

A Constraint-based Assessment Approach for Free-Form Design of Class Diagrams using UML

Nguyen-Thanh Le
Department of Informatics
University of Hamburg
le@informatik.uni-hamburg.de

Abstract. For a design problem in a modeling language like UML, there is no single correct solution. Usually, there are many solutions, which satisfy a given problem specification. In principle, the solution space can be infinite. However, current approaches evaluate student's entries by comparing them with a limited set of possible solutions and errors. Some other approaches anticipate design decisions by providing students with a set of appropriate design elements to select, thus ignoring the learning objectives of mastering the object-oriented analysis and design. We are extending the ArgoUML, an UML tool, to a learning system, which enables students to design a class diagram using UML in free-form. The core component of this system is an assessment module, which evaluates a class diagram based on design guidelines. We apply the constraint-based approach to model a solution space for the given use case, which represents a design problem. This paper describes our assessment approach and the current stage of our system which is able to evaluate elements of a class diagram: classes, associations and the multiplicity of associations.

Keywords: intelligent tutoring system, UML, design patterns, constraint-based modeling.

INTRODUCTION

Class diagrams are an important modelling type of UML. They are used during the analysis phase through the design phase to the construction phase of a software development process. A class diagram can be interpreted correctly if we know for which phase it has been created. During the analysis phase, a class diagram is created to model the objects of an application domain. A class diagram created on this level is called a conceptual model and contains class names, class attributes and class associations. During the design phase, a design model is created by enriching the conceptual model with method definitions and type information. In the implementation phase, the design model is transformed into a specific programming language to create an implementation model. Our goal is to support students to learn object-oriented analysis and design by mastering the following skills according to guidelines described in (Larman, 1998)¹:

a) To create a conceptual model:

- Identify classes, which represent domain concepts using the Concept Category List and the Noun Phrase Identification principle.
- Determine associations and class multiplicity.
- Add attributes necessary to fulfill the specification requirements.

b) To create a design model:

- Find methods for a class,
- Add type information,
- Add navigability arrows to associations and detect dependency relationships.

We want to realise the learning objectives described above by developing a system, which presents students with a list of use cases and requests students to design a corresponding conceptual model or a design model. After having evaluated the student's class diagram, our system provides students with feedback. Through iterating the process of designing a class diagram and consulting system's feedback, students should be able to improve their class diagram until it meets the specification of the given use case.

In this paper, we present a constraint-based modeling (CBM) approach, which is applied to evaluate a class diagram based on design guidelines. First, we outline current work in this domain.

RELATED WORK

Three different attempts at developing an intelligent tutoring system (ITS) for UML modeling have been mentioned in the literature. The first one (Soller & Lesgold, 2000) developed a collaborative learning

¹ Some figures in this paper are taken from the same literature.

environment for object-oriented (OO) design problems using Object Modeling Technique (OMT), a precursor of UML. Machine learning techniques are applied to monitor group members' communication patterns and problem solving actions in order to identify situations in which students effectively share new knowledge with their peers while solving OO design problems. However, this system does not evaluate the OMT diagrams themselves.

The second attempt was conducted by Blank and colleagues (Blank et al., 2005) to support students learning OO analysis and design as problem-solving skills. The evaluation component of this system observes the student's entry (class name, each attribute, each method), and tries to match them with a corresponding part of the acceptable solution(s) and possible errors. Possible solutions and errors are coded by a human tutor in advance. If the student's input is not conformed to acceptable solutions and can be mapped to a library of possible errors, the evaluation component would interpret the student's input to be erroneous. The limitation of this system is that it anticipates the student's actions by defining possible correct actions as well as possible errors in advance, but the space of errors and of acceptable solutions might be unlimited.

The third attempt for an ITS for UML was introduced by Baghaei and Mitrovic (Baghaei & Mitrovic, 2005). According to this work, students' class diagrams are evaluated based on constraints, which model the syntax of the domain UML and the semantics of the given task. However, this system does not enable students to invent their own identifiers for classes, associations, methods and attributes. Instead, a name is selected from highlighted phrases of the problem text. The decision to use a noun from the problem text should be used as a class attribute or a class name is already made by this system. Such an interface design restricts concept identification and the assignment of self-explanatory names to simple cases. Furthermore, this system does tutor UML but not design strategies which are the most important learning issue. Thus, a major learning goal from the objectives mentioned above is not considered by this system..

We apply the CBM technique to develop a system to assess class diagrams submitted by students. The CBM approach was introduced by Ohlsson in (Ohlsson, 1992) and has been proven successfully in building many Intelligent Tutoring Systems (ITS) for SQL and database design (Mitrovic, 2001), natural language (Menzel, 2006), and has also been researched in the domain of teaching UML (Baraghei & Mitrovic, 2005), data structures (Warendorf & Tan 1997) and logic programming (Le & Menzel, 2005). Our system differs from the other ones by three points:

1. Class diagrams for a design task are created in a free form.
2. The system does not only tutor UML but also design guidelines.
3. The system focuses on examining task related requirements leaving the syntactic diagnosis to an UML design tool like ArgoUML (<http://argouml.tigris.org>).

THE FREE-FORM ASSESSMENT APPROACH

We provide students with a list of use cases and ArgoUML, an UML design tool, which enables students to create appropriate class diagrams. There is no cook book, which instructs how to construct a correct class diagram. However, there are books (Larman, 1998; Fowler, 2003; Jacobson et. al 2000), which recommend best practices for designing good class diagrams.

According to (Larman, 1998), classes and attributes are identified based on the Noun Phrase Identification principle and the Concept Category List. Nouns and noun phrases in a textual use case description are good candidates to be modelled as conceptual classes or attributes. Indications for classes, attributes and associations are described in details in the books concerned above.

In addition to using the Noun Phrase Identification principle to identify classes, we should apply design patterns to solve recurring problems. Certain solutions to design problems have been expressed as a set of principles. Patterns are named problem-solution formulas that codify exemplary design principles. At present, many object-oriented software designers know fundamental object-oriented design patterns, which are also the learning objectives our system should support. We refer to design patterns, the Noun Phrase Identification principle to identify classes and attributes, and other good design practices as design guidelines.

Our problem is to analyse the student's solution and to evaluate whether the class diagram elements (classes, attributes, methods, associations and the multiplicity of associations) meet the requirements of a given use case. To solve this problem, we need 1) to understand what diagram elements of the student's solution represent and 2) to determine whether the diagram elements of the student's solution adhere the specification of the given use case. The two steps assessment for class diagrams has been realized in a module which is integrated into ArgoUML.

The shortened use case **Buy Item** from (Larman, 1998) is used to illustrate the diagnosis approach.

***Buy Item:** This use case begins when a Customer arrives at a POST (point-of-sale terminal) checkout with items to purchase. The Cashier records the Universal Product Code (UPC) from each item. The POST determines the item price and adds the item information to the running sales transaction. The description and price of the current item are presented. On completion of item entry, the Cashier indicates to the POST that item entry is complete. The POST calculates and presents the sale total.*

The solution space for a given design task

The activity of designing a class diagram rests on the given use case under consideration. In principle, design decisions are based on:

1. Design view: conceptual model, design model or implementation model²,
2. Design guidelines for finding classes, attributes, associations, multiplicities for associations,
3. The context of the use case,
4. Individual justification.

Design views are not a part of the UML specification. The notion of design view helps us to interpret a class diagram correctly. The content of a conceptual model is different than of a design model and the content of the implementation model requires more information than the design model. If a class diagram is given to be assessed, we need to know from which view it should be evaluated. Therefore, it is important to declare an agreement in advance.

There are numerous design guidelines for class diagrams. A conceptual model or a design model cannot be assessed to be absolutely correct or wrong, but more or less useful according to the given use case. To assess the usefulness of a class diagram, design guidelines are the foundation. For example, under consideration of the use case above, one may have specified a *POSTNumber* attribute in the *Cashier* type (Figure 1). According to the design guidelines “Not Attributes as Foreign Keys”, this is undesirable because its purpose is to relate the *Cashier* to a *POST* object. The better way to express that a *Cashier* uses a *POST* is with an association, not with a foreign key attribute. We cannot say that the former design is wrong, but according to design guidelines it is less useful with respect to simplification and clarification.

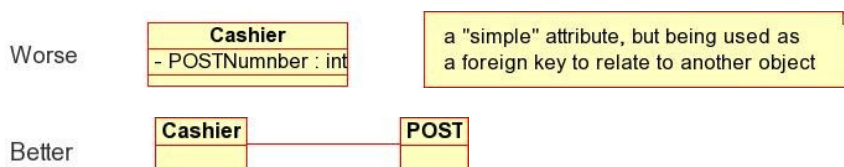


Figure 1: Design decision: relate a type using an attribute or an association.

The design decision for a class diagram is mainly derived from the context of the given use case. In the UML, the multiplicity value is context dependent. The example of *Person* and *Company* in the *Works-for* association from (Rumbaugh, 1991) indicates if a *Person* instance works for one or many *Company* instances. That is dependent on the context of the model; the tax department is interested in *many*; a union probably only *one*. For the *Buy Item* use case, one has to specify the multiplicity for the association *Records-sale-of* association between *SalesLineItem* and *Item*. Each line item might records a separate item sale, for example, one tofu package. However, it is also possible for a cashier to receive a group of like items, for example, six tofu packages, enter the UPC once, and then enter a quantity. Consequently, an individual *SalesLineItem* can be associated with more than one instance of an *Item* (Larman, 1998). As our use case does not state that the cashier is able to record the UPC from each item as well as to enter the quantity of the same item, we have to decide for the former design.

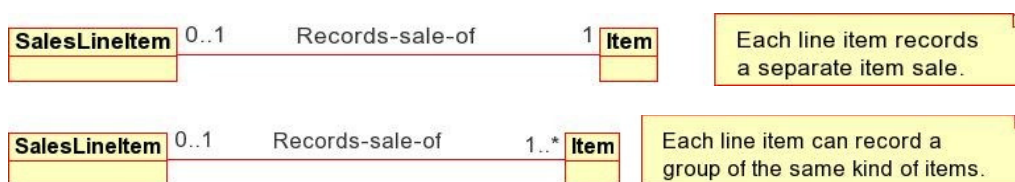


Figure 2: Design decision for the multiplicity of the association between *SalesLineItem* and *Item*.

The design of a class diagram depends not only on design guidelines, the requirements but also on the individual justification. For instance, there are two strategies to find associations for a set of concepts (Larman, 1998): 1) The knowledge of the relationship between concepts needs to be preserved for some duration can be modeled as an association (“need-to-know” association), 2) Associations derived from the *Common Associations List*³. We note that the ability to justify an association in terms of need-to-know is dependent on the requirements in the given use case. However, a strict need-to-know criterion for maintaining associations generates a minimal “information model” of what is needed to model the problem domain under consideration of the current requirements. Such models, which should play the role of a communication tool, do not convey a full understanding of the domain. For example, in Figure 3, which illustrates a conceptual model for the use case **Buy Item**, although on a strict need-to-know basis it might not be necessary to record *Sale Initiated-by Customer*, its absence leaves out an important aspect in understanding the domain, that a customer generates sales. Therefore,

² For our purposes, we just investigate conceptual model and design model.

³ The Common Associations List contains a list of associations which are often used for conceptual model.

it is recommended to find associations somewhere in the middle between a minimal need-to-know criteria and one which illustrates every conceivable relationship.

As a result, there is no single correct model. All models are approximations of the domain we attempt to understand. A good model captures the essential abstraction and information required to understand the domain in the context of the current use case. The space of good models for the given use case is therefore unpredictably open-ended and thus, the task of designing class diagrams using UML is an ill-defined domain.

The constraint-based model of a solution space

We employ the constraint-based approach and use the ideal diagram (Figure 3) to model the space of solutions, which meet the requirements of the given use case. A constraint consists of two parts: a relevance part and a satisfaction part. The first part identifies the problem state, for which a constraint is relevant. The latter examines whether these elements satisfy the conditions of a constraint. If a constraint is relevant to the student's diagram, then it must satisfy the constraint. For example, we specify a constraint to express a design guideline that designing two concepts and an association is better than designing an attribute for a concept (Figure 1) if the attribute is neither of a simple type (string, integer, boolean) nor of a pure data value type⁴ (or Data Types in UML terms).

Constraint 1:

IF the ideal diagram has a class X, a class Y, an association XY between X and Y, AND neither X nor Y represent a data type AND

The student's diagram has classes X', where X' is identical to X.

THEN the student's diagram ought to have a class Y' which is identical to Y and an association between X' and Y'.

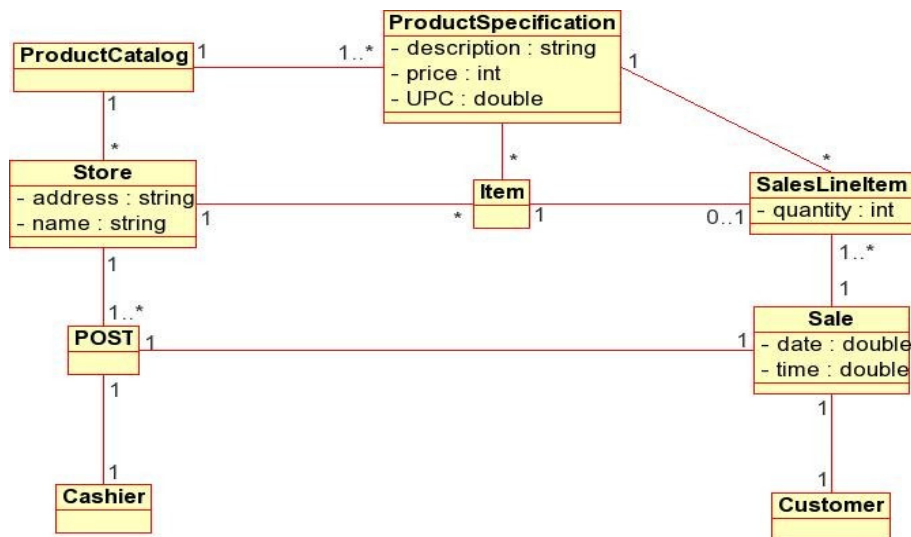


Figure 3: The conceptual model for the use case Buy Item

In addition to the existence of two classes, an association between them, Constraint 1 requires additional information about the function of the classes: none of the two classes are data types. We encode the function for each class of the ideal diagram explicitly: *domain concept*, *data type*, *generalization concept* and *specification concept*. The function of a class is *domain concept* if it is found in the given use case. Basically, one has to find concepts from the domain described in the use case and use the existing names in the territory and do not add things that are not there. A class has the function *data type*, if it is composed of separate sections (e.g. phone number, name of person) or there are operations usually associated with it, such as parsing or validation. A class has the function *generalization concept* if there are two concepts which share same attributes and we need a new class which represents the generalization of those two concepts. The *specification concept* is used for a class if there is a need to maintain the description of things and to reduce redundant information. For example, *ProductSpecification* is a specification concept required to describe many same items or products.

A constraint can be used to reflect the requirements of a use case. For example, the use case Buy Item requires that “The Cashier records the Universal Product Code (UPC) from each item.” For this purpose, we specify the following Constraint 2:

⁴ Attributes are pure data values if unique identity is not meaningful for them.

Constraint 2:

*IF the ideal diagram has a class X, a class Y, an association XY between X and Y, the multiplicity at X is M and the multiplicity at Y is N AND
the student's diagram has a class X' and a class Y', an association X'Y' between X' and Y' which are identical to classes X, Y and to the association XY
THEN the student's diagram ought to have an association X'Y' where the multiplicity for X' and Y' are M and N.*

A constraint can not only be specified to express a requirement or a design guideline but also can be used to express different possible solutions for a certain problem. For example, classes which represent pure data values may be shown in the attribute section of another concept. But since it is a non-primitive type, with its own attributes and associations, it may be illustrated as a concept in its own box (Figure 4). For this case we specify Constraint 3.

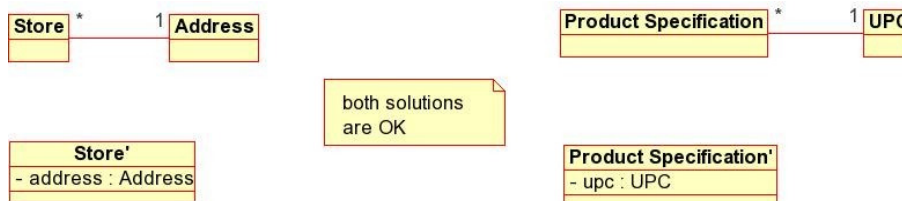


Figure 4: Pure data values can be shown in the attribute section or as their own concepts

Constraint 3:

*IF the ideal diagram has a class X, a class Y, an association XY between X and Y, where Y represents a data type AND
the student's diagram has a class X' which is identical to X
THEN the student's diagram ought to have a class Y' which is identical to Y and an association X'Y'
OR the class X' has an attribute of type Y' which is identical to Y.*

We enrich the constraint specification proposed by Ohlsson with more information: a penalty and a feedback, which are returned to students in case a constraint is violated. The penalty ranges from zero to ten, which represent the lowest severity and the highest severity of a constraint, respectively. The feedback can be used to mention the requirements of a use case, to describe design guidelines or to justify a design decision.

Beside constraints which are specified by the system engineer to model the domain of UML class diagrams, we enable exercise authors to specify constraints for individual justification. For instance, the exercise author might specify Constraint 4 which requires the existence of an association which expresses that a *Sale* is initiated by a *Customer*. This association can not be detected based on the need-to-know criterion. However, the exercise author might think that it is necessary to make the conceptual more useful and understandable, so he can specify this.

We specify for Constraint 4 a penalty of value five, because it represents a recommendation. Constraint 1, 2 and 3 should have a penalty of value ten because it represents a requirement. The penalty information is used to present feedback to students if constraints are violated. Feedback associated to violated constraints with higher penalty is shown first.

Constraint 4:

*IF the ideal diagram has classes Sale1, Customer1 AND an association between them AND
the student's diagram has classes Sale2, Customer2 which are identical to Sale1 and Customer1
THEN the student's diagram ought to have an association between Sale2 and Customer2*

Penalty: 5

Feedback: You need an association which expresses a Sale is initiated by the Customer.

We need three types of atomic constraints: class existence, association existence and multiplicity existence. Based on the penalty of the constraint, the existence of a class can be defined as optional (zero) or obligatory (ten). Similarly, the constraints for association existence and multiplicity existence ensure that a certain association or a multiplicity should exist. The constraint specifying the existence of an association can be coded as follows:

```

<constraint>
  <ae end1="SaleTransaction" end2="SalesLineItem" />
  <comment>An association between SaleTransaction and SalesLineItem is required.
</comment>
  <penalty>10</penalty>
</constraint>

```

Based on the atomic constraints, we can define complex constraints applying conjunction, disjunction and negation operations in order to specify relationships between elements of a class diagram. For instance, the following constraint requires that if an association between the class “Item” and the class “Store” exists, then the class “Item” and the class “Shop” should have a multiplicity “1” and “0_N”, respectively. The XML tag <and> conjoins two multiplicity existence requirements.

```

<constraint>
  <imply>
    <ae end1="Item" end2="Store" />
    <and>
      <me end1="Item" end2="Store" ort="end1" mult="0_N" />
      <me end1="Item" end2="Store" ort="end2" mult="1" />
    </and>
  </imply>
  <comment>A store can stock many items</comment>
  <penalty>10</penalty>
</constraint>

```

Constraints are evaluated as follows: for each constraint, the relevance part is evaluated first and if the student’s diagram conforms to the relevance part, the associated satisfaction part is evaluated too. If the satisfaction part is not satisfied, i.e. the constraint is violated; the corresponding feedback and a penalty are returned. For clarity, the list of feedback messages can be sorted based on the penalty when displayed to the student.

Identifying a Diagram’s Structure

Before the student’s diagram can be evaluated by means of constraints, the diagram’s structure needs to be identified. The identification process tries to map elements of the student’s diagram to elements of the ideal diagram as Figure 5 illustrates. First, we carry out the identification process based on the Noun Phrase Identification principle.

Applying the Noun Phrase Identification Principle

We assume that students find the candidate concepts by applying the Noun Phrase Identification principle and the Concept Category List related to the current requirements under consideration. Each class of the ideal diagram has a list of alias names, which are not used for other classes. A class of the student’s program is considered to be identified if either the name of the student’s class matches exactly, or it is an abbreviation or a misspelled variation of the name of an ideal class or one of its aliases.

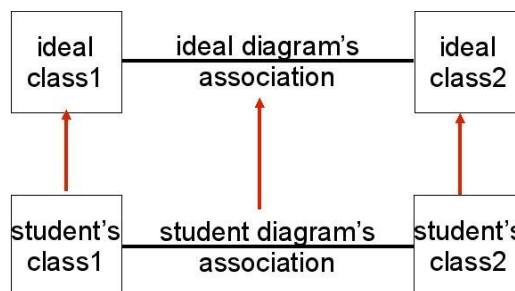


Figure 5: mapping between the student's diagram and the ideal diagram

The identification process iterates through all classes of the student’s diagram. If any student’s class can not be mapped, it is considered to be superfluous. Inversely, any class of the ideal diagram not paired with a class of the student’s diagram is considered to be missed by the student.

After having identified the classes in the student’s diagram, the identification process moves on to the level of class associations. An association of the student’s diagram is mapped to the one of the ideal diagram if the two classes at the ends of the student’s association correspond to the two classes at the ends of the association of the ideal diagram.

Combinatorial Matching

We continue the identification process to reduce the number of superfluous student's classes by combinatorial matching. An element from the set of superfluous student's classes is associated to a class of the ideal diagram where this class has not been associated to any student's class yet. After the class map is created, we derive the association map and evaluate constraints based on the class map and the association map. This process is continued until all mappings have been evaluated. The mapping which causes least penalty is taken to be the most plausible interpretation of the student's diagram. We clarify this combinatorial matching process by an example in Figure 6. Assuming, we have been able to map the class A' to the class A and the class B' to the class B based on the Noun Phrase Identification principle. Class X' could not be identified. We have to map the class X' to the set of classes X, Z and evaluate all constraints for each mapping. We assume that the penalty for each association existence constraint is of the same value. As result, we find two mappings: $Map1 = \{A-A', B-B', X-X'\}$ and $Map2 = \{A-A', B-B', Z-X'\}$. The mapping X-X' satisfies constraints, which require the association existence between class X and class A and the association existence between class X and class B. The mapping Z-X' satisfies the constraint, which requires the association existence between class Z and class B, but the association between Class X' and A' will be evaluated as superfluous. Thus, $Map1$ is hypothesized to be more plausible than $Map2$.

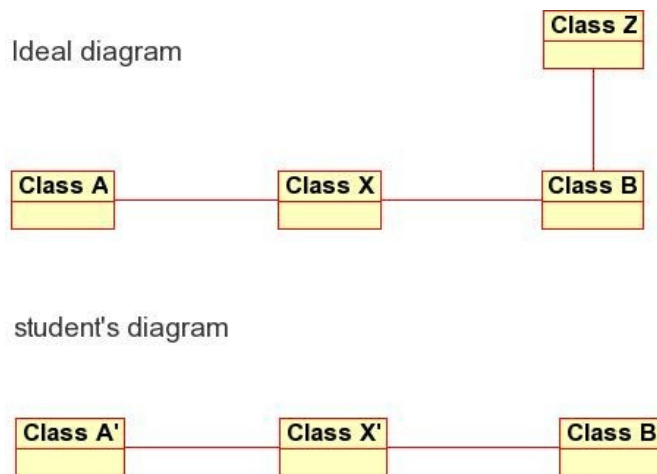


Figure 6: Combinatorial matching between a student's class and a class of the ideal diagram.

In case, some student's classes can not be identified, we can not evaluate them, because they seem to be not relevant in the solution space which is specified by constraints and the ideal diagram. We just remind the creator of these classes to consider the usefulness of them.

An Assessment Example

We demonstrate the educational capability of our system by assessing the student's diagram in Figure 7. First, the matching process is executed based on the ideal diagram. The system could not find a corresponding class in the ideal diagram for the class *Manager* in the student's diagram and returns following feedback: *The class Manager seems to be superfluous. Please, consider if you really need this class.*

Obviously, the system did not identify an appropriate class in the student's diagram for the class *POST* in the ideal diagram. But this will be evaluated by constraints. Assuming the system has only to evaluate four constraints described above.

Constraint 1 is relevant in two cases: 1) the class *Store* and the class *POST* and 2) the class *Cashier* and the class *POST* in the ideal diagram. However, Constraint 1 can not be satisfied in none of both cases because the student's diagram does not have a class which corresponds to the class *POST*. Thus, the system responds: *"According to design guidelines, the concept which represents a POST should be modeled as a class instead as an attribute because it is not a simple type."*

Constraint 2 is violated because according to the use case the association between the class *Item* and the class *SalesLineItem* should have a multiplicity value of 1 at the class *Item* and 0..1 at the class *SalesLineItem*. Our system returns following hint because Constraint 2 has been violated: *"Based on the use case specification, one instance of SalesLineItem can be associated with one Item."*

Constraint 3 is not violated because it is possible to model the concept *UPC* as a class of data type or as an attribute.

Constraint 4 is relevant and violated because the exercise author specified the requirement explicitly that an association between the class *Sale* and the class *Customer* should exist. A corresponding hint, which was

specified by the exercise author, will be forwarded to the student: “You need an association which expresses a Sale is initiated by the Customer.”

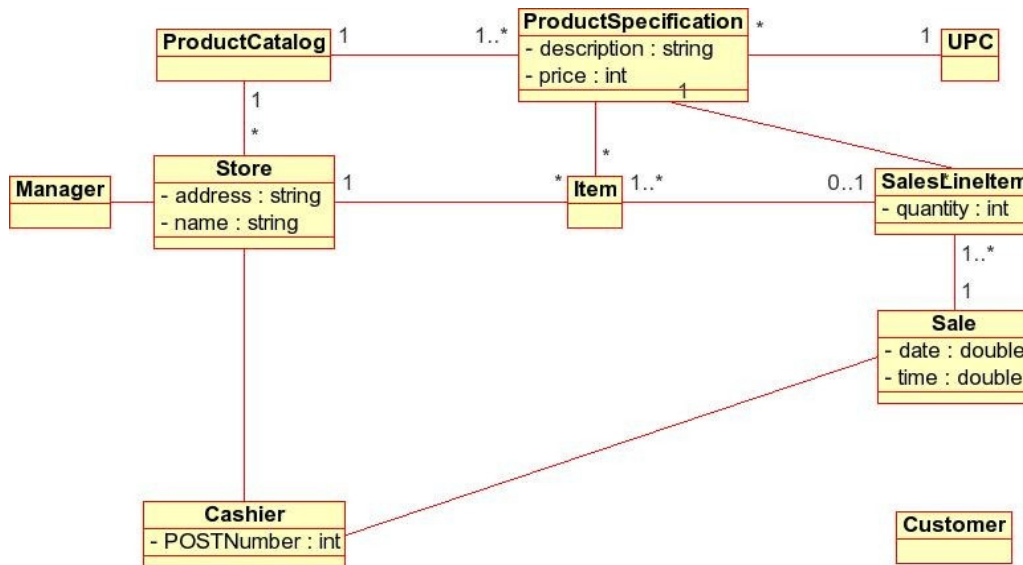


Figure 7 a student's diagram for the use case Buy Item

CONCLUSION AND FUTURE WORKS

Designing a class diagram using UML is a creative work. Thus, we provide students with a design tool like ArgoUML with which they can create class diagrams in a free-form. For a design problem in form of a use case, there are many solutions, which meet the requirements. To evaluate a solution in this domain, we apply the constraint-based approach and use an ideal diagram representing the requirements of the given use case. Constraints are not only used to specify requirements but also to describe the application of a design pattern. If the student's diagram violates a constraint, then it indicates that the student's diagram does not meet a requirement of the given use case or a design guideline has not been considered in the student's diagram. By enabling exercise authors specify constraints, the assessment can be enhanced with individual justification. Our assessment module, which has been integrated into the ArgoUML tool, supports students to develop their creativity and skill to design a class diagram and to master design patterns. At present, our assessment module is able to evaluate the following elements of a class diagram: classes, associations and the multiplicity of associations. The structure identification is currently based on the Noun Phrase Identification principle.

We plan to enrich the structure identification component with the combinatorial matching algorithm. Our goal is to extend our assessment module to be able to evaluate other elements of a class diagram: attributes, class methods, type information, and association navigation.

REFERENCES

ArgoUML homepage: <http://argouml.tigris.org>

Baghaei, N. and Mitrovic, A. (2005) *COLLECT-UML: Supporting individual and collaborative learning of UML class diagrams in a constraint-based tutor*. Accepted for presentation at KES, 2005.

Blank, G., Parvez, S., Wei, F., and Moritz, S. (2005) *A web-based ITS for OO design. Workshop on adaptive systems for web-based education tools and reusability*. 12th Int. Conference on AIED, Amsterdam.

Fowler, M. (2003) *UML Distilled. A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley.

Jacobson, I., Booch, G. and Rumbaugh, J. (2000) *The unified software development process*. Addison-Wesley.

Larman, C. (1998) *Applying UML and Patterns. An Introduction to Object-oriented Analysis and Design*. Prentice Hall PTR.

Le, N.T. and Menzel, W. *Constraint-based Error Diagnosis in Logic Programming*. In Proceedings of 13th International Conference on Computers in Education 2005.

- Menzel, W. (2006) *Constraint-based modeling and ambiguity*. In International Journal of Artificial Intelligence in Education, volume 16.
- Mitrovic, A. et al. (2001) *Constraint-based tutors: a success story*. In L. Monostori and J. Vancza, Proceeding of the 14th International Conference on Industrial Engineering Application of AI and Expert Systems, 931-940, Budapest.
- Ohlsson, S. (1992). *Constraint-based Student Modeling*. In International Journal of Artificial Intelligence in Education 3(4), 429-447.
- Rumbaugh, J. (1991) *Object-oriented modelling and design*. Englewood Cliffs, NJ: Prentice-Hall.
- Soller, A. and Lesgold, A. (2000) *Knowledge Acquisition for Adaptive Collaborative Learning Environments*. AAAI Fall Symposium: Learning How to Do Things.
- Warendorf, K. and Tan, C. (1997) *Constraint-based student modeling - a simpler way of revising student errors*. In Proceedings of ICICS, 2, 1083-1087.