasparis, E., Eden, A.H., Kazman, R.: Automated verification of design patterns with LePUS3. In: Proceedings of the 1st NASA Formal Methods Symposium (NFM09), pp. 76–85. NASA (2009)
58. Bayley, I., Zhu, H.: Formal specification of the variants and behavioural features of design patterns. J. Syst. Softw. **83**, 209–221 (2010)
59. Ballis, D., Baruzzo, A., Comini, M.: A minimalist visual notation for design patterns and antipatterns. In: Fifth International Conference on Information Technology: New Generations, pp. 51–56. IEEE (2008)
60. Ballis, D., Baruzzo, A., Cominia, M.: A rule-based method to match software patterns against UML models. Electron. Notes Theor. Comput. Sci. **219**, 51–66 (2008)
61. Balaban, M., Kifer, M.: Logic-based model-level software development with F-OML. In: Model Driven Engineering Languages and Systems, pp. 517–532. Springer, Berlin (2011)
62. Tanriover, R.: An Inspection Approach for Conceptual Models of The Mission Space in a Domain Specific Notation. PhD thesis, The Middle East Technical University, The Department of Information Systems (2008)
63. Elaasar, M., Briand, L., Labiche, Y.: Domain-specific model verification with QVT. In: Proceedings of the 7th European conference on Modelling Foundations and Applications, Vol. 6698 of LNCS, pp. 282–298. Springer, Berlin (2011)
64. OMG: Mof? query / view / transformation (qvt). Specification Version 1.1. Object Management Group (2011)
65. Elaasar, M.: An Approach to Design Pattern and Anti-Pattern Detection in MOF-Based Modeling Languages. PhD thesis, Electrical and Computer Engineering Dissertation, Carleton University (2012)
66. Elaasar, M., Briand, L., Labiche, Y.: VPML: An approach to detect design patterns of MOF-based modeling languages. Software and Systems Modeling (to appear)
67. Della, L., Clark, D.: Teaching object-oriented development with emphasis on pattern application. In: Proceedings of the Australasian conference on Computing education, pp. 56–63. ACM (2000)
68. Stuurman, S., Florijn, G.: Experiences with teaching design patterns. In: ACM SIGCSE Bulletin, vol. 36, pp. 151–155. ACM (2004)
69. Wick, M.: Teaching design patterns in CS1: a closed laboratory sequence based on the game of life. In: The 36th Technical Symposium on Computer Science Education, SIGCSE 2005 (2005)
70. Dewan, P.: Teaching inter-object design patterns to freshmen. In: ACM SIGCSE Bulletin, vol. 37, pp. 482–486. ACM (2005)
71. Pecinovský, R., Pavlíčková, J., Pavliček, L.: Let's modify the objects-first approach into design-patterns-first. ACM SIGCSE Bull. **38**, 188–192 (2006)
72. Jalil, M., Noah, S., Idris, S.: Assisting students in applying design pattern solution. In: International Symposium on Information Technology, 2008, vol. 1, pp. 1–6. ITSim 2008. IEEE (2008)
73. Huang, H., Yang, D.: Teaching design patterns: a modified PBL approach. In: The 9th International Conference for Young Computer Scientists, pp. 2422–2426. IEEE (2008)
74. Chatzigeorgiou, A., Tsantalis, N., Deligiannis, I.: An empirical study on students' ability to comprehend design patterns. Comput. Educ. **51**, 1007–1016 (2008)
75. Haack, P.: Use of positive and negative examples in teaching the concept of musical style. J. Res. Music Educ. **20**, 456–461 (1972)
76. Ali, M.A.: The use of positive and negative examples during instruction. J. Instr. Dev. **5**, 2–7 (1981)
77. Bolloju, N., Schneider, C., Vogel, D.: Asymmetrical effects of using positive and negative examples on object modeling. In: 18th International Conference on Information Systems Development (2009)
78. Kotzé, P., Renaud, K., Biljon, J.: Don't do this: pitfalls in using anti-patterns in teaching human–computer interaction principles. Comput. Educ. **50**, 979–1008 (2008)
79. Kotzé, P., Renaud, K., Koukouletsos, K., Khazaei, B., Dearden, A.: Patterns, anti-patterns and guidelines-effective aids to teaching HCI principles. In: Proceedings of The First Joint BCS/IFIP WG13. 1/ICS/EU CONVIVIO HCI Educators' Workshop 2006 (2006)
80. Stamelos, I., Settas, D., Mallini, D.: Teaching software project management through management antipatterns. In: Panhellenic Conference on Informatics, pp. 8–12. (2011)
81. BGU Modeling Group: FiniteSatUSE: A Class Diagram CorrectnessTool. http://sourceforge.net/projects/usefsverif/ (2011)

Mira Balaban received a B.Sc. in mathematics and statistics from Tel Aviv University (Israel), and a M.Sc. and a Ph.D. in computer science from the Weizmann Institute of Science. She also graduated in music from the Rubin Academy of Music in Tel Aviv. She taught at the Computer Science department in SUNY Albany NY, and currently is with the Computer Science department in Ben-Gurion University in Israel. Her research is mainly in the area of software engineering, with emphasis on modeling: Correctness of and reasoning about models, modeling languages, and model patterns. Previous research was in the areas of artificial intelligence, database semantics, and computer music.

Azzam Maraee is a postdoctorate fellow at Ben-Gurion University. He has B.Sc. in mathematics, M.Sc. in information system engineering, and Ph.D. in computer science all from Ben-Gurion University, Israel. His research focuses on software engineering, with emphasis on modeling: model correctness and reasoning, modeling languages, and model patterns.

Arnon Sturm is a faculty member at Ben-Gurion University. He received his Ph.D. in Information Management Engineering from the Technion, Israel Institute of Technology. His research interests include modeling, domain engineering, agent-oriented software engineering, and system development processes. Prior to his studies, Arnon has gained extensive experience in developing software systems in industry.

Pavel Jelnov has a B.Sc. in statistics and economics (summa cum laude) and M.Sc. in statistics from Bar-Ilan University. He is currently a Ph.D. student in economics at Tel Aviv University and was a visiting doctoral fellow at Northwestern University. Pavel's main fields of interest are empirical and applied economics, demographical history, marriage, and labor markets. His works include theoretical models and econometrical analysis. His Ph.D. dissertation deals with marriage trends in developed countries in the twentieth century. He teaches B.A. level macroeconomics and econometrics.