Teaching Basic Class Diagram Notation with UMLGrader

Dr. Robert W. Hasker, Milwaukee School of Engineering

Rob is a professor in the software engineering program at Milwaukee School of Engineering where he teaches courses at all levels. He was recently at University of Wisconsin - Platteville, where he taught for 17 years and helped develop an undergraduate program in software engineering and an international master's program in computer science. In addition to academic experience, Rob has worked on a number of projects in industry ranging from avionics to billing. He holds a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign.

Dr. Yan Shi, University of Wisconsin - Platteville

Teaching Basic Class Diagram Notation with UMLGrader

Abstract

We discuss using UMLGrader as a tool for teaching class diagram notation in the Unified Modeling Language (UML). Given a diagram which is constructed to model a tightly constrained problem, the tool compares the diagram against a standard solution and provides feedback on missing elements and other errors. This supports using canned exercises to familiarize students with UML notation. We describe the tool and discuss its use in courses at the sophomore and junior levels. In each case, we report on the results of before and after tests, showing that there was significant improvement after using the tool.

Software engineers are expected to be able to model software systems with diagrams.[1, 14] However, experience shows that teaching students to use this notation is surprisingly difficult, at least in the case of the Unified Modeling Language (UML).[3, 8, 19] The traditional method[18, 19] for teaching diagram notation is to give students open-ended design problems and have the students use the notation to design solutions. The assumption is that students will gain skills in abstract problem solving while at the same time learning to use the notation.[7] The obvious approach is to increase the size of the problems over the course of the curriculum, asking students to model systems with just a few classes in introductory courses and progressing to very large systems in advanced courses. Our experience calls this assumption into question, and we start with an example that illustrates this.

At University of Wisconsin-Platteville, where much of this work was done, students are introduced to UML class diagram notation in the introductory programming sequence with further exposure in the first software engineering course. Upper-divisional courses then expand on this, covering additional diagram types as well as design patterns.[9] In particular, one of these courses devotes considerable time on the command pattern. For several years, the first exam included a modeling question in which students applied the command pattern to a simple problem, and only a few students were able to give a satisfactory answer. In 2011, a simpler modeling problem was added to evaluate student's basic skills:

> Draw a class diagram modeling the following:
>
> - Snow removal drivers have routes and are assigned trucks with which they clear those routes. Each route is the responsibility of one driver.
> - Trucks have numbers. There are two types of trucks: dump trucks and snow plows. For dump trucks, there is a name of the type of equipment attached.
> - A route is a sequence of street segments to traverse.
> - A street has a name and is made up of a sequence of segments, where each segment is defined by the crossing streets that start and end that segment. If the street is a dead-end, the crossing street is NULL.
>
> Show classes, associations, multiplicities, attributes, operations, and generalizations.

The intended response is given in in Figure 1.[1] Since the notation had been introduced in earlier

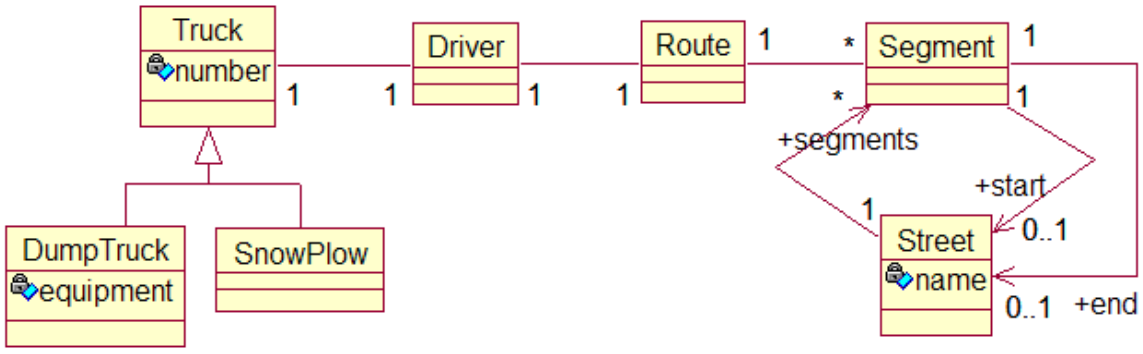

Figure 1: Snow plow class diagram

courses and reviewed within the week before the exam, it was assumed few students would find the question challenging. Furthermore, approximately half of the students had previously taken a course called "Object-Oriented Analysis and Design" which devotes nearly a third of a semester to UML notation. However, students performed rather poorly on this question. The question was graded on a four point scale, and just 7 out of 37 received a score of 3 or higher while 20 students received a score less than 2. The students had little difficulty identifying classes—only 8 failed to identify one of the classes, and none failed to identify two—but many failed to name association roles and there were many cases in which students reversed multiplicities or used the wrong notation for generalization. The students were clearly not adequately prepared for the problem.

This experience and others leads us to question the basic assumption identified in the introduction: that having students solve open-ended design problems is an effective way to teach UML diagram notation. One challenge is that the open-ended problems must be designed to exercise all of the intended notation, and if a student fails to understand where to apply a particular notation then the student misses on the opportunity to learn to use that notation. Another challenge is that grading these types of problems is very time-consuming, making it unlikely that students will receive quick feedback on how they used the notation. This paper presents an alternative approach based on teaching the notation separately from teaching its application.

Teaching notation could be done in the traditional manner: assigning exercises to create diagrams that are hand-graded. While grading constrained problems is easier than open-ended ones, the turnaround time would still be a limiting factor. We developed a tool, UMLGrader, which automates the process of comparing diagrams to give more immediate feedback. Students are given a constrained problem, use a tool such as IBM Rhapsody to draw up a solution, and submit it to UMLGrader. UMLGrader responds with a list of differences between the student's solution and the expected solution. This feedback is designed to strike a balance between guiding the students to acceptable solutions and simply telling them what to change. The result is that students can be drilled on UML notation separately from solving open-ended problems.

This paper presents evidence that this approach is effective: that using UMLGrader to learn

[1]The problem statement is ambiguous with respect to some multiplicities, and these were ignored during grading.

diagram notation improves performance on exam questions. In addition, we present data on the types of errors made by students. The ultimate question—whether using UMLGrader improves performance on more general design problems—is necessarily left as future work. However, we believe that our results show that UMLGrader is at least useful for teaching core notation.

Other approaches to teaching UML notation can certainly be considered. One is to use tools specialized for student use.[2, 16, 20] Because these tools are specialized, they can be more targeted at meeting curricular goals. However, none of the existing tools have been evaluated for teaching basic notation, and none provide support for model comparison. There are other, non-modeling tools which can detect inconsistencies between different modeling diagrams. Some of these are intended for educational use[4, 16] while others for professional use.[5, 6, 13, 15] However, each of these require constructing multiple, interrelated diagrams documenting both the static and dynamic elements of the system. Their use depends on understanding the full range of diagram types and so are not practical for introductory students.

UMLGrader Details

UMLGrader is based on UMLint,[17, 12, 11] a tool which provides feedback on UML diagrams students create for open-ended problems. UMLint flags a number of common problems in use case and class diagrams including failing to follow naming conventions, reversed generalizations, improper attributes and operations, and errors in associations.[12] The goal of UMLint is to not identify every possible error, but to catch common errors that can be detected mechanically and so encourage students to look more closely at their solutions. UMLGrader, on the other hand, is designed to critique solutions in cases where the problem is tightly constrained enough that a student can be expected to match a target diagram.

UMLGrader uses an iterative approach to matching diagrams. It starts by matching classes and then proceeds to matching associations, attributes, and methods. This reflects the noun identification technique for constructing models. More specifically, it applies the following rules:

- Classes are matched by name. The entire name must be matched exactly, though spaces, underscores, and capitalization are ignored. Requiring an exact name match helps catch the common error of using a plural noun when the class represents a singular item. To allow some variation in naming, the instructor can specify alternative names in the documentation for the class.
- Attributes, operations, and roles are also matched by name, but in these cases a substring match is sufficient. That is, a name in the sample solution is considered to be found if it occurs anywhere in the name given by the student. For instance, both ‘item_count’ and ‘count_of_items’ would match an expected attribute called ‘count’. As for class names, spaces, underscores, and capitalization are ignored during matching. As for class names, alternatives can be specified in element comments.
- When available, associations are matched based on role names. If role names are unavailable—that is, missing in either the target or student solution—then associations are matched by comparing multiplicities.
- If an association in the sample solution is missing either a role name or a multiplicity, role

names and multiplicities in the student's solution are ignored. Specifying a role when it is not required is not viewed as an error.

Placement of elements on the diagram is not considered since the focus is on notation, not organization skills. It is also not clear that rubrics for grading placement could be automated.

Ignoring spaces, underscores, and capitalization when matching names could result in students failing to learn applicable standards. To address this, UMLGrader applies the UMLint checks in addition to the above matching rules. In this way, a student can get credit for providing a required model element, but obtain additional error messages for failing to follow standards. More specifically, the systems checks for such issues as class names that start with lower case letters, methods or attribute names starting with upper case letters, misspelled words, and invalid uses of composition.[10] UMLint and UMLGrader both allow instructors to disable checks that are not applicable to a particular course or curriculum.

UMLGrader is delivered as a web service. Students upload their diagram, state which problem is being solved, and UMLGrader compares it against the specified solution. For example, suppose the diagram in Figure 2 is submitted as a solution to match against Figure 1.

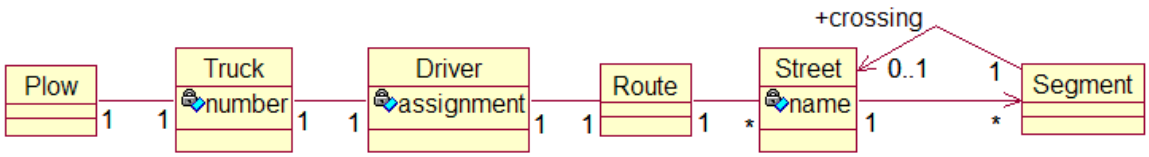

Figure 2: Snow plow solution with errors

The response is

```
Class Plow should not be in the diagram
Class Route should be associated with a different class than Street
Class Segment is missing 2 associations
Class Street should be associated with a different class than Route
Class Truck has an unexpected association with Plow
Incorrect role name for role crossing from Street
Missing classes: DumpTruck, SnowPlow
Unexpected attributes in class Driver
```

Note extra and missing classes are specified by name, and class names are used to specify associations with errors. However, errors in attributes, operations, and roles are described more vaguely to encourage the student to check the problem statement more closely when there are issues.

Evaluation

UMLGrader has been used for several semesters at UW-Platteville. The students answering the above exam question on plowing snow were in a junior-level course, Intermediate Software Engineering, which has two prerequisites: data structures and Introduction to Software

Engineering. The course covers a number of different topics, but the greatest amount of time is spent on user-centered design and software process. The course was taught in two sections with 37 students in total. While the model curriculum places the course in the junior year, there are a number of sophomores and juniors as well: 38% of the students were sophomores, 38% were juniors, and 24% were seniors.

As discussed in the introduction, the students had difficulty drawing a diagram for the snow plow question on the first exam. After this exam, an implementation of UMLGrader was developed and students were given a closed lab session using it. This lab included two problems. The first problem was to reproduce a class diagram involving six classes. Its primary purpose was to teach the students how to use the tool. The second problem mirrored the exam question, in which students were to model an electronic book reader with a display and collections, books, articles, and pages. Full details are given in the first report on UMLGrader.[10] The expected answer is in Figure 3. This lab comprised the only review of the material between the first and second exams.

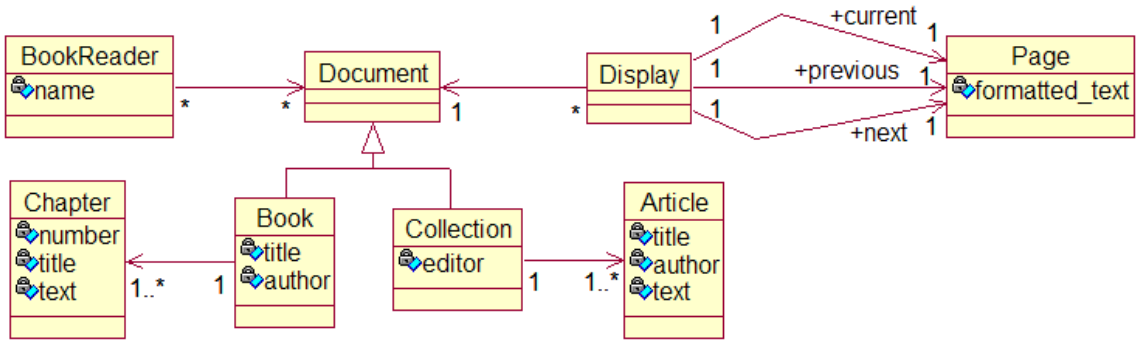

Figure 3: Expected book reader class diagram

The following problem was then given on the next exam:

> In a family pet office, each veterinarian handles two kinds of pets: dogs and cats. Each dog or cat has a name, a height and a weight. Dogs have a breed. For each pet there is a single owner, but one owner can have multiple pets. Owners have names and phone numbers. At any one time, certain pets have appointments, so some pets have no appointment, but others have a single appointment. An appointment is for a specific date and time. Each veterinarian also has multiple assistants who have names and identification numbers, and one assistant is assigned to each appointment with a second assistant assigned as backup if needed. Assistants work for just one veterinarian.
> Draw a class diagram capturing attributes, operations, associations, multiplicities, and generalization.

The expected diagram was given in Figure 4.

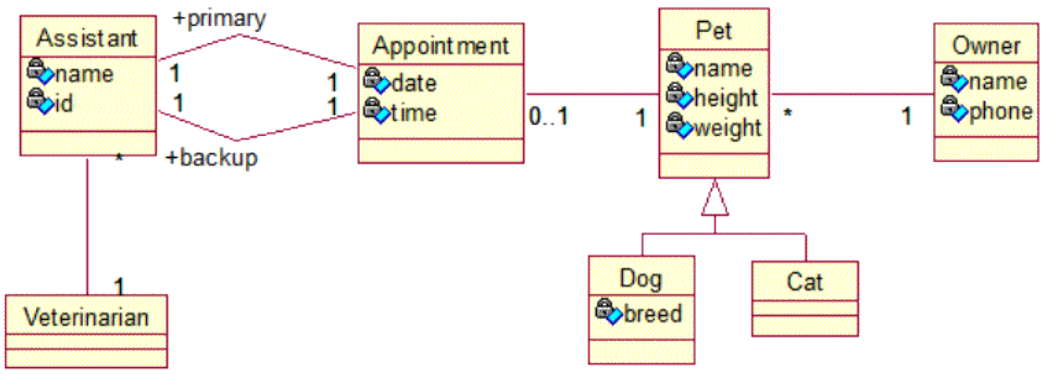

Figure 4: Expected pet office class diagram

Table 1 gives a detailed comparison of the scores from both exams. This shows a dramatic shift in

Table 1: Counts of students by score

Score	Exam 1 Count	Exam 2 Count
4	1	3
3.5	1	17
3	5	10
2.5	4	5
2	6	2
1.5	13	0
1	7	0
Ave	1.92	3.19
σ	0.78	0.50

performance. It is not clear the shift was solely due to using UMLGrader; it is possible simply seeing a correct answer during the exam 1 review was sufficient to improve the performance on exam 2. But the success of using the tool in the intermediate level class encouraged its use in lower level classes.

A second evaluation was done in spring 2013, this time in the Introduction to Software Engineering course that is a prerequisite to the intermediate course. This course has a second-semester programming course as a co-requisite, and is typically taken by sophomores in the software engineering curriculum. It is also required for certain emphases in the computer science major, and those students often take the course in their junior year. The instructor for this course was different than the instructor for the more advanced course discussed earlier in the paper, improving the generality of the results.

After a lecture on class diagrams, students were given a quiz in which they were asked to draw a class diagram based on a course registration system:

1. Each user has a unique username and password. A user also has his/her full name and email stored in the system.
2. The user should log in the system before using it, and log out after using it.
3. There are two types of users: instructors and students.
4. Each instructor has a department. Instructors offer courses.
5. Each student has an ID and major. A student can sign up and drop courses.
6. A course has its title, course ID and prerequisites. A course may have multiple sections. Each section is assigned a room and capacity.
7. Multiple instructors can teach the same course.
8. There must be at least one student taking a course.

After the quiz, students were given a lab exercise in which they constructed a class diagram, submitted it to UMLGrader, used the results to improve their solution, and repeated until no errors were reported. In this exercise, they modeled a library system with a catalog, books, DVDs, and patrons. The chart in Figure 5 categorizes the errors encountered by the students during the lab. This chart shows the percentage of students who encountered at least one instance of each type of

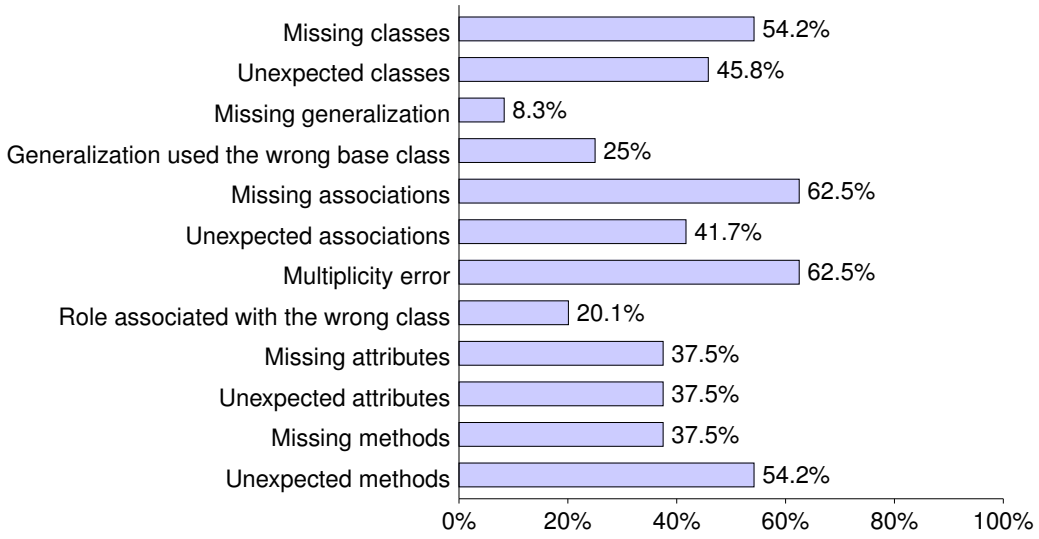

Figure 5: Percentage of unique errors by type among first submissions

error in their first submission across the 24 unique lab solutions submitted.[2] No one type of error dominated, with most types being encountered by at least a third of the students.

[2] All students submitted lab solutions; apparently two students worked together more closely than desired.

The lab was then followed by an exam which contained a problem asking students to draw another class diagram. There was no additional instruction on class diagrams during lecture, so the lab exercise was their only formal opportunity to reinforce the notation. On the exam, the students were asked to draw a class diagram for a property management system:

> Draw a class diagram modeling the following:
>
> 1. A property has a specific location and the date when it is added to the system. A location is composed of street address, city, state, and zip code.
> 2. There is one owner per each property. The system should keep a record of the owner's name, phone number and email address. An owner should have at least one property.
> 3. There are two types of properties: for rent and for sale. A rental property should list the monthly rent and whether it is vacant. A property for sale should list its sale price and property tax.
> 4. Each agent has a unique ID in the system. An agent can list multiple properties for sale. She will process the closing if a sale is made. An agent may also manage multiple rental properties. She should collect rents every month.

Both the quiz and exam question were graded on a 10-point scale. The primary statistics are summarized in Table 2, and a comparisons of the types of errors made is summarized in Figure 6. For each, the bar indicates the percentage of students who lost points in the specified category, with the top bar representing the results from the quiz and the bottom from the exam. Student's

Table 2: Quiz and exam scores in Introduction to Software Engineering.

	Count	**Mean**	**Standard Deviation**
Quiz	25	6.46	1.88
Exam	25	8.30	1.20

t-test shows that the 1.84 average improvement in scores is significant with a t value of 5.68 and a p value of $4 \cdot 10^{-6}$ (with 23 degrees of freedom). Figure 6 shows students had fewer errors in most areas, but did not improve on associations and performed worse at identifying the appropriate navigation. UMLGrader does not give feedback on navigation, so it is not surprising that there were areas in this area. It is not clear why there was no improvement in identifying associations; this could mean that additional practice is needed in this area in the lab problems or that UMLGrader fails to provide the appropriate feedback on association errors. Further investigation of this issue is needed.

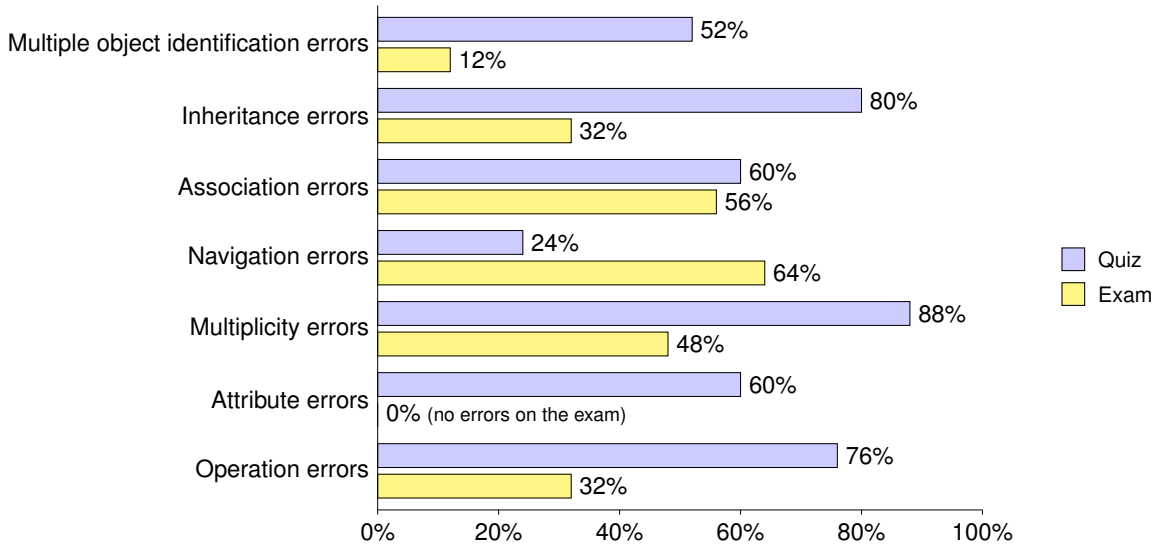

Figure 6: Quiz and exam errors categorized by type

Conclusion

Since the introductory and intermediate software engineering courses were taught by different instructors, there is some evidence that UMLGrader can be useful in teaching UML class diagram notation. However, there is an obvious flaw in this study: comparing results to teaching the same material without UMLGrader. Anecdotally, the fact that the Intermediate Software Engineering students did so poorly on the snow plow example (the example that motivated this work) suggests that traditional methods were not working well, at least at UW-Platteville. An obvious future study would be to evaluate performance on similar problems in introductory courses which do not use UMLGrader. It would also be useful to re-evaluate students in upper-divisional courses now that students are using the tool in the introductory course on a regular basis.

Additional work is being done on the tool. It currently supports diagrams created in IBM Rational Rose and IBM Rhapsody. Work is underway to provide support for Enterprise Architect. In addition, we plan to add support for comparing state diagrams and sequence diagrams.

The tool is available to instructors at other institutions via the web. Eventually, it should be possible for other instructors to set up assignments using the website, but currently assignments must be sent to us by email. Contact the first author at http://member.acm.org/~hasker for help in setting up an assignment.

References

[1] A. Abran, J. W. Moore, P. Bourque, and R. Dupuis, editors. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004.

[2] M. Auer, T. Tschurtschenthaler, and S. Biffl. A flyweight UML modelling tool for software development in heterogeneous environments. In *29th EUROMICRO Conference 2003, New Waves in System Architecture*, pages 267–272, Belek-Antalya, Turkey, Sept. 2003. IEEE Computer Society.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[4] W. Coelho and G. Murphy. ClassCompass: A software design mentoring system. *ACM Journal on Educational Resources in Computing*, 7(1):Article 2, Mar. 2007.

[5] C. R. B. de Souza, H. L. R. Oliveira, C. R. P. da Rocha, K. M. Gonçalves, and D. F. Redmiles. Using critiquing systems for inconsistency detection in software engineering models. In *SEKE*, pages 196–203, 2003.

[6] A. Egyed. UML/Analyzer: A tool for the instant consistency checking of UML models. In *Proceedings of the 29th International Conference on Software Engineering*, pages 793–796. IEEE Computer Society, 2007.

[7] G. Engels, J. H. Hausmann, M. Lohmann, and S. Sauer. Teaching UML is teaching software engineering is teaching abstraction. In J. M. Bruel, editor, *MoDELS 2005 Workshops*, volume 3844 of *LNCS*, pages 306–319, 2006.

[8] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 3rd edition, 2004.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] R. W. Hasker. UMLGrader: An automated class diagram grader. *The Journal of Computing Sciences in Colleges*, 27(1):47–54, Oct. 2011.

[11] R. W. Hasker, A. Rosene, and J. Reid. Experiences with a UML diagram critique tool. In *Midwest Instruction and Computing Symposium*, Cedar Falls, Iowa, April 2012.

[12] R. W. Hasker and M. Rowe. UMLint: Identifying defects in UML diagrams. In *118th Annual Conference of the American Society for Engineering Education*, Vancouver, BC, Canada, 2011.

[13] C. F. J. Lange. Improving the quality of UML models in practice. In *Proceedings of the 28th International Conference on Software Engineering*, pages 993–996. ACM Press, 2006.

[14] The Joint Task Force on Computing Curricula: Association for Computing Machinery (ACM)/IEEE Computer Society. Computer science curricula 2013. http://www.acm.org/education/CS2013-final-report.pdf, Dec. 2013.

[15] Z. Pap, I. Majzik, A. Pataricza, and A. Szegi. Completeness and consistency analysis of UML statechart specifications. In *Proceedings IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, pages 83–90, 2001.

[16] E. Ramollari and D. Dranidis. StudentUML: An educational tool supporting object-oriented analysis and design. In *Proceedings of the 11th Panhellenic Conference on Informatics*, 2007.

[17] M. Rowe and R. W. Hasker. The characterization and identification of object-oriented model defects. In *41st Midwest Instruction and Computing Symposium*, pages 178–192, La Crosse, Wisconsin, 2008.

[18] I. Sommerville. *Software Engineering*. Addison-Wesley, 8th edition, 2006.

[19] P. Stevens and R. Pooley. *Using UML: Software Engineering with Objects and Components, Updated Edition*. Addison-Wesley, 2000.

[20] S. A. Turner, M. A. Pérez-Quiñones, and S. H. Edwards. minimUML: A minimalist approach to UML diagramming for early computer science education. *ACM Journal of Educational Resources in Computing*, 5(4), Dec. 2005.