
AC 2011-1795: UMLINT: IDENTIFYING DEFECTS IN UML DIAGRAMS

Robert W. Hasker, University of Wisconsin-Platteville

Rob has taught at University of Wisconsin-Platteville for fifteen years where he has been a key developer of the software engineering program since its inception. He also coordinates 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 cellular billing. He holds a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign.

Mike Rowe, University of Wisconsin-Platteville

Mike has taught at the University of Wisconsin-Platteville for nine years and is a professor of Software Engineering and Computer Science. Prior to teaching he worked for 25 years in industry as a software engineer and program manager mostly in the Dallas-Fort Worth region. He earned a Ph.D. from the University of North Texas in Computer Science, a Ph.D. in Physiological Psychology from the University of North Dakota, and an MBA from Western Michigan University.

UMLint: Identifying Defects in UML Diagrams

Abstract

We present UMLint, an automated tool for detecting defects in UML diagrams. This tool is designed to improve object-oriented models developed by students. Standard tools such as IBM Rational Rose provide little feedback on model quality, so students must rely on feedback from instructors. Often there is a significant delay between completing a diagram and getting this feedback, resulting in missed learning opportunities. UMLint addresses this issue by identifying common defects, hopefully encouraging students to look more deeply for other defects. UMLint is available as a web service to allow use by both students and the community at large. This paper presents the checks made by UMLint, suggests possible future directions, and invites discussion about what standards should be expected of student-created models.

1 Introduction

Teaching students to use the Unified Modeling Language (UML)^{1, 6, 21} is challenging for many reasons. Among these is that students obtain little direct feedback on model quality from existing tools. This allows them to generate syntactically correct but semantically challenged models, such as reversing multiplicities or the direction of generalization arrows. In addition, there is often a long delay between model creation and critique. The typical classroom model is that a student works (possibly in a group) on an assignment for several days with little feedback, submits it on a due date, and then waits several more days for a graded response. The lack of immediate feedback weakens the learning experience. Further motivation comes from observing students making the same errors year after year.

A first step in improving object-oriented model quality is to use a specialized tool such as IBM Rational Rose. More general drawing tools such as Microsoft Visio can be used, but these provide few checks on diagrams. For example, Visio allows generalization arrows and associations to appear on diagrams without source or destination objects. In this sense, Rose already provides more immediate feedback.

However, Rose is not sufficient to ensure models are valid. UML has few syntactic constraints, so a model might be legal but inconsistent. This is not the fault of Rose. Tools such as Rose are designed for industrial use, so they must provide maximal flexibility to accommodate a wide variety of development processes. Furthermore, UML in industry is a tool, not an end in itself. Practices that might be discouraged in the classroom might be very appropriate for professional projects where the goal is to get the job done. This, combined with the fact that instruction on UML is somewhat inconsistent between institutions and instructors, means that professional UML tools must support a wide range of processes and standards. As a result, professional tools place few restrictions on generated models, resulting in little feedback to students with respect to standards or best practices.

This paper introduces UMLint, a tool to identify issues on demand in student-generated models. In the style of the C programming utility `lint`,¹⁰ the goal is to identify common mistakes rather than to ensure perfect solutions. Ultimately, correctness in a model will be determined by a more careful review, either by the students or a grader. The goal of UMLint is to provide quick, anytime feedback on models so that students can eliminate common mistakes and hopefully examine them closely for more significant issues.

UMLint was developed to support a variety of courses at University of Wisconsin-Platteville. As part of the software engineering major, students are introduced to UML diagrams in courses at the freshman/sophomore level, provided more in-depth training on UML diagrams at the sophomore/junior level, and use UML on large projects at the senior level. Students use IBM Rational Rose throughout, not because it is necessarily the best product available but because it is certainly adequate and IBM Rational products are commonly used in industry.

The curricula covers all major UML diagram types including use case, class, sequence, collaboration, and state diagrams. Use case and class diagrams are introduced at the freshman/sophomore level, with other diagrams being introduced in later courses. There is also a progression with respect to detail. The first use case and class diagrams contain only simple associations and major attributes and operations. Later courses introduce additional detail, and in some courses students generate full, compilable interfaces from their models. In all cases, diagrams are used to analyze and design the system in preparation for implementation, as opposed to documenting systems after the fact. Thus a major design goal of UMLint is to identify common issues across many different diagram types with varying levels of detail.

It is clear that the checks made by UMLint cannot be universal. Different instructors and institutions have different goals for UML models, and so what may be a defect for one instructor may not be a defect for another. As UMLint is used by additional instructors, we expect it will be extended to identify new types of defects and provide options to control which specific defects are identified. Similarly, students are expected to make their own decisions as to whether an identified issue is a true error. Practices that may be discouraged in the general case may be appropriate for specific projects, so it may be more appropriate to explain why a model was not changed than to change it.

This paper presents the checks currently made by UMLint. Many of these defects were identified in a previous paper,¹⁹ but UMLint introduces additional checks. This paper also presents our experiences with using UMLint in courses over the past year. Finally, the paper discusses some of its shortcomings and directions for future work.

2 Related Work

Several tools are available which perform syntactic checks on UML diagrams; for example, MagicDraw,¹⁴ UModel,²³ and Visual Paradigm for UML.²⁴ In addition to other services for the modeler, these tools identify defects that would lead to errors in code generated from

diagrams. Whether these provide additional checks that would be relevant to student-created diagrams is not clear from the literature.

There are a number of projects to identify and classify defects in UML models created by professionals.^{4, 5, 13, 17} While these projects influence UMLint, the fact that they are targeted towards professionals dramatically changes the problem. Students are simultaneously learning UML along with basic issues about modeling in general. In addition, practices that are errors for students might be acceptable for professionals and vice versa.

Another trend, again geared towards professionals, is to use formal specification languages as a tool to validate UML models.^{7, 12, 15, 11} It would be an interesting project to determine whether the types of defects identified by these methods would correspond to the types of errors commonly made by students. In any case, formal specifications often require a more rigorous framework than is practical for undergraduate students. Closely related to this is the large body of work on UML Model checking.^{e.g. 8, 18} These are typically focused on identifying inconsistencies just within behavioral models.

Existing work on defects in student models mainly focuses on errors made by students in introductory programming courses.^{22, 20} In our curriculum, CS1 is a prerequisite for all classes discussing UML, and CS2 is a prerequisite for classes in which students create detailed models. In contrast, the ClassCompass² system provides support for more advanced students. It includes automated checks, but the checks are limited. A more important feature of ClassCompass is support for reviews by other students and instructors. The intent is that these reviews would provide the primary feedback to the modeler.

3 Use case model defects identified by UMLint

We start with use case models because defects in these tend to be more obvious. Consider the use case diagram in Figure 1 written for an automated pet feeding station. This diagram

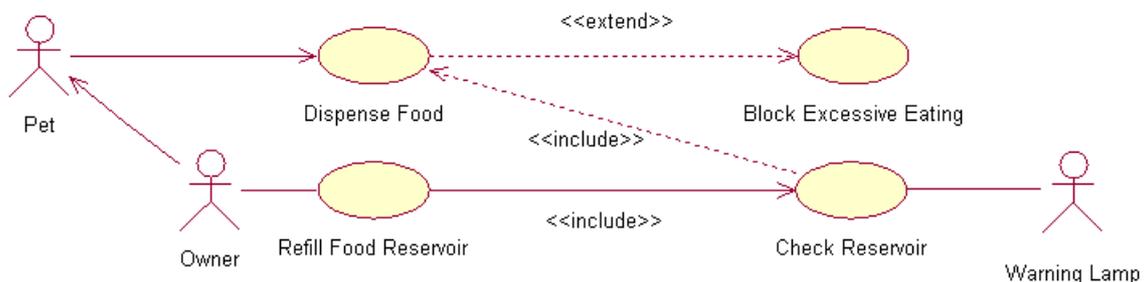


Figure 1: Use case model with defects

contains a number of issues:

- The association between Pet and Owner should be documented in a class model instead.
- The association between Pet and Dispense Food should be undirected.
- One of the <<include>> dependencies is drawn using a regular association.
- The <<extend>> dependency is backwards.

UMLint analyzes the model file produced by Rose³ and identifies each of these defects. The first three are apparent based on a simple inspection of the model file. Rose records the Pet and Owner objects as

```
(object Class "Pet"
  quid          "4D2CC780008D"
  stereotype    "Actor")
(object Class "Owner"
  quid          "4D2CC78B039A"
  stereotype    "Actor")
```

and the association between the two as

```
(object Association "$UNNAMED$3"
  quid          "4D2CC8CD039A"
  roles        (list role_list
    (object Role "$UNNAMED$4"
      quid          "4D2CC8CE03AA"
      supplier      "Use Case View::Owner"
      quidu         "4D2CC78B039A"
      is_navigable  TRUE)
    (object Role "$UNNAMED$5"
      quid          "4D2CC8CE03AC"
      supplier      "Use Case View::Pet"
      quidu         "4D2CC780008D" )))
```

Additional information captures that the association is part of the use case model. Noting this is an association between two actors is straightforward. The other three cases use similar information.

Detecting the backwards dependency is trickier but important given that we find such reversals are among the most common errors. To detect these defects, we assume every use case must involve at least one actor at least indirectly. Since <<extend>> dependencies should point from the exception to the normal case, Block Excessive Eating apparently has no associated actors. This leads to the conclusion that the <<extend>> is backwards. UMLint uses the same logic to check for backwards <<include>> dependencies, requiring each to point towards the

service being provided. But while checking for connections to actors works for Block Excessive Eating, it does not work for Check Reservoir because of its association with Warning Lamp. Complete automation of this check would require extensive domain knowledge.

The diagram in Figure 1 illustrates how UMLint can play a role in helping students learn UML. None of the above defects were detected by Rose, and the absence of negative feedback can lead students to conclude their model is acceptable. As a result, students may learn the wrong rules for use case diagrams. However, it is not clear Rose is at fault in these cases. Professionals may have good reason to break these “rules,” such as supporting legacy modeling conventions on very large projects. The designers of Rose are justified in being reluctant to classify particular constructs as errors. UMLint bridges the gap between a professional tool and the needs of academia by enforcing stricter standards.

Figure 1 also contains a more subtle error: the space in the actor name “Warning Lamp.” It is common for actors to become classes in models, so spaces in names can result in generating code that will not compile. Resolving this in just the code creates a naming conflict. To encourage students to develop the habit of using legal identifiers throughout the model, UMLint warns users of spaces and other illegal identifier characters. Reserved words are also flagged. The current rules are specific to Java and C++, but support for other languages is planned.

In addition to the above checks, UMLint flags actors and use cases with no documentation. IBM Rational Rose allows every element to be documented using free-form text, and a strong case can be made that it is more appropriate to document project elements at the model level than just in code. This is especially true when generating code from the model. Checking for missing documentation is tedious. UMLint flags elements with missing documentation. This does not prevent meaningless documentation, and tools to enforce good quality documentation may need to be developed.

Additional use case model defects identified by UMLint include adding `<<include>>` and `<<extend>>` stereotypes on regular associations, including multiplicities on associations, and missing `<<include>>` and `<<extend>>` stereotypes on dependencies.

4 Class model defects identified by UMLint

Figure 2 illustrates several class model defects identified by UMLint:

- The generalization arrows between Pet, Dog, and Cat are reversed. While some languages support multiple inheritance, it is rarely needed for class models created by students. On the other hand, reversing the direction of the arrow is reasonably common. This check is limited to cases involving at least three classes; checking for reversed inheritance with two classes would require domain knowledge.
- The composition relationships between Bed, FoodAndWaterDish, and Pet indicates that

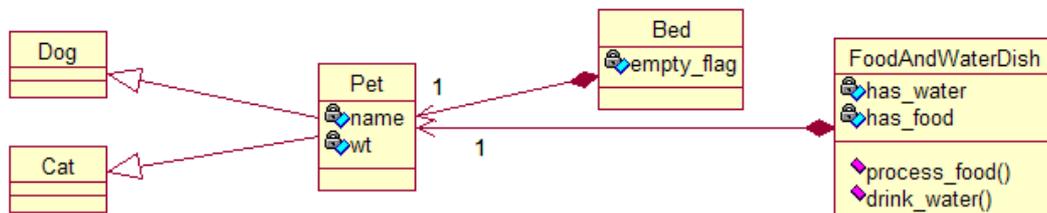


Figure 2: Class model with defects

Pet is “owned” by two classes. This could indicate either an improper use of composition or a reversed association.

- Non-dictionary words such as “wt” in Pet often either indicate simple misspellings or unnecessary abbreviations. Such identifiers are flagged to improve readability.*
- Nondescriptive words such as “flag” and “process” (in Bed::empty_flag and FoodAndWaterDish::process_food) are flagged when they appear in attribute and operation names to encourage students to find more descriptive names.
- The word “and” in FoodAndWaterDish is interpreted as an indication of a class with low cohesion. UMLint suggests either introducing additional classes, such as separating Food and WaterDish, or renaming the class to cover its multiple roles more naturally.

UMLint also issues warnings for objects with no associations. Such objects would be rare in student models.

Additional issues identified by UMLint are classified by the element upon which they are based: classes, class members, and associations.

4.1 Class issues

Identified issues related to classes include

- Checking for uses of “or” in class names. This also can indicate low cohesion, and UMLint suggests either renaming the class or introducing inheritance.
- Checking that class names follow the convention of starting with an upper case letter such as “Pet” rather than “pet”.
- As for actor names, flagging spaces and other illegal characters in class names. Reserved words as class names are also flagged, though this is a relatively uncommon mistake given the convention of using upper case letters for class names.

*We are indebted to an anonymous reviewer for this suggestion.

- Identifying classes without documentation. Documenting classes with their responsibilities should probably be a minimal requirement for diagrams.
- Identifying varying conventions in class names: some designers use underscores to separate words within names (such as “Food_Bowl”) while others do not. Either convention is acceptable, but students should apply a chosen convention or standard consistently.

4.2 Class members

UMLint checks attributes and operations for many of the same issues: avoiding illegal identifier characters, failing to document elements, and improper capitalization convention. For attributes and operations, UMLint applies the convention that names should start with a lower case letter. As mentioned above, UMLint also checks for nondescript words appearing in names.

Figure 3 illustrates additional issues detected by UMLint:

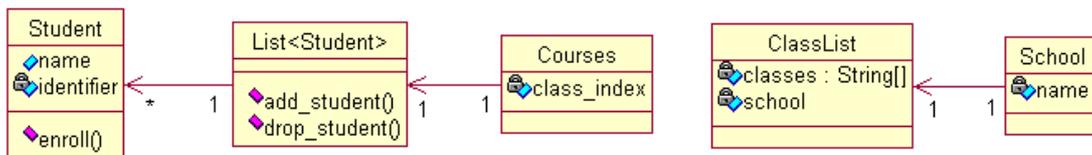


Figure 3: Class Element Defects

- Public attributes such as `name` in class `Student`. In older object-oriented languages, these typically result in broken abstractions. Because newer object-oriented languages support explicit properties (such as in C#, Ruby, and many others), this check should be refined to depend on the target programming language.
- Additional operations added to standard class libraries: `add_student` and `drop_student` are not standard `List` operations in Java. It certainly is possible to add such operations to an existing class, but not likely in student projects. In most cases this indicates a missing container class.
- Complex attributes such as `school` in class `ClassList`. A common convention is that only attributes with simple types such as numbers, strings, or dates are listed as class members. Data members with complex types should be represented by an association with a separate class, and there is no need to document that relationship in two places. This is an example of a “rule” which depends on modeling practice, and future versions of UMLint will likely allow disabling this check. UMLint detects such attributes by finding occurrences of class names within attribute names. This method can lead to false positive matches, but the frequency of the error justifies the check.

A similar issue occurs when an array is a data member as for `ClassList::classes`. If an array is an appropriate container for a type, an implication is that the position within that array is significant. This also implies that the items in the array have identity, suggesting that each item should be an instance of some class. If an array is necessary for performance reasons, the student should document that decision to ensure more robust designs were at least considered.

- Index variables used to refer to an object's location in a collection. In an object-oriented model, there is no need for the indirection: objects should simply point to their targets rather than contain indirect references. For example, the `class_index` member of `Courses` is probably intended to index into `ClassList::classes`. In this case, the index potentially hides an important association between `Courses` and the classes contained by `ClassList`.

4.3 Associations

Besides classes having more than one owner, Figure 4 illustrates most of the issues identified from associations:



Figure 4: Association defects

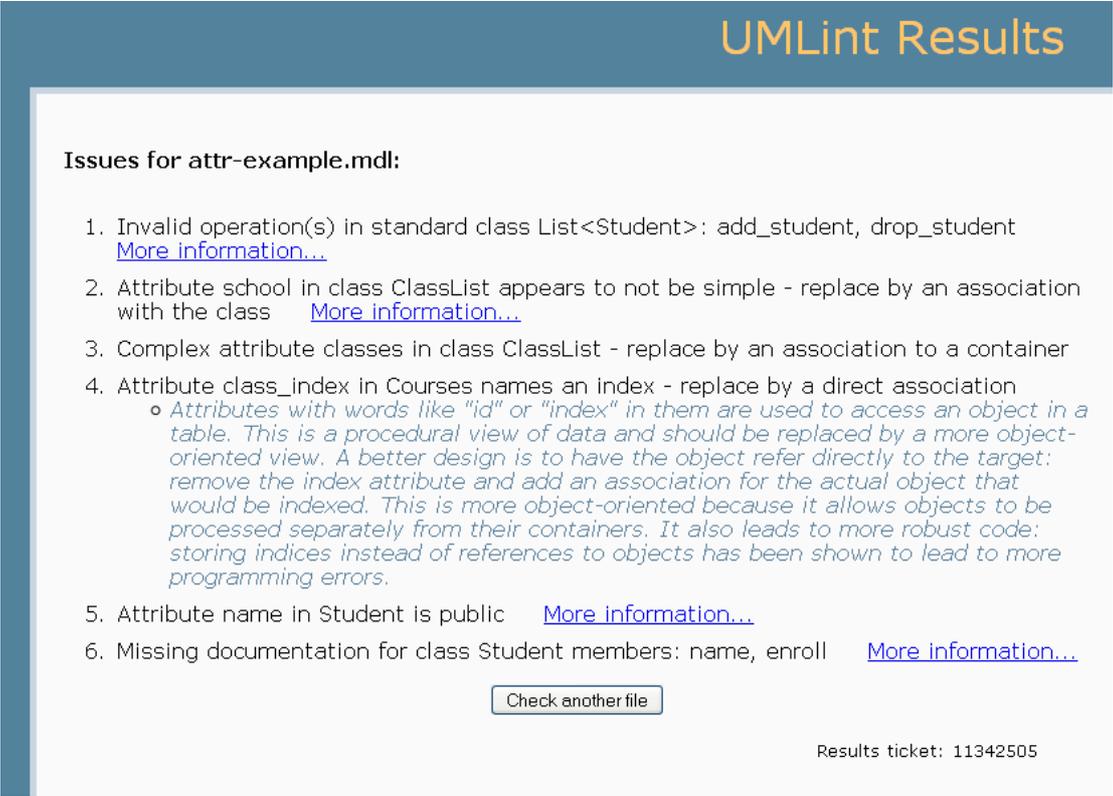
- Improper multiplicities for composition; since a class can have just one owner, composition implies a multiplicity of one.
- Using `0..*` rather than just `*`. Presumably `0..*` is supported by Rose for backwards compatibility.
- Using a dynamic association such as between `Ingredient` and `MixingBowl`. While these are very important in some contexts, they are discouraged in our courses since overuse can lead to cluttered diagrams. This is likely another example of a check that would need to be disabled for some instructors.
- Unspecified multiplicities such as between `MixingBowl` and `AutoStirrer`. While professionals may adopt a convention of assuming a multiplicity of one, our experience has been that such a convention in student work makes it too difficult to determine if the student failed to consider multiplicity altogether.

In addition, UMLint warns against using a multiplicity of 0. It also identifies associations and other references involving elements that are not shown on the diagram. Rose allows deleting

an element from a diagram without actually removing it from the model. Such hidden elements erode maintainability and cause problems in automated code generation.

5 Using UMLint

UMLint is available as a web service: students visit the website, browse to their model file, and click a button to obtain a list of defects. For example, checking the model for Figure 3 results in the output shown in Figure 5. The defects are in a priority order with the highest priority items (such as associations between actors and reversed generalization arrows being listed first and the lowest priority items (missing documentation) last. The priorities for defects are clearly



The screenshot shows a web interface titled "UMLint Results". Below the title, it says "Issues for attr-example.mdl:". There is a list of six issues, each with a "More information..." link. The fourth issue is expanded, showing a detailed description and recommendation. At the bottom of the list, there is a "Check another file" button and a "Results ticket: 11342505" label.

UMLint Results

Issues for attr-example.mdl:

1. Invalid operation(s) in standard class List<Student>: add_student, drop_student [More information...](#)
2. Attribute school in class ClassList appears to not be simple - replace by an association with the class [More information...](#)
3. Complex attribute classes in class ClassList - replace by an association to a container
4. Attribute class_index in Courses names an index - replace by a direct association
 - *Attributes with words like "id" or "index" in them are used to access an object in a table. This is a procedural view of data and should be replaced by a more object-oriented view. A better design is to have the object refer directly to the target; remove the index attribute and add an association for the actual object that would be indexed. This is more object-oriented because it allows objects to be processed separately from their containers. It also leads to more robust code: storing indices instead of references to objects has been shown to lead to more programming errors.*
5. Attribute name in Student is public [More information...](#)
6. Missing documentation for class Student members: name, enroll [More information...](#)

Results ticket: 11342505

Figure 5: UMLint output

subjective, but this groups defects by general type and does let the student focus on issues that must be changed for the model to be meaningful before considering less critical issues.

The “More information” links provide a more detailed description of the defect along with recommendations on how to improve the model. For illustration, this additional information has been expanded for the fourth defect. A future version will provide snippets of diagrams illustrating the issues to make the concepts more concrete. The output also contains a “ticket” number; this is used with a log to allow a student to provide evidence that the tool was run.

Interested readers are invited to use the tool for their own courses. Either contact the authors or simply search the web for “UMLint”.

6 Results

The tool has been used in courses at the sophomore, junior, and senior levels at UW-Platteville. The authors (who also teach each of these classes) consider the resulting diagrams to be improved over previous semesters. While diagrams still contain defects, they tend to be more sophisticated defects. A more rigorous evaluation of the tool’s effectiveness is needed and is planned as future work.

When we first introduced the tool, we did find students were reluctant to use it. Explicit motivation was needed, generally in the form of additional lost points for defects that the tool identifies. This is to be expected: students rarely want to hear that their work is incorrect! However, it does seem that once the students get used to the tool, they begin to see its usefulness.

As part of our preliminary evaluation, we requested written feedback from students using the tool in a sophomore/junior-level course. A number of comments reflect the need for additional work on the tool: making it more robust when syntactic errors are found in input files, improving the identification of defects in diagrams, and the need to include illustrations of the error types. Comments about the effectiveness of the tool include

- “Fixed numerous errors that [the instructor] did not address in class such as using * instead of 0.*. Also found errors in the syntax of the classes. attributes, operations, and the documentation of them.”
- “The tool helps with discovering some errors, for example in the Associations. We found a composition which was drawn the wrong way around. It also helped to discover missing documentation.”
- [After discussing a syntax error message, the student writes:] “Other than that, the utility was able to detect missing documentation and suggested when it was appropriate to use a navigation variable. This was very helpful when we were doing our final revisions before submittal.”
- “Running UMLint on our Rose Model file resulted in no new errors being reported. ... Having the ‘Okay’ from UMLint let us wrap up this deliverable with relative ease.”

Since these comments appeared in graded project reports, strongly negative comments would be surprising. Future evaluations will be anonymous. But it does seem students found the tool at least somewhat helpful.

7 Conclusion and Future Work

We have introduced a tool, UMLint, which identifies certain defects in student-generated object-oriented models. The tool checks for a number of issues in use case and class models, and our experience in using it has generally been positive.

In addition to ongoing maintenance and improving the interface to UMLint, we see several ways in which UMLint could be improved:

- Introduce checks for interaction and state diagrams and add support for the XML Metadata Interchange (XMI) format.¹⁶
- Define a mechanism for specializing checks on the basis of the target programming language, both for identifiers and libraries. This mechanism should allow users to specify identifier syntax and library classes.
- Provide a course profile mechanism to enable or disable specific checks.
- Revise how feedback is given so that issues appear as notes directly on the submitted diagrams.
- Incorporate natural language processing tools to provide more semantic-oriented checks such as described in our previous work.¹⁹ For example, the tool could warn against using verbs to name classes.
- Using metrics such as those surveyed by Genero⁹ to identify common design flaws such as concentrating all processing in one or two classes.

As discussed above, we are also planning a careful evaluation of the tool's effectiveness: to what extent the tool improves object-oriented models generated by students and to what extent the tool contributes positively to a student's understanding of modeling.

While the immediate goal of UMLint is to improve student learning, it is also hoped that UMLint will initiate a dialog regarding what standards should be enforced in UML diagrams. We anticipate that as other instructors use this tool in their classes, their feedback will result in extending and refining the checks made by UMLint.

References

- [1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [2] W. Coelho and G. Murphy. ClassCompass: A software design mentoring system. *ACM Journal on Educational Resources in Computing*, 7(1):Article 2, Mar. 2007.

- [3] M. Dahm. Grammar and API for Rational Rose petal files. <http://crazybeans.sourceforge.net/CrazyBeans/doc/grammar.pdf>, 2001. Retrieved January, 2011.
- [4] 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.
- [5] 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.
- [6] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 3rd edition, 2004.
- [7] R. France. A problem-oriented analysis of basic UML static requirements modeling concepts. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 57–69. ACM Press, 1999.
- [8] P. Gagnon, F. Mokhati, and M. Badri. Applying model checking to concurrent UML models. *Journal of Object Technology*, 7(1):59–84, Jan. 2008.
- [9] M. Genero, M. Piattini, and C. Calero. A survey of metrics for UML class diagrams. *Journal of Object Technology*, 4(9):61–92, 2005.
- [10] S. Johnson. Lint, a C program checker. Technical Report 65, Bell Laboratories, Dec. 1977.
- [11] K. Kaneiwa and K. Sotah. Consistency checking algorithms for restricted UML class diagrams. In *Lecture Notes in Computer Science*, volume 3861, pages 219–239. Springer-Verlag, 2006.
- [12] S. Konrad and B. H. C. Cheng. Automated analysis of natural language properties for UML models. In *Lecture Notes in Computer Science*, volume 3844, pages 48–57. Springer-Verlag, 2006.
- [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] MagicDraw. Available at <http://www.magicdraw.com/>. Accessed Jan., 2011.
- [15] T. Massoni, R. Gheyi, and P. Borba. A UML class diagram analyzer. In *3rd International Workshop on Critical Systems Development with UML*, pages 143–153, 2004.
- [16] Object Modeling Group. MOF 2.0/XMI Mapping, Version 2.1.1. Available at <http://www.omg.org/spec/XMI/2.1/PDF>, Dec. 2007.
- [17] 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.
- [18] O. Pilskalns, A. Andrews, S. Ghosh, and R. France. Rigorous testing by merging structural and behavioral UML representations. In *Proceedings of the 6th International Conference on the Unified Modeling Language*, pages 234–248, 2003.

- [19] 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.
- [20] K. Sanders and L. Thomas. Checklists for grading object-oriented CS1 programs: Concepts and misconceptions. In *ITiCSE '07*, pages 166–170, Dundee, Scotland, June 2007. ACM Press.
- [21] P. Stevens and R. Pooley. *Using UML: Software Engineering with Objects and Components, Updated Edition*. Addison-Wesley, 2000.
- [22] B. Thomasson, M. Ratcliffe, and L. Thomas. Identifying novice difficulties in object oriented design. In *ITiCSE '06*, pages 28–32, Bologna, Italy, June 2006.
- [23] UModel. Available at <http://www.altova.com/umodel.html>. Accessed Jan., 2011.
- [24] Visual Paradigm for UML. Available at <http://www.visual-paradigm.com/product/vpuml/>. Accessed Jan., 2011.