Experiences with a UML Diagram Critique Tool

Robert W. Hasker, Andrew Rosene, and James Reid
Computer Science and Software Engineering
University of Wisconsin-Platteville
Platteville, WI 53818

Abstract

We evaluate UMLint, an automated tool for detecting defects in UML diagrams. This tool is designed to improve object-oriented models developed by students. Standard diagramming tools provide little feedback to the user on diagram quality. UMLint addresses this for an academic environment by identifying common mistakes made by students. We present experiences by students in using UMLint in a variety of upper level courses. This experience will be used to improve the tool and to hopefully provide useful information for integrating UMLint into other curricula.

1 Introduction

Since code is an expensive way to capture and evaluate system designs, most students are at least introduced to diagraming systems in some way. However, teaching students diagraming skills is difficult. In our experience, students resist learning new notation, misunderstand the more subtle aspects, and often fail to appreciate the fundamental relationships between the notation and implementations. A related issue is that the tools we use to teach diagramming are often the same ones used by practitioners in industry. These tools are powerful, but they must encompass a wide variety of processes. The result is that they provide less direct feedback than might be helpful to students, introducing more challenges into the process of learning to diagram systems. Instructors can certainly provide the missing feedback, but grading diagrams is complex and takes time.

We have developed a tool, UMLint, which attempts to provide direct feedback on diagram quality to students. The tool focuses on use case and class diagrams using the Unified Modeling Language (UML) [1]. For both, it addresses issues ranging from failing to follow naming conventions to identifying missing elements. It currently supports only diagrams drawn by Rational Rose, but support for alternative diagramming tools is in development. Students draw diagrams and submit them to UMLint through a web interface. The result is that students can get more timely feedback for certain diagramming issues.

The tool has two goals. The obvious one is to help students avoid frequent errors. Many of these issues are stylistic, but there are a number of structural issues that can be identified. Of course, rules have exceptions, and so ultimately it is up to the instructor to determine which errors are truly significant. There are also many errors that cannot be detected mechanically. However, UMLint allows students to correct the relatively simple errors before a final submission. This leads to a second, less obvious goal. By directing the students to fix relatively simple issues, it is hoped that they will be encouraged to examine their diagrams closely to find deeper errors.

The curriculum at UW-Platteville presents the major UML diagrams: use case, class, sequence, collaboration, and state diagrams. Use case and class diagrams are introduced at the freshman/sophomore level, and other diagrams introduced in subsequent courses. Later courses also introduce additional detail. In the first course, use case and class diagrams contain only simple associations and major attributes and operations. Later courses discuss alternative types of associations and other details. In a junior-level course, students generate full, compilable interfaces from 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.

UMLint has been used at UW-Platteville for over a year. This report discusses the tool from the student's point of view: what has worked, where the limitations are, and how the tool might be improved. It is hoped that these experiences will help other instructors determine whether UMLint would be applicable to their curricula.

1.1 UMLint Checks

UMLint currently critiques two types of diagrams: use case and class diagrams. For use case diagrams, the tool identifies issues such as directed associations between actors and use cases, using the wrong type of association for extends and includes, and attempting to name a use case with a single word. For example, the diagram in Figure 1 includes a

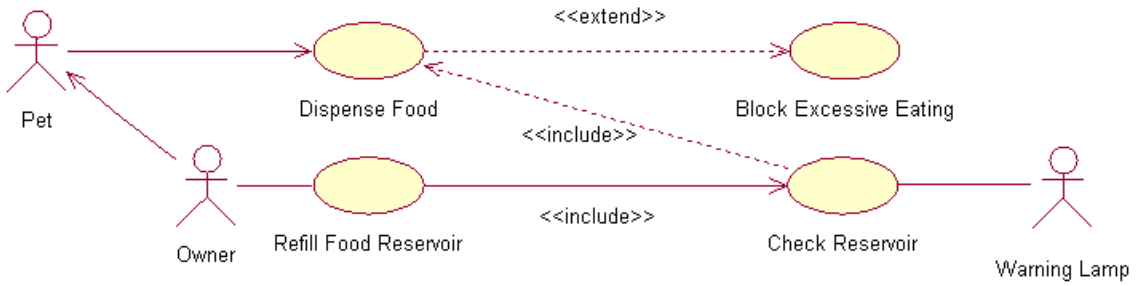

Figure 1: Use case model with defects.

number of issues:

- Using a directed association between Pet and Dispense Food.
- Documenting an association between Pet and Owner that should be in a class model instead.
- Having a reversed `<<extend>>` dependency.
- Using a regular association (as opposed to a dashed line) for an `<<include>>` dependency between Refill Food Reservoir and Check Reservoir.

Use case diagrams are relatively simple, so the number of potential errors is small. There is also a legitimate question about the usefulness of use case diagrams [5, p. 104]. However, these diagrams do provide a good opportunity to teach basic diagramming concepts to students.

Figure 2 illustrates some of the class model defects identified by UMLint:

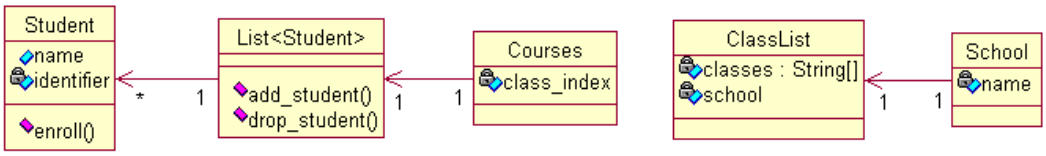

Figure 2: Class model with defects.

- Public attributes such as `name` in class `Student`.
- Additional operations added to standard class libraries: `add_student` and `drop_student` are not standard Java `List` operations.
- 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.
- Index variables such as `Courses::class_index` which indicate poor object-oriented design. In this case, the index potentially hides an important association between `Courses` and `ClassList`.

Additional checks include potential misuses of multiple generalizations, nondescript names such as "flag," poor cohesion exhibited by 'or' or 'and' in class names, failing to follow capitalization standards, classes having more than one owner, unassociated classes, and failing to include multiplicities or documentation. See [16] and [8] for details about the checks and the reasons for introducing those checks. It is well understood that standards for diagrams are likely to vary by instructor, and UMLint allows instructors to define checklists that are specific to their needs.

1.2 Related Work

Most of the existing tools to identify defects in UML models are designed for use by professionals, [3, 4, 11, 14]. These influence UMLint, but practices that are errors for students are often acceptable for professionals and vice versa. A number of student-specific tools do exist such as MagicDraw [12], UModel [19], and Visual Paradigm for UML [20]. However, these tools focus on defects that would lead to errors in code generated from diagrams. The errors in UMLint are generally independent of code generation, and study would be needed to determine whether additional checks from these tools would be relevant.

Another method, again geared towards professionals, is to use formal specification languages as a tool to validate UML models [6, 9, 10, 13]. It would be interesting to determine the extent to which these identify errors commonly made by students. In any case, formal specifications often require a more rigorous framework than is practical for most undergraduate students. Closely related to this is the large body of work on UML Model checking [7, 15]. 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 [17, 18]. 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 ClassCompass [2] 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.

2 Experiences

Two of the authors of this report used the tool as students in junior and senior level classes. We will examine three of these projects.

The first is an assignment given in a capstone project course in the software engineering major. The prerequisites for this course include data structures, Intermediate Software Engineering, and Object-oriented Analysis and Design. The goal of the assignment was to determine the extent to which students had retained their skills on drawing use case and class diagrams. In this assignment, students were asked to model a somewhat fanciful system in which self-guide automobiles were designed to deliver rescue supplies to victims in sparsely-populated, arid areas. The victim calls for help, and the operator plans a route for the vehicle to drive to the victim's location. The vehicle would have a limited ability to drive around obstacles such as trees and animals blocking the route.

The student's initial diagram is given in Figure 3. This diagram contains a number of issues identified by UMLint:

- Directed associations between use cases and actors such as Operator/chooseRoute and messageOperator/Operator.
- Regular classes such as `RescueCar`, `Camera`, and `Route` that should not appear on a use case diagram.
- Regular associations instead of `<<extend>>` and `<<include>>` dependencies between use cases.
- Appending '()' to use case names.

The student was able to use the issues to relearn basic use case diagram notation. The repaired diagram is given in Figure 4. The extraneous classes were removed, the associations between actors and use cases were repaired, and dependencies were introduced. An obvious remaining issue is that the diagram contains flowchart elements. UMLint has since been improved to warn of use cases which are only remotely associated with an actor, and running the tool on this diagram now results in the message

> Use case(s) isLeftClear, isRightClear more than 3 links from an actor

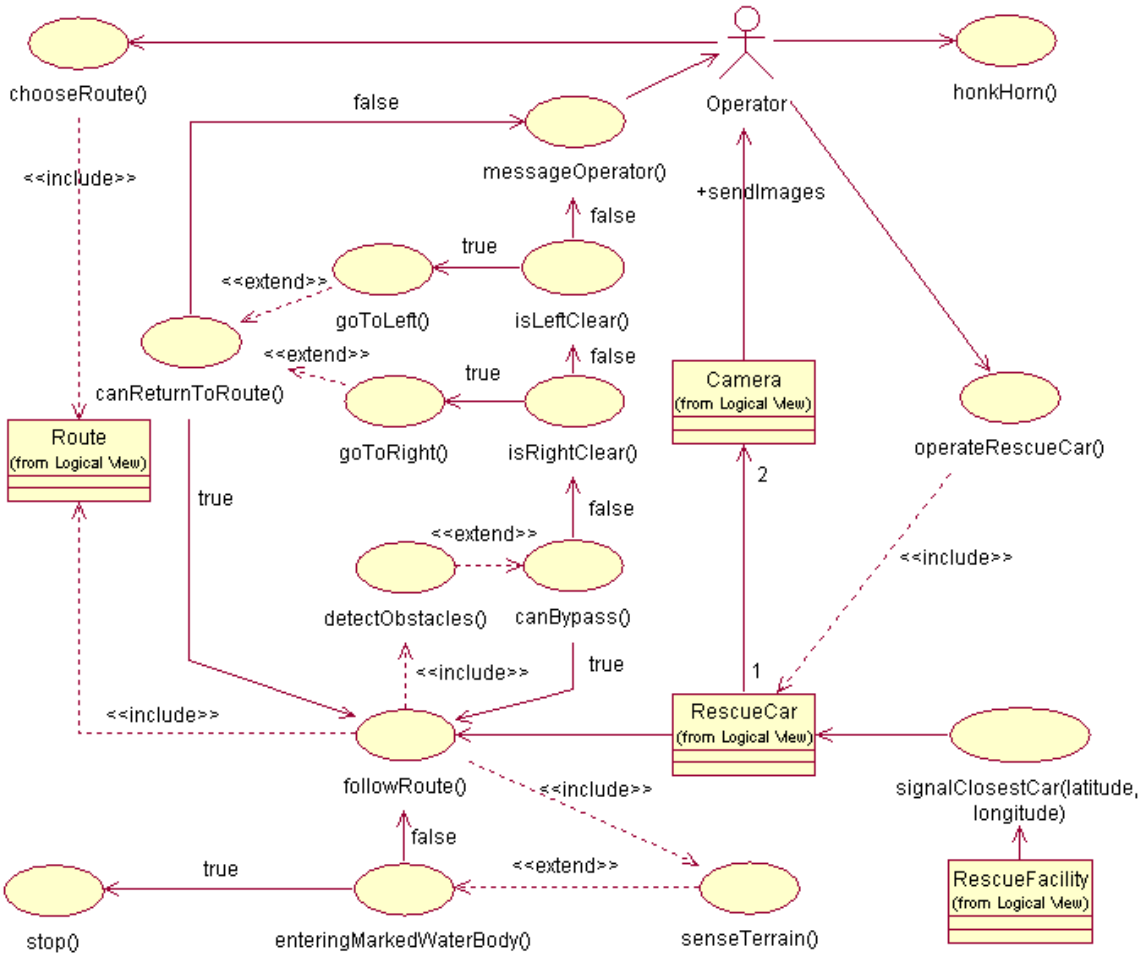

Figure 3: Initial rescue use case model.

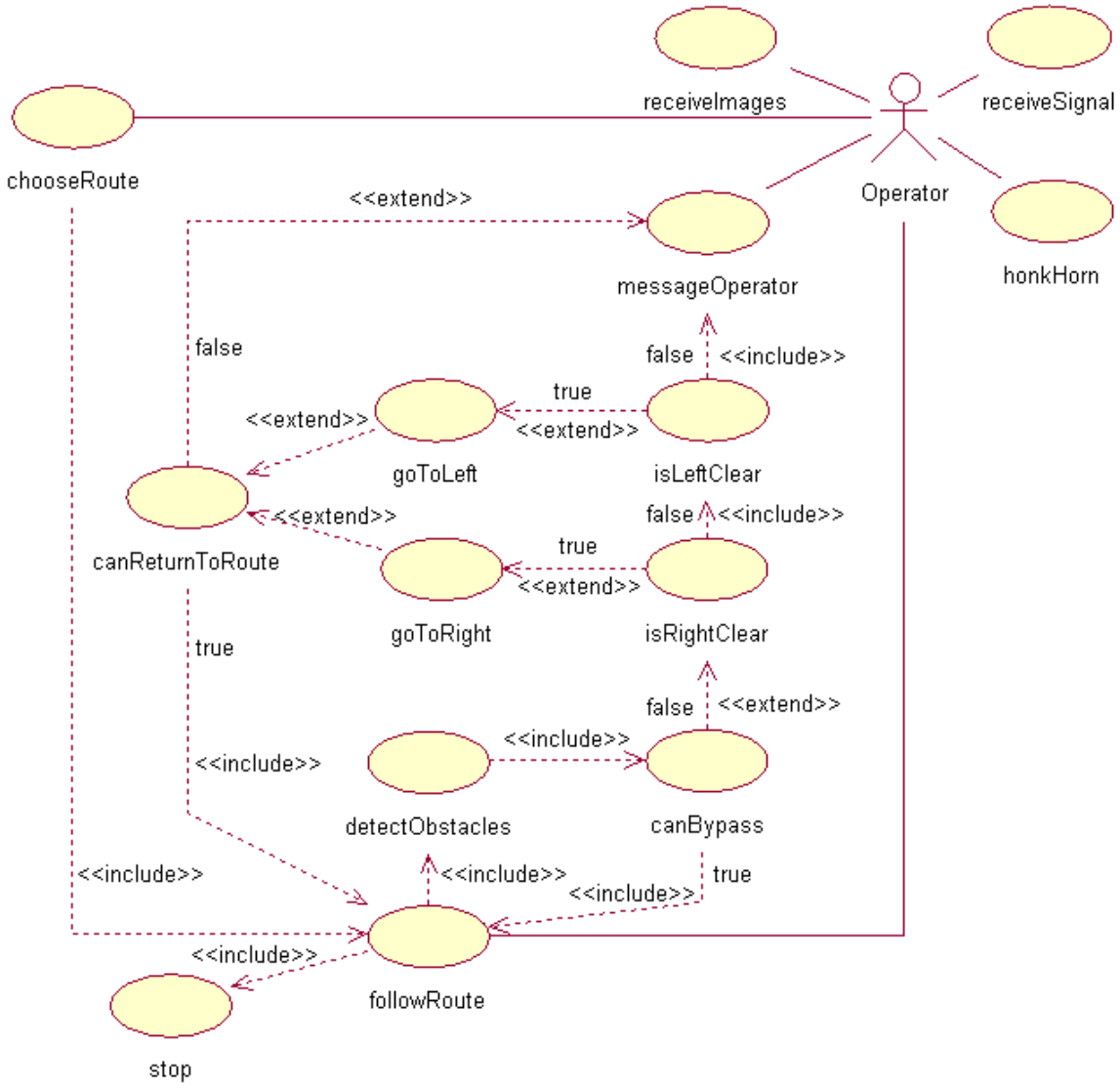

Figure 4: Final rescue use case model.

Repeating the experiment in a future semester will help determine if this is sufficient to guide students towards eliminating flowchart features from use case diagrams.

The second example is from a larger design project done in the same course. The project was to construct a phone app for ordering food from on-campus restaurants. Foods are grouped into categories, with each food item having optional condiments as well as optional components such as extra cheese. One group developed the diagram shown in Figure 5. This diagram is complex enough that it may be a challenge to find the inconsistencies:

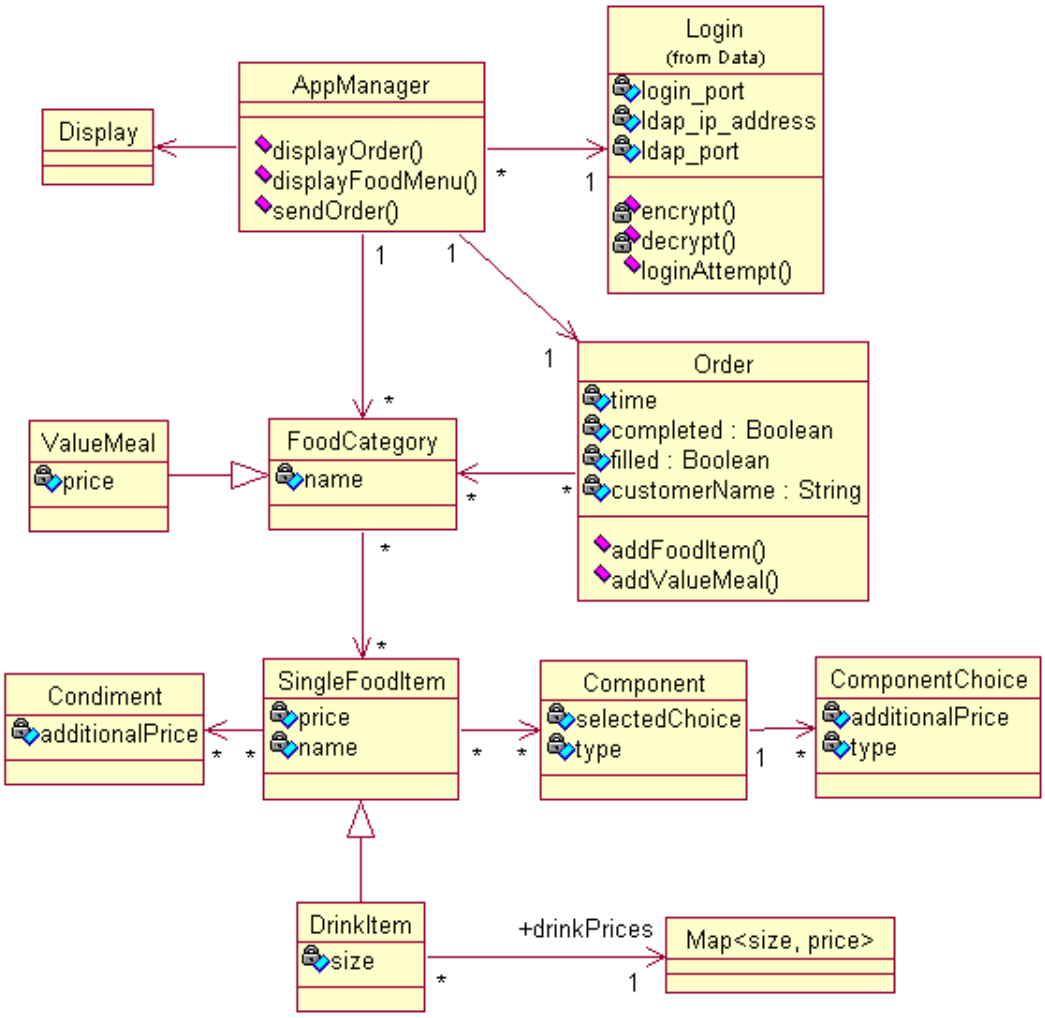

Figure 5: Food app class diagram.

- Multiplicities are missing between `Display` and `AppManager`.
- Most attributes use "camel case," in which capital letters are used to separate words. However, the attributes in class `Login` use underscores.

On a large project, such inconsistencies can slow down development.

The last two examples were developed by a different group in the same course. The first, in Figure 6, is an alternative model to support ordering food. In addition to a number of

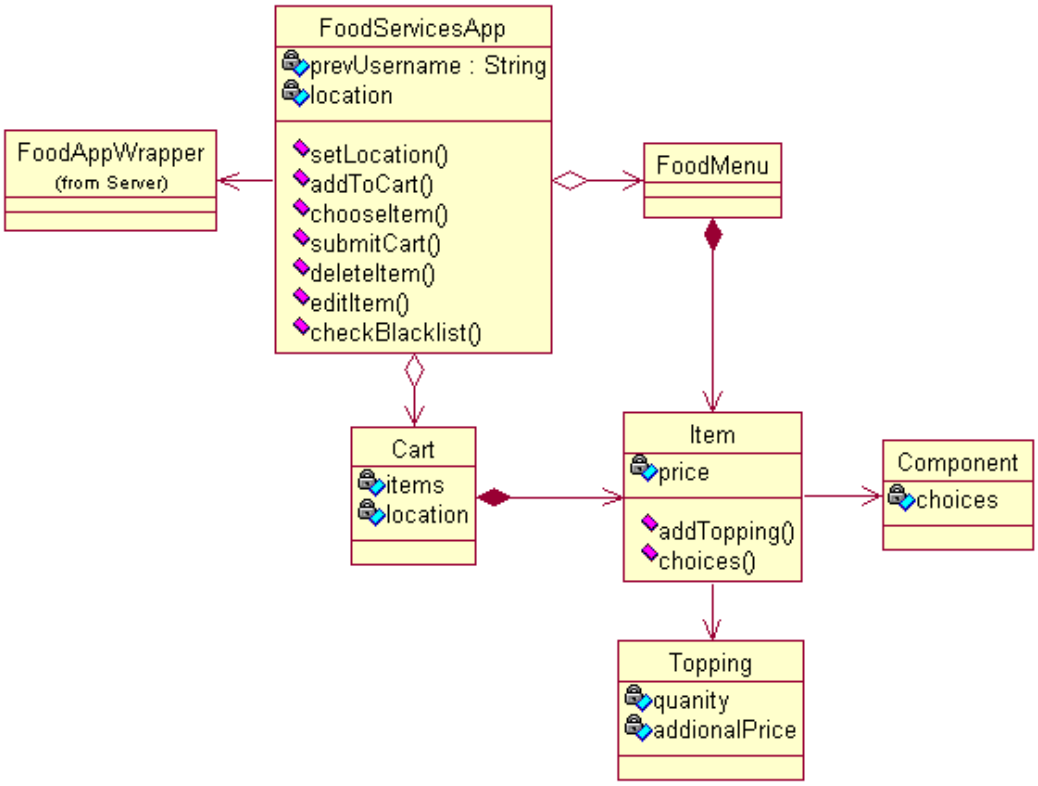

Figure 6: Initial classes to support an alternative food ordering system.

missing multiplicities, the issues identified by UMLint are

- Attribute `items` in class `Cart` appears to not be simple - replace by an association with the class.
- Class `Item` has two owners: `Cart` and `FoodMenu`.
- Unrecognized words in identifiers: `addional`, `prev`, `quanity`.

These issues are a bit more subtle than those in Figure 5. The first reflects a common standard that only simple attributes be listed in classes. Complex objects are to be shown by association only. In this case, UMLint notes a problem because an attribute (`items`) contains a class name. Clearly this heuristic could result in false positives; it is up to the user to decide when this is a true error. The composition relationships between `Item` and both `FoodMenu` and `Cart` indicate each item is "owned" by both; this is clearly inconsistent. Finally, the misspelled words in the `Topping` attributes add unnecessary complexity to development.

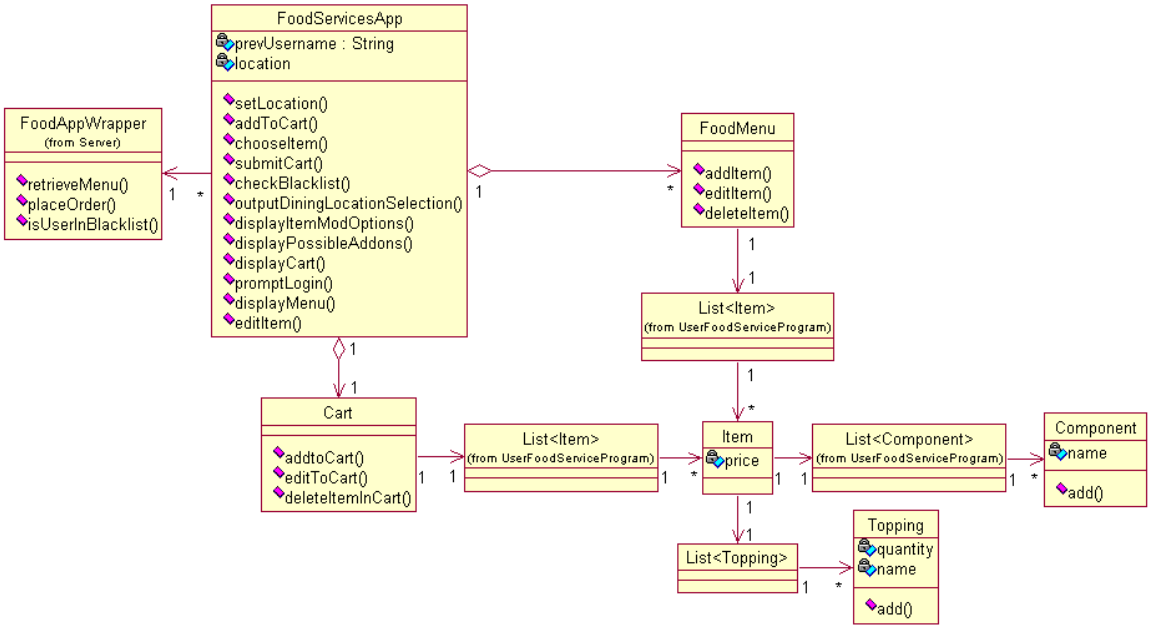

Figure 7: Final classes for the alternative food ordering system.

The final version of this diagram is given in Figure 7. Fixing the association between `FoodMenu` and `Item` and adding multiplicities resulted in a many-to-one association between `FoodMenu` and `Item`. Since this was intended to be a detailed diagram, UMLint was configured to note that in this is a case of "Failing to use a container class where one is needed." This and other, similar fixes lead to introducing `List` classes in a number of places. In the process, the group re-examined and revised the placement of methods in classes.

A second diagram from the same project illustrates further additions. Figure 8 gives the initial version of this server. In addition to requiring multiplicities, students were asked to

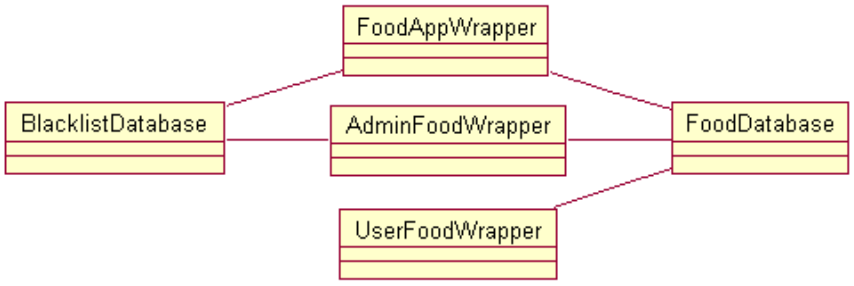

Figure 8: Original server for food ordering app.

add documentation to each class. UMLint can be configured to list classes (and optionally, attributes and methods) which have no documentation. In the process of adding this documentation, the group was motivated to add a number of methods to the classes. The resulting diagram is in Figure 9.

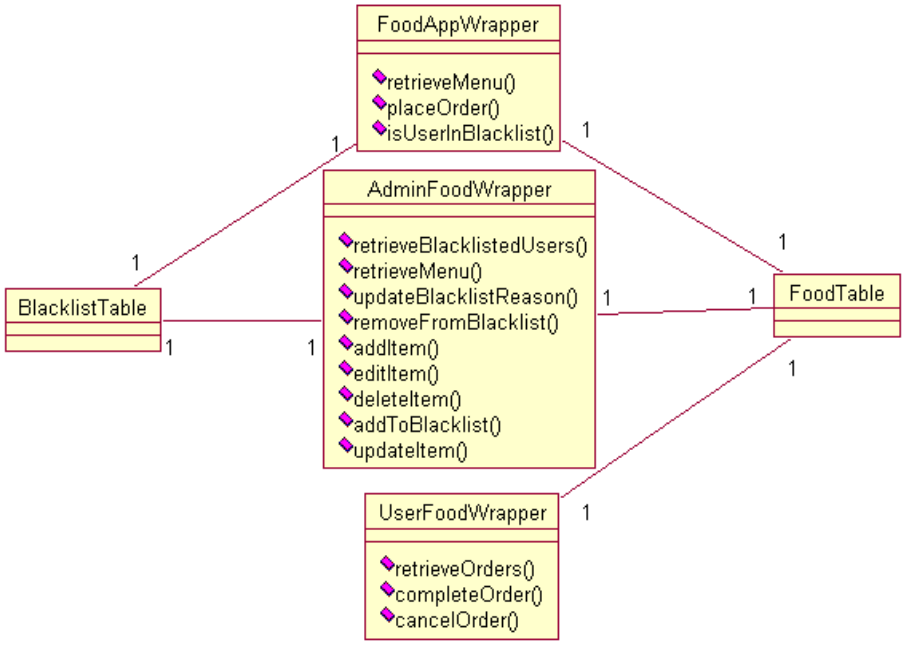

Figure 9: Original server for food ordering app.

3 Discussion

In the introduction, two goals were given for UMLint: helping students avoid frequent errors and encourage them to examine their diagrams more closely. A full evaluation of the success at achieving these goals would require more study. This report represents an more informal evaluation based on the authors' direct experiences with using the tool in individual and group projects.

We have presented both the first and last diagrams submitted to UMLint for a number of similar problems. Diagrams created for other courses were also examined in developing this report. Ideally we would find cases in which new domain-level classes were identified, significant changes were made to association relationships, or additional uses of generalization were found. No such cases were identified. Generally, these types of issues were identified by the groups before submitting the initial version to UMLint.

However, while there was no evidence of gross structural changes, a large number of smaller issues were identified. These ranged from failing to follow naming conventions to incorrect associations, missing multiplicities, and missing container classes. The tool did help correct a significant number of details that may have been difficult to identify otherwise. We also found that the existence of the tool helped clarify discussions. Often group members would point out that certain choices would be flagged by UMLint. In these cases at least, the tool helped groups maintain a consistent level.

The tool also proved helpful on individual assignments. The individual assignment in this

report—delivering an autonomous vehicle to a victim—was designed to exercise students in drawing diagrams rather than identifying classes. More open problems would provide more opportunities for the tool to help students identify structural issues. But the students did find the tool useful in reminding users about how to use the notation. As stated by one of the authors, "Basic stuff, but it's that basic stuff you forget."

This analysis revealed some additional features of UMLint that may need revisiting:

- While UMLint does check that every object has documentation, that check is very simplistic. Currently it allows a simple space to satisfy this check. A spot-check by the instructor should reveal cases in which such tricks have been used, but a better solution would be to introduce more stringent checks for appropriate documentation. A simple solution would be to use a natural language parser and require that documentation contain at least one sentence.
- Rational Rose allows model elements to be deleted from a diagram but left in the underlying model. Such hidden elements can cause problems with code generation because they are difficult to inspect and maintain. UMLint flags hidden elements for deletion by the user. However, little direction is given on where to find these elements. More information needs to be given to the user to find them.
- A command line version of UMLint would be useful to groups because it would often be easier to use than the web interface. It is possible that the web interface could be improved to simplify re-checking diagrams after the initial check. It may be possible to provide a command-line version of the tool for student use, but as further analyses are integrated (especially those performed by external tools such as natural language processors) the installation will become more complex.

It is hoped that additional use by instructors from other institutions will result in further improvements.

References

[1] BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

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

[3] DE SOUZA, C. R. B., OLIVEIRA, H. L. R., DA ROCHA, C. R. P., GONÇALVES, K. M., AND REDMILES, D. F. Using critiquing systems for inconsistency detection in software engineering models. In *SEKE* (2003), pp. 196–203.

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

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

[6] FRANCE, R. 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* (1999), ACM Press, pp. 57–69.

[7] GAGNON, P., MOKHATI, F., AND BADRI, M. Applying model checking to concurrent UML models. *Journal of Object Technology 7*, 1 (Jan. 2008), 59–84.

[8] HASKER, R. W., AND ROWE, M. UMLint: Identifying defects in UML diagrams. In *118th ASEE Annual Conference & Exposition* (Vancouver, BC, Canada, June 2011).

[9] KANEIWA, K., AND SOTAH, K. Consistency checking algorithms for restricted UML class diagrams. In *Lecture Notes in Computer Science*, vol. 3861. Springer-Verlag, 2006, pp. 219–239.

[10] KONRAD, S., AND CHENG, B. H. C. Automated analysis of natural language properties for UML models. In *Lecture Notes in Computer Science*, vol. 3844. Springer-Verlag, 2006, pp. 48–57.

[11] LANGE, C. F. J. Improving the quality of UML models in practice. In *Proceedings of the 28th International Conference on Software Engineering* (2006), ACM Press, pp. 993–996.

[12] MagicDraw. Available at http://www.magicdraw.com/. Accessed Jan., 2011.

[13] MASSONI, T., GHEYI, R., AND BORBA, P. A UML class diagram analyzer. In *3rd International Workshop on Critical Systems Development with UML* (2004), pp. 143–153.

[14] PAP, Z., MAJZIK, I., PATARICZA, A., AND SZEGI, A. Completeness and consistency analysis of UML statechart specifications. In *Proceedings IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop* (2001), pp. 83–90.

[15] PILSKALNS, O., ANDREWS, A., GHOSH, S., AND FRANCE, R. Rigorous testing by merging structural and behavioral UML representations. In *Proceedings of the 6th International Conference on the Unified Modeling Language* (2003), pp. 234–248.

[16] ROWE, M., AND HASKER, R. W. The characterization and identification of object-oriented model defects. In *41st Midwest Instruction and Computing Symposium* (La Crosse, Wisconsin, 2008), pp. 178–192.

[17] SANDERS, K., AND THOMAS, L. Checklists for grading object-oriented CS1 programs: Concepts and misconceptions. In *ITiCSE '07* (Dundee, Scotland, June 2007), ACM Press, pp. 166–170.

[18] THOMASSON, B., RATCLIFFE, M., AND THOMAS, L. Identifying novice difficulties in object oriented design. In *ITiCSE '06* (Bologna, Italy, June 2006), pp. 28–32.

[19] UModel. Available at http://www.altova.com/umodel.html. Accessed Jan., 2011.

[20] Visual Paradigm for UML. Available at http://www.visual-paradigm.com/product/vpuml/. Accessed Jan., 2011.