

Generating and Evaluating Object-Oriented Designs for Instructors and Novice Students

Sally Moritz and Glenn Blank¹
Computer Science and Engineering Department
Lehigh University
Bethlehem, PA 18015 USA
001-610-758-4085, 001-610-758-4867
{sgh2, gdb0}@lehigh.edu

Abstract. Creating object-oriented designs for even simple problems of the type assigned to beginning students is a challenging task. While rules can be applied to identify key design components, multiple acceptable solutions often exist, and instructors may have different preferences for which variation(s) are best. We describe a tool we developed to assist instructors in designing solutions for problems to be assigned to novice CS1 students. The tool generates a design template of acceptable solutions which the instructor may modify and add comments. Given an updated template, the Expert Evaluator component of DesignFirst-ITS assists novice students as they design a class diagram that models the assigned problem.

Keywords: intelligent tutoring system, unified modeling language, object-oriented design, CS1, natural language parsing.

1 Introduction

Learning object-oriented design and programming is a challenging task for beginning students. Data collected from CS1 courses in numerous studies [7, 12] have shown that, in spite of new curricula and tools that focus on object concepts, many students still have difficulty understanding and applying these concepts to solve problems. High school teachers find gaining the necessary mastery of the subject challenging. Reaching high school students is a must if we are to attract able students to the field.

Our approach introduces the design of UML (Unified Modeling Language) very early in the CS1 curriculum. In the “design-first” curriculum [9], students learn how to create a class diagram that represents the solution for a problem. Students learn procedural code only after they generate code stubs from the class diagram, as they implement each design element. Many tools generate code stubs from class diagrams. But students often need guidance as they translate a problem description into a design. We developed DesignFirst-ITS, an intelligent tutoring system (ITS) that observes students as they create UML class diagrams and offers assistance when they need it.

¹ This work was supported by the National Science Foundation under Grant No. EIA-02317687 and the Pennsylvania Infrastructure Technology Association (PITA).

This paper focuses on two components of DesignFirst-ITS: 1) an Instructor Tool and Solution Generator, which lets an instructor enter a problem description in English, then generates a solution template, and 2) an Expert Evaluator, which compares a student's actions to the template. The solution template represents class diagrams that model multiple valid designs for the problem. Through a web-based front end, the instructor can modify the template and add his own comments to be displayed to students in DesignFirst-ITS. Thus, the Solution Generator provides for multiple valid design variations, while allowing the instructor to incorporate his own preferences into the final model of acceptable solutions and providing a mechanism for the instructor's voice to be included in the feedback given to the student.

2 Related Work

Like DesignFirst-ITS, Collect-UML helps students who create class diagrams from problem description [3]. When a student adds a design element to her diagram, she selects a name by highlighting a word or phrase in the problem description. No free-form entry of element names is allowed. This approach avoids the need to understand the student's intent from natural language terms. A pedagogical drawback of this approach is that the student must eventually learn how to transfer to a less structured problem solving environment. DesignFirst-ITS, on the other hand, lets a student use natural language terms and abbreviations to develop her own UML class diagram.

Collect-UML evaluates student solutions by applying a set of constraints which must hold. Feedback text that explains the error in general terms is stored with each constraint. Since constraints do not model problem-solving knowledge, Collect-UML offers feedback offered only when a student submits a complete or partial solution, which may be too late to identify the flaw in the student's process. Constraint Based Modeling researchers have shown that a system can include constraints to be applied at intermediate steps in the problem-solving process; the NORMIT tutor uses this technique [8]. However, this approach cannot easily provide feedback after each solution step or suggest a next step to a student who is stuck. We believe that modeling the problem-solving process is crucial to help novices.

We studied software engineering research to define a process for translating a problem description to an object-oriented design. A long popular technique has been to identify potential classes, attributes and methods using parts-of-speech analysis. Abbott defined a model for identifying parts of speech in a problem specification and using them to derive the components of an Ada program [1]. Booch suggested that parts of speech, such as nouns and verbs, were often good candidates for design elements such as objects and methods [4]. Object Modeling Technique [13] defined a process to build a list of candidate classes from the nouns in a problem description and apply a set of rules to identify those that did not belong. It went on to define how to identify attributes and methods for the remaining classes.

These ideas have been used in automated tools to assist experienced analysts and developers in the design process. Examples of such tools include Circe [2], LIDA [10] and Metafor [6]. Although used as brainstorming tools, none were applied to educating students in learning object-oriented design.

3 The Instructor Tool and Solution Generator

The Instructor Tool lets teachers enter problem descriptions in English. The Solution Generator then automatically generates class diagrams as potential solutions. The Instructor Tool displays the results and lets the teacher view details (the source of each method or attribute in the problem description) and delete elements. The Tool aids the teacher in creating clear and concise problem descriptions for students. It also allows her to revise the solution based on her preferences, and to add her own voice to the ITS by entering comments and explanations to be used as feedback to the student.

The Solution Generator incorporates two external software packages: the MontyLingua natural language processor [6] and the WordNet lexical database [5]. MontyLingua identifies verb/subject/object tuples, which are in turn the basis for extracting classes and their attributes and methods. It also performs part-of-speech tagging, which the Solution Generator uses to determine the role each word should play in the design. The parts of speech are also passed to WordNet to find the definition that matches the word's use within the text. Natural language processing is difficult, and the current state of the art still has shortcomings. Thus it is not surprising to find a number of limitations in MontyLingua. The most notable are compound clauses, dependent clauses and appositives. Rules for structuring the problem text so as to avoid these shortcomings are outlined in an Instructor Tool User Manual.

WordNet provides services similar to a dictionary and thesaurus. It groups nouns, verbs, adjectives and adverbs into synonym sets. The Solution Generator uses WordNet to identify semantic information, such as nouns that are Actors, by searching for phrases in a WordNet definition that indicate a person, such as "someone who..." or "a person who..." When multiple definitions exist for a word, the Instructor Tool chooses the most common use; the teacher can select a different definition and regenerate the solution from the problem description based on the new definition. WordNet also identifies synonyms within the context of the problem, which the Instructor Tool suggests as synonyms to the teacher as she reviews the generated solutions. The synonyms that are accepted by the instructor are saved as "related terms" for the component in the solution template. The Expert Evaluator compares a student entry against these related terms when looking for a match in a solution template. JWordNet, a Java-based WordNet implementation developed at George Washington University by Kunal Johar, was used to integrate WordNet into the Instructor Tool (www.seas.gwu.edu/~simhaweb/software/jwordnet/).

After using MontyLingua to parse tuples from text, the Solution Generator algorithm tags sentence subjects and objects as actors, attributes (based on the WordNet definition, plus a database of simple values, such as money, amount, balance), methods (if a noun that is a verb form) and potential classes. Verbs are tagged as potential methods. Classes for which no attributes or methods are defined are removed. It adds optional get and set methods for each attribute that doesn't already have one plus standard classes for character-based or graphical user interface.

The generated design typically has more attributes and methods than required. The teacher can delete such elements with the Instructor Tool, providing explanations why they are not appropriate. The instructor can mark an element optional, supplying comments explaining why it is allowed but not optimal.

4 The Expert Evaluator

DesignFirst-ITS provides a relatively unstructured environment that allows students freedom to build their designs. Students design classes, methods and attributes in the LehighUML editor, which we developed as a plug-in for the Eclipse integrated development environment. (A stand-alone version of LehighUML has also been created, for use outside of the complex Eclipse environment.) As the student designs a solution for a given problem in the plug-in environment, LehighUML reports each student action in a database on a server. DesignFirst-ITS analyzes student actions and provides feedback to the student as necessary.

The Expert Evaluator (EE) evaluates each of the student's steps in the background by comparing it with the solution template, and generates an information packet for a correct student action and an error packet for an incorrect action. The packet also contains the concept applied, the type of error committed, and a recommended alternative to the incorrect action. The Student Model analyzes these packets to determine the knowledge level of the student for each concept and attempts to find reasons for student errors [14]. The Student Model updates the student profile and passes the EE's packets along with a reason packet that contains possible reasons for the student's error to the Pedagogical Advisor [11].

The EE uses the following algorithm to match the student's component name to a name in the solution template. First, it compares the student's component with each solution component and its related terms. A solution component name may be all or part of the student's name. Abbreviations and spelling errors for each term are considered by applying two string similarity metrics from the SimMetrics open source algorithms (www.dcs.shef.ac.uk/~sam/stringmetrics.html). If none exceed the similarity criteria, there is no match. If there is more than one match, then we consider adjectives to break ties; for example, distinguishing between attributes such as "customer balance" and "total balance." If the student's element matches an expert's deleted component, mis-matches a components (e.g., a student's attribute matches an expert's method) or if it has the wrong data or return type, generate an error packet; otherwise generate an info packet. If the student element does not match anything, generate an unknown element error packet.

Consider an example scenario. Suppose a student has successfully added a class she calls TicketMachine. She has also added valid attributes movie and price. Now she adds an attribute cash with data type int to the class TicketMachine. The EE attempts to match cash to attributes of TicketMachine not yet matched, which now includes (customer) money. It then compares cash to money and its related terms. WordNet did not return cash as a synonym when the Solution Generator added the attribute money. So what happens next depends on whether the instructor entered cash as a related term of money. If cash does match, then the EE makes sure that it is not a duplicate in the student's complete solution. Then it compares the data type for the matched element (in this case double) to the student's data type (int). In this case, int is not listed as an acceptable data type, so the EE creates an error packet with the code INC_DATATYPE. On the other hand, if cash does not match as a related term, then the EE compares cash to attributes for other classes, then to other classes, parameters, actors and methods. If there were a match, an error packet would indicate any match that is found (there is none); otherwise, an error packet indicates an unknown element.

5 Evaluation

Five computer science teachers evaluated the Instructor Tool and Solution Generator. The experts were given the Instructor Tool User Manual, an overview of DesignFirst-ITS and the Instructor Tool, rules for writing a problem description and examples. The evaluators then each entered one or two problems of their own invention, using the tool to refine the description and the solution until it met with their own satisfaction. They then completed an interview and questionnaire.

The teacher evaluations of the Solution Generator focused on how well the software's interpretation of the problem description matched the teacher's intent and the quality of the resulting class diagram. The problem descriptions they entered provided examples of different types of problems as well as different language styles, which was invaluable in measuring and improving the Solution Generator's ability to interpret a range of voices and perspectives.

The teachers listed specific difficulties or bugs they found in solution generation. These most frequently listed had to do with MontyLingua's sentence parsing. Sometimes MontyLingua correctly parsed subordinate clauses such as "The team that accumulates the most points wins" but sometimes it did not. The most common result was that the information contained in the subordinate clause was dropped. The teachers were able to work around the problem by rewording their text. However, avoiding subordinate clauses completely is not always possible, even for the most simple subjects. Natural language parsing is where the most improvements can be made as better software becomes available. A second source for bugs was pronoun resolution: the simple strategy of using the subject of the prior sentence as a pronoun's antecedent worked well except for one case in which the antecedent was taken from the subject of a subordinate clause. A third source of difficulties was interpretation or word meanings. In one case, WordNet did not return a valid definition. This happened for the word "alarm" – a simple word which one would expect to be found even in an abridged version of the WordNet database. WordNet matched another word. One remedy is to download a larger version of the WordNet database; another remedy is that the instructor can add terms with definitions and related terms. In a second case, WordNet returned a set of suitable definitions for the word, but the most common definition was not appropriate for the context. This occurred in a problem description of a basketball game: "The team to accumulate the most points wins." WordNet's most frequently used sense for point is "A geometric element that has position but no extension." But the appropriate definition in this context is "The unit of counting in scoring a game or contest." This difficulty isn't truly a bug; understanding the context of the problem is beyond the scope of the Solution Generator. In this case, the instructor was able to choose the correct sense for the word and regenerate the related terms, adjectives and data type for the element.

The bugs detected represented a small portion of the teachers' text. They did not prevent successful use of the tool, as all three evaluators were able to create an acceptable solution. The evaluators all rated the quality of the solution as a 4 on a scale of 1 to 5 (1 being poor, 5 being excellent), and all said the solution generation met or exceeded their expectations. One commented that learning how to properly word the problems for optimal interpretation would take some time, but the learning curve was not too cumbersome. Another suggested adding other commonly used

methods, such as toString and equals. They observed that the tool encouraged instructors to think more deeply about how the problem should be worded so that students have a clear and complete understanding and that the tool created a better solution than the teacher had initially developed.

The EE was tested with 33 students in a CS1 course at Lehigh University in November 2006. These students attended an optional workshop in which they used the DesignFirst-ITS to create a class diagram for the Movie Ticket Machine problem. Every action taken by each student was recorded in the ITS database.

Table 1: Expert Evaluator Results

Number of Actions	Number EE Marked Correct	Number EE Marked Correct in Error	Number EE Marked Incorrect	Number EE Marked Incorrect in Error	Number EE Marked Unknown
1035	540	16	454	79	41

For 1035 student actions, the EE had 16 false positives (3%), 79 false negatives (17%) and 41 unknowns (4%), for an overall error rate of 13%. An accuracy rate of 87% is quite high and acceptable for the overall performance of DesignFirst-ITS.

The EE marks an action as unknown when it cannot match the action to any component within the solution template. In this population of data, this included the use of words not anticipated by WordNet or the instructor (“pay” for an “enter money” method, “dollars” for “money”); the use of single-letter or nonsense component names (“p”, “asdf”); and the use of abbreviations not recognized by the string similarity metrics (“tix” for “tickets”).

When the EE identified an action as correct, it was right 97% of the time; only 3% of the actions marked correct by the EE were not. This makes sense; most matches were on terms taken directly from the problem description. All of the instances in which actions were erroneously marked correct involved ambiguous names used by the student. “Display” or “enter” are only partial method names; they should indicate what is to be displayed or entered. When the EE encounters ambiguous names, it tries to match on data type or return type. If that does not resolve the ambiguity, it chooses a match at random. A better strategy may be to mark the action unknown to allow the Pedagogical Advisor to ask the student for clarification.

When the EE identified an action as incorrect, 17% of the time the action should have been accepted as correct. There are two main causes of an incorrect diagnosis: a misinterpretation of the student’s action, or a failure to recognize an acceptable solution that the action represents. There were only a few unique errors that occurred within this population of students; those same errors occurred many times, and in some cases triggered other errors (such as when an error with an attribute leads to errors interpreting methods involving the attribute).

All of the misinterpretation errors occurred with components that had very similar names or were similar in purpose. For example, there was some confusion over “number of available seats/tickets” as an attribute and the number of tickets requested by the customer. These are difficult even for a human instructor to resolve, especially when students would need to use long names to distinguish between them. This

trickled down to confusion with methods similar to “get” and “set” functions for these attributes—such as “enterNumberTickets,” which the student may have intended as a method to request the number of tickets the customer wants rather than set the attribute representing the number of tickets available for sale. A better way to handle these cases might be to mark such components as unknown and allow the Pedagogical Advisor to ask the student for clarification. The EE can store the closest match in the recommended action fields, so the Advisor can ask “Did you mean...?”

There were three distinct errors classified as failure to recognize alternate solutions. One involved the return type of the “print tickets” method. Fourteen students coded String as its return type, which was not allowed as an optional value in the solution template. In this case the teacher who taught the class had not prepared the solution template in the Instructor Tool, and the question of what data is returned from the print ticket method is wide open to interpretation. It is the instructor’s prerogative to allow more or less flexibility in the finer details of a solution.

A second variation entered by one student includes two attributes related to the number of seats in the theater. A student had “seatsAvailable” and “seatsRemain” as attributes; the EE marked the second as a duplicate of the first. An instructor might agree that there is no need for two attributes. Or an instructor might want to accept both attributes, and could do so by adding wording to the description such as “The machine keeps track of the total number of seats in the theater, and the number of seats remaining.”

Overall, the error rate in the data collected does not significantly hinder the useful operation of the ITS. The error rate presented could be improved by using the Instructor Tool to accept simple variations and synonyms that were rejected. In actual classroom use, this would be a routine and expected task for the instructor. There were four simple changes that would reasonably be made for the Ticket Machine problem: 1) adding alternate return data types for the printTickets method; 2) adding “tix” as a synonym for “tickets”; 3) adding “cash” as a synonym for “money”; 4) adding “dollars” as a synonym for “money.” The first change alone would yield 14 fewer errors; the second 7, the third and fourth one each. The result for the input student actions would be 56 actions marked in error incorrectly, instead of 79. These changes would improve the overall error rate to 11%, or an 89% accuracy rate, including unknowns (93% accuracy excluding unknowns).

A simple revision to the EE’s logic that would also improve its accuracy would be to pass ambiguous elements to the Pedagogical Advisor for clarification instead of making best guess matches. This should not be too annoying for the student. Indeed, asking the student for clarification more often might be beneficial in inducing the student to think more deeply about her design elements, and thus improve learning.

6 Conclusions

The research summarized in this paper applied software engineering principles to develop tools that assist both students and teachers in learning object-oriented design. The Instructor Tool analyzes problem descriptions in English and generates a class diagram representing a design template that supports multiple acceptable solution

variations. Three experts found that the tool generated valid design elements, including some which they had not anticipated. Experts rated both the tool's ease of use and completeness of functionality as a 4 on a scale of 1 to 5.

The EE analyzes student actions with an accuracy of 87%. DesignFirst-ITS lets students solve problems naturally, using their own terms in their UML design. Students are unencumbered by scaffolding they don't need; students who have difficulties are provided with assistance as they need it. Students are not locked into a single solution; multiple solutions that are represented in the problem description are recognized. Feedback on student errors is relevant to the design process, since recommended actions are based on the Solution Generator's analysis.

The tools, curricula and dissertation are available at designfirst.cse.lehigh.edu.

References

1. Abbott, R.: Program Design by Informal English Descriptions. In: Communications of the ACM, November 1983, pp. 882-894. (1983)
2. Ambriola, V., Gervasi, V.: Processing Natural Language Requirements. In: Proceedings of the 12th International Conference on Automated Software Engineering, pp. 36-45 (1997)
3. Baghaei, N., Mitrovic, A., & Irwin, W.: Problem-Solving Support in a Constraint-based Intelligent System for Unified Modeling Language. In: Technology, Instruction, Cognition and Learning Journal, Vol 4, No 1-2 (2006)
4. Booch, G.: Object-Oriented Development. In: IEEE Trans. on Software Engineering, 12(2), pp. 211-221 (1986)
5. Fellbaum, C.: WordNet: An Electronic Lexical Database. MIT Press (1998)
6. Liu, H., Iebberman, H.: Metaphor: Visualizing Stories as Code. Proceedings of the ACM International Conference on Intelligent User Interfaces, pp. 305-307 (2005)
7. McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagen, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., Wilusz, T.: A Multi-National Study of Assessment of Programming Skills of First Year CS Students. In: SIGCSE Bulletin 33(4), pp. 125-140 (2001)
8. Mitrovic, A., Martin, B., & Suraweera, P.: Intelligent Tutors for All: The Constraint-Based Approach. In: Intelligent Systems: IEEE, Vol. 22, No. 4, pp. 38-45 (2007)
9. Moritz, S., Blank, G.: A Design-First Curriculum for Teaching Java in a CS1 Course. In: ACM SIGCSE Bulletin (inroads), June 2005, pp. 89-93 (2005)
10. Overmyer S.P., Lavoie B., & Rambow O.: Conceptual Modeling through Linguistic Analysis Using LIDA. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), pp. 401-410 (2001)
11. Parvez, S., Blank, G.: Individualizing Tutoring with Learning Style Based Feedback. In: Proc. 9th International Conference on Intelligent Tutoring Systems (2008)
12. Ratcliffe, M., Thomas, L.: Improving the Teaching of Introductory Programming by Assisting Strugglers. In: Proc. 33rd ACM SIGCSE Technical Symposium (2002)
13. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice Hall (1991)
14. Wei, F., Blank, G.: Student Modeling with Atomic Bayesian Networks. In: Proc. of 8th International Conference on Intelligent Tutoring Systems, pp. 491-502 (2006)