The Characterization and Identification of Object-Oriented Model Defects

Mike Rowe and Robert W. Hasker
Computer Science and Software Engineering Department
University of Wisconsin – Platteville
Platteville, Wisconsin 53818
rowemi@uwplatt.edu, hasker@uwplatt.edu

Abstract

This paper presents a study of defects that commonly occur in object-oriented modeling. The study is based on experience from teaching more than a dozen sections of an Object-Oriented Analysis and Design course to sophomore and junior-level Software Engineering and Computer Science majors over the last eight years. The students use IBM (Rational) Rose as the design tool.

The goal of this research is to eventually provide real-time and anytime feedback for students as they develop their object-oriented models. It is hoped that this instant feedback will help students by discouraging them from developing bad habits and guiding them in the development of superior software models.

1 Introduction

While teaching the Object Oriented Analysis and Design course, we have observed that many software modeling defects occur multiple times, year after year. This paper categorizes these common defects by type of model: Use Case, Class, and Interaction and State. Examples of these defects as well as manual methods and potentially automated techniques for identifying some of these defects are described. This paper serves two goals: to provide a (non-exhaustive) catalog of errors that could be made available to students so they might be less likely to introduce the same defects, and to generate feedback from other instructors about these errors.

Eventually, it is hoped that satisfactory automated techniques will be made available for students and their instructors to help rapidly detect these defects and provide feedback to remedy the problem. The automated techniques proposed by this paper are of two types. The first automated approach involves the generation of C++ source code from the models, compiling it and studying the compilation errors. The second approach involves parsing Rose model (MDL) files to find specific defects directly.

The defects cataloged in this paper are based on having taught multiple sections of an Object-Oriented Analysis and Design course over the past eight years. This course covers modeling using the UML notation, though of course the concepts extend beyond any particular syntax. The students in the course are typically at the sophomore and junior level. The prerequisites are a course on fundamental data structures (CS2) and a project-based course on software engineering. Thus students are quite familiar with object-oriented programming and have already been introduced to some basic issues of object-oriented design. This course attempts to move students from applying the techniques on small problems towards modeling larger systems.

2 Related Work

Automated detection of defects in UML diagrams has received attention from a number of researchers [1][3][7][8][9][10][11][12]. However, these works focus on improving diagrams for professional developers. This paper focuses on errors typically made by students – people who are not only learning UML, but also learning basic issues about how to apply modeling in general.

This paper discusses errors typically made by students. [2] and [13] also examine errors made by students, but these catalog only a few errors and some of these are controversial. [5] and [14] discuss common errors in more depth, but focus on errors made by CS1 students. This paper discusses errors made by students at the sophomore/junior level. These are students who already have a basic understanding of object-oriented programming but do not yet appreciate certain subtleties of both modeling and object-oriented design.

3 Use Case Model Defects

There are a number of frequent defects in use case models: failing to use verb phrases for cases, misusing extends and includes, and use cases which capture insignificant interactions.

3.1 Use Case Titles That Are Not Verb Phrases

Description: Use cases are common uses of a system that describe how an actor or actors obtain a significant benefit from a system. Since they describe this process of obtaining a benefit, they are best titled as verb or verb-object phrases.

Detection: Use case titles are generally concatenated strings of two or more words. This makes it difficult to parse the strings into separate words unless some syntax is enforced such as "camel case" or underscore word separators. If the syntax allows parsing, the component words in the use case title can be compared to a lexicon or use natural language technology to recognize parts of speech. If a use case title is used inside of the scenario, parsing the use case title in the syntax of the sentence may be able to reveal its part of speech.

3.2 Misuse of Extends and Includes

Description: Use case extensions and inclusions are often confused. Most students can give the strong definitions of these constructs, but often confuse the direction of the arrows when producing use case diagrams. For an "include" relationship, the arrow should be on the included use case side, whereas with an "extension" relationship, the arrow should be on the far side of the extending use case.

Detection: This may be difficult to detect with only the use case diagram. A solution may be to cross check the diagram against use case scenarios. Formal use case scenarios typically list extensions and inclusions. By locating the use case scenario title in the use case model, we can analyze the model from this point, comparing the model's extends and includes against the scenario's. The automatic checking for this defect would rely on very specific use case scenario templates.

3.3 Insignificant Use Cases

Description: The goal of a use case is to provide a significant benefit to an actor associated with that use case. Use cases without a significant benefit should be combined with others.

Detection: The above definition does not meet many of the IEEE STD 830 [6] characteristics of "good" requirements. To say the least, this does not meet the test of verifiability, in that we would be hard pressed to obtain universal consensus on precisely what is meant by "significant benefit". We believe that certain cases can be defined and recognized, but further research is needed in this area.

4 Class Model Defects

Because they are richer, the number of potential errors in class models is larger.

4.1 Non-noun Class Names

Description: Classes are the stuff from which objects are made, and objects are nouns. Students frequently name classes using verbs, possibly because they are focusing on the actions performed as part of the class rather than the object which performs those actions.

Detection: The detection of non-noun class names could be partially automated by parsing scenarios, identifying which words are used most frequently as nouns, adjectives, or verbs, and using this information to identify misused words in class names.

4.2 Reversed Multiplicities

Description: The collection class of an aggregation or composition should have a multiplicity of '1', whereas the parts can have any multiplicity. This mistake is also commonly seen with other class relationships.

Detection: This can be detected with aggregation and composition, when the collection class has a multiplicity not equal to '1'. See Figure 1 for a UML example of reversed multiplicities and Figure 2 for a corresponding MDL file snippet. With non-collection relationships, this cannot be easily detected without domain knowledge about the classes and their relationship.

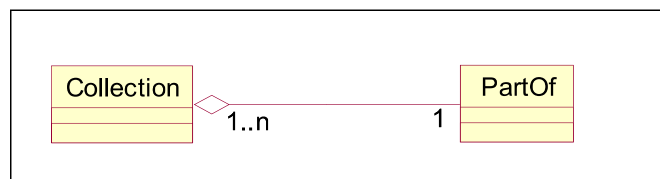

Figure 1: UML model with reversed multiplicities

```
// snippet from MDL file
root_category     (object Class_Category "Logical View"
  quid              "47BD93AC01B0"
  exportControl     "Public"
  global            TRUE
  subsystem         "Component View"
  quidu             "47BD93AC01B2"
  logical_models    (list unit_reference_list
    (object Class "Collection"
        quid        "47BD942B03D2")
    (object Class "PartOf"
        quid        "47BD942F0346"
        documentation  "A part of the collection.")          // *
    (object Association "aggregation"
```

```
quid        "47BD943502E8"
roles       (list role_list
  (object Role "$UNNAMED$0"
        quid        "47BD94360384"
        supplier              "Logical View::PartOf"
        quidu                 "47BD942F0346"
        client_cardinality  (value cardinality "1")     // multiplicity of part of: 1
        is_navigable         TRUE)
   (object Role "$UNNAMED$1"
        quid        "47BD94360395"
        supplier              "Logical View::Collection"
        quidu                 "47BD942B03D2"
        client_cardinality  (value cardinality "1..n") // multiplicity of Collections: 1..n
        is_navigable         TRUE
        is_aggregate         TRUE))))                  // indicates Collection is an aggregate
```

Figure 2: Rose MDL file snippet for defective aggregation multiplicities.

4.3 Only Public Operations and Attributes in Implementation-level Models

Description: An implementation-level class diagram should be at the detail that goes beyond interfaces and public operations. If almost all operations across all classes are public, then either the model not have enough detail to include private operations or the designer has not properly identified which functions should be private. In either case, there is a problem. Generally, it is hard to defend the concept that any class attribute should be public.

Detection: The detection of this defect is as simple as calculating a percentage of public to non-public operations in a class model. As long as the percentage is larger than some arbitrary value (well less than 100%) then this defect may be present. An acceptable percentage will depend on the domain, the design, and the expected level of detail in the design. Recognizing any public attributes can also be flagged as very likely defects

4.4 Classes, Operations, and Attributes without Documentation

Description: Rose class models allow the designers to add documentation to classes, operations, attributes, and operation parameters. This documentation is inserted into the source code that is generated from a class model. Documenting the elements while designing the system is an excellent practice since this is when designers are most likely to be intimate with both the requirements and the elements of the design that satisfy those requirements.

Detection: Identifying missing documentation is straightforward. The line marked by an asterisk in Figure 2 marks the documentation for class part of ("a part of the collection"). In contrast, class Collection, listed a couple lines earlier in the file, is missing its documentation.

4.5 Associations without Navigation Attributes

Description: Navigational attributes are reference or pointer variables that allow one class to access another class. In a class model, they are part of the reference and on the side of the navigational arrow. When code is generated from a class model, a navigational attribute produces a reference or pointer class variable of the type of the associated class.

Detection: Detection of missing navigational attributes can be detected by processing the Rose MDL file's references for missing navigational attributes. See Figure 3 for UML model, Figure 4 for MDL file associated with the UML, and Figure 5 for C++ code auto-generated by Rose from the model.

This "rule" is somewhat controversial: some instructors would prefer students to generate diagrams with less redundancy in them. This illustrates a basic requirement for any automated system: it must be possible for instructors to select which rules to apply for a particular course or even assignment.

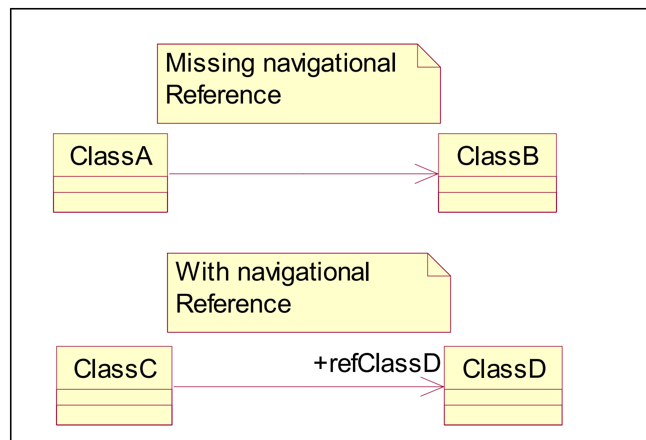

Figure 3: UML without and with navigational references

```
logical_models    (list unit_reference_list
        (object Class "ClassA"
            quid       "47BDA0820067")
        (object Class "ClassB"
            quid       "47BDA08500A6")
        (object Class "ClassC"
            quid       "47BDA087023C")
        (object Class "ClassD"
            quid       "47BDA08A00F4")
        (object Association "$UNNAMED$0"
            quid       "47BDA0B803B3"
            roles      (list role_list
              (object Role "$UNNAMED$1"  // Un-named navigational reference
                   quid       "47BDA0B903E2"
                   supplier          "Logical View::ClassB"  // type of reference
                   quidu             "47BDA08500A6"
                   is_navigable      TRUE)  // Navigable
```

```
          (object Role "$UNNAMED$2"
                quid        "47BDA0B903E4"
                supplier              "Logical View::ClassA"
                quidu                 "47BDA0820067")))
    (object Association "$UNNAMED$3"
          quid        "47BDA0BD01EE"
          roles       (list role_list
          (object Role "refClassD"  // named navigational reference
                quid        "47BDA0BE021D"
                label       "refClassD"
                supplier              "Logical View::ClassD" // type of reference
                quidu                 "47BDA08A00F4"
                is_navigable          TRUE)  // Navigable
          (object Role "$UNNAMED$4"
                quid        "47BDA0BE021F"
                supplier              "Logical View::ClassC"
                quidu                 "47BDA087023C"))))
```

Figure 4: MDL file snippet of relationships without and with navigational references

```
//##ModelId=47BDA0820067
class ClassA   // notice no reference to Class B is generated
{
};

class ClassC
{
 public:   // this is the generated reference to Class D
   //##ModelId=47BDA0BE021D
   ClassD *refClassD;
};
```

Figure 5: Code auto-generated from Rose without (ClassA) and with (ClassC) navigational references.

4.6 Attributes and Operations that are not Typed

Description: At the implementation level, class model attributes, operations, and operation parameters need to be typed to support code development.

Detection: This defect can be detected by using Rose to generate source code from the class model and compiling it. Non-typed identifiers are not permitted in many high-level languages and will produce syntax errors. For example, the C++ code generated for NonTypedClass in Figure 6 is

```
Class NonTypedClass { public: opp(void agr1); private: att1; };
```

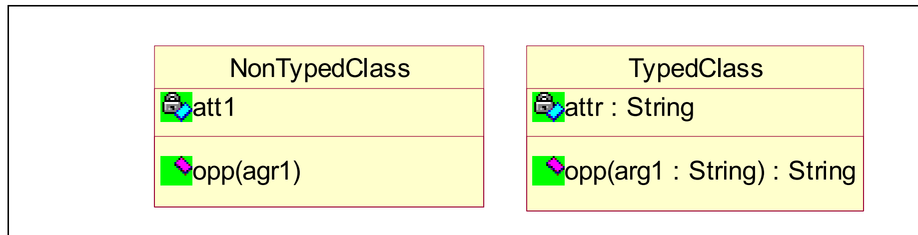

Figure 6: UML without and with typed attributes, operations, or operation arguments

4.7 Illegal Identifiers for Target Language

Description: Classes should not be a dead end in a software development process. The class names, attributes, operations, and parameters of the model produce the identifiers in the generated source code. If illegal identifiers are used in the model, they will appear in the generated code.

Detection: This defect can be detected by using Rose to generate source code from the class model and compiling it. The compiler will produce errors relating to these illegal identifiers. Below is an example of a class with illegal C++ identifiers as they contain embedded spaces, "attr One", "opp Two", and "arg Three". When the Rose-generated code is compiled, parse errors result on the identifiers.

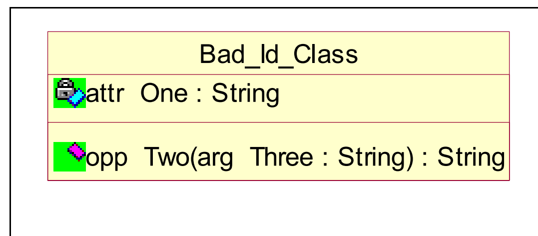

Figure 7: A class with incorrect identifiers for an attribute, an operation, and a parameter of the operation

4.8 Inheritance Arrows in the Wrong Direction

Description: A common student error is that to reverse the inheritance arrows, placing the inheritance arrow on the child rather than the parent side of the relationship.

Detection: In general it could be very difficult to identify reversed inheritance arrows automatically. However, it is possible in some cases: given that multiple inheritance is rare in student solutions, a class with multiple outgoing generalization arrows suggests the presence of an error. Common cases in which multiple inheritance is encouraged, such as for the Composite Pattern [4] shown in Figure 8, can be detected as special cases.

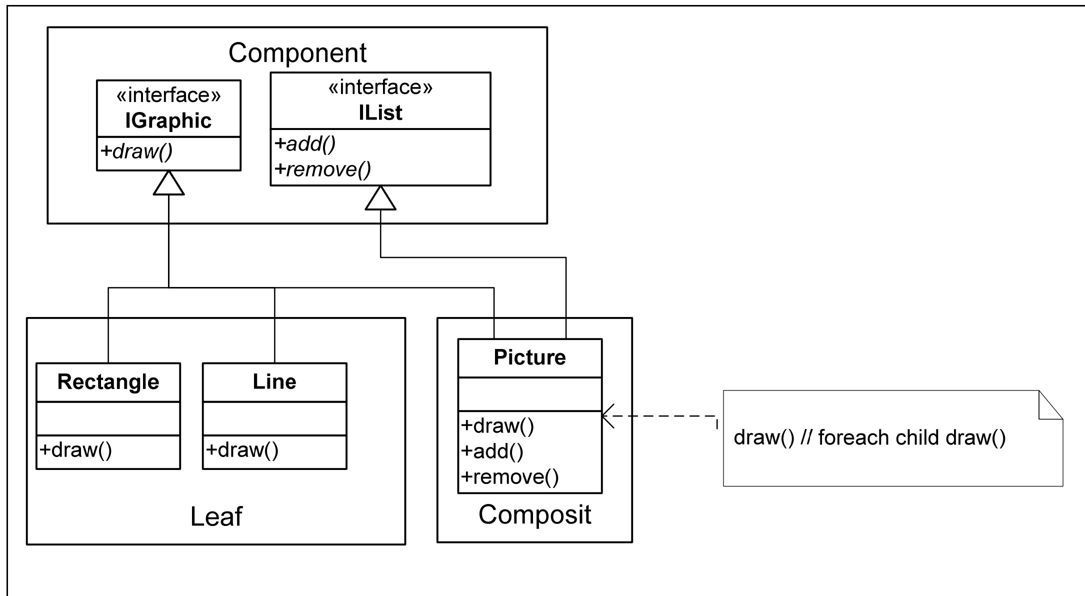

Figure 8: An application of the Composite Pattern in which Picture inherits two interfaces

4.9 Duplicate Operations in Multiple Classes

Description: It is relatively rare for an operation to be duplicated in multiple classes (except to support inheritance) in class projects. Duplicated operations often indicate poor class cohesion or misplaced operations. While duplicated operations certainly do not indicate an actual defect, they can trigger a message discussing the concern.

Detection: This type of defect can be detected by scanning the Rose model file for duplicate operation names. The danger is leading students to believe that all duplication is invalid. Some generic operations, such as "sort" or "find" are likely to appear in several classes. Filtering out common operation names is likely to be necessary to avoid teaching students invalid concepts.

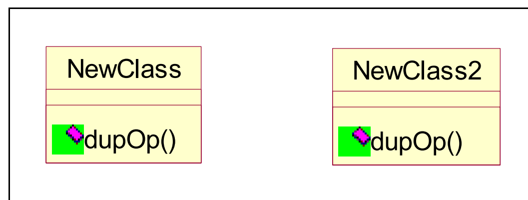

Figure 9: UML of two classes with the same operation

4.10 Classes without Attributes and/or Operations

Description: Once a model gets to the design or implementation phases, it should have at least one unique attribute or operation. Classes without attributes or operations imply either that the class may have been motivated by the analysis but not be relevant to the final system or that the model is incomplete.

Detection: This type of defect can be spotted by scanning the Rose MDL file for classes that do not have "(object operation…" for class operations or "(object ClassAttribute…" class attribute entries.

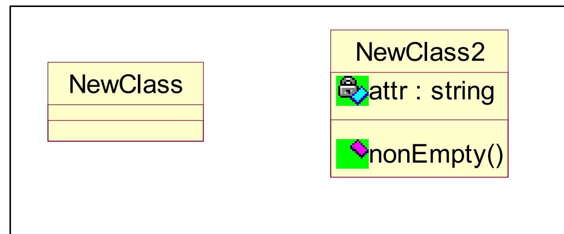

Figure 10: UML of an empty and non-empty class. The non-empty class contains the operation nonEmpty( ).

```
root_category     (object Class_Category "Logical View"
        quid      "47BD9B0402AA"
        exportControl     "Public"
        global    TRUE
        subsystem         "Component View"
        quidu             "47BD9B0402AC"
        logical_models    (list unit_reference_list
          (object Class "NewClass"    // empty Class
                quid      "47CC0AAE03D8")
          (object Class "NewClass2"   // non-empty Class
                quid      "47CC0AB102EE"
                operations        (list Operations
                  (object Operation "nonEmpty"  // non-empty Class has an Operation
                        quid      "47CC10170157"
                        result    "void"
                        concurrency       "Sequential"
                        opExportControl  "Public"
                        uid       0))
                class_attributes  (list class_attribute_list
                  (object ClassAttribute "attr"  // non-empty Class has an Attribute
                        quid      "47CC204E0177"
                        type      "string"))))
```

Figure 11: Rose MDL file snippet of empty and non-empty classes.

4.11 Classes that are not Associated with Other Classes

Description: Classes interact with each other, providing and using services. To be useful in a system, classes need to be associated with other classes.

Detection: This type of defect can be spotted by scanning the Rose MDL for classes which are never referenced in the "(object Association . . ." lists. See the lines annotated with "//" comments in Figure 12.

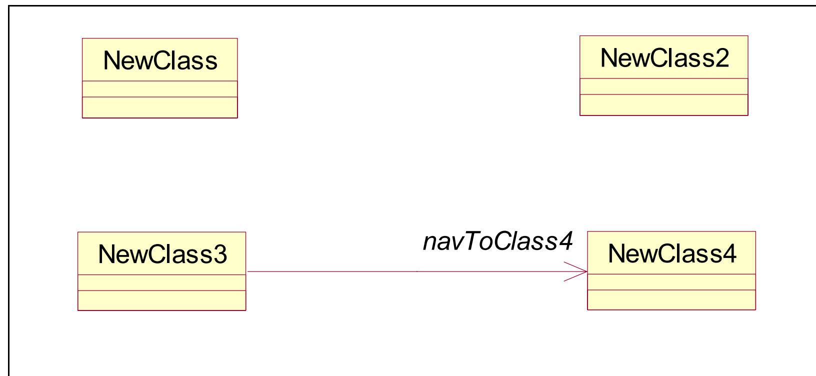

Figure 12: UML example of two classes without associations and two classes with associations.

```
root_category     (object Class_Category "Logical View"
        quid      "47BD9B0402AA"
        exportControl     "Public"
        global    TRUE
        subsystem         "Component View"
        quidu             "47BD9B0402AC"
        logical_models    (list unit_reference_list
          (object Class "NewClass"
                quid      "47CC0AAE03D8")
          (object Class "NewClass2"
                quid      "47CC0AB102EE")
          (object Class "NewClass3"
                quid      "47CC21E900DA")
          (object Class "NewClass4"
                quid      "47CC21EE038A")
          (object Association "navToClass4"
                quid      "47CC21F501B5"
                roles     (list role_list
                  (object Role "$UNNAMED$0"
                        quid      "47CC21F60109"
                        supplier          "Logical View::NewClass4"  // Associated Class
                        quidu             "47CC21EE038A"
                        is_navigable      TRUE)
                  (object Role "$UNNAMED$1"
                        quid      "47CC21F6010B"
                        supplier          "Logical View::NewClass3"  // Associated Class
                        quidu             "47CC21E900DA"))))
```

Figure 13: MDL file of two classes without associations (NewClass1 and NewClass2) and two classes with associations (NewClass3 and NewClass4).

4.12 Very High Class Coupling

Description: Good object-oriented design strives for low coupling and high cohesion. High coupling is associated increased maintenance costs because when one class changes, the coupled classes are more likely to require changes.

Detection: The detection of high coupling is rather subjective in that the amount of acceptable coupling depends on the problem domain. However, a coupling metric can be computed for each class by processing the Rose MDL file associations and counting the number time each class name appears in the "(object Association(roles (object Role supplier)))" fields. A simple statistical analysis can be used to indicate which classes might be candidates for being coupled to too many others.

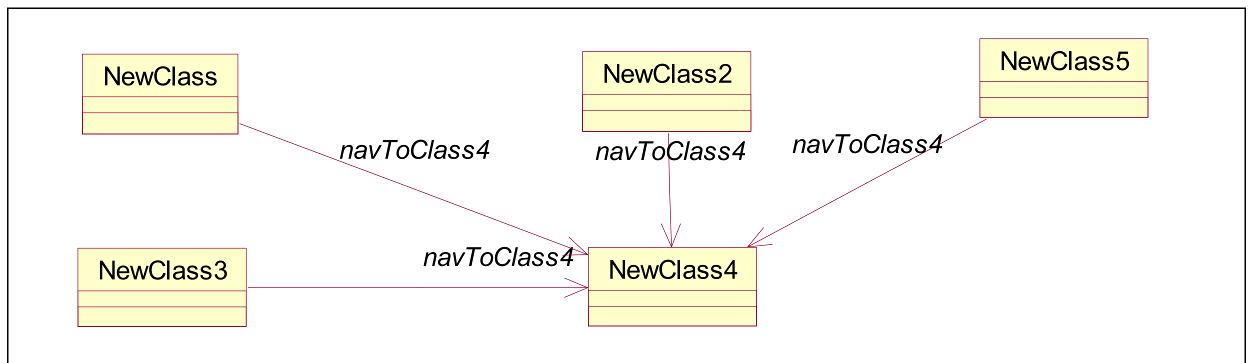

Figure 14: UML showing NewClass4 associated with the other four classes – high coupling.

```
logical_models    (list unit_reference_list
        (object Class "NewClass"
              quid      "47CC0AAE03D8")
        (object Class "NewClass2"
              quid      "47CC0AB102EE")
        (object Class "NewClass3"
              quid      "47CC21E900DA")
        (object Class "NewClass4"
              quid      "47CC21EE038A")
        (object Class "NewClass5"
              quid      "47CC24F702AF")
        (object Association "navToClass4"
              quid      "47CC21F501B5"
              roles     (list role_list
                 (object Role "$UNNAMED$0"
                        quid      "47CC21F60109"
                        supplier          "Logical View::NewClass4"
                        quidu             "47CC21EE038A"
                        is_navigable      TRUE)
                 (object Role "$UNNAMED$1"
                        quid      "47CC21F6010B"
                        supplier          "Logical View::NewClass3" // NewClass3 associated with
                                                                    // NewClass4
```

```
                    quidu               "47CC21E900DA")))
        (object Association "navToClass4"
                quid    "47CC24E8034B"
                roles   (list role_list
                   (object Role "$UNNAMED$2"
                        quid    "47CC24EA01C5"
                        supplier            "Logical View::NewClass4"
                        quidu               "47CC21EE038A"
                        is_navigable        TRUE)
                   (object Role "$UNNAMED$3"
                        quid    "47CC24EA01C7"
                        supplier            "Logical View::NewClass" // NewClass  associated with
                                                                     // NewClass4

                        quidu               "47CC0AAE03D8")))
        (object Association "navToClass4"
                quid    "47CC24EE0148"
                roles   (list role_list
                   (object Role "$UNNAMED$4"
                        quid    "47CC24F100CB"
                        supplier            "Logical View::NewClass4"
                        quidu               "47CC21EE038A"
                        is_navigable        TRUE)
                   (object Role "$UNNAMED$5"
                        quid    "47CC24F100CD"
                        supplier            "Logical View::NewClass2" // NewClass2 associated with
                                                                      // NewClass4

                        quidu               "47CC0AB102EE")))
        (object Association "navToClass4"
                quid    "47CC24FE02AF"
                roles   (list role_list
                   (object Role "$UNNAMED$6"
                        quid    "47CC2500000F"
                        supplier            "Logical View::NewClass4"
                        quidu               "47CC21EE038A"
                        is_navigable        TRUE)
                   (object Role "$UNNAMED$7"
                        quid    "47CC25000011"
                        supplier            "Logical View::NewClass5" // NewClass5 associated with
                                                                      // NewClass4
                        quidu               "47CC24F702AF"))))
```

Figure 15: Rose MDL file showing NewClass4 associated with the four other classes.

5 Interaction and State Model Defects

This section discusses the most significant defect in dynamic diagrams: failing to be consistent with static diagrams. Identifying additional issues is left as future work.

5.1 Message Arcs and Class/Objects that do not Correspond to the Class Model

Description: In a project, all of the object-oriented models model the same domain objects and such should be based on the same model components. Many students fail to make this connection. As a result, they build each model from scratch and ignore the work already done in previous modeling. Frequently, students will have wonderfully refined class models and then use different identifiers for classes, attributes and operations when producing interaction and state models. On the other hand, Rose and many other design tools provide mechanisms for ensuring consistency between diagram types. In Rose, drop-down menus give appropriate suggestions based on components from the class model – all a developer needs to do is click on the appropriate identifier. Students need encouragement to use such assistance.

Detection: The detection of this defect can be achieved by scanning the interaction and state model parts of the Rose MDL file to determine if all components are already part of the class model.

6 Conclusion

We have identified a number of frequent errors made by students when constructing UML diagrams. This list is certainly not intended to be exhaustive, but in our experience these defects have the distinction of being both easily recognized (at least by instructors) and very common. Future plans include developing tools to automatically recognize many of these defects. The intent is that students would use the tools to get anytime feedback on their models, presumably resulting in improved submissions. Thus the goal is an automated assistant: developing a system to identify relatively simple errors. This will hopefully allow instructors to spend more time on more significant issues.

References

[1] Cleidson, R. B., et al., Using Critiquing Systems for Inconsistency Detection in Software Engineering Models. SEKE 2003, pp. 196-203.

[2] Coelho, W. and Murphy, G., ClassCompass: A Software Design Mentoring System. *ACM Journal on Educational Resources in Computing*, Vol. 7, No. 1, Article 2, March 2007.

[3] Egyed, A., UML Analyzer Tool. Information available at http://www.alexander-egyed.com/tools/uml_analyzer_tool.html. Accessed March 7, 2008.

[4] Gamma, Helm, Johnson, and Vlissedes, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[5] Holland, S., Griffiths, R., and Woodman, M., Avoiding Object Misconceptions. *SIGCSE Bull.* 29, 1 (Mar. 1997), pp. 131-134.

[6] IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specifications, http://standards.ieee.org/reading/ieee/std_public/description/se/830-1998_desc.html, IEEE, 1998.

[7] Kaneiwa, K., and Satoh, K., Consistency Checking Algorithms for Restricted UML Class Diagrams. In Proceedings of the Fourth International Symposium on Foundations of Information and Knowledge Systems (FoIKS 2006), *Lecture Notes in Computer Science,* Volume 3861, Springer-Verlag, 2006, pp. 219-239.

[8] Konrad, S. and Cheng, B.H.C., Automated Analysis of Natural Language Properties for UML Models. *Lecture Notes in Computer Science*, Volume 3844, Springer-Verlag, 2006, pp. 48-57.

[9] Lange, C., Improving the Quality of UML Models in Practice. In *Proceedings of the 28th international Conference on Software Engineering* (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM, New York, NY, 993-996.

[10] Lindland, O., Sindre, G., Understanding Quality in Conceptual Modeling. *IEEE Software*, March 1994, pp. 42-49.

[11] Pap, Zs., Majzik, I., Pataricza, A., and Szegi, A., Completeness and Consistency Analysis of UML Statechart Specifications. In Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'2001), Győr, Hungary, 18-20 April, 2001, pp. 83-90.

[12] Pilskalns, O., and Andrews, A., Rigorous Testing by Merging Structural and Behavioral UML Representations. In 6th International Conference on the Unified Modeling Language (UML 2003), San Francisco, USA, October 20-24, 2003.

[13] Thomasson, B., Ratcliffe, M., and Thomas, L., Identifying Novice Difficulties in Object Oriented Design. ITiCSE'06, June 26-28, 2006, Bologna, Italy, pp. 28-32.

[14] Sanders, K., and Thomas, L., Checklists for Grading Object-Oriented CS1 Programs: concepts and misconceptions. ITiCSE '07, June 23-27, 2007, Dundee, Scotland, pp. 166-170.