Generating and Evaluating Object-Oriented Designs

in an Intelligent Tutoring System

by

Sally H. Moritz

Presented to the Graduate and Research Committee
of Lehigh University
in Candidacy for the Degree of
Doctor of Philosophy

in

Computer Science

Lehigh University

March 17, 2008

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

April 2, 2008
Date

Glenn D Blank
Dissertation Director

April 3, 2008
Accepted Date

Committee Members:

Glenn D Blank
Dr. Glenn Blank, Chair

Dr. Alec Bodzin

Jeff Heflin
Dr. Jeff Heflin

Edwin J Kay
Dr. Edwin Kay

Dr. Hector Munoz-Avila

Acknowledgements

I wish to thank my husband, Craig Moritz, for his work proofreading this dissertation, as well as his past and continuing support and encouragement. I also wish to thank Shahida Parvez for her contributions to the DesignFirst-ITS project and her friendship as we both completed our research.

Table of Contents

List of Tables

List of Figures

Abstract

Learning object-oriented design and programming is a challenging task for many beginning students. In recent years, this has motivated the development of new curricula and tools to support their pedagogies. However, student achievement in first-semester courses, as well as enrollment and retention rates (especially for women and minorities), remains a concern. New tools and techniques are needed to engage students and enable more of them to be successful in a CS1 course. This need inspired two avenues of research for this dissertation: a new "design first" curriculum that uses elements of Unified Modeling Language (UML) to teach students problem-solving and design skills before procedural code, and an intelligent tutoring system (ITS) that observes students as they create UML class diagrams and offers customized assistance when they need it.

An ITS must be able to solve the problems that are presented to the student, or at least evaluate the student's work. Several models exist for accomplishing this task, but all are limited in the number of different solutions they accept as correct. This is not a problem in many domains, but it is in object-oriented design. For any given problem, there are often many different solutions that are acceptable. While scaffolding is desirable to guide novices through a new procedure, students who follow a different but fruitful path should not be deterred. The goal of this research is to build an expert module for an ITS for object-oriented design that can generate its own solutions, and also recognize and accept a variety of valid solutions. A working expert module of an ITS for object-oriented design was built. A tool to assist teachers in the creation of student exercises and valid solution models was also produced.

I. Introduction

The increasing importance of object-oriented design in modern software development has made it a crucial skill for computer science students to learn. Yet it is a complex activity, both for beginners who are learning to create small, standalone programs and for professional developers who work on large, interconnected systems. Data collected from CS1 courses in numerous studies (McKracken et al, 2001), (Ratcliffe & Thomas, 2002) have shown that many students have difficulty understanding object concepts and applying these concepts to solve problems. One approach to addressing this problem is through the curriculum. A variety of curricula have been developed (Bruce 2004), with varying degrees of success. The foci of these curricula range from teaching procedural coding skills before introducing objects, to teaching object concepts first. Many tools to support these curricula have also been developed, such as specialized integrated development environments (IDEs) that simplify the coding and testing processes and help students visualize concepts and experiment with code.

Both anecdotal and experimental evidence supports the hypothesis that the "objects first" approach is effective in teaching Java programming (Kölling & Rosenberg, 2001), (Decker, 2003), (Blank et al, 2003). However, this approach still emphasizes coding and syntax rather than problem solving. When presented with a problem, many students don't know where to begin. They need guidance on how to analyze a problem and develop a plan for solving it. This idea led to the development of a "Design First" curriculum for Java (Moritz & Blank, 2005), (Moritz et al, 2005), which was taught and evaluated over two semesters at an inner-city high school.

Students learned to use elements of UML (use cases and class diagrams) to design a solution to a problem before writing code. Objects were still introduced early, and object concepts were emphasized throughout the semester. The students also used an IDE with a component that allowed entry of class diagrams and generated code stubs from them. The curriculum and its evaluation are described in detail in a later section.

Teaching standard design practices such as UML presents the appropriate skills, but adds yet more content to an already challenging curriculum. The high school students in the Design First with Java classes did well, but they benefited from a low student-teacher ratio (two teachers for 13 students). This is not typical in high school or college settings. Also, students often work on assignments outside of the classroom and at times when instructors, teaching assistants and other students are not available. Tools like the UML interface used in the high school classes simplify entry of a design and hide some syntax complexities by generating code stubs, but they do not give advice, guide a student to a good design, nor evaluate a student's work. Adding such capabilities could make a difference to a student who is struggling, help one progress even when a teacher or other students are not available, and provide a critical resource for distance learners. These are the primary goals of an intelligent tutoring system, or ITS.

Intelligent Tutoring Systems

An ITS is educational software with one or more intelligent components. Most ITS possess expert knowledge of the instructional domain, and the ability to apply that knowledge to solve the problems assigned to the student. That knowledge is also used

to analyze the student's solution, identify the student's reasoning, and determine from the student's errors what it is he misunderstands. This information is then used to provide customized feedback to the student. An ITS also keeps a history of student performance so that the student's progress can be monitored, and subject matter that the student does not understand can be reviewed or presented again in different ways.

Research on ITS has been ongoing since the 1970s, and successful systems have been developed in areas such as high school algebra and geometry, college physics, and programming languages. Evaluations of such systems have shown that while still not as effective as a human tutor, ITS can result in much quicker and deeper learning than using only traditional methods such as workbooks, lectures, and standard computer aided instruction. Advances in artificial intelligence, as well as faster and cheaper hardware, have enabled the development of more sophisticated systems, and have lowered their cost, making them more feasible for use in public schools as well as other educational institutions. ITS have been most successful in domains which are well defined and finite in scope. But they have also proven beneficial in complex problem-solving domains. For example, the Andes tutor has been demonstrated to improve test scores in college-level introductory physics, especially for humanities majors with less prior experience than science or engineering majors in the same course (Gertner & VanLehn, 2000).

A number of successful ITS have been built in the domain of computer programming. These include ELM (Weber & Schult, 1998) and LISPITS (Corbett & Anderson, 1992) for LISP; MENO-II for Pascal (Soloway et al, 1981); and tutors for concepts in C++ (Kumar 2002) and (Kostadivov & Kumar, 2003). A few tutors for

Java have recently been built or are under development, including an expression evaluation tutor (Kumar, 2005) and JITS (Sykes & Franek, 2004). All of these tutors, including those for C++ and Java, teach procedural concepts; none addresses object-oriented design.

This research discovered only one ITS that teaches object-oriented design: Collect-UML (Baghaei & Mitrovic, 2005). Collect-UML provides an interface in which students enter a class diagram as a design for a given problem. Feedback ranging from a simple statement about a design's correctness to hints about all errors in the solution is displayed after the student submits her work for evaluation. A student may also request to see the full correct solution at any time. Collect-UML is a standalone system; it does not link to an IDE to tie design to code. It does not support multiple solutions (student solutions are compared to a set of constraints that define an ideal solution). It also provides a high degree of scaffolding. For example, component names must be selected from words or phrases in the problem description; free-form entry is not permitted.

Most ITS have three required components: an expert system which can solve the problems presented to the student; a student model which tracks the student's knowledge; and a pedagogical module, which carries out the teaching strategies of the system (Shute & Psokta, 1996). (A fourth module, the user interface, is also required for a complete system.) Another type of ITS, constraint-based tutors, do not have a distinct expert model. Instead, student work is evaluated by applying a set of constraints that must be met for a solution to be correct. The constraints that are violated comprise a partial student model that focuses on the concepts which the

student has not applied correctly. The main advantage is efficiency: a complete student model which attempts to identify both the knowledge mastered by the student and the gaps is computationally complex and potentially intractable (Mitrovic et al, 2001). Proponents of constraint-based modeling, or CBM, argue that good human tutoring focuses on correcting student errors, rather than explaining misconceptions or understanding all aspects of a student's mental state. They also acknowledge that lacking an expert module limits the pedagogical advice that can be offered. For example, the system cannot recommend a next step to a student who is lost, nor can it offer advice as a student works. Collect-UML is a constraint-based tutor and thus does not have these capabilities. Recent work has attempted to mitigate these shortcomings for procedural domains by evaluating student work at predetermined steps in the problem-solving process (Mitrovic, Martin & Suraweera, 2007). Collect-UML does not use this technique.

The functionality available in a constraint-based tutor is just one variation in a vast array of behaviors and perspectives offered by different ITS, even for the same domain. The degree of scaffolding, the type and frequency of feedback presented and the manner of presentation are all aspects which vary widely based on the pedagogical philosophy and goals of the system. The ITS built in conjunction with this research is designed to fit into the "Design First" curriculum, and its pedagogy is targeted to beginners who may need guidance in the early steps of solving a problem to avoid becoming frustrated and discouraged with the subject. Thus, the ability to analyze the student's steps and recommend next steps is required. These are the jobs of an expert module. More specifically, the expert module in the domain of object-oriented design

must be able to analyze the same problems presented to the student and generate its own solutions; it must interpret a student's actions as he builds a design, and evaluate each step for correctness; and it must identify and diagnose errors. It must also possess a high degree of flexibility to accommodate different terms used by students for the same meaning, and different complete solutions, all of which may be acceptable.

Expert Module Designs

Prior research has measured the effectiveness of various types of expert modules in the tutoring process.[1] A "black box" expert module creates a solution without revealing its own reasoning to the student or to other parts of the system; thus it cannot suggest strategies that the pedagogical module could include in its advice to the student. Early ITS which used existing expert systems as the expert module often faced this problem. These ITS had to employ tutoring strategies that were independent of the problem-solving strategy. "Issue-based" tutoring is one such strategy, in which a student who executes an incorrect step is counseled on broad issues about the step, such as why the correct step can be a good choice in general. One ITS that uses issue-based tutoring is WEST (1982), which is based on the board game "How the West was Won." In the game, students use basic arithmetic to combine three random numbers to result in a move that advances them toward the end of the board. They compete with the computer to reach the end first. If a student lands on an occupied square, he can "bump" his opponent backwards. Using WEST, a student who moves ahead in a situation where bumping would be better would receive an explanation of why

[1] The following summary of various types of expert modules in ITS over time is based on (Anderson 1988).

bumping can generally be a useful strategy (but not how it applies in the specific situation).

Expert systems sometimes function as "glass box" systems (they do not hide their logic), but unless an expert system is built with tutoring in mind as a possible use, it would not model a student's problem-solving process. An example is MYCIN, an expert system for diagnosing bacterial infections. MYCIN was incorporated into an ITS called GUIDON, but only after GUIDON's developer added "t-rules": rules that applied to the tutoring process. The t-rules did not replace MYCIN's expert rules; in fact, they played no part in MYCIN's problem solving, and the expert rules played no part in developing advice for the student. The t-rules were designed specifically to model the human problem solving process, and to construct advice appropriate to steps in the student's thought process.

The next step in the evolution of the expert module is the use of cognitive models that attempt to imitate human problem solving processes. One technique for building cognitive models in domains which require procedural problem solving is model tracing. In model tracing tutors, the problem-solving domain is described by a set of production rules and a repository of declarative knowledge. Each production rule specifies an action to be taken when a given condition is met. More than one production rule may apply to a condition. Thus, when the rules are applied to solve a specific problem, more than one complete solution can be generated. Rather than creating solutions up front, model tracing tutors follow each step a student takes in solving a problem. The tutor attempts to match the student's action to a production rule by matching the student's result to the result obtained by the system as it applies

each rule. Common student errors are modeled in "buggy" production rules. If the student's action matches a buggy rule, the tutor advises the student by addressing the misconception modeled in the rule (Anderson, Corbett, Koedinger & Pelletier, 1995).

Model tracing tutors provide a structured process for the student to follow, and depend on scaffolding in which the student shows each step in his work. This can work very well for domains in which a well-defined procedure or set of rules are used. However, it can also lock students into following the process that is modeled, inhibiting creativity and holding back students who may be ready to use more sophisticated problem-solving strategies.

The importance of matching the cognitive model used by the expert module to the strategies being taught is illustrated by a comparison of two geometry tutors developed at Carnegie-Mellon University in (Koedinger & Anderson, 1993). The first model implemented for the geometry tutor did an exhaustive search of all possible combinations of the 27 rules (definitions, postulates, and theorems) available for use. Even with some heuristics to eliminate impossible combinations, it was slow. A new expert module was built with a library of partial proofs. Patterns in a given problem were then matched to the partial solutions, from which a complete proof was assembled. The student interface and tutoring strategy were also revised to emphasize matching the givens with an applicable pattern. Not only was performance much faster, but the tutor was now able to teach a useful strategy rather than merely point out errors and give hints. Students using the latter version did better than students using the initial version at choosing a successful proof strategy.

This result suggests an important criterion for the expert module of an ITS for object-oriented design: it should employ the same strategies that the student is expected to use to solve a problem. This requirement steered the search for a strategy for the Expert Evaluator to software engineering research.

Software Engineering

Students tasked with creating a program to solve a given problem face the same challenges as experienced developers: they must identify and understand the requirements of the problem and determine the objects within the scenarios to be handled. They must also identify the attributes and behaviors of all objects and the interactions between objects. Unified Modeling Language (UML) was developed as a structured process for accomplishing these tasks and many others, from gathering and documenting requirements through analysis and design (Booch, Jacobson & Rumbaugh, 1997). UML consists of many elements focused on individual steps in the process, and each one can be used independently of the others (although connections between elements make using them together beneficial). Rather than using specialized languages or complex techniques, components of UML are relatively straightforward; those that deal with requirements gathering and documentation are non-technical enough for end users who are not software engineers to understand. This makes these techniques accessible to beginning students. UML gives students direction on how to get started when presented with a problem, and techniques that they can continue to use throughout their computer science education and as professional developers.

This research concentrates on creating an object model from a problem description. In UML, the object model is represented by a class diagram. Class diagrams come from OMT, or Object Modeling Technique. OMT is an object-oriented analysis methodology developed prior to and later incorporated into UML. (UML is the combination and extension of three methodologies that were developed separately: Booch, OMT and Objectory. The end result is a comprehensive set of techniques and notations for the requirements gathering, analysis, and design phases of system development.) OMT is fully described in (Rumbaugh, Blaha, Premerlani, Eddy & Lorensen, 1991); the process described below is taken from the book.

OMT's process for determining the object classes in a system is to identify all possible candidate classes. These candidate classes are nouns in the problem description. The list is narrowed down by eliminating any that fall into these categories:

Redundant classes – words that are synonyms for the same concept should be grouped together under one term.

Vague classes – words or phrases that don't match a tangible object or well-defined idea in the problem.

Attributes – words that are commonly ways of describing something, like name or color.

Operations – words that define a task or procedure.

Roles – words that define a person's or thing's behavior in a situation, like customer or manager. These might be actors (people or things that interact with the system).

Irrelevant classes – items that are not directly related to the problem.

After narrowing down the list of classes, the next step is to identify relationships, called associations, between classes. Candidates are identified as verbs or verb phrases that associate two or occasionally three classes. Again, some candidates are eliminated based on criteria such as whether they are irrelevant to the problem, able to be derived from another association, or represent an action that an object can execute.

Finally, attributes are identified from nouns that were eliminated as classes in the first step, from adjectives that refer to specific attribute values, or from nouns that precede possessive phrases (as in "the price of a movie ticket").

Notice that OMT's strategy is very dependent on identifying parts of speech. This idea was not new; in a 1986 paper, Booch suggested that parts of speech, such as nouns and verbs, were often good candidates for design elements such as objects and methods (Booch, 1986). The ultimate extension of this analysis of natural language is generating complete programs from English. This dream goes far back in the history of programming languages and software development. In the 1960s, one of the goals in the design of COBOL was to make it English-like and thus intuitive to use. (That is not to imply that COBOL actually met this goal!) In 1977, Halstead proposed a technique to identify operators and operands from English prose, based on a categorization of words developed by Miller, Newman, and Friedman in 1958 (Halstead, 1977); (Miller, Newman & Friedman, 1958).

In 1983, Abbott defined a model for identifying parts of speech in a problem specification and using them to derive the components of an Ada program. His model

is based on object-oriented paradigms from Smalltalk76 and Simula67 and is very close to Booch's and Rumbaugh's models. Abbott identifies data types (classes) from common nouns, objects (instances) from proper or direct reference nouns, and operators (methods) from verbs, attributes, predicates or descriptive expressions (Abbott, 1983). He presents his technique as a process for software developers rather than an algorithm for an automated process, and acknowledges that assumed "common knowledge" about the domain that is not explicitly stated in a problem definition is a significant challenge to automation.

Abbott's work inspired Booch, Rumbaugh, and others who designed processes to define and standardize software development. As the field of natural language processing advanced, it also inspired researchers to build automated tools to aid in the analysis and design phases. Examples of such tools include Circe (Ambriola & Gervasi, 1997) and LIDA (Overmyer, Lavoie & Rambow, 2001). Both of these tools use parts of speech tagging to identify candidate classes, attributes and methods, but they depend heavily on a human analyst to verify and refine the results. The greatest challenge to further automation is in semantic analysis for the meaning of terms, and the application of background knowledge. These are needed to make decisions to remove unsuitable candidates. LIDA acknowledges this but does not attempt to use either to refine its choices; all refinement must be done manually. Circe does some refinement based on a domain-dependent data dictionary that must be entered by an analyst; this requires a significant amount of up-front work for each domain. Manual review and revision is still required to generate an acceptable design.

Metafor (Liu & Lieberman, 2005) is the most ambitious tool so far of all attempts to automate the design process. In Metafor, the user enters a "story" describing what the program should do. Metafor displays code segments (such as class definitions) as the story is entered. The user can modify or add to the story to resolve ambiguous statements and revise the code. Metafor fills in the background knowledge assumed in the story by accessing a repository of "commonsense" knowledge to refine its results. The system has produced encouraging results when applied to small problems similar in scope to those assigned to beginning computer science students. It is, however, still intended as an interactive tool whose results are verified and refined by a human analyst. Metafor is being evaluated as a brainstorming tool for intermediate-level programmers and non-programmers.

One of the benefits of Metafor and the earlier tools is that they aid in the identification of missing information in the requirements specification. This allows analysts to go back and update the requirements to make them more complete. The entire process promotes a better match between the original natural language requirements and the system design. In a classroom environment, this can help teachers create more complete and precise problem descriptions for their students. Indeed, explicit problem statements are essential for beginning students, who do not have the prior experience from which to draw knowledge of implicit or assumed functionality.

The process of identifying design components from the problem text can also provide useful information about shortcomings in a student's design. Has the student taken a word or phrase that represents an object in the design and made it an attribute?

If so, this information provides clues to the student's understanding of both class and attribute, and can help the ITS' pedagogical module more accurately tailor its feedback to the student's needs.

In addition to interpreting the problem description entered by the teacher, the ITS built in conjunction with this research must also interpret the student's actions. The expert module must be capable of two types of flexibility. First, it must be able to recognize different terms for the same concept, since students are allowed to enter their own names for components. Related terms must be matched to the student's intended meaning; misspellings must also be recognized. A lexical database such as WordNet (Fellbaum, 1998) can be used to resolve the former, and a spell checker can be employed for the latter. Second, even very simple problems can have multiple valid solutions. Multiple solutions must be generated and recognized as valid. Variations among components within a solution, not just complete distinct solutions, must also be allowed.

Hypotheses

Software engineering standards and practices such as UML have helped developers build systems that are more complete and accurate implementations of their requirements than in the past. These practices can also help novice students learn object-oriented programming by giving them procedures they can use to analyze a problem description, identify design elements, and logically divide functionality into manageable pieces.

Some of these same software engineering processes are being automated to generate program designs as a starting point or brainstorming tool for experienced developers and their clients. Teachers can also benefit by using such a tool to verify and refine project descriptions, to gain insight into alternate solutions, and to expand their own knowledge of object-oriented design. The generated solutions can be used by an ITS to evaluate student work while allowing for multiple variations, and complete analyses of each component within a problem description can provide diagnoses for student errors.

Research Questions

1. *Can learning how to create use cases and class diagrams help novices learn object-oriented design and programming?*
2. *Can the automation of software engineering processes and principles be applied to solve simple problems of the type generally assigned to beginning computer science students? Specifically, can it:*
    a. *generate multiple solutions for a given problem?*
    b. *provide flexibility in identifying and evaluating novel but correct solutions created by students?*
    c. *interpret the steps a student applies in solving a problem,*
    d. *provide a diagnosis for student errors and*
    e. *recommend next steps when the student is stuck?*

Experimental Design and Evaluation

The first research question is addressed by the development of the "Design First" for Java curriculum and its evaluation over three semesters at Dieruff High

School in Allentown, PA. The curriculum developed includes everything needed to teach the course: lesson plans, student projects, practice exercises, assessment instruments and an IDE specifically chosen to support the processes being taught. At the end of each semester, student performance on exams and projects was analyzed to determine how well students grasped specific categories of concepts, including problem solving, objects and related terminology, program design and procedural code. Improvements based on those evaluations were made after each semester, but the basic structure of the course did not change: the same topics continued to be covered in their original order. Changes that were made included the addition of more practice exercises and quizzes, and the assignment of different projects.

All three groups of students performed well on object concepts and design. The first group was weak in procedural code; the emphasis on the object and design concepts may have not allowed enough time for practice using procedural statements. Additional time was allocated to review and practice procedural concepts in the second semester; as a result, this group of students performed better on procedural coding, and still did well on object and design concepts. Further improvements were seen in the third group of students. The complete results for all students will be included in Section IV of this dissertation.

The second research question (and its related sub-questions) are addressed by the development of an expert module that generates design solutions for a problem, evaluates a student's solution, and provides diagnoses for student errors. This expert module evolved into two components: an Instructor Tool that generates solutions interactively with the instructor, and an Expert Evaluator that matches student actions

with components in the generated solutions. The Instructor Tool aids the teacher in creating clear and concise problem descriptions. 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 Instructor Tool was evaluated by instructors using it to create actual assignments, and assessing the quality of the solutions it generated. The Expert Evaluator served as part of CIMEL-ITS (later renamed DesignFirst-ITS), which assists students as they enter a class diagram representing the design of a program to solve a problem. Actual student work within the tool was used to measure the Expert Evaluator's accuracy.

The IDE chosen for the ITS is Eclipse[2], an open-source, freely available IDE supported by the Eclipse Foundation (a non-profit corporation whose directors include IBM, HP, Intel and other major technology companies). Eclipse is widely used in both academia and industry. Built initially as a development environment for Java, Eclipse is based on an extendible architecture that supports the integration of plug-ins, or modules that add functionality to the system. Plug-ins have been created for many functions, including C++ development, data modeling (entity-relationship diagrams), and UML. This allowed Eclipse to be customized for the curriculum. Thus, two plug-ins were added: a UML editor for entry of class diagrams (originally, a commercial plug-in called Omondo was used; it was later replaced by a UML plug-in developed at Lehigh); and DrJava, a beginner's tool that allows for interactive execution of single lines of code (Allen et al, 2002); (Reis & Cartwright, 2004). With DrJava, students can quickly see what a line of code does, or test a method without creating a main method

[2] www.eclipse.org

to run it. The UML plug-in supports the process of creating a class diagram first. It generates code stubs from the diagram, using default values for options such as public, private, and static. Thus students don't need to learn the complexities of syntax or the meaning of various keywords until later, when they are better prepared to understand them.

At Dieruff High School, students worked on their project assignments in class, where two teachers were present to answer questions and help students who got stuck. Observation of students working in Eclipse yielded a wealth of information on the types of difficulties students encountered and the advice which was effective in helping them move forward. This information was useful in making the decision to focus the ITS on creating class diagrams in the UML editor of Eclipse, and in defining the ITS' requirements.

One of the desired functions of the ITS is to refer the student to other sources of information for review of concepts with which he or she needs help. An earlier project at Lehigh, CIMEL[3], had created multimedia courseware covering a breadth of topics for a CS0 or CS1 course (Blank et al, 2003). Some of CIMEL's content on objects and classes was used in the high school; as an online resource, it is easily available to students for individual study outside the classroom. CIMEL also contains interactive exercises and quizzes, which offer additional information on the student's knowledge of concepts. Thus CIMEL can provide to the ITS both instructional content and input on the student's knowledge level. Additionally, an intelligent tutoring system seemed a logical successor to a project that offered diverse media and

[3] Constructive, collaborative Inquiry-based Multimedia E-Learning.

interactive tools, but did not solve problems with the student or adapt to individual student knowledge levels or learning styles. Thus, the project to build the ITS was originally named CIMEL-ITS.

The DesignFirst-ITS architecture (shown in Figure 1) is based on the traditional ITS model with four components: expert module, student model, pedagogical model, and user interface. A Curriculum Information Network was added to store the concepts that comprise the tutoring domain. This dissertation's research focuses on the expert module, called the Expert Evaluator, or EE. Two other dissertations focus on the development of the student model (Wei, 2007) and pedagogical model (referred to as "Pedagogical Advisor" in DesignFirst-ITS) (Parvez, 2008).

The Expert Evaluator observes and analyzes each action the student enters in the Eclipse UML editor and then creates a packet of data about the step. The data packet identifies the step as correct or incorrect, and the concept applied in the step. If incorrect, the packet also contains the type of error committed, and a recommended alternative to the incorrect action. This information is available to the student model and pedagogical advisor components of the ITS. A separate interface is provided for the entry of new problem descriptions by instructors. The Instructor Tool works through this interface to analyze a problem description and present potential solutions to the instructor, who may revise and annotate the solutions before saving them in a problem database, or modify the description and regenerate solutions.

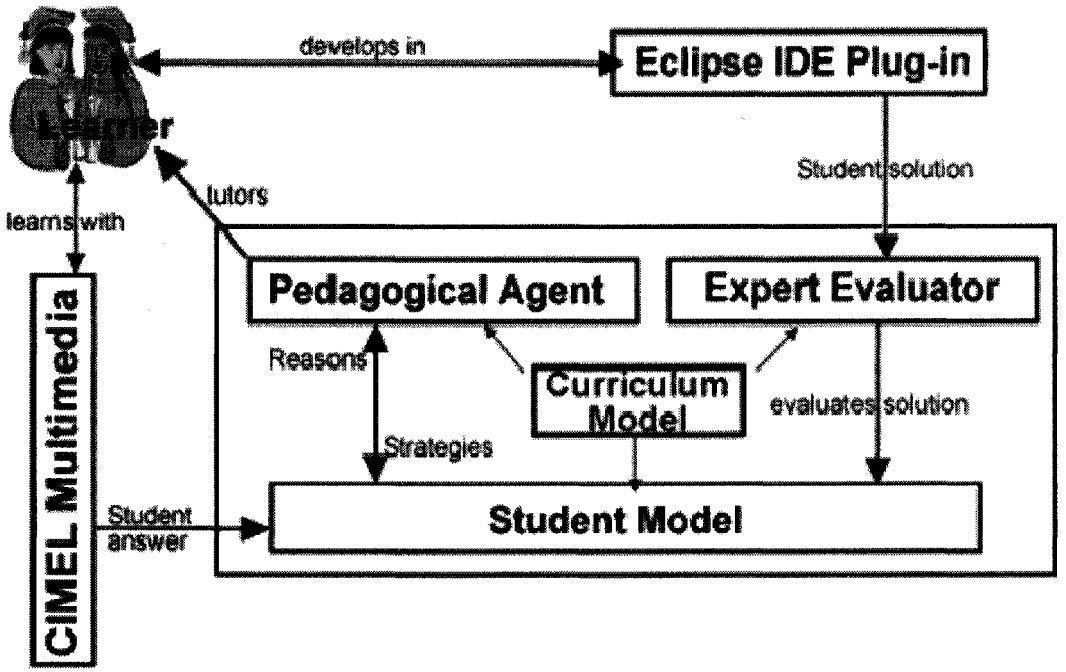


Figure 1: DesignFirst-ITS Architecture

Research question 2a deals specifically with the Instructor Tool's ability to generate solutions to problems specified in English prose. Three independent evaluators (two Java instructors and a software engineer) used the tool to analyze their own problems. They completed a questionnaire on the correctness and completeness of the solutions, and the ease of use of the interface. Six problems suitable for a CS1 course (including actual assignments used in the participating classes) were input into the tool.

Questions 2b, c, d and e all pertain to evaluating student work. Thus, actual student solutions and data on students' intermediate work are required to measure the Expert Evaluator's effectiveness. Nine students in CSE12 at Lehigh during the Fall 2005 semester used the ITS to design a solution to one problem (an automated teller machine). Each step entered into the UML editor was recorded and processed by the

EE; the results were analyzed and used to improve the EE. In Spring 2006, additional data was collected from five Lehigh students in CSE432 using the ITS, which provided input that led to further refinement of the EE. In the Fall of 2006, students in a first-semester computer science course at Lehigh (CSE15) participated in an optional workshop in which they used the ITS; data collected from these students were used in answering questions 2b, c, d and e.

Question 2b deals with the EE's accuracy in identifying multiple novel correct solutions. This was measured by observing the number of solution variations reflected in the solution templates generated by the Instructor Tool on at least three problems. Another way to measure this aspect of the tool is to count the number of correct solution variations in a group of student solutions that were mistakenly marked in error by the EE. Such an analysis of student data is also useful in answering questions 2c, d and e, which all pertain to the EE's accuracy in interpreting a student's intent for each action taken. As students used the ITS to complete an assignment, every action was recorded in a database. The EE analyzed each action and recorded its diagnosis in the database. From this data entire student problem-solving episodes were recreated, and the diagnosis of each step was evaluated by a human expert. Specifically, the expert made his own determination of correctness, and for errors, decided on the concept applied and the recommended action to replace the error. The percentage of actions for which the expert agrees with the EE's diagnosis is a measure of the EE's accuracy. In summary, the analysis determines:

- How many errors were present in the solutions; of those, how many were identified by the EE;
- How many correct actions were incorrectly flagged as errors;

- How many different good solutions were represented in the group.

Although the participating students used the ITS and received pedagogical advice from it, measuring the efficacy of the advice is not necessary to determine the EE's accuracy and is beyond the scope of this dissertation. Also, these experiments are not testing the effect on student performance of using the ITS, or of using the Eclipse/UML IDE alone. Thus, a control group of students who do not use the ITS is not necessary.

Contributions

This research contributes to several domains. First, the Design First with Java curriculum is an innovative approach to teaching a first programming course. The curriculum has been disseminated within the Computer Science Education community through a journal article (Moritz & Blank, 2005), conference presentation (Moritz et al, 2005), and workshop (Blank et al, 2005). Chad Neff, the teacher at Dieruff High School who partnered with Sally Moritz in teaching the Java course, presented the curriculum to other high school teachers at the National Academy Foundation's annual conference in July 2005. The curriculum and supporting materials are posted on the Lehigh Valley Partnership for STEM Education website at www.lehigh.edu/stem/teams/dieruff/.

Automated design and code generation is in an early stage of development. A few systems have attempted to automate the extraction of design elements from plain text requirements, but all depend on a significant amount of user intervention to refine

their results. Most are intended as assistants to experienced developers, or as brainstorming tools to help end users identify all requirements of a desired system.

Despite these shortcomings, early code generation systems have provided results that are useful examples of the design process. Yet educational use of these systems has been infrequent and limited to illustrating the process of identifying design components. By tailoring the techniques to the domain of small beginner problems and applying the processes that students are taught to follow, automated code generation can be used to validate designs generated by students and provide pedagogically useful diagnoses of errors. Successful use of these techniques in an ITS and as a general learning tool is innovative and would demonstrate their value beyond their current applications. Its educational use is not limited to students using the ITS. Several evaluators suggested the Instructor Tool as a learning aid for the teacher, when the tool generated design components they hadn't considered. The tool expanded their own understanding of object-oriented design, as well as coached them toward crafting better problem descriptions for students.

A third domain to which this research contributes is intelligent tutoring systems. An effective ITS for object-oriented design would be a valuable resource for computer science students. DesignFirst-ITS provides a relatively unstructured environment that allows students freedom to build their designs. Students who do well are unencumbered by scaffolding they don't need; students who have difficulties are provided with assistance as they need it. Students are also not locked into a single solution; multiple solutions that adhere to the rules of design and 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. Finally, the Expert Evaluator is useful as a grading tool. A teacher could use its evaluation as a first pass in grading student solutions, and review the results to verify them.

In summary, the contributions of this research are:

- A novel use of automated code generation techniques, which provides additional evidence supporting the value of these techniques and encouraging their continued development and use in new software engineering applications.

- A unique expert module for an intelligent tutoring system that overcomes the limitations of a constraint-based tutor through a rich analysis of the problem to be solved. The EE allows for creativity in a complex domain by recognizing multiple solutions, and it provides error diagnoses that can be used to recommend next steps and customize advice when a student is stuck.

- The application of software engineering principles and UML to help beginners learn problem solving and object-oriented design.

- An innovative curriculum for teaching object-oriented design and programming to beginners, complete with lesson plans, student materials, and an IDE to support it, which has been evaluated in a high school setting.

- A tool to assist teachers in creating complete, clear problem descriptions, in increasing their understanding of those problems and object-oriented design in general, and in evaluating student work.

II. Related Research

The prior work related to this research falls into 3 categories: Computer Science Education (specifically teaching a first programming course); Intelligent Tutoring Systems and the development of the expert module; and Object-Oriented Software Engineering methodologies and automated code generation. Each is covered in its own subsection within this chapter.

Computer Science Education: The First Programming Course

This section will briefly cover the challenges of teaching an object-oriented programming language such as C++ and especially Java; the "Objects First" approach and the tools developed to support it; and the problems students still encounter with this approach. First, a history of pedagogies in Computer Science Education is warranted.

Beginning in 1968, the Association of Computing Machinery (ACM) has published recommendations for the undergraduate course of studies in Computer Science. Updated curricula were published in 1978, 1991 and 2001; the last two were developed jointly with the IEEE-Computer Society. The curricula include identification of the knowledge and skills a student should possess at the completion of the undergraduate program; the core subject areas that should be required; additional subject areas recommended as electives; and the basic skills required of every beginning student. Each Computing Curricula was developed by a committee with input from educators, subject matter experts and researchers. The

recommendations reflect the state of Computer Science at each point in time. The ACM continues to review computing curricula, and is developing recommendations for related domains, including Information Technology, Information Systems and Software Engineering. Details on work in progress may be found at http://www.acm.org/education/curricula.html.

The goals of the undergraduate course of study as stated in CC'68 and CC'78 were similar. Both focused first on programming: students should be able to "write programs in a reasonable amount of time that work correctly, are well documented, and are readable" (Austing et al, CC '78). Secondary goals were to understand basic computer architectures, and prepare for further education or specialization. Thus, the CS1 course focused on algorithms to solve problems, and the syntax and semantics of a programming language. In the 1970's, structured programming was the prevailing methodology for software engineering; CC '78 recommends that structured programming techniques be incorporated throughout the curriculum, including CS1. But it does not provide any details on which aspects of structured programming should be emphasized, nor how they should be applied in CS1. Neither CC '68 nor '78 recommend a specific programming language for CS1 (CC '78 specifies only a "high-level" language). The dominant languages of the day were procedural; while object-oriented programming was developing in the 1960s (Simula) and 1970s (Smalltalk), it remained within the domains of research and a few specialized applications. Meanwhile, Niklaus Wirth developed Pascal to implement and enforce important principles of structured programming. Pascal quickly became one of the most popular languages for undergraduate CS courses.

Structured programming continued as the predominant methodology for software analysis and design through the 1980s, although object-orientation gained more attention during this time. As software systems grew in size and complexity, structured programming became less effective in managing their development and maintenance. Building objects that combined the data and behaviors of components, protected their inner parts through encapsulation, and promoted code reuse offered greater promise in controlling software complexity and improving quality. A number of object-oriented languages were developed during this time. ObjectiveC, an object-oriented superset of C, was created by Brad Cox at StepStone Corporation. Bertrand Meyer, a prominent researcher in software engineering, developed Eiffel as a simple, robust language that strictly adhered to the object methodology defined in his 1988 book *Object-Oriented Software Construction.* Modula-3 was developed at DEC's Systems Research Center and found some use as a teaching language in academia. Many other lesser-known languages based on object-oriented concepts were developed as research projects at universities.[4] It took the clout of Bell Labs (later AT&T Research Systems) to give one object-oriented language – C++ – predominance, and bring object-oriented design and development to mainstream business and industry.

Bjarne Stroustrup developed C++ at Bell Labs in the early 1980s. Stroustrup found that the object-oriented features of Simula made it an excellent language for large systems development, but it suffered from slow performance. So Stroustrup started with C – a very fast, general-purpose language also developed at Bell Labs –

[4] See www.wikipedia.org for a summary of each of these programming languages.

and added support for classes. The first commercial version was released in 1985. C++ gained popularity not only among users of C, but also among developers who wanted to shift away from structured programming to object-oriented design and development. Both public domain and commercial C++ compilers for many different platforms quickly became available.

With the movement from structured programming to object-oriented development well under way, the ACM and IEEE-CS collaborated on an updated version of the Computing Curricula, released in 1991. CC'91 is significantly different from its predecessors in its structure and focus. Instead of providing a single set of courses divided into requirements and electives, CC'91 lists nine subject areas which should be included in the undergraduate curriculum. From these subject areas, eleven knowledge units are developed in some detail to specify the concepts from each subject area that are required. An institution is free to combine those knowledge units into courses as best fits the program's educational goals. CC'91 recognizes the separate branches that had developed by this time, including computer science, information technology, computer engineering and software engineering. Sample curricula in each of these disciplines are provided.

Another significant difference from earlier Computing Curricula is the role of programming. CC'91 defines programming as "the entire collection of activities that surround the description, development, and effective implementation of algorithmic solutions to well-specified problems." This includes identifying requirements, problem solving, verifying correctness of a solution, and even maintenance of code. However, because programming pervades every knowledge unit, students must develop fluency

in a language early in the program. Yet CC'91 includes "Introduction to a Programming Language" as an optional knowledge unit and makes the claim that a programming language can be taught as part of a course devoted to other content. The report goes so far as to suggest that most students acquire at least some background in programming in high school.

CC'91 also does not recommend any particular language or methodology. The closest to any such recommendation can be found in the sample curricula the curriculum committee included in an appendix to the CC'91 report. Only in software engineering courses is object-oriented development mentioned; one course description states that either functional or object-oriented analysis may be used. To paraphrase the report, an institution may choose to use an object-oriented language, although historically either procedural or functional languages have been used. Clearly, the curriculum committee saw object-orientation as an alternative methodology, but still not widely accepted or preferred over structured programming. A look at the sample CS1 and CS2 courses shows little change from the content recommended in CC'78, and no suggestion on changing from a procedural or functional programming language.

By the time the 2001 Computing Curricula task force was formed in 1998, the need to revise the existing curriculum recommendations was urgent. The 1990s was a decade of great change. The emergence of the World Wide Web, more powerful personal computers, sophisticated graphics and multimedia, and many more advances had a profound effect on modern life, and spawned new areas of interest and research. Object-oriented programming became a widely accepted and foundational paradigm

for software design and development. The great technological changes alone were sufficient to require revision of the curricula; additionally, the CC '01 task force found that many departments had difficulty translating CC'91's knowledge units into course descriptions (ACM/IEEE-CS Joint Curriculum Task Force, 2001). CC'91 had strived to be flexible enough to accommodate programs with varying goals and focus; instead, it was too vague to help many departments with defining course content and breakdown. This was especially true for CS1 and CS2; at many institutions, these courses had changed little from the CC'78 recommendations.

The CC'01 task force pledged to identify the fundamental skills and knowledge all computer science students should possess, and to offer guidance in course design. The task force identified all knowledge areas appropriate for an undergraduate program, which they divided into 14 major areas. Within these major areas, lower-level topics were identified and determined to be either required or elective. All required topics make up the core subject matter for all students. The other topics may be included at the institution's or student's discretion for an individual's complete course of study.

CC'01 acknowledges the importance of programming as a skill that all students must master early in their careers by identifying "Programming Fundamentals" as a core knowledge area. The specific topics within this area include the basic syntax and semantics of a high-level language, algorithms and problem solving strategies, elementary data structures and recursion. Also included is object-oriented programming, which comprises at least 10 hours of the Programming Fundamentals unit. CC'01 clearly affirms that the object-oriented programming

paradigm is required knowledge which should be taught within the introductory sequence of courses. The task force backs off from endorsing either an objects-first or procedural-first approach. Instead, it offers two different course sequences. In the procedural approach, object-oriented programming is taught in a CS2 course, after a CS1 course which covers the basic procedural constructs traditionally taught. In the objects-first approach, object concepts are introduced at the beginning of CS1 and intertwined throughout the sequence.

CC'01 officially affirmed that the object-oriented paradigm had become required fundamental knowledge for all computer science students. Many institutions had already been teaching object-oriented programming before CC'01 was published, and the debate on when and how to teach it was well underway.

Even though CC'91 paid scant attention to the growth of object-oriented technology in the 1980's, many colleges and universities did take notice. By the late 80's, papers on incorporating objects into the curriculum began appearing. One of the earliest and most committed adopters of object technology was Carleton University in Ottowa, Ontario, Canada (Pugh et al, 1987). Carleton began by using an object-oriented approach to abstract data types in the Data Structures course (taken in the sophomore year). Initially, the use of Pascal was continued even though the design of data structures was taught with an object-oriented approach; students simulated concepts like encapsulation by commenting fields as public or private, and implemented data type operations as functions with the message receiver as the first argument. This proved too cumbersome, so Smalltalk was adopted as the programming language. Carleton then took the next logical step, and replaced Pascal

with Smalltalk in the CS1 course (Skublics & White, 1991). In addition to the procedural constructs traditionally taught in CS1, Carleton's course included classes and instances, defining new classes and inheritance. Student grades in the object-oriented CS1 were not significantly different from the grades in the procedural version of the course. Students were able to apply the object-oriented paradigm in other advanced courses, as well, including Programming Languages, Operating Systems, and special projects.

Object-oriented development was introduced at Michigan State University first in an advanced software engineering course in 1990 (Reid, 1991), then in the CS1 course the following year (Reid, 1993). Both courses used C++. In the introductory course, students were provided with classes that implemented a graphical interface; the focus was on creating instances using existing classes to solve problems, rather than creating new classes.

Most other early implementations of object-oriented development in the curriculum remained in the intermediate and advanced courses. Data Structures was a popular place for the first introduction to object-orientation, at schools such as Pace University (using C++) (Bergin, 1992), and a group including Allegheny and Bowdoin Colleges, University of Connecticut, and West Chester University of Pennsylvania who participated in an NSF-sponsored project that evaluated a breadth-first curriculum incorporating object-oriented methodologies (Epstein and Tucker 1992). Aalborg University in Denmark taught object-oriented techniques in a fifth semester course (out of ten semesters), which introduced program development in Eiffel to students already familiar with Pascal (Nørmark, 1995). Other schools did not introduce it until

the junior or senior year of the program, including Indiana University – Purdue University at Fort Wayne (Temte, 1991), and Pratt Institute (Bellin, 1992). Both these schools used a special seminar course as their first foray into teaching the object-oriented paradigm, and recognized that this was but a first step toward incorporating the paradigm into other courses throughout the curriculum.

The widespread interest in object-orientation's role in the CS curriculum converged in an Educators' Symposium at OOPSLA '92 (Conference on Object-Oriented Programming Systems, Languages, and Applications – the annual conference of the ACM Special Interest Group on Programming Languages, or SIGPLAN). Educators shared their experiences in implementing object technology in their curricula, and reflected on the open issues that prevented a simple answer to the question of how the object paradigm should be incorporated. Perhaps the largest of these issues was that object methodologies themselves were not yet mature (Northrop, 1992). Much work on design and analysis techniques was in progress; many object-oriented languages (which implemented the paradigm to varying extents and in different ways) were competing for dominance; there was a dearth of books on the subject (especially ones suitable as textbooks or student references).

These problems, and others, are very well documented in a survey of the use of object-orientation in the undergraduate curriculum by Dorothy Mazaitis (Mazaitis, 1993). First, there are several factors which slow the adoption of any new technology or paradigm: a desire to wait to see what the new paradigm's long term role will be (making sure that it is not a passing trend); the need to evaluate how the new approach should fit into the curriculum as a whole; the bureaucratic processes required to

implement curriculum change at most institutions; and the time and monetary costs in educating faculty and developing new course material and tools. These issues were aggravated by the fact that the object-oriented paradigm represents a completely different way of approaching software design, and permeates all aspects of software development.

As with any new technology, supporting materials such as appropriate textbooks and development tools did not yet exist. Students at Carleton used the Digitalk Smalltalk manual and notes written by the instructors in lieu of a textbook; Temte at Indiana also had to supplement course materials with his own notes. A textbook (using C++) for the Data Structure course was written at Pace; Epstein's and Tucker's working group also planned to author a textbook. Biddle and Tempero, writing about their experience teaching the object-oriented paradigm in the intermediate and upper level courses at Victoria University of Wellington in New Zealand, lament that in spite of the large number of C++ texts available, none applied the object-oriented paradigm in explaining the features of the language (Biddle & Tempero, 1994).

Suitable tools to support student work were also lacking. Integrated development environments were just appearing, and they were still based on a simple text editor for entry of code. Managing multiple files (required to support one file per class) was still essentially a manual task. Testing individual units of code was not possible without writing code stubs. The only exception was the Smalltalk environment. One of the earliest graphical development environments, it presented a single unified facility in which to enter, compile and run programs. The Smalltalk

class library was built-in, so a programmer could search for an existing class without leaving the IDE. The IDE was large – too large, in fact, for students, who faced a steep learning curve when presented with its large number of unfamiliar options. The extensive class library was also overwhelming. Both Skublics and White at Carleton and Temte at Indiana reported student difficulties in using the Smalltalk IDE. Students at Aalborg University had similar problems with the Eiffel development environment and the complex class library it provided.

The issue of tools leads to a central question in developing any programming course: which language to use. All the early adopters discussed the pros and cons they weighed in their decision. Those who chose C++ cited its wide acceptance in industry, and students' prior familiarity with C; proponents of Smalltalk and Eiffel chose them for their complete object-oriented nature. Some chose extended versions of Pascal, also because of students' (and instructors') familiarity with standard Pascal. Unfortunately, all of these choices had their drawbacks. C++ was (and still is) not a pure object-oriented language; built on top of a procedural language, one can easily use it to develop purely procedural code, which makes it all too easy for students to fall back on old habits. Both Smalltalk and Eiffel have very small user bases, and very few textbooks to support student learning. Both are also inseparable from complex programming interfaces and powerful but overwhelmingly large class libraries, which provide additional challenges for beginning students. A number of versions of "object" Pascal were developed, all quite different from each other. Pascal was not widely used outside of academia, so it would not have practical use outside of the classroom.

In spite of all the difficulties, almost all the early adopters of the object-oriented paradigm were committed to moving forward with it, and most agreed with introducing it as early in the curriculum as possible. A common theme was the difficulty in switching from a procedural way of thinking to the object-oriented way. Tempe noted "the new mind-set did not come naturally to students with ingrained notions of problem solving from a procedural perspective." Skublics' and White's students at Carleton who had prior experience in procedural programming "reported difficulty in changing their way of thinking to an object-oriented approach." Stroustrup asserted that experienced programmers would take between six and eighteen months to switch from a procedural mindset to an object-oriented way of thinking (Stroustrup, 1994). However, students who learned an object-oriented approach first did not have difficulty learning procedural concepts (Skublics & White, 1991), (Decker & Hirshfield, 1994). This suggests that introducing objects earlier in the curriculum saves time and effort in learning a new way of thinking, without incurring greater difficulty in mastering procedural concepts.

Before the "objects first" philosophy could be put into practice by most institutions, the lack of a suitable language, tools and textbooks for teaching had to be resolved. One of the leaders in filling in the gaps was Michael Kölling. He addressed the first two issues: language and tools. Kölling identified ten requirements for a suitable first year teaching language (Kölling et al, 1995), summarized as:

1. Basic object-oriented concepts such as encapsulation and inheritance should be supported in a clear, concise way.

2. The language should support "pure" object-orientation. It should not provide those features as an extension to a procedural language, so that programs could be written without object-orientation.

3. It should include safeguards like static type checking and detection of common errors such as uninitialized variables.

4. It should not include advanced features whose purpose is efficiency rather than support of the development methodology (for example, pointers and dynamic memory allocation).

5. It should be easy to compile and execute.

6. It should have an easy to read syntax. For example, keywords rather than symbols should be used. (Consider C and its cryptic shorthand notations.)

7. There should be no redundancy (only one way to express a command).

8. It should ease the transition to other popular languages. Thus, it should follow common conventions and styles.

9. It should support features that improve code reliability, such as assertions, pre and post conditions, and debugging options.

10. Its development environment should be easy to learn, and include development aids such as debugging facilities.

Kölling evaluated four object-oriented programming languages: C++, Smalltalk, Eiffel and Sather. Problems with the first three of these have already been discussed. Kölling describes Sather as "a cross between C++ and Eiffel." Sather has a number of features designed to improve its efficiency over Eiffel. These are confusing for a beginning student (a violation of item 4 above). It also does not include a number of safeguards (such as type checking of all objects), again in the name of efficiency, but resulting in a reduced safety net for beginners. Like C++, it is more geared toward experienced developers who need access to lower level functions, rather than to beginners focused on learning the concepts of object-oriented development.

Kölling saw these shortcomings of existing languages as a call to develop a new object-oriented language for teaching. So, he developed Blue (Kölling & Rosenberg, 1997). Blue was designed to meet Kölling's ten requirements. It is a pure object-oriented language: everything is a class. There are no primitive datatypes; basic datatypes such as integer and real are implemented by predefined classes. There is also no concept of a program. All classes can create instances of other classes. A top-level class can be created to simulate a main program that kicks off execution. Simple inheritance and generic classes are also supported. Other features include basic procedural constructs for condition, selection and iteration, basic interactive and file input and output, and assertions, preconditions and postconditions. Blue also had its own development environment (Kölling and Rosenberg 1996b), which included features defined by Kölling as required to support beginners (Kölling & Rosenberg, 1996a). The Blue IDE emphasized visualization of classes and instances, and supported experimentation in creating and manipulating instances.

Students see all the classes that exist for a single application (called a "project" in Blue). The graphical depiction also shows relationships between classes (in Figure 2 we see that circle, triangle and rectangle are subclasses of graphical object, and square is a subclass of rectangle). From the diagram, students can edit the code for a class (using the Edit button on the left, or simply double-clicking the class). An instance can be dynamically created using the Create button; a dialog prompts for any parameters defined by the class's constructor method. A student sees the instances she's created in the Object Bench at the bottom. These instances can be manipulated through the Interface Routine Menu, which shows the methods that can be invoked for the

instance. The student can also view changes in the instance's state by viewing its instance variables and their values.

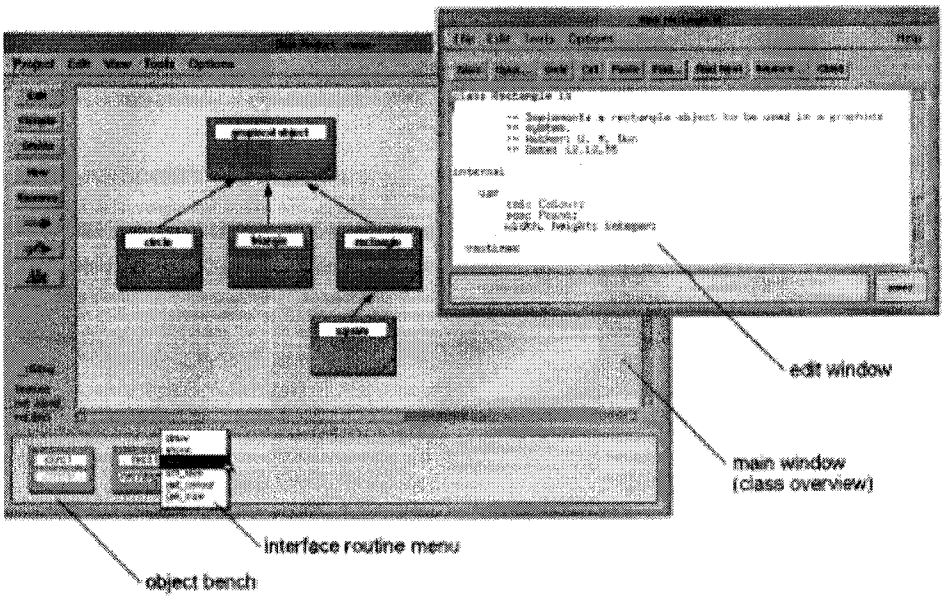


Figure 2: The Blue Development Environment
(taken from (Kölling, 1996b))

A student begins creating an application by creating the classes and their relationships in the graphical window. Only after the classes are added here can students enter code to implement methods (within code stubs that are generated automatically). This encourages students to think about the design and organization of a solution before writing code. The simplicity of the environment (the interface includes only options needed by a beginning student, within clearly labeled buttons or on standard menus) allows students to concentrate on developing a solution rather than learning a complex interface.

Kölling's work received much attention in the Computer Science education community through ACM's SigCSE conferences and publications. At about the same time, a new object-oriented language began to gain prominence: Java. First released in

1996, Java was Sun Microsystems' object-oriented language for the Web. The project that ultimately created Java was initially tasked with developing a language for smart appliances. Such a language would need to be highly reliable, run on many platforms, and have a small footprint. It also needed to support multithreading in a distributed environment. A version was pitched to the TV and cable industries, but it failed to generate interest. So the project's target was changed to web applications, and Sun was able to partner with Netscape to have Java support included in the Netscape browser. With the backing of both Sun and Netscape, Java quickly became popular for development of both front and back end web applications.

One of Java's goals was to be easier to learn and less error-prone than C++. Manual memory management is a difficult and tedious task in C++, and the cause of many code failures. In Java, automated garbage collection eliminates that problem, and is quite efficient.

The semantics of Java are similar to C++, presumably to ease the learning curve for experienced programmers moving to Java. But Java more strictly adheres to the object-oriented paradigm. All components, including the main method, must be implemented within classes. A C++ programmer can create purely procedural programs; a Java programmer cannot. Java has included an extensive class library since its first release, thus promoting code reuse. C++ included only a modest library based on the C standard library; the Standard Template Library (STL) was developed independently, although it was made freely available (by Hewlett Packard, where much of its development occurred) in 1994 and included in the ANSI standard C++ of 1998.

Java's other major goal is platform independence, and this is accomplished by compiling Java to bytecode rather than machine language. The Java bytecode is then interpreted and run by a Java Virtual Machine (JVM) on the target platform. While this ensures portability of Java across platforms, it does exact a performance cost at runtime. This cost has been reduced through improved versions of the JVM and faster hardware; however, it is still cited as a reason to choose C++ over Java for high-performance applications[5].

Through the late 1990's, C++ remained the most popular language in undergraduate courses, in spite of (or perhaps because of) its weaknesses as a teaching language. Certainly its widespread use in industry was a major factor. Students wanted to learn a language that would be immediately useful in gaining employment. Because C++ is built on top of C, a CS1 course using C++ could cover procedural programming first. In many courses, classes were not introduced until the second half of the semester. Most CS1 textbooks use this approach. So even though students learned an object-oriented language, many did not learn to develop solutions using the object-oriented paradigm.

Java's popularity pushed the issue of object-oriented software development to the forefront. Academia's interest in Java came from two directions: the rapid growth of industry's use of Java (especially for web applications), and proponents of teaching the object-oriented paradigm, who saw Java as a better alternative to C++. While Java still did not meet all ten of Kölling's requirements for a teaching language, it was a stricter implementation of object-orientation than C++, and it eliminated some of the

[5] For a summary of Java's history, see www.wikipedia.org.

unnecessary complexities of C++, such as pointer arithmetic and manual memory management. Java's popularity grew so quickly that in the 2003-04 academic year, the College Board switched its Advanced Placement exams in Computer Science to Java from C++.

Despite Java's emphasis on classes, most Java textbooks and many CS1 courses still took the same approach of teaching procedural concepts first, and then introducing classes and objects. Many texts still used examples from Pascal or other procedural languages ("Hello World!" for instance), which were simply unsuitable to demonstrate object-oriented techniques. There was also a lack of tools to support teaching object concepts, and no IDE that was suitable for beginners.

The Computer Science Education community began working on these needs in earnest. Kölling adapted his Blue development environment for Java and called it BlueJ (Kölling et al, 1999). He also co-authored a textbook for CS1 based on BlueJ, *Objects First with Java* (Barnes & Kölling, 2003). DrJava was developed at Rice University as a simplified IDE for students (Allen et al, 2002). It includes basic editing, compile, and testing features useful for students without the many other complex features often built into professional IDEs. Additionally, it allows interactive execution of single lines of code, including creation of instances and method calls. This enables experimentation by students, as well as a simple means of unit testing individual classes and methods.

Another type of tool to support beginning students is a specialized class library for novices. Predefined classes can give students practice using classes, creating instances and calling methods before they are ready to create their own. Class libraries

also enable students to create interesting applications with graphics or other complex actions very early in their education, thus providing quick gratification and motivation to learn more. Many of the libraries that have been developed implement simplified graphics, such as Java Power Tools (Rasala et al, 2001). Widget classes draw simple shapes (Roberts & Picard, 1998); ObjectDraw (Bruce et al, 2001) enables the creation of simple animations and interactive games.

Program examples and assignments were created from these libraries, and from other sources. Some instructors provided classes that partly implemented a larger project; this enabled students to create more complex and interesting programs than they could by starting from scratch, and it taught students how to use pre-built class libraries (an important skill for professional software developers). Many such examples were shared via conference papers (Reges, 2000), (Popyack et al, 2000) and through SigCSE (see the "Nifty Assignments" web page, nifty.stanford.edu).

Another type of tool that has become popular in teaching object-oriented concepts is the microworld. Microworlds use visualization to help students understand concepts without the added complexity of learning code. For example, Alice is a graphical environment in which students create physical objects (instances) from categories or templates (classes). These instances are manipulated through a simplified language that's entered by selecting commands from pull-down menus in a highly-structured code editor (Cooper et al, 2003). Alice's developers compared the performance of novice students in a CS1 course using Alice to a second group of similarly experienced students who did not use Alice. The Alice group did significantly better in CS1, and more of them stayed on to take CS2. Another

microworld, Karel J. Robot, is an object-oriented adaptation of Karel the Robot, a microworld designed to teach procedural programming concepts. Karel J. Robot is a simulation in which the student moves one or more robots on a grid. Robots can pick up objects and interact with one another. A curriculum for a first object-oriented programming course based on use of the robot world is presented in the textbook *Karel J. Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java* (Bergin et al, 2005).

Finally, many educators have shared their curriculums and experiences through venues such as the ACM's Special Interest Group in Computer Science Education (SigCSE). But in spite of the successes of some instructors, many others still report difficulties experienced by students (McCracken et al, 2001). The reasons for this are a subject of ongoing debate in a variety of venues, including the SigCSE members' mailing list (Bruce 2004). Some of the major points debated in the exchange were:

- Object-oriented concepts incorporate a higher level of abstraction than procedural concepts, making them more difficult to learn.
- Many faculty members lack an in-depth background in object-oriented programming, such as experience developing applications on a larger scale than those typically assigned in beginning or even higher-level CS courses.
- Most faculty lack training in teaching techniques specific to the requirements of object-oriented programming.

The last two points were posed by two educators with many years of experience teaching freshman and sophomore computer science courses and writing textbooks. The object-oriented paradigm poses a significant learning curve for experienced computer scientists, whether within academia or industry. At many

schools, the change to Java forced a move to an object-oriented approach in a short timespan, making it difficult to gain such experience before designing and teaching a course. While an instructor might be learning a new language (and programming paradigm), he must also be assembling a new curriculum from a variety of sources. Dozens of curriculums, materials, program assignments and examples, class libraries, IDEs and microworlds exist. However, very few of them are presented as complete courses (with textbook, tools, and examples integrated in the curriculum), and very few have more than anecdotal evidence as an evaluation of their effectiveness. Most instructors are thus on their own to put together a complete curriculum from the many pieces that are available. This underscores the need for a complete solution to help instructors deliver a pedagogically effective course.

The *Objects First with Java* textbook along with the BlueJ IDE provide the most complete curriculum solution for a truly objects-focused Java course currently available. Object concepts are emphasized throughout and above all other topics, including procedural code structures. Primitive datatypes and variables, assignment statements, arithmetic, conditionals and looping are introduced in the context of example problems. Practice exercises for these concepts are limited to applying them to expand the functionality of the classes in the current sample application. No standalone exercises are provided. Thus, the textbook is a radical change from the traditional approach. This is almost certainly one of the reasons the text has not been more widely adopted; many instructors are wary of making such a drastic change in approach. Another reason may be the lack of scientifically gathered evaluation data to indicate whether such a curriculum is truly effective.

So with very little evaluation data on which to base a decision, what should be included in an effective curriculum? (McCracken et al, 2001) provides some clues. This study, conducted by ten computer science instructors from around the world, developed an exam which was taken by 216 students at four universities. The students had completed the first year of an undergraduate computer science program, and had learned either C++ or Java. The test required the student to complete a program to solve one of three problems (of her choice) in the programming language she had learned. The programs were scored on four criteria: *execution* (did the program run), *verification* (did it generate the correct output for a given input), *validation* (did the program do what the requirements specified) and *style* (were proper coding and commenting conventions followed). A second evaluation was also done, in which the program was assigned an overall score from 5 (complete, correct working program) down to 1 (not even close to a solution; indicates student had no idea how to approach problem). The results of both evaluation methods were dismal: an overall average of 22.9 out of 110 points for the four criteria combined score, and 2.3 out of 5 on the single evaluation score. There were differences between the four universities, but all had disappointing scores. Interestingly, there was no significant difference between the scores of students who used C++ and those who used Java. A characteristic of the data at all four schools was a bi-modal distribution: a large group of students scored at the bottom of the range (a single evaluation score of 1-2), with another group appearing at the top (a score of 4-5). While the paper explores possible reasons for this (different student backgrounds, for example), it does point out the possibility that the

teaching methods used may not be reaching students of varying needs and learning styles.

Perhaps most interesting is how the students did on each of the four criteria individually. Overall, they did best on style (an average 46.2%, as a percent of the maximum possible). Next was execution at 23.9%. Verification (the program produced the correct output for the given input) was 2.8% and validation (the program did was what specified) was 3.2%. This suggests that students could write syntactically valid code, following the style taught by their instructor. But they could not produce code that solved the problem. This suggests that the students' difficulty was with problem solving: understanding the problem's requirements and developing a logical plan to solve it.

The students were also asked to rate the problem's difficulty and write comments about how they approached the problem or what aspects they found most challenging. Those who scored a 1 on the single evaluation all considered the problem difficult, and none of their comments indicated that they applied any process or plan in devising a logical solution. Students who scored 5 often mentioned specific steps (for example, figuring out how to handle special input cases), or referred to a plan for the solution. This supports the hypothesis that lack of problem solving skills was a major cause of failure for the poor students. The study's authors come to a similar conclusion, and observe:

> "Students often have the perception that the focus of their first-year courses is to learn the syntax of the target programming language. This perception can lead students to concentrate on implementation activities, rather than activities such as planning, design, or testing... Students often skip the early stages in the problem-solving process,

> perhaps because they see these steps as either difficult or unimportant. It is also possible that instruction has focused on the later stages, with an implicit assumption that the earlier stages are well understood or easy to understand."[6]

Few CS1 courses, whether using a procedural or objects-first approach, emphasize design and problem solving. Even the BlueJ text, while introducing all concepts through examples implemented as sets of objects, examines Java code from day one. Rushing to write code seems to be a common impulse for novice and experienced developers alike. But novices often don't know where to begin when presented with a problem. A design methodology provides scaffolding that both gets a student started and guides him through the problem solving process. It is also language-independent; thus students learn a skill that will remain useful long after today's most popular language is replaced by the next. Before UML emerged as a standard set of processes for object-oriented design, instructors either created their own notations (Parker, 1995) or used an existing format with some modification (Barrett, 1996). Both of these instructors required students to turn in design diagrams, either with their completed programs or before. While neither did empirical experiments to measure the effectiveness of this approach, both provide anecdotal observations on their students' performance.

Some instructors have continued using a design methodology they developed specifically for their CS1 courses. (Adams & Frens, 2003) describe a methodology they call object-centered design, or OCD. OCD does not incorporate standard methodologies or diagramming techniques, but it does provide a logical process for

---

[6] (McCracken et al 2001), p. 133.

students to follow from the very start of tackling a problem. Adams and Frens initially used OCD to teach C++ programming, then refined it for use with Java; however, the process is not tightly tied to a particular language. Anecdotal evidence is presented to support the benefits of the process (fewer students report that they are at a loss of how to begin solving a problem). Students can also continue to use and benefit from the process in subsequent CS courses.

UML has become a standard as a set of tools and methodologies for experienced developers to apply to software design and problem solving. If beginning students learn even a subset of these processes, they can carry and build on that knowledge throughout their careers. (Tabrizi et al, 2004) at East Carolina University present this argument for teaching UML to beginning students. In a CS1 course, students created activity diagrams using Rational Rose. The instructor observed students in the lab spending less time on trial and error tinkering with code; students were able to clearly explain how their programs worked. He also noted that these students performed better in CS2, and attributed this to their continued use of the design skills they had learned in CS1. Unfortunately, in spite of having a distinct population of CS1 students taught by another instructor who did not teach design, the authors did not include any empirical data comparing the retention rates or performance of both groups in CS1 and CS2.

One drawback to teaching even a subset of UML using a professional tool such as Rational Rose is the time needed for students to gain sufficient skill with the tool. This takes time away from other topics in CS1. An alternative is to provide a simpler UML tool developed specifically for beginners and the curriculum being used. This is

the approach of instructors at the State University of New York at Buffalo. (Alphonce & Ventura, 2002) use UML class diagrams in CS1 to illustrate the design of solutions before any coding is taught. Objects and classes, inheritance, polymorphism and even simple design patterns are covered before any procedural code constructs including selection and looping. To ease students into creating their own designs, they are given designs and code stubs which they complete by adding code. The size of the starting design is reduced until students are expected to create their own, either from scratch or by applying the patterns they were taught. To reinforce the value of creating a design first, and to enable design creation to seamlessly flow into coding, the authors created their own interface called QuickUML (Alphonce & Ventura, 2003). QuickUML supports entry of class diagrams and generates code stubs. It also reverse-engineers class diagrams from code. This encourages students to use an iterative design process, in which an initial design can be modified based on implementation issues or changes in requirements that occur during coding.

The idea of teaching design as a problem solving methodology seems like a logical choice for beginners. Yet there are those who contend that beginners need to learn basic building blocks before they are able to grasp "big picture" issues such as design. (Buck & Stucki, 2000) support this argument using Bloom's taxonomy of cognitive learning. They correlate various software development skills with Bloom's six cognitive levels. Bloom maintains that before a student can learn the skill at a particular level, a minimum grasp of each prior level's skills must be attained. The CS skills Buck and Stucki attribute to each level in the taxonomy are coding-specific; only "system analysis", which they define as complex design for very large system

development, is included at the highest level. They do not mention problem solving as a prerequisite to coding, nor do they acknowledge problem solving as a skill required to produce even the simplest of programs.

Those who argue that computer science is really about problem solving rather than programming need to counter Buck and Stucki with successful examples of incorporating design as a pedagogy to teach problem solving. (Tabrizi et al, 2004) cite the successful teaching of problem-solving skills in other domains. (Popyack et al, 2000) describe their success teaching design skills to students in a breadth-oriented introduction to CS course. In spite of the limited time spent on programming, students were able to design and build a simple but interesting application using JavaScript. These individual successes support the idea that learning design is essential in helping beginning students build problem solving skills and reinforce object concepts.

All the beginning courses described so far that emphasize design teach it early, but either after or in conjunction with objects and classes. They use class or activity diagrams, both of which are at least one step removed from the problem specification. This research proposes a curriculum that begins design at the time the problem is presented, by creating use cases to guide the student through understanding the problem, identifying all the functionality it entails, and documenting that functionality in a more structured format. Objects are still introduced early, and in a context that provides students with heuristics to identify the objects, their attributes and behaviors in a given problem. This step in the design process is supported through the use of class diagrams, from which code stubs are generated, and smoothly leads into code implementation. Unit testing of individual objects is emphasized before coding a user

interface to tie the pieces together. This gives the student a clear path to follow from the initial problem presentation through completing its implementation. Empirical data on student performance were collected to confirm the curriculum's effectiveness.

Intelligent Tutoring Systems

1. Background

Computer technology has been used to develop a vast array of educational software, from early computer-based training systems to web-based adaptive hypermedia, multimedia courseware, and educational games. These systems have given students access to a great variety of pedagogical approaches that both supplement classroom learning and provide resources outside the classroom. This variety has been helpful in reaching students who don't do well with traditional lecture and textbook instruction.

Intelligent tutoring systems, or ITS, is a distinct category of educational software that meets two criteria. First, an ITS must provide individualized instruction. That means that the feedback, sequencing of material and other interactions with the student must be uniquely adapted to that student's needs, based on his interactions with the system over time and his knowledge of the subject matter. This individualization is more specific than in adaptive hypermedia, which chooses the order and format in which pre-defined material is presented based on the student's behaviors.

Second, an ITS must have one or more intelligent components. (Shute & Psotka, 1996) define three required components:

- An expert module, which is able to solve the problems presented to the student, and can interpret and evaluate the student's actions;
- A student model, which tracks the student's actions and maintains a model of the student's knowledge level of each concept in the domain;
- A pedagogical module, which chooses the content and timing of feedback based on the history of feedback provided so far, and the student's knowledge level

Intelligent tutoring systems are designed to simulate the actions of a human tutor. Bloom cites one-on-one tutoring as the most effective form of instruction; on average, performance of students receiving instruction from an experienced tutor exceeds that of students receiving only classroom instruction by 2 standard deviations (Bloom, 1984). Unfortunately, providing a tutor to every student who needs one is not possible. Software that mimics a human tutor, even if only partly as effective as an experienced tutor, could make a significant difference to a student who is struggling. This has inspired the development of many tutoring systems in a variety of subject areas. The following section explores several successful ITS in problem-solving domains.

2. Successful Systems

Carnegie Mellon University is the birthplace of a number of highly successful ITS, and many of those systems are based on the work of John Anderson. Anderson holds degrees in Psychology and Computer Science, and his primary research interest is in modeling human cognition. He developed a theory called ACT-R (Adaptive Control of Thought – Rational), which describes a model for memory, problem solving and learning (Anderson, 1993).

ACT-R defines two different types of long-term memory: declarative and procedural. Declarative memory consists of facts; procedural memory is the knowledge of processes, or how to do things. In ACT-R, procedural knowledge is represented by production rules that consist of condition-action pairs (essentially if-then statements). Each production rule also has a goal, which is the desired end result of the action. Many problem-solving tasks can be represented by a model that includes the necessary declarative knowledge and a set of production rules. Each production rule in a problem-solving episode represents a sub-goal, each of which contributes to achieving a final goal.

ACT-R models are translated to LISP for execution. An ACT-R model for a domain solves specific problems by selecting a production rule that matches the conditions that hold at each step. This matching is a fuzzy one; the closest match is chosen for each step. In addition to solving problems within an ITS, the ACT-R model can also be used to identify each step the student employs in a problem-solving episode. Each intermediate step the student enters is compared to the result of other production rules if it does not match the correct step. Thus the ITS can identify the faulty reasoning applied by the student, and tailor its feedback to correcting that specific misstep.

The most successful tutor built on ACT-R is the Pump Algebra Tutor, or PAT (Koedinger, 2001). PAT is based on the Pump curriculum, which was developed jointly by Pittsburgh public school teachers and Carnegie Mellon researchers. The Pump curriculum teaches beginning algebra in the context of applying it to practical real-world problems. One of the basic principles of Pump is that students already have

some informal knowledge of applying mathematics to solve word problems. Often students can correctly solve a simple problem, but not construct an equation that represents their calculations symbolically. By asking a student to provide a simple solution before modeling the problem with an equation, then reflect on what she did to solve it, a teacher can build on the student's informal knowledge to help her learn how to model the algebraic equation. The curriculum also requires the student to model each problem several ways, with graphs and tables, before writing equations. This guides the student to reflect on the problem at a deep level and from several different perspectives. Most class time is spent on hands-on problem solving and collaborative learning in small groups.

In PAT, a problem description is displayed in one window; in two additional windows, the student creates a table and a graph. The table has columns for the relevant quantities. The student can fill in values for the quantities, and must eventually enter formulas for each. The student must also identify units for each quantity. Next, the student graphs the tabular data, and uses these representations and formulas to answer questions about the problem.

As a student works through a problem, PAT models each step with a production rule. It compares the result of the student's action to the actual solution; if the student makes an error, feedback is given immediately. PAT's production rule set contains "buggy" rules that represent common errors; if one of these rules is matched to the student's action, advice can be specific to the particular misunderstanding. A student can ask for help at any time; because PAT builds a model of the student's

reasoning, it knows exactly where the student is in the process and can recommend an appropriate next step.

To enter new problems into PAT, a teacher types in a problem description and the complete solution. PAT matches table and graph elements to phrases in the problem text; these matches are verified, and if necessary, revised by the teacher. PAT then builds a set of production rules to create the solution.

Experimental evaluations of PAT and the Pump curriculum in schools in two cities over three years showed that students using PAT scored fifteen to twenty-five percent better than students in traditional algebra classes on standardized tests; on problem-solving tests they did between fifty and one-hundred percent better (Koedinger et al, 1997); (Corbett et al, 1999). PAT and Pump are used at over 75 schools around the country. They are commercially available through Carnegie Learning, a company formed to market and support PAT and other products for high school mathematics. Similar tutors have been developed for advanced algebra, probability and statistics and geometry.

Another successful tutor in a problem-solving domain is Andes (Gertner & VanLehn, 2000). Andes helps students solve problems in Newtonian physics. It is designed for use in an introductory college-level or high school Advanced Placement physics course. It was developed at the University of Pittsburgh and tested at the U. S. Naval Academy.

A student using Andes is presented with a text description of a physics problem and a picture of the situation. Below the problem is a graphic editor in which the student enters a free body diagram. To the right of the problem are two windows:

one in which the student enters equations used to solve the problem, and another in which every variable used in the equations is defined. To add a variable, the student can add a label to an item in the free body diagram, or he can use a variable definition menu. The student is prompted to choose an element from the problem description when defining a variable, to ensure that Andes can interpret what the variable represents.

As the student enters solution components, correct components are colored green and incorrect steps red. The student may request help to correct an error. On the first request, a short hint that reveals only a small amount of information is displayed. Successive requests for help yield more specific hints, and eventually bottom-out at telling the student how to fix the error. By providing less information early on, the system encourages the student to correct the error on her own.

A student who is stuck may also ask for help. Andes will select a next step appropriate to the student's partial solution, and provide a general hint on the topic related to the step. As in the error help, each successive request for procedural help will yield a more detailed hint until the student is able to enter a valid next step.

Andes stores the solution(s) to a problem in a solution graph. The graph may include nodes for more than one solution, and buggy nodes that represent common misconceptions. An instructor interface is provided to add new problems, both in the text plus graphics format that the student sees, and in a specially-coded form that Andes uses to generate the solution graph. To determine whether a step entered by a student is correct, Andes attempts to match it to a node in the graph. The order in which the student enters the steps need not match the order of nodes in the graph

(there is often no required order of equations for a problem). If the student's step matches a buggy node, or no node, Andes flags the step as an error. Andes also finds the node that's the closest fit to the student's action. It uses that node to identify the specific error in the step, and provide hints targeted to that error.

When the student requests procedural help, Andes selects a node that is related to the last step completed by the student. If there is more than one node that is a good candidate, the node that the student is less likely to know (based on the student model) is chosen.

Evaluations of Andes were conducted in the introductory physics class at the U.S. Naval Academy every fall from 1999 to 2003 (VanLehn et al, 2005). Some sections of the course used Andes to complete homework assignments; control sections did their homework using pencil and paper. Although the sections had different instructors who did not assign the same problems, all students took the same exams and final. The students using Andes did significantly better on the exams and final than the students in the control sections. When students were grouped by major, the difference for engineering and science majors was not significant, but the humanities majors showed a great improvement using Andes. Students were asked on a questionnaire whether they liked using Andes and found it helpful; many engineering majors said they preferred using pencil and paper, but the humanities majors preferred Andes and indicated that it helped them successfully complete the homework. This suggests that the proficient students did not need the extra help and that Andes' scaffolding got in the way of completing the homework efficiently. That same scaffolding did provide appropriate support to those students who needed it.

VanLehn and his co-authors (VanLehn, 2005) compare the Andes results to the PAT evaluations (Koedinger et al, 1997), and postulate on why the Andes results, while good, are not as strong as PAT's. One major difference is that PAT was used with the Pump curriculum. Andes is not aligned with any particular curriculum, and is designed to be used with any textbook. Additionally, Andes covered only about seventy percent of the curriculum in the Naval Academy course. Comparing exam results only for the topics covered by Andes might have shown a greater benefit. Alternatively, Andes' effectiveness may be enhanced by aligning it more closely with the classroom instruction.

A number of interesting tutors have been developed for the domain of computer programming. One of the earliest is MENO-II, which helped students develop iterative loops in Pascal (Soloway et al, 1981). MENO-II focused on errors to diagnose student understanding and direct tutoring. Its main component was a "bug finder" that identified errors in a student's work from a catalog of known bugs. Each bug was associated with one or more misconceptions and advice on how to correct the bug. Unfortunately, this simplistic approach did not work well; in tests conducted in an introductory programming course, MENO-II found only twenty-two percent of actual errors (Soloway et al, 1983). However, an important lesson learned was that focusing solely on errors was not sufficient; an effective tutor would need much more sophisticated knowledge of problem-solving strategies, both to identify the missteps in a student's work and diagnose what those errors reveal about the student's problem-solving process.

LISP programming was a popular and much more successful domain for ITS. One of the first tutors based on ACT-R was LISPITS (Corbett & Anderson, 1992). LISPITS analyzes the student's code as he enters it, and provides immediate feedback when an error is detected. The feedback reminds the student of the current goal (which the tutor knows from the production rule that applies) and points out what is wrong with his code so far. If the student attempts to correct the error and fails, additional advice is given. The tutor will eventually give the student the correct next step rather than allow the student to flounder. LISPITS accompanies an introductory LISP textbook (Anderson et al, 1987) and was used for many years in the LISP course at Carnegie Mellon.

LISPITS provides little scaffolding for the student. A function template appears after the student types in a function name; the next part of the code that the student should work on is highlighted. Tutoring advice is limited to reaction to errors. The code is transferred to a LISP interpreter for execution after it is complete; the student may experiment within the interpreter, but the tutor is not active.

ITS researchers at Princeton saw a need for a LISP tutor that could walk a student through the reasoning behind creating a program, and so created GIL (Reiser et al, 1992). GIL provides a graphical interface in which the student links the steps to solve a problem in a graph. The student may work forwards from the starting condition, or backwards from the final goal. This helps the student visualize the reasoning process, and reflect on each action and its results. It also keeps the big picture in front of the student, so he doesn't get lost while focused on a particular detail.

GIL's knowledge representation is in the form of rules similar to those of ACT-R. But when GIL constructs a solution to a problem, it creates a graph that can be traversed in either direction, allowing GIL to interpret a student's steps no matter what direction the student takes. GIL also replaces the generic sub-goals of each step with the specific goal in the problem, and uses it to construct more context-sensitive feedback than LISPITS. More than one rule may apply at any point in the solution, and so a graph may have multiple paths leading to the end goal. As a student works on a problem, GIL constructs the student's solution graph, matching nodes in it to nodes in the solution graph. Thus GIL is aware of where the student is at in his solution, which enables GIL to provide "next step" help when the student requests it, as well as responses to errors.

The contrast between GIL's approach and that of LISPITS highlights that the design of a tutor depends heavily upon the curriculum and pedagogical goals on which it's based. GIL seems to be better suited than LISPITS to beginning students who are struggling with basic problem solving skills in the LISP paradigm. However, evaluations to determine how well students transfer from the heavy scaffolding of GIL to writing LISP programs on their own have not been done.

A third example of a successful LISP tutor is ELM (Episodic Learner Model). ELM is different from the previously described tutors in that it uses case-based reasoning, or CBR, to maintain a history of the student's past performance and to base feedback on that history. ELM has been used in two formats: ELM-PE (Programming Environment), an integrated development environment for LISP, and ELM-ART

(Adaptive Remote Tutor), a web-based introductory LISP course[7] (Weber & Schult, 1998).

Case-based reasoning uses a database of previous solutions for similar problems to derive a solution for a new problem. It is well-suited for domains in which procedural and declarative knowledge can not provide a complete definition. Chess is an example of such a domain. Expert chess players do not rely on procedural knowledge, but look for patterns in the position of pieces on the board. In fact, chess instruction makes extensive use of examples to teach strategies for different situations. Weber and Schult point out that examples are also used extensively in teaching programming. Analogy is frequently used to help a student solve a new problem by comparing it to a previously-solved, well-understood example. This is called case-based learning.

When working on a problem in ELM, a student enters his code into a LISP editor that provides code templates and catches syntax errors, so that ELM's analysis always begins with syntactically correct code. Like the other LISP tutors, ELM applies rules to its own representation of the problem to match the student's code so far. These rules include correct, bad (technically correct but not preferred) and buggy (representative of common errors) rules. If more than one correct rule applies, ELM chooses the one most frequently used by the student in past problem-solving episodes, or, if insufficient history exists for the student, the most frequently used rule in general. As the student works, ELM also builds a derivation tree of all the concepts and rules used by the student. The derivation tree for a complete solution is merged

[7] Available at http://www.psychologie.uni-trier.de:8000/projects/ELM/elmart.html.

into a single tree representing the episodic learner model. Individual problem derivation trees are not stored together, but are divided into snippets. Each snippet is merged into the learner model, which is organized by concept. The complete derivation tree for a single problem can still be extracted, but as the model is typically accessed by concept, the structure better supports retrieving past behavior for specific steps in new problems.

If a student is observed having difficulty with a step in a new problem, the concept related to that step is retrieved from the derivation tree, and a previously completed problem that applied that concept is identified. Feedback to the student in the form of a reminder of how the student applied that concept in the previous problem is presented. If the student did not use that concept in an earlier problem, a problem from the current chapter in the web-based course is used as the example, since the student would have seen it a short time before attempting the exercise.

Evaluations of ELM are limited to comparisons between students using ELM-PE (the standalone environment) and ELM-ART (the ITS integrated with a web-based course) (Weber & Brusilovsky, 2001). Not surprisingly, students using ELM-ART did better on exams and programming projects than those using ELM-PE. ELM-ART is a complete course that provides examples and explanations that the student can review as needed; this supplemented the classroom instruction to which students using ELM-PE were limited. A better way to evaluate the example-based tutoring aspect of ELM would be to compare ELM-PE to LISPITS, since the environments of both tutors are similar, and they could be used within the same or similar curricula.

Only a small number of tutors have been built for C++ and Java. One long-running research project that has resulted in tutors for a variety of topics in C++, Java and C# is Armuth Kumar's "problets" project[8]. Problets are tutors that model a single concept, such as expression evaluation, if/else statements and loops. A problet generates problems consisting of code segments that employ the concept. The student is then asked to interpret the code by showing the flow of control through the statements and the output they generate. The problet software analyzes the student's answer, finds errors, and supplies an explanation of the correct answer.

Problets can be used to generate quizzes or provide practice exercises. For the latter, students are given immediate feedback, then presented with a new problem to try again. This corrects student misconceptions early and reinforces the correct concepts through timely application. The tutor adapts the number and type of problems presented based on the student's performance (Kumar, 2005a).

Evaluation of problets has shown that they are effective at helping students understand procedural constructs and the logical actions they model (Kumar, 2005b). But problets do not help students apply their knowledge in writing programs. A tutor designed with this latter goal in mind is JITS (Sykes & Franek, 2004b). JITS (Java ITS) focuses on a small subset of procedural Java (variables, operators and loops). It presents a student with a problem, and a partial code segment which the student must complete. After the student enters his code, JITS parses it (essentially using the same process as a compiler). When an error is found, a module called an Intent Recognizer further analyzes the error to generate feedback that is more detailed and pedagogically

[8] www.problets.org

appropriate than typical compiler errors. Once the code has no syntax errors, JITS determines its correctness based on the results of its execution. To find specific errors, it uses a decision tree that represents each step in the creation of the solution and executes it, comparing the result of each step to the student's solution.

Published papers on JITS do not describe how problems are entered into the system, or how solutions are generated. No progress on JITS has been reported since 2004, and no evaluation results published. Therefore, it appears that JITS has not been developed beyond a prototype.

Even though object-oriented programming languages are the domains of Kumar's problets and JITS, neither addresses object concepts nor design. This research has found only one tutor that teaches object-oriented design: Collect-UML (Baghaei et al, 2006). Collect-UML assists students in creating a class diagram from a textual problem description. Its main window is a graphical editor in which the student enters classes and draws relationships using standard UML syntax. The problem description is displayed above the editor, and a separate window shows the tutor's feedback. When a student adds a class, attribute or method 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 is done for both pedagogical and practical reasons. Pedagogically, it emphasizes that requirements are the source of design elements, and that the user's terminology should be used whenever possible. Practically, it avoids the problem of natural language understanding to determine the student's intent from the name entered.

To obtain feedback on her design, the student submits her solution for evaluation. The five levels of feedback are 1) stating whether the student's solution is correct; 2) identifying the type of element (class, attribute, method, relationship) that is not correct; 3) providing a hint for one error at a time; 4) providing hints for all errors; 5) showing the complete correct solution. Each successively higher level of feedback is shown on each resubmission of the solution. However, the student can request a different feedback level at any time.

Student solutions are evaluated by applying a set of constraints which must hold. These constraints are defined generically and apply to all problems. Feedback text that explains the error in general terms is stored with each constraint. Specific solution information for a problem is entered in the form of tags that map words and phrases to design components (classes, attributes, etc.). These tags are embedded in the text but do not display in the student interface. Thus a single solution is entered for each problem, since a word or phrase can be assigned only one tag.

A short-term student model containing feedback history for the current problem is maintained until the student completes the problem. A history of all constraints applied to student work so far, including how often each was satisfied or violated, is also maintained. This long-term history is used to identify the types of problems for which the student needs additional practice, and to determine when problems that use new concepts may be introduced.

Collect-UML was evaluated in a study of 38 volunteers from a second-year software engineering course. The students had two weeks of classroom instruction and some hands-on practice in UML modeling. Each student took a pre-test, used the

system in a two-hour lab session, and then took a post-test. Student performance on the post-test was significantly better than on the pre-test (Baghaei & Mitrovic, 2005).

3. Expert Modules in ITS

The systems reviewed so far vary significantly in how they represent domain knowledge and how they apply it to both solution generation and student evaluation. This section will consider how the various approaches influence the pedagogical goals and effectiveness of the tutor.

Many early tutoring systems had "black box" expert systems that solved the problems presented to the student, but could not analyze the student's problem-solving strategies (Anderson, 1988). These types of systems could only verify if the student's action was correct or incorrect; they could not provide the reason(s) why. Students could be given hints suggesting the correct action, and even explanations as to why another action is the better choice, but student misconceptions could not be addressed directly. Counseling a student based on broad issues relating to an error is known as issue-based tutoring. This technique was used quite effectively in WEST, the implementation of the arithmetic game "How the west was won." When a student made an error, WEST suggested an alternate move and explained why that move would be advantageous. This approach encourages the student to reflect on his own strategy and why he made the error.

Issue-based tutoring may have been a good fit for a game like WEST, where each move the student makes is visible to the tutor. But in a more complex problem-solving task like an algebra problem, understanding the student's strategy is crucial to

providing corrective feedback. Bug catalogs, which list common errors made by students and the faulty reasoning that causes them, were quickly found to be inadequate in identifying actual errors in student work. The Pascal tutor MENO-II demonstrated that even an extensive bug catalog could identify only a small percentage of student errors.

A good human tutor follows a student's steps as she works through a problem. Doing so allows the tutor to identify which step has the error, and then provide corrective instruction specifically for that step. This is the rationale behind model tracing. Model tracing is the process of building a model of the student's reasoning as she goes along. Each step in the student's model is compared to a model of the correct reasoning that leads to a solution. The correct reasoning can include multiple pathways to the answer. The student's step at a given point is compared to all the valid branches. If a match is not found, feedback can be provided immediately.

Model tracing is not specific to any particular means of expressing the problem-solving logic of a domain. Andes uses a graph to represent the steps needed to solve a physics problem; PAT and other tutors based on ACT-R use production rules to model the logic. GIL and ELM are also model tracing tutors that use their own rule-based logic to analyze the student's work.

Model tracing requires the intermediate steps of a student's work to be transparent. In Andes, the student must show all equations. In PAT, the student constructs a graph, then a table of the variables in the problem. In LISPITS, the system watches as each element of a program is entered. (GIL takes this even further by having the student construct a graph of the steps used to reach a solution.) For

beginners first learning a new procedure, the additional scaffolding is beneficial: new students are reminded of the steps they should follow. The ability to identify the step in which the student first erred is also very beneficial. Feedback is focused exactly where the student needs it, and the tutor can intervene immediately, before the student goes down a wrong path and becomes confused or frustrated.

While bug libraries by themselves are inadequate in diagnosing errors, they can be helpful in giving targeted feedback for common misconceptions. An effective teacher recognizes common errors and knows how to address them; an automated tutor should too. Thus, many model tracing tutors include buggy rules that identify common errors, with feedback tailored to the misconceptions they reveal.

A model tracing tutor employs the same problem-solving process that the student is expected to follow. Thus, it is vital that the tutor use the same process being taught to the student. Defining this process is the first step in building a model-tracing tutor, as Koedinger and Anderson demonstrate in (Koedinger & Anderson, 1993). They describe two versions of a tutor to help students learn how to create proofs in geometry. The first version, called GPT (Geometry Proof Tutor), solved a problem by searching through all the definitions, postulates and theorems that could be applied to each step of a proof. With 27 rules defined in the tutor, an exhaustive search quickly became time-consuming: at the third level of a solution tree, over 100,000 inferences were possible. Although forward and backward searching and heuristics were used to limit the number of rules tested at each level, the solution generator was still slow.

System performance was the initial motivator for revisiting how the tutor solved the geometry problems. But Koedinger and Anderson realized that the original

process also limited the tutoring provided to the student. To become proficient, a student needs to develop more sophisticated strategies than simply searching for a rule that fits at each step.

To define a new strategy, Koedinger and Anderson asked an expert to verbally explain his thought process as he solved a problem. Rather than applying one step at a time, the expert made several observations about the givens of the problem, then noted general conclusions which could be made from those givens. One of those conclusions provided the supporting statement that led directly to the goal statement. As soon as the expert recognized this, he declared himself done with the problem, then went back and filled in the details.

Based on the expert's strategy, Koedinger and Anderson built a new cognitive model using ACT-R. The new process generates a proof by first building a schema that highlights the givens of the problem. The schema is compared to generalized schemas representing partial or whole scenarios stored in the expert module's database. Creating a solution involves matching schemas at successive steps in the process to yield a general solution. Details are filled in after the general solution plan is complete.

Changing the problem-solving strategy also required changing the tutoring strategy and the user interface that supports it. Icons representing generalized schemas were added to the interface to help students identify and apply them to a problem. The tutoring component focused on helping the student understand the general configurations and choosing the correct match for a problem. This change was natural because feedback is linked to the ACT-R production rules.

The new tutor, called ANGLE, was much more efficient at generating solutions than GPT. Students using ANGLE did as well as students using GPT on a posttest given after working with the tutor for eight hours. The ANGLE students did better on identifying the overall strategy for each proof, although they made more errors in the details of individual steps. The tutoring provided for filling in the final details to a complete strategy could be improved to address this weakness.

All of the model tracing tutors described so far offer convincing evidence of the benefits that model tracing brings to the tutoring process. The newest model for tutoring systems, constraint-based modeling (Ohlsson, 1994), claims to provide effective tutoring without a model of the problem-solving process. Constraint-based modeling does not model the process of deriving a solution; instead, it models the characteristics of a correct solution. A domain is modeled as a set of constraints of the form (C_r, C_s) where C_r is a relevance condition that must exist for the satisfaction condition C_s to be required to hold true. In other words, if C_r is true, then C_s must also be true. A set of constraints is created for an entire domain, then applied to individual problem solutions to discover errors. Feedback is attached to each constraint; this feedback is presented to the student for each constraint violation in the student's solution.

CBM was developed to replace both the expert module and student model of the traditional ITS architecture. Both of these components tend to be very complex, and require a significant amount of computation time. Constraint-based tutoring is very fast, since the processing required to evaluate a solution is essentially just matching patterns specified by the conditions to patterns within the solution. The

resulting model of the student solution is a list of violated constraints; a long-term student model can be created from the concepts relating to constraints that were applied correctly and those that were violated.

Ohlsson argues that effective tutoring focuses on student errors (Ohlsson, 1996), and therefore, identifying errors and providing targeted feedback for those errors is sufficient. Ohlsson also argues that a tutor should be flexible in allowing students to follow more than one problem-solving strategy, since multiple paths may yield the correct result. While this flexibility may be desirable for intermediate-level students who have enough experience to reflect upon and experiment with different problem-solving processes, beginners need to master a basic process before they can develop more sophisticated techniques.

Since CBM has no problem-solving knowledge, it cannot provide feedback while a student constructs a solution. It also cannot suggest a next step to a student who is stuck. Feedback is 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. CBM researchers have recognized this weakness, which is especially apparent in domains requiring procedural problem-solving. One approach used to address it has been implemented in ASPIRE (Mitrovic et al 2005), an authoring tool to assist educators in developing constraint-based tutors in any domain. The ASPIRE tool allows for entry of procedural steps, each of which is modeled by a set of constraints. The student's work is evaluated at each step, thus providing guidance at intervals throughout the problem-solving process. This technique was used with NORMIT, which teaches data normalization in relational database theory (Mitrovic, Martin & Suraweera, 2007).

Data normalization is a step-by-step process, with a well-defined result after each step. It is thus well suited to this approach.

Constraint-based tutors have been built and evaluated in several domains, including software engineering (relational database topics such as SQL and Entity-Relationship Modeling, and object-oriented design) and English grammar (pluralization and capitalization) (Mitrovic et al, 2001), (Mitrovic, Martin & Suraweera 2007). Numerous studies have shown that students using these tutors did indeed show significant learning improvement over students who did not use the tutors. Collect-UML, the object-oriented design tutor, is of most significance to this dissertation.

Collect-UML was not implemented using procedural steps, and thus offers advice when a student submits a solution rather than while the student constructs it. Collect-UML was subsequently enhanced to support collaborative learning in which students work together in small groups (Baghaei & Mitrovic, 2007), but its model as a single set of constraints to be evaluated at the completion of a problem did not change. Both the initial and collaborative learning versions were evaluated with second-year software engineering students, not programming novices. An interesting comparison might be to evaluate both the single-user and collaborative versions of Collect-UML with novice students, and compare the results to an evaluation of a tutor such as DesignFirst-ITS that provides advice while the student works.

Learning well-defined procedures for problem solving is central to developing proficiency in many domains, including algebra and physics. Novices may need guidance throughout the process, rather than at predetermined steps or the end of a

problem-solving episode. While constraint-based tutors offer advantages over model tracing tutors such as efficiency of operation and less complexity to build, further studies are needed to determine the domains and students for which constraint-based tutors are best suited.

Object-Oriented Software Engineering

1. Background

One of the primary goals of the field of Software Engineering is to provide a methodology to standardize the process of creating software, just as other engineering disciplines use standard practices and procedures to provide consistency, improve quality and gain efficiency in the process of designing and manufacturing products. The need for a software engineering discipline emerged early in the history of computer science. Initially, language and program complexity was limited by hardware, but software complexity grew rapidly as processors grew in power and sophistication. By the 1950s and early '60s, software systems reached a size and complexity which resulted in development taking much longer than planned, and quality becoming much more difficult to achieve. A famous example is IBM's OS/360 operating system, which was delivered one year late and millions of dollars over budget. That project was the basis of Fred Brooks' classic book *The Mythical Man Month* (Brooks 1975, 1995). Additionally, software had to be adapted to changing requirements, but maintenance of complex systems was a difficult and costly endeavor. The idea of applying engineering principles to software development began to be seen as a way to control complexity, standardize the programming process and

simplify maintenance of existing code. The term *software engineering* was formally coined at the 1968 NATO Software Engineering Conference.[9]

Early software engineering research focused on developing processes and procedures to tame code complexity and improve quality. Perhaps because these problems were so urgent, software engineering had a practical focus to provide timely solutions that could be implemented fairly quickly. As the predominant programming paradigm was procedural, early software engineering worked to improve the development process within that paradigm, rather than critically evaluate procedural programming in comparison to other paradigms. It took another two decades of research and experience in object-oriented programming to recognize its potential to address software complexity and quality issues.

Structured programming was the predominant software engineering methodology of the time. Its ideas included simple top to bottom flow of control with a single entry point and exit point, elimination of the GOTO statement, and division of logic into small code segments that could be tested individually. (A detailed catalog of the tools, techniques and processes proposed during this period, and a discussion of how they differed is beyond the scope of this dissertation.)

Structured programming influenced software development at all levels, including beginning programming, where one of its most significant impacts was seen in the Pascal programming language. Pascal was designed to facilitate adherence to structured programming principles. Its adoption by many colleges and universities

[9] www.wikipedia.com

supported the Computing Curriculum '78 recommendation that structured programming techniques be incorporated throughout the curriculum, including CS1.

As its name implies, structured programming focused on the *programming* phase of development rather than analysis or design. It defined how clear, concise code should be structured, not how to design a solution given a problem definition or requirements. Professional developers recognized that defining the requirements of a system and designing a solution are extraordinarily difficult and time-consuming steps in software development, and that allocating too little time and effort to these steps had a negative impact on later phases of a project. They also recognized that requirements continually change throughout the life of a software package. Thus software needed to be easily extendible without requiring major modifications to existing code. Structured programming made code easier to understand and modify, but it did little to minimize the changes needed to existing code to accommodate even minor changes to data or functionality. It also did little to promote code reuse in different systems. By the 1980s, software engineers began to understand how object orientation could address these problems, and research in object-oriented software engineering took off.

The following sections describe the dominant methodologies for object-oriented design.

2. Analysis and Design

Every program starts with requirements expressed in human language. Those requirements must go through several major translations to become an executable

software system. (Rumbaugh et al, 1991), (Jacobson, 1992) and (Booch, 1994) similarly define the necessary major steps.

Analysis is the process of creating a "precise, concise, understandable, and correct model of the real-world."[10] The idea that the object paradigm models the real world is important. Bertrand Meyer defined object-oriented software as an "operational model of some aspect of the physical world."[11] This implies a strong relationship between the prose requirements and the object-oriented model that represents them.

The analysis process serves several purposes in addition to translating the requirements into an object-oriented model. It helps the developer understand the requirements and verify their consistency and completeness. Thus, analysis is an iterative process: it identifies weaknesses in the requirements so they can be addressed, and the analysis applied again. The result is a more accurate and complete model.

Analysis does not take into account implementation issues such as programming language or hardware platform. These issues are addressed in the design phase. In design, the model is revised as needed to account for factors such as the user interface, physical hardware requirements and performance issues. The resulting model contains everything needed to define the architecture of the system and begin its construction.

---

[10] (Rumbaugh 1991), p. 148.
[11] (Meyer 1988), p. 51.

The current standard methodology for object-oriented analysis and design is the Unified Modeling Language, or UML (Booch, Jacobson & Rumbaugh, 1997). UML defines a process for developers to follow. It also provides a set of nine diagrams that support the process and document the resulting system design. Few developers use all nine diagrams in their models, but rather choose those that complement their own style and express the portions of the design they consider most important. The nine diagrams, with comments on their use, are:

1. Use case diagrams – These show the interactions between actors and the individual use cases in the system. Actors are external entities that interact with the system; they can be human users of the system or other computer systems or devices which provide services to or receive services from the system. A use case is a single interaction between an actor and the system. A complete use cases definition includes the steps performed in the interaction, including all decision points and alternative actions based on the result of those decisions. Use cases are very popular because they are written from the user's point of view, in the user's language. Thus, they can be developed jointly by both analyst and client, and the client can verify their correctness and completeness.

2. Class diagrams – These represent the classes that comprise the system, the structure of each class (attributes and methods), and the relationships between classes. The class diagram is a pictorial representation of the actual classes, and directly correlates to the implementation code. These diagrams are used in many tools which generate code in various target languages.

3. Object or Instance diagrams – These diagrams show instances of classes rather than the classes themselves. A common use is to illustrate examples of complex relationships between classes.

4. Sequence diagrams – While class and object diagrams show static class definition or examples of instances, sequence diagrams are a type of interaction diagram that shows how objects interact with one another during the course of a use case. Sequence diagrams specify when instances are created and show the messages that are passed between them and the order in which actions are executed.

5. Collaboration diagrams – These convey the same information as sequence diagrams, but in a different format. Generally, sequence diagrams are preferred.

6. Statechart diagrams – Statechart diagrams portray all the possible states of an object, the actions that initiate a change in state, and the order in which they occur in a given scenario.

7. Activity diagrams – Activity diagrams are similar to flowcharts. They show the steps in a use case graphically. The flow of logic is shown, including when instances are created, the actions carried out by each instance and the order in which they occur. They are useful in illustrating various objects' roles in use cases, and in further refining the logic in a use case as an intermediate step before coding.

8. Component diagrams – Component diagrams show the overall structure of the packaged code: all modules and how they are packaged (grouped). It is a high-level, physical view of the executable system.

9. Deployment diagrams – Also a physical view, a deployment diagram portrays the physical structure of a system, such as the location of all parts of an installed system, especially for distributed systems whose components reside on different hardware.

UML was developed for professional software engineers working on medium- to large-scale systems. Yet the benefits it offers, such as helping developers better understand requirements and providing a process for creating an object-oriented model, are invaluable to students as well. Some elements of UML are a better fit for beginners than others. Use cases are particularly well-suited, since they are expressed in natural language; there is no need to learn a special language or notation. Use cases provide a straightforward process to divide a large problem that combines multiple functions into smaller problems that can be solved separately. The act of writing out use case steps requires students to reflect on the requirements at a detailed level.

Activity diagrams have been used in at least one beginner course (Tabrizi et al, 2004), and their structure suggests why. The notation used is borrowed from flowcharts, which is quite simple and may already be familiar to some students. Activity diagrams are a visual representation of the steps in use cases, which may be helpful to students whose learning style is visual rather than verbal. They also show the objects and their roles in each step of a use case, which can help students understand how the objects participate in executing the tasks of the system.

Class diagrams are also a good fit for beginners. Class diagrams allow students to document the objects they identify, their attributes and methods, and relationships between objects. Their notation is simple, and portions can be introduced as needed; beginners only need to know the symbols that express the features they use. Class diagrams are direct representations of code structure; the ease with which students generate code stubs from a class diagram encourages them to design the objects first rather than jump straight to coding.

3. Identifying the Objects in a Problem

Use cases provide an excellent technique for breaking down the functionality of a system into smaller, self-contained actions. But they do not offer a procedure to identify the objects in a problem. Much has been written on the task of identifying the objects in a problem description; we discuss the most prominent research next.

One of the earliest techniques for creating programs from English descriptions was presented by Abbott (Abbott, 1983). His methodology includes generating procedural code as well as identifying the objects in a problem. He chose Ada as his

target language because of its "useful program design constructs" – particularly its support for object orientation. Ada terminology calls classes "data types" and instances "objects" of those data types. "Operations" (functions or procedures) are methods. Ada supports encapsulation through private data types and the separation of specification from implementation. Code reuse is supported through defining data types in packages, which can be compiled separately and stored in a sharable library.

Abbott's methodology starts with an informal English statement of the problem, which includes the steps that comprise the actions described. The second step is to "formalize the strategy" – extract from the English description the key programmatic components of the solution. He first identifies the data types, then objects (instances), then operators, and then selects control structures to carry out the logic of the solution using calls to the operators.

Candidate data types are identified as common nouns in the problem. A common noun refers to a general class of things in the real world. Those things must be discrete objects, not a mass. (What Abbott calls a "mass noun" is identified as a quantity rather than a collection of individuals; "water" and "money" are mass nouns, as opposed to "employee" or "car.") A common noun is distinguished from a direct reference, which refers to a specific occurrence of the noun ("a date" versus "today's date"). Direct references are also important in Abbott's methodology – they suggest objects (instances) of the data types.

Abbott identifies operations as verbs in the problem description. A verb indicates an action that is to be performed on the object(s) contained in the sentence. For example, a sentence "When the customer has completed his order, the system

prints an invoice" would yield an operation "print_invoice." Abbott also creates operations for three other cases. "Predicates," or statements that can be evaluated as true or false ("If the Submit button is pressed..."), yield functions that return a Boolean value. Attributes are characteristics of an object whose values can be queried. For example, "display the customer's current account balance" would yield a "get_balance" method for account. "Descriptive expressions" are phrases that describe a condition that exists or can be determined involving one or more objects; for example "the number of days between two dates" would yield a "days_between" method.

Abbott's technique first identifies parts of speech, but then depends heavily on semantics to make important distinctions. Word meanings must be applied to distinguish between common and mass nouns and identify direct references. The definitions of descriptive expressions and predicates are based on semantics. Abbott acknowledges that although his strategy may appear mechanical, it requires real-world and domain knowledge, and its automation is beyond the "current (1983) state-of-the-art computer science."[12] It does, however, provide a clear process for human developers creating well-designed, object-oriented Ada programs.

(Booch, 1986) is a paper which heralds the benefits of object-oriented development at a time when functional decomposition and procedural programming were still the dominant methodologies. Booch presents a clear definition of object-oriented development, an explanation of the major steps in the object-oriented design process, and a comparison to a functional approach that highlights the advantages of

[12] (Abbott 1983), p. 884.

object-orientation. He presents five major steps of object-oriented design and then applies them to a real problem. Booch's methodology not only is the basis of his work in developing more detailed methods and design tools which led to UML, but it also inspired many other developers and researchers.

Booch emphasizes that object-oriented software models the real world. He adheres to object-oriented ideals more strictly than Abbott, but employs some of Abbott's ideas in his own process. Since objects are representations of things and ideas in the real world, Booch looks at the nouns in a problem description to suggest possible objects in a system (the first step of his process). He refines his list of objects based on the semantics of the problem, and an understanding of the domain.

The second step in Booch's methodology is to identify the operations performed by and required of each object. While Booch does not refer to parts of speech, he does return to the problem definition to find the operations each class must do. But analysis of the problem statement is not sufficient to identify all the elements of a complete and accurate model. He emphasizes an understanding of the problem statement *and* domain, and states the need for coupling methodologies for defining and analyzing requirements with the object-oriented design process.

Booch focused on developing methodologies and tools for human analysts. Other researchers worked on creating simpler, more "cookbook" type procedures that made the process less intuitive and skill/experience-intensive, and more mechanical. Many of these attempts employed and expanded Abbott's ideas. (Wirfs-Brock et al, 1990) outlines a procedure employing grammatical analysis, then demonstrates it with two example problems. Wirfs-Brock begins with finding the noun phrases, while

carefully examining the structure of each sentence and applying some standardization. Sentences in passive voice should be restated in the active voice, to ensure that any implied subject is explicitly stated. Plural noun forms should be changed to singular, and synonyms grouped together; the most meaningful term should be chosen as the standard to represent the object. Caution should be used in handling nouns that appear with different adjectives: do they represent different classes (perhaps related through inheritance), or different instances of the same class? For example, a problem that describes checking accounts and savings accounts is best modeled with two separate but related classes (both subclasses of class account); an air traffic control system that identifies the initial approach fix (position) and final approach fix for an instrument approach into an airport is describing different instances of the same object (position).

Some of the nouns in the list will be attributes of other objects; Wirfs-Brock determines this based on semantics. The data type of each attribute is determined to decide if an existing class or primitive data type can be used, or if a new class must be created.

A final source to consider while adding candidate classes to the list is the user interface, and any other interfaces between the system being developed and external systems or databases. The latter will likely be described in the problem and will appear on the list; the user interface class(es) may need to be added based on what the developer knows about the hardware on which the system will be deployed. These classes may follow standard patterns based on the implementation platform.

Once the list of objects has been refined, Wirfs-Brock turns to identifying the responsibilities of the objects. She defines responsibilities as both the data the object

must maintain and the actions it performs. Some attributes may come from nouns that were rejected as classes; others are determined through the semantics of the problem description and domain knowledge. Verb phrases should be examined as possible behaviors of objects; here again Wirfs-Brock's methodology is dependent on the semantics of the description, and is expressed as rules-of-thumb for human analysts rather than an algorithm that might be automated.

Bertrand Meyer, in his 1988 book *Object-Oriented Software Construction*, acknowledges Booch's previous recommendations on parts-of-speech analysis, and offers some heuristics on improving those results (Meyer, 1988). While nouns do provide a good first cut at a list of candidate classes, that list is likely to contain too many. It may include nouns that represent simple values, with no interesting (from the problem's perspective) behaviors. While sometimes this may be an obvious determination (for example, the balance in a customer's bank account is a simple value which most likely would not merit being a class in any scenario), in other cases it may depend on the problem specifics. In an air traffic control system that tracks the position and speed of airplanes, position may simply be a pair of latitude and longitude values, or it may also have necessary operations (a calculation of the distance from one position to another, for example).[13] In the latter case, position should be defined as a class.

Another common error that Meyer points out is defining a class that should really be a procedure. A class should be defined around an object that has characteristics (data) and services that can be performed on the object; it should not be

[13] (Meyer 1988), p. 328.

defined around an action. For example, withdrawal is a noun (as in "make a withdrawal from a bank account"). But a withdrawal is really an action, not an object. It is better suited as a service performed on an object (as in "*withdraw* an amount from an *account*").

Meyer refined his ideas on the process in the second edition of his book (Meyer, 1997). Interestingly, he criticizes the idea of grammatical analysis of prose requirements as far too simplistic to be useful. He restates the problem of the nouns in the description suggesting too many classes, which can be reduced by the heuristics described in his earlier edition. But classes can also be missed, for several reasons. One reason is simply the way the problem is stated. For example, in a system that controls the elevators in a building, a sentence might read, "A record is written every time the elevator moves from one floor to another." "Move" is a verb in this sentence, yet the system looks at a move as a noun: a move from one floor to another. A move can have attributes like starting floor, ending floor and time of move; it might have an action like "record move." Restating the problem can fix this. ("A record is written for every move of the elevator from one floor to the next.") But this imposes restrictions on the language used to describe the requirements.

A second source of missed classes is that useful abstractions may not be directly stated in the requirements. A text editor has a notion of commands which may not be explicitly defined, yet is important to the design of a solution. A third source of missed classes is not including external sources of useful classes, such as libraries and already existing parts of a larger system. This misses opportunities to reuse code, one of the most valuable benefits of object-oriented design.

These points are serious problems to be addressed in any attempt to automate design creation. However, Meyer does provide some additional ideas that could be used in automating the design process. For example, he recommends using design patterns whenever possible. Design patterns are designs that have been recognized as standards for implementing certain types of problem configurations that appear in many different applications. An example is the Model-View-Controller pattern, which separates the user interface (View) from the application logic (Model). The Controller acts as the intermediary between the two, invoking the appropriate portions of the Model based on user actions in the View, and selecting the appropriate portions of View based on the result. Patterns are a way to standardize the construction of software, much like standard processes in other engineering disciplines that have increased manufacturing efficiency and improved the consistency and quality of the end product. (Gamma et al, 1995) was the first widely accepted published catalog of design patterns for object-oriented software engineering.

Even while discounting the use of grammatical categories, Meyer's 1997 book still makes rule-of-thumb recommendations. Terms that occur frequently in the requirements and terms having explicit definitions are good candidate classes. Inappropriate classes may be eliminated by verifying that the class name does not reflect an action or primarily function as a verb (such a class should probably be a method). It should also have at least one method, not just simple data items, and it should represent a single physical thing or idea.

These ideas, as well as those of Abbott, Booch, Wirfs-Brock and others, have fueled numerous attempts at automating the design process. Some of these projects will be reviewed next.

4. Automating the Design Process

There are several significant benefits to automating the software design process that make even a partially successful attempt attractive. Software design is a time-consuming, labor-intensive process requiring highly-skilled analysts. A tool that could analyze prose requirements input by a non-technical end user could create a basic starting design which a human analyst could further refine. It could also find gaps in the requirements through an evaluation of the generated design and request more information from the client without intervention by the analyst, thus assisting the user in defining more complete and logically correct specifications. Additionally, an automatically generated design can be trusted to be an accurate representation of the requirements. If the design can be modified only by changing the requirements and regenerating, one can ensure that the two are in sync with each other.

These last two benefits – helping the user define more complete and correct requirements and tying the requirements more closely to the system design – are the goals of Circe, a web-based requirements engineering tool developed at the University of Pisa (Ambriola & Gervasi 1997a, b). Circe allows non-technical users to enter system requirements in natural language, with a minimum of structural rules that constrain how the data is entered. The requirements are then processed and five diagrams, including an OMT object diagram, are created. In addition, several error

reports that flag inconsistencies in the requirements (such as unused data and ambiguous or redundant requirements) are also generated. The core of the system is Cico, a natural language understander built specifically for Circe.

The requirements input into Circe must be broken down into simple sentences, each of which expresses a single requirement that makes sense by itself. Pronouns are not allowed. All domain-specific terms used in the requirements must be pre-defined in a glossary. Each glossary entry contains a natural language definition (used only as a human-readable comment), one or more Circe-defined tags that describe the role of the term (is it an action, a data value, input or output, or a calculated value, for example), and an optional list of synonyms. Cico processes the requirements one at a time by standardizing the text (removing articles, converting plural nouns to singular and verb forms to present tense), tokenizing it, and substituting synonyms and adding tags from the glossary.

Circe contains a database of rules called MAS (Model, Action and Substitution). These rules serve as templates that model different types of requirements and how they might be expressed. Each MAS rule also specifies one or more actions to be taken when the requirement is processed. Cico uses a similarity metric to find the closest-matching MAS rule(s) for a processed expression. More than one rule may apply to any single requirement. The actions specified by the rules are used by Circe to generate the design.

Cico works very differently from most natural language processors. Ambriola and Gervasi admit that it is a relatively simplistic system, but claim it is well-suited to the application and has performed well in their testing. They acknowledge its

limitations: it requires the pre-populated glossary and MAS-rules database to work for a given domain; its lack of "common sense" knowledge requires that all system requirements, even seemingly obvious ones, must be explicitly stated; and the format of the requirements has some restrictions (including that each requirement must roughly match one or more MAS-rule). In response to these points, they provide a MAS-rules database that they claim meets the needs of most applications, since the way in which requirements are stated differ little by domain. The ability to add new MAS-rules allows for flexibility in unique circumstances. They also maintain that the explicit statement of all requirements is desirable rather than restrictive, since one of the goals of the system is to provide complete requirements documentation that is accurately reflected in the resulting design. This also ensures that all design elements can be traced back to one or more specific requirements. (Any developer responsible for maintaining a large system has at least once encountered some odd feature whose existence is a mystery, as no one can recall the reason for its inclusion.)

Circe's interface allows for an iterative process. A user can process a set of requirements, review a list of inconsistencies, update the requirements, and reprocess. The user can also add to the glossary at any time. These steps require minimal assistance from a technical analyst. Special training or technical knowledge is limited to understanding the glossary tags, entering MAS-rules (which should be needed infrequently), and interpreting the error reports. Of course, the resulting design is intended for the analyst/developer, but the analyst's time needed to complete or correct the design should be minimized.

Circe's approach to converting natural language requirements to designs appears effective in the framework for which it was developed. But parts-of-speech analysis offers the promise of handling less restrictive prose input. A tool that employs this technique is LIDA (Overmyer, Lavoie & Rambow, 2001). LIDA (Linguistic assistant for Domain Analysis) is a tool for analysts, not end users. Any documentation describing a system is input as unrestricted text. LIDA then applies a general-purpose parts-of-speech (POS) tagger. It identifies multi-word phrases that likely refer to the same base concept ("temporary employee," "part-time employee" and "employee", for example) and counts occurrences by base form. LIDA then displays the text with all nouns and noun phrases highlighted, and, in a separate window, all base nouns in order of occurrence. The analyst then either tags each noun as a class, attribute, operation or role (actor) or skips it if it is not a part of the design. The analyst can easily see the context in which a base noun is used by requesting a display of all sentences containing the noun. She can also display all forms of a noun, and tag any form as a class. (For example, "part-time employee" and "employee" can both be tagged as a class; a subclass relationship can be added in a later step.) After reviewing all base nouns, the analyst does the same for adjectives and verbs.

When all parts of speech have been reviewed, the analyst selects the Model Editing Environment within LIDA. A window listing all the design components by type is shown next to a graphical editor that displays the start of a class diagram, with a box for the first class. The analyst adds attributes and operations to the class by selecting each from a list sorted by relevance (relevance is determined by proximity of the word to the class's base noun in the text). She can also add new attributes and

operations by typing in a new name. As each class is completed, the analyst adds a new class from the list to the diagram. At any time after at least two classes are defined in the diagram, relationships between classes can be added using standard UML notation. When the class diagram is complete, LIDA generates a text description of the model that can be reviewed with the end user to validate that the design meets the user's requirements.

LIDA's focus is assisting the analyst in the design process, in contrast to Circe's focus on assisting the user in creating complete and accurate requirements. Circe allows design changes only via changes to the requirements; LIDA allows new elements to be added directly to the design, and elements extracted from the requirements to be excluded. Another significant difference between the two systems is how they address the need for both semantic and domain knowledge. In Circe, semantic knowledge is stored in the MAS-rules used to translate each requirement; the glossary provides the domain knowledge. LIDA has only limited semantic knowledge via the third-party parts-of speech tagger it employs; it side-steps the need for deeper semantic and domain knowledge by delegating the tasks that need it to the human analyst. (LIDA uses MXPOST, a POS tagger developed at the University of Pennsylvania; see (Ratnaparkhi, 1996) for more information.)

A third and much more recently developed natural language design generator is Metafor (Liu & Lieberman, 2005). Whereas Circe's main concern is the completeness and accuracy of the requirements and LIDA's is assistance in the analysis process, Metafor attempts to provide some benefit to both.

Liu and Lieberman call a prose description of the functionality of a program a "story." An end user or analyst types into Metafor a story that describes activities that occur and objects that interact within a program. The story is expressed as though describing scenarios in the real world. One of the examples used to illustrate the process is a description of a Pacman game. "Pacman is a character who runs through a maze and eats dots," yields a class Pacman with methods run and eat, a class dot and a class maze. As each sentence is entered, Metafor interprets it and displays to the user the results in natural language and non-technical terms. "I created a new agent Pacman...I added the ability for Pacman to run..." The code that is generated is displayed in a separate window. The current version creates Python code, although the system is designed so that modules to generate code in other languages could be added easily.

The first step in Metafor's processing of a sentence is done by MontyLingua (Liu 2003), a general-purpose natural language processor. MontyLingua tags text using the Penn Treebank[14] tagset (Santorini, 1990) (Marcus et al, 1993), identifies proper nouns, standardizes word forms (converts verbs to present tense and nouns to singular) and extracts verb/subject/object tuples. It is based on Brill's trainable rule-based tagger (Brill, 1994). To improve the selection of the meaning and role of a word by taking into account its context, MontyLingua also uses ConceptNet. ConceptNet (Liu & Singh, 2004a, b) is a semantic network that connects words based on concept domains; its goal is to represent commonsense knowledge in a form that is accessible to computer applications.

[14] See http://www.cis.upenn.edu/~treebank/home.html

The verb/subject/object tuples are input to an interpreter that extracts objects and their related components and recognizes structures that suggest procedural logic (if-else, repetition, etc.). ConceptNet is used again to extract additional meanings, such as categories for adjectives (identifying that "red" is a color, for example). As code structures are identified, a solution model is maintained, and the effect of each change on the existing model is evaluated before it is applied. The interpreter also maintains a history of the discourse between Metafor and the user, and tracks changes in context as the dialog progresses. A separate code renderer translates the solution model into the target language.

Metafor has a feature to describe any code object using the user's own terms from her story. This feature, and the ongoing natural language explanations of how Metafor interprets each sentence, is generated by another component.

Metafor has undergone limited testing with intermediate programmers and non-technical users as a brainstorming tool. Input has been limited to simplistic problems, and output to code stubs rather than executable modules. But, the results are sufficient to suggest that the system can handle problems within the scope of a one-semester beginning programming course, and can generate program designs rather than code.

Novices in any discipline should be taught good practices from the start. Generating code directly from a problem description sets a poor example of software development practice. As currently designed, Metafor is not a good teaching tool for novices. But features like Metafor's natural language explanations and its interactive nature have significant pedagogical value, and their successful implementation in

Metafor suggests that similar functionality can be implemented in other environments designed to support a specific curriculum and pedagogical goals.

III. Methodology

Design First CS1 Curriculum

An outline of the final version of the Design First curriculum for Java, as taught at Dieruff High School during the 2005-06 school year, is included as Appendix A. The complete curriculum, including all lesson plans and student worksheets, is available at www.lehigh.edu/stem/teams/dieruff. This website also contains the curriculum used in 2004-05, which follows the same format and covers the same concepts, with different projects. While details such as the content and quantity of exercises and quizzes changed, the core curriculum and approach to teaching did not. Neither did the core principles on which the curriculum is based: learning object concepts *and* developing problem solving and design skills before coding. The basic tools (Eclipse with DrJava and UML plug-ins) and techniques (pair programming and project-based learning) also did not change.

Pair programming is a technique used in Extreme Programming (Beck, 2000). Extreme Programming (also called XP) is a software development methodology designed to be more responsive to changing requirements, while producing higher quality code than traditional methodologies. In pair programming, two developers are teamed together. One plays the role of "driver," entering code at the keyboard, while the "observer" reviews the code and makes suggestions. The programmers regularly switch roles.

Pair programming has been used with beginning students as a form of collaborative learning. Studies of pair programming in CS1 labs at North Carolina

State University reported that students found the labs more productive when working in pairs, and that the pair programming students actually did better on projects and exams than students working alone (Williams et al, 2002). The goal of pair programming is to encourage students to talk through problems together and learn from one another. Regular switching of roles discourages the same student assuming the bulk of the work. Supervision within the classroom at Dieruff also kept students on track when they worked in pairs.

Project-based learning is also an important technique in the Design-First curriculum. New concepts are introduced in the context of a larger project in which they are immediately applied. This conveys to students the relevance of what they are learning, and ties concepts to their practical application.

An overview of each unit in the curriculum follows:

Unit 1 introduces students to software engineering and the importance of planning before building. Students work through the steps of building a house, then compare those steps to the software development process. Unit 1 also introduces object concepts, including attributes, methods and instances. An exercise using simple shapes classes allows students to experiment with creating instances and manipulating them via method calls, while getting familiar with Eclipse and DrJava.

Unit 2 teaches elements of UML and the development process through a realistic problem (the Movie Ticket Machine and Automated Teller Machine problems were used). The students write use cases and identify actors; they also identify class(es) and their attributes and methods. The concept of unit testing is introduced at this point, and the students create a list of tests for their methods. Finally, the students

enter their class diagram into the Eclipse LehighUML plug-in. The importance of documentation is stressed, and students are reminded to enter comments for all their work, starting with the class diagram and continuing as they write their code.

Unit 2 also introduces the first procedural coding concepts: variables, arithmetic and assignment statements. Return statements are also covered in conjunction with methods. These concepts are reinforced through exercises completed individually. The students are now ready to write their first simple methods for the program's primary class. They work in pairs, switching typing and observing roles each day. Unit testing of each method is done through the DrJava interface.

Unit 3 teaches character-based printing and if-statements, which provide enough procedural coding knowledge to finish the remaining methods. All coding and unit testing, following the students' test plans, is completed.

Unit 4 covers how to create a simple character-based user interface, using the Scanner class for reading input, and the Integer and Double classes for data conversion. While and do-while loops and scope of variables are also covered. Students create a user interface for their program that allows for repeated execution. They now have a complete working program for the introductory problem, and are ready to go through the process on their own.

Unit 5 assigns students a new problem. Working in pairs, they go through the entire process, from use cases and class diagram, to a final working program with a character-based interface. Key deliverables along the way ensure that students stay on track, and that they get help with difficulties before their progress is hindered.

Unit 6 covers Java Swing, and teaches how to create a standard graphical user interface. The most commonly used elements (including text fields, radio buttons and dialog boxes) are covered. Students create a graphical interface for the project they built in Unit 5. An introduction to the online Sun Java documentation was included in the first iteration of the curriculum, but was moved to Unit 7 for the third offering of the course. The instructors included it at a point in which it fit in with the specific needs of the project; for example, students who implemented a calculator in Unit 5 used the Math class, and learned how to look up the functions they needed in the online documentation.

Units 7 and 8 cover topics addressed in traditional CS1 courses: sequential files, arrays, searching and sorting. These topics were covered in a second-semester extension of the Java course taught at Dieruff, during the spring 2006 term. They may be included in a first semester course based on the students' level and pace of the class.

Design First ITS

1. User Interface

The Eclipse IDE (integrated development environment) is the basic environment in which students enter and test their programs. The DrJava plug-in is also used to provide an easy means of experimenting with code one line at a time, and to facilitate interactive unit testing of methods. Since design is the focus of the curriculum, the IDE needed an additional interface that would allow students to enter a class diagram. It should also serve as a bridge between design and code by generating

the program structures into which students can add procedural code to implement their complete solutions. This interface was implemented as an Eclipse plug-in and called LehighUML. Figure 3 shows the UML editor window in which students enter a class diagram.

When a student saves her class diagram, a Java file for each class is generated. The java files contain the basic code framework for the class and method definitions. Attribute declarations are also included.

The UML plug-in can be used with or without the ITS. An environment variable that specifies the location of the ITS database is required to activate it; if present, the student is asked to log in to the ITS when the UML interface is opened. The Eclipse software, bundled with the LehighUML and DrJava plug-ins, may be downloaded from www.lehigh.edu/stem/teams/dieruff.

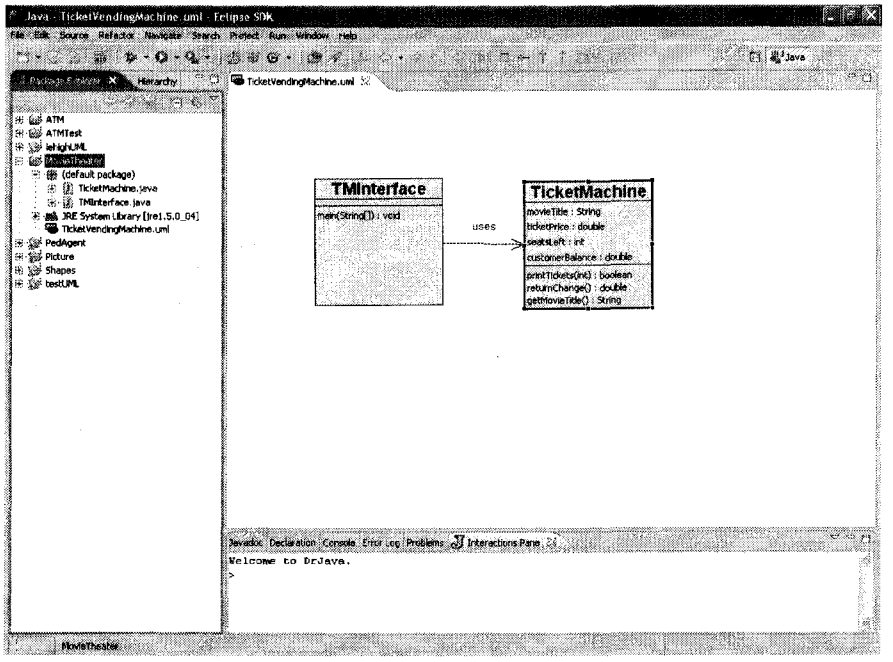


Figure 3: Lehigh UML Plug-in

2. ITS Architecture

The components of the ITS are shown in Figure 1 on page 21. When a student enters or changes a design component (class, attribute or method) in the UML editor, a record containing the details of that action is written to the ITS database. That record is read by the Expert Evaluator portion of the Expert Module, which maintains its own copy of the student's solution as it is built. The Expert Evaluator incorporates the action into the student's solution, then evaluates it for correctness. An "info" or "error packet" is written to the database depending on whether the student's action was deemed correct or incorrect. Both packets contain all the details of the student's action and the concept applied; the error packet also contains diagnostic information, including an error code and an alternate recommended action. The packet is written to the ITS database and read by both the student model and pedagogical module.

Expert Module

This section will describe how the Expert Module works. Each piece, the Instructor Tool and Expert Evaluator, will be covered separately. The general algorithm used by each component will be presented, followed by an example. First, two external software packages that are used by the Expert Module are described.

External Software

1. MontyLingua

MontyLingua is the natural language processor used in Metafor (see pages 93 through 95 of this document). The Instructor Tool uses it as the first step in extracting

design elements from an instructor's problem description. Specifically, the verb/subject/object tuples it identifies are the basis on which classes and their corresponding attributes and methods are determined. It also performs part-of-speech tagging, which is needed by the Solution Generator's algorithm to determine the role each word should play in the design. The part of speech is 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:

1. Compound sentences are only partially processed: "The ticket machine prints tickets and returns the customer's change." MontyLingua drops the second verb/object clause. The recommended workaround is to split the compound sentence into two simple sentences: "The ticket machine prints tickets. It returns the customer's change."

2. Dependent clauses are not recognized: "If the tickets are not sold out, the machine prints tickets for the customer." This should be split into two sentences. Another example is "The machine counts the number of tickets sold." "...tickets sold" implies "tickets that are sold," a dependent clause that describes tickets. This also should be reworded to avoid more than one verb in the sentence.

3. Appositives are not recognized: "An automated teller machine, or ATM, dispenses money." Do not include "or ATM." Specifying synonyms in the problem text is not necessary because they are pulled in from WordNet. If a desired synonym is not included automatically, it can be added manually to the solution template.

Rules for structuring the problem text so as to avoid these shortcomings are outlined in the *Instructor Tool User Manual*, which is included in this dissertation as Appendix C.

2. WordNet

WordNet (Fellbaum, 1998) is an English-language lexical database developed at Princeton University. It provides services similar to a dictionary and thesaurus. Additionally, it groups nouns, verbs, adjectives and adverbs into synonym sets, which are semantically linked. Thus words can be grouped by context. This helps determine the sense of a word in a given context, even when a word is polysemous (has more than one meaning). WordNet has analyzed large corpora and gathered statistics on the senses in which words are used; the synonym sets for each sense are ranked by frequency.

The Instructor Tool uses WordNet to clarify the role of a word or phrase in the solution. For example, sentence subjects are considered possible classes, but are eliminated based on specific criteria (see step 4 of Instructor Tool's Logic in the next section); objects are candidate attributes, except when they meet similar criteria (step 6). Nouns that refer to a person are labeled actors; this is determined by retrieving WordNet's definition and searching for key words or phrases that indicate a person, such as "someone who..." or "a person who..." A noun that represents an action or is frequently used as a verb is labeled a candidate method. This is also determined based on WordNet's definition. When multiple definitions exist for a word, the Instructor Tool chooses the most common use; the user can select a different definition and regenerate the solution based on the new definition.

WordNet is also used to identify synonyms within the context of the problem (eliminating duplicates as in steps 3 and 6), and to suggest synonyms to the instructor as she reviews and refines 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[15], was used to integrate WordNet into the Instructor Tool.

Instructor Tool

The Instructor Tool consists of two parts: the *Instructor Interface*, into which the instructor enters problem descriptions and views and revises generated solutions, and the *Solution Generator*, which comprises the core logic to analyze the problem text and generate a solution. The Instructor Tool may be accessed at moritz.cse.lehigh.edu/its/instr1.php.

1. Instructor Interface

The prose description of a problem serves two purposes. It is input to the Solution Generator for its analysis and solution generation, and it is the assignment which is given to the student. This ensures consistency between what the Solution Generator builds and what is expected of the student.

Problem statements must meet the following requirements:

1. *All aspects of the problem must be included in the description; that is, no implicit or assumed knowledge is required.*

[15] www.seas.gwu.edu/~simhaweb/software/jwordnet/

Pedagogically, this helps to clarify the problem for the student, and it sets a level playing field for all students. No student has a disadvantage due to lack of assumed prerequisite knowledge about the problem domain.

For the Solution Generator, this helps reduce the scope of the problem. The Solution Generator's natural language processor has a very limited amount of "common sense" knowledge of language semantics to improve the accuracy of the text analysis. It does not possess "real world" knowledge that extrapolates beyond the stated requirements. Any such capability is beyond the scope of this research.

Finally, this requirement allows the Instructor Tool to become an assistant to the teacher, to verify that all required information is included in the problem description and is clearly stated.

2. *No details beyond those required for a complete and correct solution may be included.*

This also has benefits for both the Solution Generator and the student. The ability to determine the relevance of distinct parts of the problem description is beyond the scope of the Solution Generator. Likewise, beginning students can not be expected to distinguish between pertinent and extraneous information. Providing unnecessary details can cause confusion and detract from the assignment's learning objectives.

3. *Details relating to program implementation (coding) must be excluded.*

DesignFirst-ITS emphasizes design *before* coding. Its primary objective is to teach object-oriented design. Coding is beyond the scope of both the Solution Generator and the ITS. The Design First curriculum introduces code elements and

procedural logic *after* students understand object concepts and can apply them to simple problem designs.

4. *The problem should be stated using simple sentence structures and active voice.*

The natural language processing software performs best with this type of language, and it is clearer for human students to understand, too. Compound sentences, dependent clauses and appositives should not be used, in order to avoid the shortcomings of the MontyLingua software as described on page 102.

5. *Problems must be at the beginner level. They should need no more than 5 classes (including the interface). The objects modeled are within the scope of "things" – either physical objects (like a ticket machine or a timer) or simple concepts that are easily understood (an account or invoice, for example). Inheritance is beyond the scope of the system.*

A brief overview of the Instructor Tool's interface and capabilities is now presented. A detailed discussion of how to use the Tool, including two examples stepping through the process, is in Appendix C: *Instructor Tool User Manual.*

The Instructor Interface is a web-based application developed using PHP. Problem descriptions entered through the interface are stored in a MySQL database, along with their generated solutions. The actual generation of solutions is performed on the server. Figure 4a shows the initial Instructor Tool menu, which allows selection of an existing problem for editing, or addition of a new problem. Figure 4b shows the problem description screen.

After entering a description, pressing the "Submit Problem Text" button at the bottom of the screen triggers the Solution Generator to analyze the text. The result of the analysis is a class diagram, as shown in Figure 5.

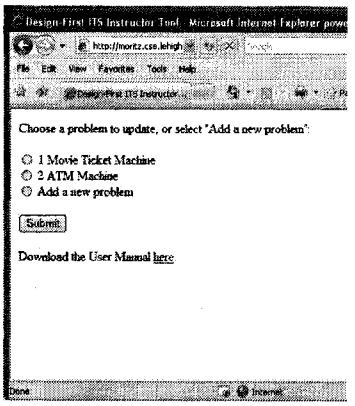


Figure 4a: Instructor Tool Menu

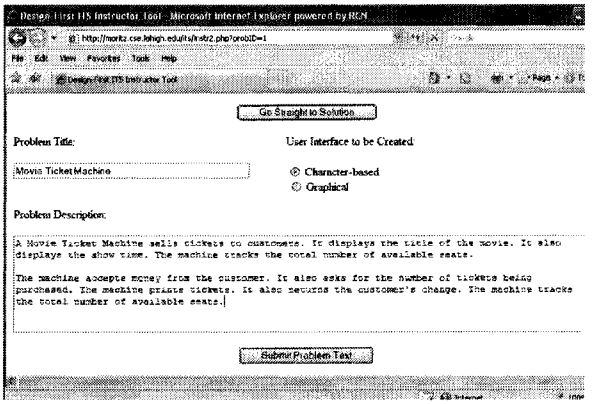


Figure 4b: Problem Description Entry

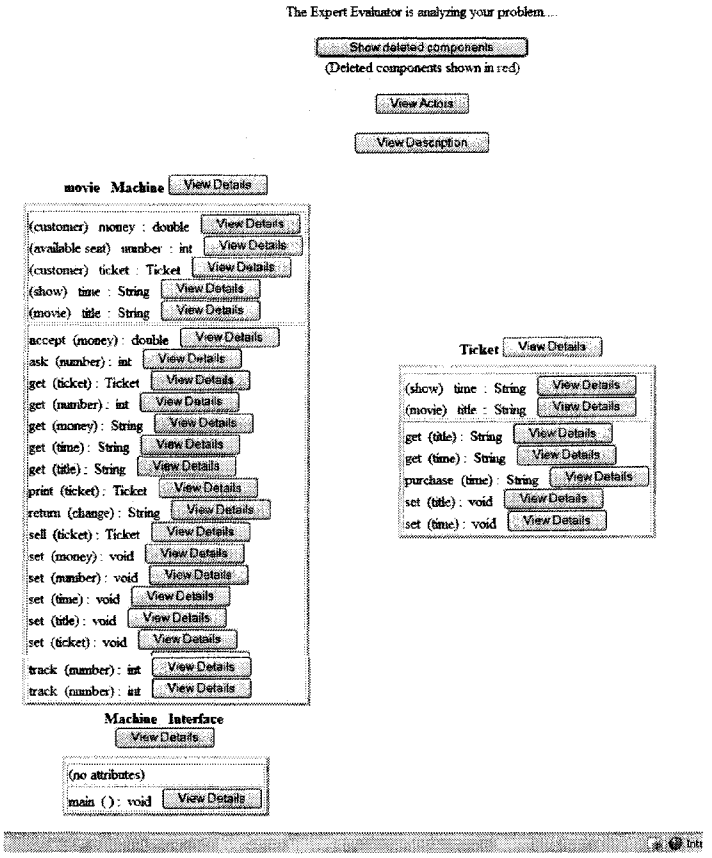


Figure 5: Class Diagram showingGenerated Solution

The instructor now reviews the solution. If there are missing components, the instructor should go back to the problem description, revise it and submit it for

reanalysis. This is consistent with the requirement that all aspects of the problem must be explicitly stated. This should be done before making any changes to the components, since resubmitting the text deletes the old solution and generates a new one from scratch. (A desirable future enhancement would be to analyze only text which was added to the description.)

The instructor should view the details of each component, starting with the classes. With the exception of datatype and returntype, the same fields are stored for classes, attributes, and methods. The fields are:

1. Component type – Class, attribute, method.
2. Component name – Always taken from the problem text.
3. Datatype plus two optional datatype fields – For attributes only.
4. Returntype plus two optional returntype fields – For methods only.
   Primary datatype/returntype is set by the Solution Generator, but can be changed by the instructor. The instructor enters values for the optional fields.
5. Source – The first sentence appearing in the text from which this component was derived.
6. Adjectives – Words that more specifically qualify the component name. These are taken from the problem text and from the WordNet definition for the component name. They can be added or deleted by the instructor.
7. Related terms – Synonyms for the component name. Taken from WordNet. Can be added or deleted by the instructor.
8. Senses – All the WordNet definitions for this component name, based on its part of speech as used in the text. The definitions are numbered in order of frequency of use. The Solution Generator always chooses the first sense; the instructor may choose a different sense, which invokes the Solution Generator to reanalyze the component.
9. Required/optional – Whether the component must appear in a correct solution. The Solution Generator labels get/set methods as optional, all other components as required. May be changed by the instructor.

10. Instructor Comments – Any comments the instructor may want to display to the student as feedback. Required for deleted components.

In addition to the class diagram components, the Solution Generator also identifies actors within the problem. They are accessed by pressing the "View Actors" button on the class diagram display. The instructor may add comments or delete an actor. An example of an actors list is shown in Figure 6. Only the fields shown on this screen are populated for actors.

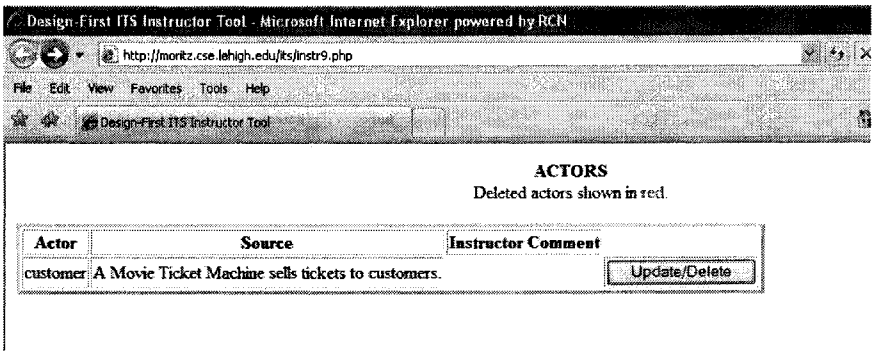


Figure 6: Actors Display

2. Solution Generator Logic

The Solution Generator portion of the Instructor Tool uses the following algorithm to derive design elements from the description:

1. Convert text to list of (verb, subject, object1, … ,objectn) tuples. This processing is done by MontyLingua.

Steps 2 through 5 deal with subjects:

2. Convert each subject that is a pronoun to its antecedent. This is done using simple "best guess" logic. For example, a pronoun used as the subject of a

subordinate clause will be replaced with the subject of the sentence's main clause.

3. Compile a list of subjects with no duplicates.
4. For each unique subject:
    a. Determine if it is a person. Examples include customer, manager, client. This will be done using WordNet. These will be tagged as actors.
    b. Determine if it is a simple value. This includes nouns like money, amount, balance, count, number, name, title. This will be done with a library of common "value" words that I will build and maintain. These will be tagged as possible attributes.
    c. Determine if it is a verb form. Examples are deposit, withdrawal, purchase. This will be done using WordNet. These will be tagged as possible methods.
    d. Remaining subjects are candidate classes.
5. Order the candidate classes by the number of times each appeared as a subject. Those that appear most often will be considered the most likely classes, and those with the highest frequencies are processed first.

Steps 6 through 10 deal with objects:

6. For all the objects in each tuple for a candidate class:
    a. Combine prepositional phrases with the preceding object they describe. This is needed because Montylingua lists each

prepositional phrase as a separate object in a tuple. For example, "name" and "of movie" would be returned as two separate objects.

b. Verify that the remaining objects are nouns or noun phrases. This is done as a double-check of Montylingua's output.

c. Tag verb forms as methods. Verb forms are identified using WordNet as in step 4c.

d. Remove actors.

e. Remove duplicates.

f. Remaining objects are candidate attributes of the class.

7. Set datatypes for the candidate attributes based on a set of rules. An internal database of commonly used nouns, such as number, count, price, cost and name will be used to look up the datatypes best suited for each.

8. Once the objects for all candidate classes have been processed, remove those classes that have no attributes.

Steps 9 and 10 add methods.

9. Examine each verb for each tuple for a candidate class.

   a. Create a method for the verb/object pair.

   b. Set the return type based on rules for common predefined verbs. For example, "display" will be associated with a "get" method, and the return type will be set to the data type of the object.

   c. Mark each method required.

10. Add get and set methods for each attribute that doesn't already have one. Mark each of these optional.

11. Add standard classes for the user interface. One class is added for a character-based interface; two classes (a window display and a listener) are created for a graphical interface.

The algorithm does not apply design rules based on implementation. For this reason, it does not generate parameters. For example, a rule of thumb for deciding whether a data element is an attribute or a parameter is whether the value needs to persist over the life of the object. Is it needed to perform a single task? Can it have a different value each time that task is performed, or must its value be maintained over several operations? As a result, the generated design typically has more attributes than required. Extra methods may also be generated, such as when an action is logically suited as part of another method. These situations are handled through the Instructor Tool, where the instructor can delete such elements and enter an explanation of why they are not appropriate. The instructor also has the flexibility to allow an element by marking it optional, and entering comments explaining why it is allowed but not optimal. Specific examples are discussed in the "Ticket Machine" problem in the next section.

3. Solution Generator Example

We now step through the processing for a partial text from a problem description:

> "The movie ticket machine displays the movie title. It also displays the show time and it displays the price of a ticket. A customer enters money and the number of tickets into the machine. The machine prints the tickets. The ticket has the show time and movie title. The ticket machine returns the customer's change. The machine also tracks the number of seats in the theater so it can tell the customer when the tickets are sold out."

1. Parse the code into (verb, subject, object) tuples using MontyLingua:

   ("display","ticket machine","movie title","showtime", "price","of ticket")
   ("enter","customer","money","number","of ticket", "into machine")
   ("print","machine","ticket")
   ("return","machine", "customer's change")
   ("track","machine","number","of seat","in theater")
   ("tell","it","customer")
   ("sell","movie")

2. Use "best guess" logic to resolve pronouns. In this case, "it tells customer" is a subordinate clause to "machine tracks number of seats in theater."

   ("tell","it","customer") -> ("tell","machine","customer')

3. List subjects with no duplicates:

   "ticket machine" (occurrences of "machine" are identified to be the same as "ticket machine")
   "customer"
   "movie"

4. Analyze each subject using WordNet and internal lexicon:

   a. "customer" is identified by WordNet as a person/role, so tag it an actor.

   b. Neither of the remaining subjects matches a simple value.

   c. Neither is a verb form.

   d. "ticket machine" and "movie" remain as candidate classes.

5. Order the remaining subjects by frequency:

   "ticket machine" appears 5 times
   "movie" appears once

6. Examine the objects in each tuple for "ticket machine." We have:

   "movie title"
   "showtime"

"price", "of ticket"
"ticket"
"customer's change"
"number","of seats", "in theater"
"customer"

a. Combine prepositional phrases with the preceding object they describe:

"price of ticket"
"number of seats in theater"

b. Verify that all remaining objects are nouns or noun phrases. (They are.)

c. Determine if any objects are verb forms, and if so, tag them as methods. (None are.)

d. Remove actors. We have one: "customer."

e. Remove duplicates.

We are now left with:

"movie title"
"showtime"
"price of ticket"
"ticket"
"customer's change"
"number of seats in theater"

7. Set data types for each attribute:

"number" -> int
"price", "change" -> double
all others set to String

8. Remove candidate classes that have no attributes. "movie" is removed because it is the subject of only one tuple, and that tuple has no object.

9. Examine each verb in each tuple.

a. Create a method for each verb/object pair:

"display", "movie title"

"display", "showtime"
"display", "price of ticket"
"print", "ticket"
"return", "customer's change"
"track", "number of seat in theater"

b. Set the return type based on rules for common predefined verbs. "display" and "return" both match "get" so the return type will be set to the data type of the object for those methods. All other methods' return types are set to void.

c. Each method is marked required.

10. Add get and set methods for each attribute that doesn't have one. Mark these methods optional:

    set "movie title"
    set "showtime"
    set "price of ticket"
    get/set "ticket"
    set "customer's change"
    get/set "number of seat in theater"

11. Add standard class(es) for the user interface.

The end result is shown in Figure 5 on page 107. Two problem-specific classes were created: Movie Machine and Ticket. Movie Machine is a required class; Ticket could certainly be a valid class that represents the physical layout of a ticket. A more advanced student might reasonably use it in her design, but an instructor may consider it acceptable, especially for a beginner, to forgo the Ticket class and embed the ticket layout in the print ticket method. The instructor indicates this by marking the Ticket class optional.

The Movie Machine has five attributes, four of which are necessary. The attribute "ticket" should not be included. Even if the Ticket class was considered a required part of the design, the Movie Machine has only one method that handles tickets: print ticket. It creates tickets when needed and dispenses them to the customer. They are not kept for later use. Thus ticket need not, and should not, be kept as an attribute. This is an example of a design decision related to the program's implementation. The instructor provides an explanation in the comment he enters when deleting the attribute.

Six methods (other than get/set) were generated for the Movie Machine. Three are necessary: accept money, print ticket and return change. The others must be evaluated by the instructor based on implementation considerations. Method "track number" is a good example. The problem states that "The machine also tracks the number of seats in the theater so it can tell the customer when the movie is sold out." To determine whether tracking the number of seats requires a separate method, one needs to know what steps it requires. In this problem, it is simply subtracting the number of tickets sold to the current customer from the number of available seats. An instructor could decide that because it requires only one line of code, it should not be a separate method. Or, he could mark the method as optional and enter "decrement" as a related term for the method.

The output of the Solution Generator, edited by the instructor using the Instructor Tool, is used by the Expert Evaluator as a template to which to compare students' solutions. It represents alternate solutions through the use of optional elements, and it provides diagnostic data by storing deleted elements.

Expert Evaluator

1. Expert Evaluator Logic

The Expert Evaluator receives a "Student Action" record from the LehighUML plug-in for each action completed by the student. The Student Action record contains:

- action ("add_class", "modify_attribute")
- name of the element
- its related element (for an attribute, its class)
- its datatype or return type (if applicable)
- student ID
- problem ID
- date/time
- unique sequential ID

The Expert Evaluator uses the Student Action record to update its own model of the student's solution. It then applies the logic outlined below to determine whether the action is correct or not. The result is an "info packet" for correct actions or an "error packet" for errors. Each packet contains:

- Identifiers (packet ID/count, student ID, problem ID, date/time)
- Student action, name, related element and data/return type as added by the student
- Concept applied
- Solution element it matched
- Instructor comments from the matched element

Additionally, error packets contain:

- Error code
- Text from the problem description pertaining to the element it matched, if any
- The matched field contains a recommended action, if no solution element matched.
- Correct concept, if different from the student concept

The algorithm below describes how the student's component name is matched to a name in the solution template. The same algorithm is used for all component types.

1. Compare the solution component name and each of its related terms to the student's name. The solution component name may be part or all of the student's name. Abbreviations and spelling errors for each term are also considered by applying two similarity metrics from an open-source library of string similarity metrics called SimMetrics, developed by Sam Chapman at the University of Sheffield, UK[16]. A similarity score is assigned to the comparison, with a value of 1 representing an exact match. Solution elements with a similarity > .8 are saved as possible matches. If none meet that criteria, no match is returned. If one or more is an exact match, all other matches are discarded. If there is more than one match, go on to step 2.

2. For each possible match, pull off the part of the student's name that matched in step 1 and compare the rest to the adjectives stored for all possible matches. If the comparison yields a similarity > .8, add that similarity divided by 4 to the solution component's total score. The adjective is given only 20% of the overall weight to the score; it is used primarily to break a tie on matches of the name or related term (for example, distinguish between attributes such as "customer balance" and "total balance"). The solution component with the highest overall similarity value is chosen as the match.

[16] www.dcs.shef.ac.uk/~sam/stringmetrics.html

The overall process to determine correctness is as follows:

1. Attempt to match the student's element to an element of the same type in the stored solution(s). Search elements not yet matched first.

2a. A match was found.

   a. If it's a deleted component, generate an error packet and return.

   b. Verify that it's not a duplicate in the student's solution. If it is, re-evaluate the previous duplicate element to verify that the Expert Evaluator did not misidentify it. (A change in the interpretation of a previously processed action will result in a new packet being generated for the changed element.)

   c. If it isn't a duplicate, compare the data or return type and check if the student entered a comment in the Javadoc field.

   d. Create an info or error packet based on the results of b and c.

2b. A match was not found.

   a. Compare the name to other element types in a predefined order. For example, if the element added was a class, compare it to actors, then attributes, then parameters[17] and finally methods.

   b. If a match is found, create an error packet. If the matched element is not deleted, insert the matched element name and type in the correct action fields.

   c. If no match is found, set the error code to indicate that the Expert Evaluator cannot determine the student's intent.

[17] The Expert Evaluator does not identify method parameters. To test the Expert Evaluator, method parameters were entered into the database manually. To fully integrate this functionality, adding a data entry page linked to the method page in the Instructor Tool is sufficient. A useful addition to the Instructor Tool would be an option to reclassify attributes as parameters.

2. Expert Evaluator Example

Suppose the student has added a class called "TicketMachine" which was identified as correct. She has also added valid attributes "movie" and "price." The next student action is adding attribute "cash" with data type int to the TicketMachine class.

1. Attempt to match "cash" to attributes of TicketMachine not yet matched:

   (customer) money
   (available seat) number

   Compare "cash" to "money" and its related terms, and to "number" and its related terms. "Cash" was not returned by WordNet when the Solution Generator added the attribute money, but the instructor could have entered cash as a related term. If he did not, the Expert Evaluator would not match cash to either attribute.

   If a match is not found, compare "cash" to attributes already matched using the same process.

2a. A match was found in step 1.

   a. Verify that it is not a duplicate in the student's complete solution.
   b. Compare the data type for the matched element (in this case double) to the student's data type (int). int is not listed as an acceptable data type.
   c. Create an error packet with the code "INC_DATATYPE".

2b. A match was not found in step 1.

   a. Compare "cash" to attributes for other classes, then to classes, parameters, actors and methods.
   b. Create an error packet with an error code indicating whether a match was found.

3. Evaluate Complete Solution

The logic outlined in the above two sections is carried out while the student's work is in progress. When the student has completed her design, she may request an evaluation of her final solution. This is initiated by the LehighUML plug-in, which sends a stream of "add" Student Action records (as though the student is re-entering her solution one element at a time). The Expert Evaluator processes the Student Action records exactly as described above, and generates corresponding info and error packets. An indicator field on the packets identifies them as part of the student-requested evaluation, so that the student model and pedagogical module can handle them appropriately. After the last record in the stream has been processed, a final comparison of the solution template to the student's solution is done to find any required elements that are missing from the student's design, and generate an error packet for each.

4. Operational Notes

The Expert Evaluator, which runs on a centralized server, depends on coordination with the student's work in the LehighUML plug-in, which executes on a networked PC. A student may enter a partial design, save her file locally, then resume work at a later time. The student's solution is also saved on the server, but the Expert Evaluator must verify that its copy of the student's work matches the local copy. Many events can cause the copies to be out of sync, such as a network failure, or the student could simply choose to work without the ITS by not logging in for a session. To handle these situations, the filename and timestamp of the local UML file are saved

in the ITS database. When the student opens an existing UML file, the LehighUML plug-in compares the stored name and timestamp to those of the local file. If they do not match, the plug-in generates a series of "add" Student Action records that the Expert Evaluator uses to rebuild its copy of the student solution. The Expert Evaluator also performs its normal processing, and generates packets with an indicator identifying them as part of the synchronization process.

IV. Evaluation

Separate evaluations were conducted for the Design-First curriculum, the Instructor Tool and the Expert Evaluator. Each is covered in its own section below.

Design-First Curriculum

The Design-First curriculum was taught over three semesters between Fall 2004 and Fall 2005 at Dieruff High School in Allentown, Pennsylvania. Each class was evaluated on project and final exam grades; the second two classes' evaluations included quiz scores. The project scores were an overall average of scores on each project deliverable: use cases, class diagram, program methods and unit testing, and completed program with character-based interface. The students also completed course evaluations. Each semester's evaluation results were used to refine and improve the curriculum for the following semester. Student average scores are shown in Table 1.

Semester	Quizzes	Project	GUI	Final Exam
Fall '04 (6)	N/A	92.9	95.0[1]	82.5
Spring '05 (10)	75.4	94.5	100.0[2]	78.7
Fall '05 (13)	80.0	88.0	N/A	76.9

Number of students per class are shown in parentheses after semester.
[1]5 students
[2]6 students

Table 1: Design-First Student Grades

None of the students had prior programming experience, although most had taken other information technology courses at Dieruff, including an introduction to computer hardware, networking and database courses. The first class was limited to six twelfth grade students who had registered for the existing programming course,

had proven themselves to be better than average students, and were highly motivated. Success with that class resulted in the Design-First curriculum replacing the existing programming course (which had used Scheme).

Classes met daily for 40 minutes. Because the first class was so small, sessions were informal. Lecture comprised only about one-quarter of class time overall; the rest was spent working on project assignments or completing scripted exercises or small standalone assignments within Eclipse. The students worked in pairs at their own pace, and all were able to complete the assigned project. One student did not complete the graphical user interface due to an extended absence.

The final exam covered both object concepts and procedural code. In the objects portion, students were asked to identify key components (such as objects, attributes and methods) within a sample problem. They were also given a class and asked to write code to create and manipulate instances. In the procedural section, they were asked to write short code segments which required them to declare variables, perform input and output and arithmetic calculations, and branching and looping logic. Sample exam questions are shown in Appendix B.

On the final exam, three students scored 90% or higher. One student ran out of time, but did well on all but the last problem. All students did very well on the portion of the exam that covered object-oriented concepts; most of the errors were on procedural code. No other tests or quizzes were given. One student actually recommended having tests to reinforce the material over the course of the semester.

On the course evaluations, students said they enjoyed working in pairs. As observed in the lab, the students did have substantive discussions about the code they

were writing. This observation was repeated in the other two classes as well. The students also reported that they enjoyed the hands-on work, but they asked for more exercises and in-class review of the more challenging topics. As a result, quizzes were added to the next iteration of the course, and more worksheets and practice exercises were assigned.

The second class had ten eleventh and twelfth grade students, and was more diverse in terms of ability and motivation. Students were allowed to pair with whom they chose, and generally teamed with students of similar academic ability. As a result, some pairs progressed much more quickly than others. By the end of the semester, there were two groups of students: one that advanced quickly and completed the project, the graphical interface and in some cases extra assignments, and another group that completed only the project. The distribution of the final exam grades also divided the students into the same groupings. Once again, the students who did poorly on the exam had more difficulty with the procedural code than with the object concepts. Even though much more time was spent on procedural topics in this class than in the prior one, the students did just as well on the objects portion of the exam as on the procedural portion.

The students in the final class were a much more homogenous group, all with average overall grades in their high school careers. They were also a mix of eleventh and twelfth graders. This group spent the most time on procedural concepts, and their quiz grades, about 80% of which covered procedural code, reflect it. Their project average is slightly lower than the previous two classes, possibly due to receiving less individual assistance from the instructors in the lab. (Even though there were only

three more students in the third class than in the second, three of the teams in the second class needed very little assistance, freeing up the instructors' time for the slower students.) The final exam average was lower than expected based on the students' work during the semester. However, the students' performance on the object portion of the exam was still stronger than on the procedural portion. Seven students in this class were motivated to continue the programming course into the second semester, and they completed Units 6, 7 and 8 of the Java curriculum. One student worked beyond the curriculum and learned the basics of Java graphics, which he used to create the geometric shapes for a Tetris game, including implementing their rotation and movement down the screen.

These results indicate that average high school students without prior programming experience can learn and apply object concepts before learning procedural code, and that Java is accessible to beginners. Many of these students did not have a strong mathematics background, which traditionally has been considered a prerequisite for success in computer science. Thus, this curriculum would be accessible to younger students who may have had no more than basic algebra. Offering the course to ninth grade students could open the door to more students having a solid foundation in computer science, thus making the field a more accessible and attractive option when they move on to college.

The curriculum is still in use by Chad Neff at Dieruff High School, and is also freely available to others via www.lehigh.edu/stem/teams/dieruff. An additional avenue of dissemination has been the National Academy Foundation, or NAF (www.naf.org). NAF supports high school "academies" which specialize in a variety

of subject areas, including Information Technology. Dieruff has its own Academy of Information Technology and is a member of NAF. Another NAF teacher, Alvin Kroon at Kamiak High School in Mukilteo, WA, used the curriculum in the 2006 spring semester, and presented his results in a workshop at the 2006 NAF conference. The workshop, titled "Design First, a Java Curriculum and a University Partnership," described the collaboration between Kamiak, Dieruff and Lehigh and presented the curriculum to NAF teachers from across the United States.

Instructor Tool

1. Evaluator Results

The Instructor Tool was evaluated by three experts in computer science education and/or software engineering. Sharon Kalafut is a Professor of Practice at Lehigh University, where she teaches first year computer science students using Java. Chad Neff teaches mathematics and computer science at Dieruff High School, and participated in the development and teaching of the Design-First curriculum. John Haselsberger is a software engineer with 25 years experience developing large systems, primarily for AT&T, Lucent and Agere; he currently develops Java applications for Cegedim Dendrite.

The evaluators were given the Instructor Tool User Manual and attended a brief presentation, which included an overview of the DesignFirst-ITS, the goals of the Solution Generator, and the rules for writing a problem description. The presentation also stepped through one example, from writing the description, to generating a solution and editing it with the Instructor Tool. The evaluators 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 a verbal interview and written questionnaire (included as Appendix D).

The evaluators rated the Solution Generator and Instructor Tool separately. Their evaluations of the Solution Generator focused on how well the software's interpretation of the problem description matched the instructor's intent, and the quality of the resulting class diagram. In addition, the problem descriptions they entered (along with those entered by two other informal testers) 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 evaluators were asked to list specific difficulties or bugs they found in the solution generation, and the 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 other times it did not. The most common result was that the information contained in the subordinate clause was dropped. The evaluators 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. The natural language parsing task is the most complex part of the processing, and is the area in which the most improvements can be made as better software becomes available.

Other bugs found, and how they were corrected, are listed below:

1. Pronoun resolution (one occurrence) – The simple strategy of using the subject of the prior sentence as a pronoun's antecedent worked exceptionally well except for one case in which the antecedent was taken from the subject of a subordinate clause: "The math drill program asks the number of problems the user wants to do. It asks what type of problem the user wants." In this case, the Solution Generator chose "user" rather than "math drill program" as the antecedent of "it" in the second sentence. This is a good example of the importance of accurately identifying subordinate clauses. Correct identification of "the user wants to do" as a subordinate clause corrected this bug and supports the soundness of the original strategy for pronoun resolution.
2. Relative pronoun resolution (two occurrences) – "That," "what" and "who" are examples of relative pronouns. They refer back to nouns mentioned earlier in the sentence: "The elevator has a door that can open or close." The problem with correctly interpreting this sentence is related to the issue of parsing subordinate clauses, but even when MontyLingua correctly parsed "that can open or close," "that" cannot be resolved using the same logic as subjective pronouns. In this example, the evaluator was easily able to circumvent the problem by rewording: "The elevator has a door. The door can be open or closed." However, a code solution to handle relative pronouns will allow the Solution Generator to properly interpret more natural-sounding English text.
3. Interpretation of specific words (three occurrences) – All three of these bugs fell into different categories. In the first 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 "alar" for "alarm" and returned the definition for "alar." A simple remedy is to download a larger version of the Wordnet database. Another remedy incorporated into the Solution Generator is a database extension that allows the user to add his own words along with their definition and related terms. This is useful to add terms that may be specific to an area of study or local jargon familiar to students within a problem's context.

In the 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 description of a basketball game: "The team to accumulate the most points wins." The 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 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 datatype for the element.

In the third case, "how many" in the sentence "It asks how many problems the user wants to do." was not interpreted as a quantity. A quick fix for the evaluator was to reword the sentence to "It asks the number of problems the user wants to do." The Solution Generator code was corrected to look for adjectives referring to quantity, and treat the phrase as referring to a count.

The bugs encountered represented a small portion of the text that was entered. 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 that the learning curve was not too cumbersome. Another suggested adding other commonly used methods, such as toString and equals. General comments included that the tool encouraged the instructor 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 instructor had initially developed.

The evaluators were also asked about the tool's user interface. Suggestions for improvements included graying out the "View Actors" button if no actors were found; displaying the number of actors on the button; and aligning the "View Details" buttons on the class diagram. One suggested adding the ability to undo delete of a component. None of them felt that the ability to add components was necessary. They rated both the tool's ease of use and its completeness of functionality as a 4 (also on a scale of 1 to 5).

2. Design Variations

Three problems were chosen to examine the variations identified by the Solution Generator. These are the Movie Ticket Machine (used as an example in the Methodology section and the problem for which student data was collected to assess the Expert Evaluator), the Automated Teller Machine (Example #2 in the Instructor Tool User Manual) and the Clock.

The Movie Ticket Machine problem definition can be found on page 112 (and is repeated here):

> "The movie ticket machine displays the movie title. It also displays the show time and it displays the price of a ticket. A customer enters money and the number of tickets into the machine. The machine prints the tickets. The ticket has the show time and movie title. The ticket machine returns the customer's change. The machine also tracks the number of seats in the theater so it can tell the customer when the tickets are sold out."

The unedited solution template is shown in Figure 5 on page 107. The most obvious variation is the presence of a Ticket class. Defining a class for a Ticket might be considered unnecessary, since the print ticket method would contain all the logic defining the look of a ticket. However, it is valid within the rules of object-oriented design, and an advanced student working on a more complex ticket machine would likely choose a Ticket class to contain the definition of a ticket's layout. So an instructor could accept the Ticket class as a reasonable variation, and mark it optional so that solutions with and without the class would be valid.

Of the 33 student solutions for the Ticket Machine problem, only one significant variation (entered by three students) was found that was not recognized as a correct solution by the Expert Evaluator. A method "seatsAvailable" (also called "openSeats") returning a boolean should have been accepted as correct. The Solution Generator could not predict this method because an explicit reference to its action was not contained in the problem description. Adding a sentence such as "The machine verifies enough seats are available" would have resulted in a "verify seats" method being added. This is a case where a reasonable variation was not foreseen by the

Solution Generator because of the inherent limitation that it cannot infer actions based on real world knowledge or implementation details not explicitly stated.

The Automated Teller Machine problem is:

> "An ATM dispenses money from an internal supply. It reads a customer's card number. It also reads the customer's password. It verifies that the card number and password are correct. The ATM contacts the bank's central database for the verification and customer balance.
>
> The ATM can display the customer's balance. It can deposit money into the customer's account. It also withdraws money from the account. The ATM prints a receipt showing the type of transaction and account number. The receipt also shows the new account balance."

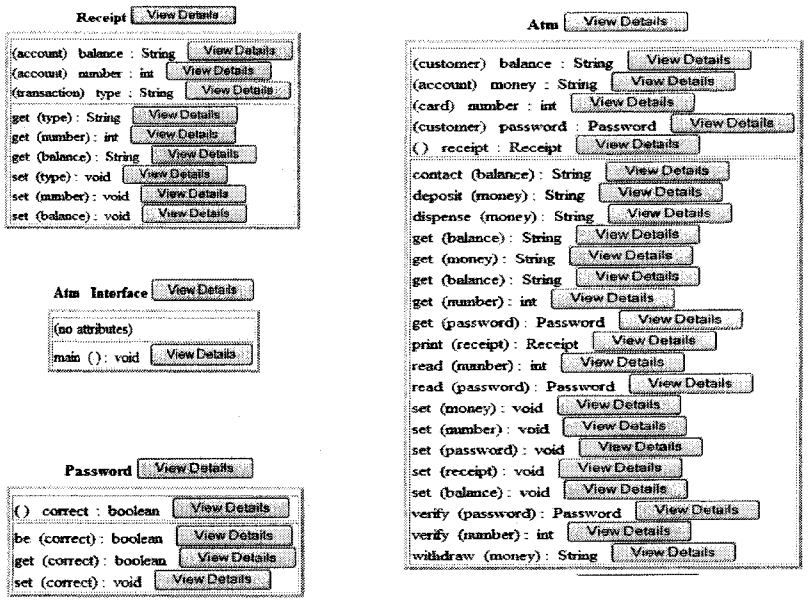


Figure 7: Class diagram generated for ATM problem

The generated class diagram is shown in Figure 7. In addition to the ATM class, a Receipt class and a Password class were also added. The Password class is likely not needed, since a password is typically a number or character string. The

Receipt class may be acceptable, for the same reason the Ticket class could be in the Movie Ticket Machine problem.

The third problem, the Clock, reads as follows:

> "A clock shows the current time. It shows hour and minutes. Time consists of hours and minutes. The clock advances the time."

This small segment of a problem description generates two variations (shown in Figure 8): one with hour and minute as attributes of clock, and another with an attribute time, which is represented by a Time class that includes hour and minute. The instructor now has the option of allowing both to be valid designs, or deleting one of

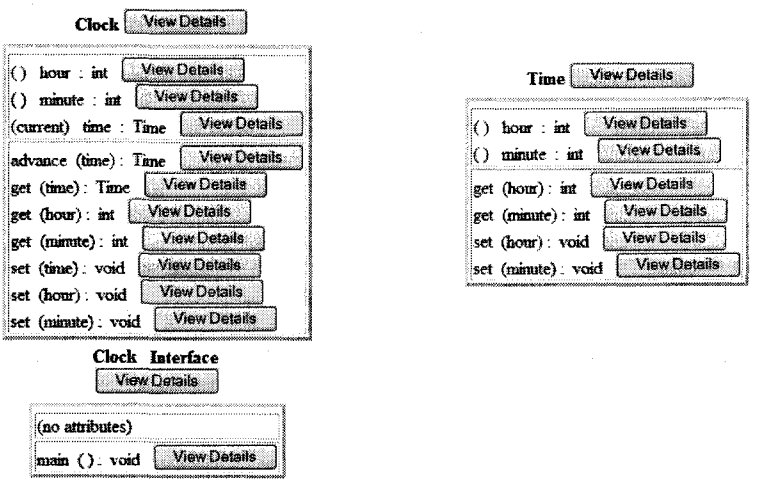


Figure 8: Class diagram generated for Clock problem

the variations. If he deletes hour and minute in the Clock class, the instructor can enter a comment such as "Is hour really an attribute of a clock? The clock shows time. Time is made up of hours and minutes." This explanation would be displayed to the student who entered hour as an attribute of Clock. Alternatively, the instructor could mark hour and minute as optional, with comments telling the student "Your design with hour as an attribute of Clock will work, but there is a better way to model the Clock's time."

Expert Evaluator

The Expert Evaluator was tested using data from 33 students in a first-semester Computer Science course (CSE-15) 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 (as described under "ITS Architecture" in the Methodology section on page 101). These "student action records" were saved for future use. At the time of the workshop, the Expert Evaluator used a solution template that was generated manually, but followed the same format as that created by the Solution Generator. Later, when the Solution Generator and Instructor Tool were complete, the student action records were processed using a solution template created by the Solution Generator and modified through the Instructor Tool. The info and error packets produced were then analyzed by manually comparing each packet to its corresponding student action, and determining its accuracy. The results are shown in Table 2.

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

Overall error rate = 136/1035 = 13% (87% accuracy), including unknowns
95/1035 = 9% (91% accuracy), not including unknowns
Error rate identifying correct actions = 16/540 = 3% (97% accuracy)
Error rate diagnosing incorrect actions = 79/454 = 17% (83% accuracy)
Unknown rate = 41/1035 = 4%

Table 2: Expert Evaluator Results

For any action, the EE provided an accurate diagnosis 87% of the time. 4% of the time it could not determine the student's intent ("unknown" in Table 2). 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"). Since the EE's accurate interpretation of the student's intent is necessary to provide a correct diagnosis, the EE's interpretation (see research question 2c on page 16) must have been correct at least 87% of the time.

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 have to use long names to distinguish between them. This trickled down to confusion with methods similar to "get" and "set" functions for those 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 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 PA can ask "Did you mean…?"

There were 3 distinct errors that were 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. The student had "seatsAvailable" and "seatsRemain" as attributes, with the second marked as a duplicate of the first. There is no need to keep the original value of the number of seats available, and an instructor might agree that there is no need for two attributes. Another 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."

The most significant design variation not accepted by the EE was the addition of a method "seatsAvailable," returning a boolean. This was discussed in the "Design Variations" section on page 133 as a case that illustrates a limitation of the Solution Generator.

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 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. That improves

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.

V. Conclusions

Learning object concepts and object-oriented design is difficult, yet this knowledge is an essential foundation for success in computer science. Instructors, especially high school teachers who specialize in other areas but are called upon to teach a programming course, can find gaining the necessary mastery of the subject challenging. Reaching students in high school is a must if we are to attract able students to the field.

The goal of this dissertation was to apply software engineering principles to develop tools and resources that assist both students and teachers in learning object-oriented design. Intelligent tutoring systems have been shown to be beneficial to students learning other complex concepts in fields such as algebra, physics and procedural programming. Software engineering principles have been used to automate parts of the design process and build tools to aid developers and software end-users in designing complete, well-structured systems. This research combined ideas from both fields to build a design tool for teachers and an intelligent tutoring system for students. It accomplished this goal through answering the research questions posed in Section I as described below.

1. *Can learning how to create use cases and class diagrams help novices learn object-oriented design and programming?*

This question was answered by developing the Design-First curriculum for a first-semester course using Java. The curriculum was refined and evaluated over three semesters at Dieruff High School. The progress of average students with no prior programming experience was measured. Through exams and project assignments, the

students demonstrated an understanding of object concepts, the ability to create well-structured object-oriented designs for small problems, and the ability to implement those designs in Java. The curriculum has been taught with favorable results by an independent instructor at a high school in Washington, and continues to be used at Dieruff High School.

2. *Can the automation of software engineering processes and principles be applied to solve simple problems of the type generally assigned to beginning computer science students?*

This question was answered through the development of the Instructor Tool, which analyzes problem descriptions input as plain text and outputs a class diagram representing a design template that includes multiple acceptable solution variations. When used by three evaluators and other informal testers, the tool generated design elements not initially identified but recognized as valid by the user. This addressed the first two specific sub-questions: *Can automation*

   a. *generate multiple solutions for a given problem?*

   b. *provide flexibility in identifying and evaluating novel but correct solutions created by students?*

Sub-question b was further answered by the Expert Evaluator implemented as part of Design-First ITS. The EE uses Wordnet definitions and synonyms to interpret the language used in the problem description and the words used by the student for the names of design components. Thus, the student is not locked into a heavily-scaffolded process. This promotes a more thoughtful and creative design process, rather than a mechanical one. Because the EE recognizes multiple options for a correct design, variations that are explicitly represented within the problem text are accepted.

The remaining sub-questions are also answered by the EE. The EE's primary function, to

c. interpret the steps a student applies in solving a problem,

was accomplished with an 88% accuracy rate in the student data analyzed in Section IV. Sub-questions d and e,

d. provide a diagnosis for student errors and

e. recommend next steps when the student is stuck?

are also handled by the EE. The EE determines the correctness of each student action. When an action is deemed incorrect, the EE may determine any one of the following diagnoses: 1. the component is valid, but some aspect of it is incorrect (data type or return type, for example); 2. the component matches a deleted component in the solution template; 3. the component matches a different component type in the solution template; or 4. the component is not recognized as part of any valid design. In case 1, the EE passes the recommended correction to the ITS's Pedagogical Advisor; in case 2, the EE sends the instructor's comments (recorded through the Instructor Tool) to the PA; in case 3, the EE sends a recommended correct action in the form of the type of component the student's element should be; in case 4, the EE chooses a likely next correct action (for example, if the student attempted to add an attribute, the EE chooses another attribute as a recommended alternate action). Case 4 is not same as the EE's inability to determine the meaning of the student's component name; that is communicated to the Pedagogical Advisor separately, so the PA can ask the student for clarification.

This dissertation's contributions overlap three fields: computer science education, intelligent tutoring systems and software engineering. The specific contributions are:

- A novel use of automated code generation techniques, which provides additional evidence supporting the value of these techniques and encouraging their continued development and use in new software engineering applications.

- A unique expert module for an intelligent tutoring system that overcomes the limitations of a constraint-based tutor through a rich analysis of the problem to be solved. The EE allows for creativity in a complex domain by recognizing multiple solutions, and it provides error diagnoses that can be used to recommend next steps and customize advice when a student is stuck.

- The Design-First curriculum: An innovative curriculum for teaching object-oriented design and programming to beginners, complete with lesson plans, student materials, and an IDE to support it, which has been evaluated in a high school setting. Its evaluation and use by an outside instructor also provide evidence supporting the effectiveness of using software engineering principles and UML to help beginners learn problem solving and object-oriented design.

- A tool to assist teachers in creating complete, clear problem descriptions, and in increasing their understanding of object-oriented design.

Future Work

Potential improvements to the Solution Generator and Expert Evaluator components of the expert module provide several avenues for future research. In the Solution Generator, there are three particular areas with potential for improvement.

The first area is in natural language processing. The ability to handle more complex language structures and to improve aspects of interpretation, such as pronoun resolution, would make the composition of problem descriptions much easier for the instructor, and the end result more readable for the student. Natural language

processing is a huge field, and although a natural language processor was used to implement the Solution Generator, specific research into any aspect of this field was beyond the scope of this dissertation.

The second area for improving the Solution Generator's results is the application of real-world knowledge to provide a more accurate interpretation of problem descriptions. The Solution Generator currently requires all aspects of a problem to be *explicitly* described in the text; components which could be derived indirectly from the text currently cannot be identified. An example is the "seatsAvailable" method added by several students working on the Ticket Machine problem (described in Section IV, Evaluation, under Design Variations on page 133). The sentence "The machine also tracks the number of seats in the theater so it can tell the customer when the tickets are sold out" implies that a method to check whether there are enough seats left for a customer's request is valid, but recognizing it as such requires an understanding of the scenario beyond the text. In this case, one must extrapolate the idea of checking for available seats from the required ability to "tell the customer when the tickets are sold out." The Solution Generator, working from problem text to solution, could use this knowledge to identify more variations up front, and the Expert Evaluator could use it to better understand the student's intent and evaluate whether the student's idea is a valid addition to the design. Encapsulating "common sense" knowledge that identifies different types of relationships between words and concepts, recognizes associations and makes inferences has been attempted by projects such as ConceptNet (Liu and Sing 2004a, b). Can ConceptNet or any other knowledge representation network provide the capability for the Expert Evaluator to

infer valid design components, or recognize inferred components as valid? Can novel designs not previously identified but verified as valid be "learned" by the EE to improve future performance?

The third area is exploring ways to enable the Solution Generator to "learn" from new correct solution variations submitted by students. Input from a human instructor's evaluation of student work could be used to identify additional designs and incorporate them into an existing solution template. Rather than rely on the experiences of one instructor's classes, the results of many students at different schools using the DesignFirst-ITS could be combined to build very robust solution templates. A library of problems and solutions could be made available to teachers, who could download them and revise their own copies to incorporate individual design preferences and styles. Techniques for using data gleaned from experience should be studied to determine how the Solution Generator can both improve the solution template for the current problem, and apply that knowledge to other problems.

Design patterns have proven very useful in speeding up the design process through reuse, and in improving the overall quality of systems by promoting the use of designs that have been proven superior in many applications. Could design patterns be useful in small, beginner-level problems? Can design patterns specific to this domain be identified? Could the Solution Generator recognize whether a specific pattern applies to a problem?

Inheritance is covered in many first-semester programming courses. Extending the scope of the Solution Generator and Expert Evaluator to include inheritance would

significantly expand the problem set and the curriculum within which the DesignFirst-ITS is useful.

In computer science education, questions of how to most effectively teach object concepts and design, and how to increase student interest in computer science are still wide open. The Design-First curriculum supported the use of UML and design methodologies to teach novice students in grades 11 and 12. Does the curriculum help students who go on in computer science do better than students whose first programming course is procedural-based or objects-first? Is the curriculum effective with younger students in grades 9 and 10? Could a design-based curriculum adapted for students in middle school be effective? These avenues of research could also inspire the development of much needed projects and activities that are both educational and fun.

Bibliography

ACM Curriculum Committee on Computer Science. (1968). Curriculum 68: Recommendations for the Undergraduate Program in Computer Science. *Communications of the ACM, March 1968,* 151-197.

ACM/IEEE-CS Computing Curricula 2001. (2001). Online at http://www.sigcse.org/cc2001/.

ACM/IEEE-CS Joint Curriculum Task Force. (1991). *Computing Curricula 1991.* New York, NY: ACM Press. Abridged versions reprinted in *Communications of the ACM , June 1991* and *IEEE Computer, November 1991.* Online at http://www.acm.org/education/curr91/homepage.html.

Abbott, Russell (1983). Program Design by Informal English Descriptions. *Communications of the ACM, November 1983,* 882-894.

Adams, J., & Frens, J. (2003). Object-Centered Design for Java: Teaching OOD in CS-1. *Proceedings of the 34th Technical Symposium on Computer Science Education (SigCSE 2003), Reno, NV, 2003,* 273-277.

Allen, E., Cartwright, R., & Stoler, B (2002). DrJava: A Lightweight Pedagogic Environment for Java. *Proceedings of the33rd Technical Symposium SIGCSE Conference on Computer Science Education (SigCSE 2002), Cincinnati, OH, March 2002,* 137-141.

Alphonce, C., & Ventura, P. (2002). Object-Orientation in CS1-CS2 by Design. *The Seventh Annual Conference on Innovation and Technology in Computer Science Education, Aarhus, Denmark, June, 2002,* 70-74.

Alphonce, C., & Ventura, P. (2003). Quick UML: A Tool to Support Iterative Design and Code Development. *Proceedings of OOPSLA '03, Anaheim, CA, October 2003,* 80-81.

Ambriola, V., & Gervasi, V. (1997). An Environment for Cooperative Construction of Natural-Language Requirements Bases. *Proceedings of the 8th International Conference on Software Engineering Environments, Los Alamitos, CA, April 1997,* 124-130.

Ambriola, V. & Gervasi, V. (1997). Processing Natural Language Requirements. *Proceedings of the 12th International Conference on Automated Software Engineering, Los Alamitos, CA, November 1997,* 36-45.

Anderson, John (1988). The Expert Module. In Martha Polson & J. Jeffrey Richardson (Eds.), *Foundations of Intelligent Tutoring Systems* (pp. 21-53). Hilldale, NJ: Erlbaum.

Anderson, John (1993). *Rules of the Mind.* Hilldale, NJ: Erlbaum.

Anderson, J. R., Corbett, A. T., Koedinger, K., & Pelletier, R. (1995). Cognitive tutors: Lessons learned. *The Journal of Learning Sciences, Vol. 4,* 167-207.

Anderson, J. R., Corbett, A. T., & Reiser, B. J. (1987). *Essential LISP.* Reading, MA: Addison-Wesley.

Austing, R., Barnes, B., Bonnette, D., Engel, G., & Stokes, G. (1979). Curriculum '78: Recommendations for the Undergraduate Program in Computer Science – A Report of the ACM Curriculum Committee on Computer Science. *Communications of the ACM, Vol. 22, Issue 3 (March 1979),* 147-166.

Baghaei, N., & Mitrovic, A. (2005). COLLECT-UML: Supporting Individual and Collaborative Learning of UML Class Diagrams in a Constraint-based Intelligent System. *Proceedings of the 9th International Conference on Knowledge-Based & Intelligent Information & Engineering Systems, Melbourne, Australia.*

Baghaei, N., & Mitrovic, A. (2007). From Modeling Domain Knowledge to Metacognitive Skills: Extending a Constraint-Based Tutoring System to Support Collaboration. *User Modeling 2007,* 217-227.

Baghaei, N., Mitrovic, A., & Irwin, W. (2006). Problem-Solving Support in a Constraint-based Intelligent System for Unified Modeling Language. *Technology, Instruction, Cognition and Learning Journal, Vol 4, No 1-2.*

Barnes, David & Kölling, Michael (2003). *Objects First with Java.* Essex, Great Britain: Pearson Education (Prentice Hall).

Barrett, M.(1996). Emphasizing Design in CS1. *Proceedings of the 27th Technical Symposium on Computer Science Education (SigCSE 1996), Philadelphia, PA,* 315-318.

Beck, Kent (2000). *Extreme Programming Explained.* Reading, MA: Addison-Wesley.

Bellin, D. (1992). A Seminar Course in Object-Oriented Programming. *Proceeding of the 23rd Technical Symposium on Computer Science Education (SigCSE 1992), Kansas City, MO,* 134-137.

Bergin, J. (1990). The Object-Oriented Data Structures Course. *Proceedings of SOOPPA 1990*, 100-111.

Bergin, J., Stehlik, M., Roberts, J., & Pattis, R. (2005). *Karel J. Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java.* Foster City, CA: Café Press.

Biddle, R. & Tempero, E. (1994). *Teaching C++ Experience at Victoria University of Wellington.* New Zealand: University of Wellington. (Technical Report CS-TR-94/18)

Blank, G. D., Moritz, S. H. & DeMarco, D. W. (2005). Objects or Design First? *Nineteeth European Conference on Object-Oriented Programming (ECOOP 2005), Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts, Glasgow, Scotland, July 2005.*

Blank, G. D., Parvez, S., Wei, F., & Moritz, S. (2005). A Web-Based ITS for OO Design. *12th International Conference on Artificial Intelligence in Education, Workshop on Adaptive Systems for Web-Based Education: Tools and Reusability, Amsterdam, The Netherlands, June 2005,* 59-64.

Blank, G. D., Pottenger, W. M., Sahasrabudhe, S. A., Li, S., Wei, F., & Odi, H. (2003). Multimedia for computer science: from CS0 to grades 7-12, *EdMedia, Honolulu, HI, June 2003.*

Bloom, B. S. (1984). The 2 Sigma Problem: The Search for Methods of Group Instruction as Effective as One-to-one Tutoring. *Educational Researcher, 13(6),* 4-16.

Booch, G. (1986). Object-Oriented Development. *IEEE Trans. on Software Engineering, 12(2),* 211-221.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications (second edition).* Menlow Park, CA: Addison-Wesley.

Booch, G., Jacobson, I., & Rumbaugh, J. (1997). *The Unified Modeling Language.* Rational Software.

Brill, Eric (1994). A Report of Recent Progress in Transformation-Based Error-Driven Learning. *Proceedings of the Twelfth National Conference on Artificial Intelligence,* 722-727.

Brooks, Fred (1975; republished 1995). *The Mythical Man-Month.* New York, NY: Addison-Wesley.

Bruce, Kim (2004). Controversy on How to Teach CS1: A Discussion on the SIGCSE-members Mailing List. *Inroads – The SIGCSE Bulletin, December, 2004*, 29-34.

Bruce, K, Danyluk, A., & Murtagh, T. (2001). A Library to Support a Graphics-Based, Object-First Approach to CS 1. *Proceedings of the 32nd Technical Symposium on Computer Science Education (SigCSE 2001), Charlotte, NC*, 6-10.

Buck, D. & Stucki, D. (2000). Design Early Considered Harmful: Graduated Exposure to Complexity and Structure based on Levels on Cognitive Development. *Proceedings of the 31st Technical Symposium on Computer Science Education (SigCSE 2000), Austin, TX*, 75-79.

Cooper, S., Dann, W., & Pausch, R. (2003). Teaching Objects-First in Introductory Computer Science. *Proceedings the 34th Technical Symposium on Computer Science Education (SigCSE 2003), Reno, NV*, 191–195.

Corbett, A., & Anderson, J. (1999). Intelligent Computer Tutors: Out of the Research Lab and into the Classroom. Paper presented at the Annual Meeting of the American Educational Research Association, Montreal, Canada, April 19-23, 1999.

Corbett, A., Koedinger, K., & Anderson, J. (1992). LISP Intelligent Tutoring System: Research in Skill Acquisition. In J.H. Larkin & R.W. Chabay (Eds.), *Computer-assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches* (pp. 73-109). Hillsdale, NJ: Erlbaum.

Decker, A. (2003). A Tale of Two Paradigms. *Journal of Computing Sciences in Colleges, 19(2)*, 238-246.

Decker, R. & Hirshfield, S. (1994). The Top Ten Reasons Why Object-Oriented Programming Can't be Taught in CS 1. *Proceeding of the 25th Technical Symposium on Computer Science Education (SigCSE 1994), Phoenix, AR,* 51-55.

Epstein, R. & Tucker, A. (1992). Introducing Object-Orientedness into a Breadth-First Introductory Curriculum. *Proceedings of OOPSLA 1992*, 293-298.

Fellbaum, Christiane (1998). *WordNet: An Electronic Lexical Database*. Cambridge, MA: MIT Press.

Gertner, Abigail & VanLehn, Kurt (2000). Andes: A Coached Problem Solving Environment for Physics. *Proceedings Fifth International Conference, ITS 2000, Montreal, Canada,* 133-142.

Halstead, M. (1977). *Elements of Software Science.* New York, NY: Elsevier North-Holland, Inc.

Koedinger, Kenneth (2001). Cognitive Tutors as Modeling Tools and Instructional Models. In Forbus, Kenneth & Feltovich, Paul (Eds.), *Smart Machines in Education* (pp. 145-167). Cambridge, MA : AAAI Press/MIT Press.

Koedinger, Kenneth & Anderson, John (1993). Reifying Implicit Planning in Geometry: Guidelines for Model-Based Intelligent Tutoring System Design. In Lajoie, Susanne & Derry, Sharon (Eds.), *Computers as Cognitive Tools* (pp. 15-45). Hilldale, NJ: Lawrence Erlbaum.

Koedinger, Kenneth, Anderson, John, Hadley, William, & Mark, M.A. (1997). Intelligent Tutoring Goes to School in the Big City. *International Journal of Artificial Intelligence in Education, 8(1),* 30-43.

Kölling, M.., Koch, B., & Rosenberg, J. (1995). Requirements for a First Year Object-Oriented Teaching Language. *Proceeding of the 26th Technical Symposium on Computer Science Education (SigCSE 1995), Nashville, TN*, 173-177.

Kölling, M. & Rosenberg, J. (1996). Blue – A Language for Teaching Object-Oriented Programming. *Proceedings of the 27th Technical Symposium on Computer Science Education (SigCSE 1996), Philadelphia, PA*, 190-194.

Kölling, M. & Rosenberg, J. (1996). An Object Oriented Program Development Environment for the First Programming Course. *Proceedings of the 27th Technical Symposium on Computer Science Education (SigCSE 1996), Philadelphia, PA*, 83-87.

Kölling, M. & Rosenberg, J. (1997). Blue Language Specification. Victoria, Australia: Monash University. (Technical Report TR97/13)

Kölling, M. & Rosenberg, J. (2001). Guidelines for Teaching Object Orientation with Java. *Proceedings of the Sixth Annual Conference on Innovation and Technology in Computer Science Education*, 33-36.

Kostadinov, R. & Kumar, A.N. (2003). A Tutor for Learning Encapsulation in C++ Classes. *Proceedings of ED-MEDIA 2003 World Conference on Educational Multimedia, Hypermedia and Telecommunications, Honolulu, HI*, 1311-1314.

Kumar, A.N. (2002). Model-Based Reasoning for Domain Modeling in a Web-Based Intelligent Tutoring System to Help Students Learn to Debug C++ Programs. *Intelligent Tutoring Systems (ITS 2002), Biarritz, France*, 183-199.

Kumar, A.N. (2002). A Tutor for Using Dynamic Memory in C++. *Proceedings of 2002 Frontiers in Education Conference (FIE 2002), Boston, MA*, T4G 32-36.

Kumar, A.N. (2005). Results from the Evaluation of the Effectiveness of an Online Tutor on Expression Evaluation. *ACM SIGCSE Bulletin, Vol. 37, Issue 1*, 216-220.

Kumar, A.N. (2005). Rule-Based Adaptive Problem Generation in Programming Tutors and its Evaluation. *Workshop on Adaptive Systems for Web-Based Education: Tools and Reusability, 12th International Conference on Artificial Intelligence in Education (AI-ED 2005), Amsterdam, The Netherlands*, 35-44.

Lieberman, Henry (Ed.) (2001). *Your Wish is My Command: Programming by Example*. San Francisco, CA: Morgan Kaufmann.

Liu, Hugo (2003). MontyLingua: Commonsense-Informed, Natural Language Understanding Tools. Available at: http://web.media.mit.edu/~hugo/montylingua/

Liu, Hugo & Lieberman, Henry (2005). Metafor: Visualizing Stories as Code. *Proceedings of the ACM International Conference on Intelligent User Interfaces, IUI 2005, San Diego, CA,* 305-307.

Liu, Hugo & Singh, Push (2004). Commonsense Reasoning in and over Natural Language. *Proceedings of the 8th International Conference on Knowledge-Based Intelligent Information & Engineering Systems (KES'2004,. Wellington, New Zealand,* 293-306.

Liu, Hugo & Singh, Push (2004). ConceptNet: A Practical Commonsense Reasoning Toolkit. *BT Technology Journal, Volume 22*, 211-226.

Mazaitis, D. (1993). The Object-Oriented Paradigm in the Undergraduate Curriculum: A Survey of Implementations and Issues. *ACM SIGCSE Bulletin (inroads), September 1993*, 58-64.

Marcus, Mitchell, Santorini, Beatrice & Marcinkiewicz, Mary Ann (1993). Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics (Special Issue on Using Large Corpora), Volume 19, Number 2,* 313-330.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagen, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A Multi-National Study of Assessment of Programming Skills of First Year CS Students. *SIGCSE Bulletin 33(4)*,125-140.

Miller, G., Newman, E., & Friedman, E. (1958). Length Frequency Statistics of Written English. *Information and Control, Vol. 1*, 370-389.

Mitrovic, A., Martin, B., & Suraweera, P. (2007). Intelligent Tutors for All: The Constraint-Based Approach. *Intelligent Systems: IEEE, Vol. 22, No. 4,* 38-45.

Mitrovic, A., Martin, B., Suraweera, P., Zakharov, K., Milik, N., & Holland, J. (2005). ASPIRE: Student Modeling and Domain Specification. New Zealand: University of Canterbury, Intelligent Computer Tutoring Group, Dept. of Computer Science and Software Engineering. (Technical Report TR-08-05)

Mitrovic, A., Mayo, M., Suraweera, P. & Martin, B. (2001). Constraint-based Tutors: A Success Story. *14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-2001), Budapest,* 931-940.

Moritz, S. & Blank, G. (2005). A Design-First Curriculum for Teaching Java in a CS1 Course. *ACM SIGCSE Bulletin (inroads), June 2005,* 89-93.

Moritz, S., Wei, F., Parvez, S., & Blank, G. (2005). From Objects-First to Design-First with Multimedia and Intelligent Tutoring. *The Tenth Annual Conference on Innovation and Technology in Computer Science Education, Monte da Caparica, Portugal,* 99-103.

Nørmark, Kurt (1995). *An Evaluation of Eiffel as the First Object-Oriented Programming Language in the CS Curriculum.* Denmark: Aalborg University.

Northrop, L. (1992). Finding an Educational Perspective for Object-Oriented Development. *Proceedings of OOPSLA 1992,* 245-249.

Ohlsson, S. (1994). Constraint-based Student Modeling. In Greer, J.E. & McCalla, G. (Eds.), *Student Modeling: The Key to Individualized Knowledge-based Instruction* (pp. 167-189). Heidelberg, Germany: Springer-Verlag.

Ohlsson, S. (1996). Learning from Performance Errors. *Psychological Review, 103(2),* 241-262.

Overmyer S.P., Lavoie B., & Rambow O. (2001). Conceptual Modeling through Linguistic Analysis Using LIDA. *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001),* 401-410.

Parker, D. (1995). Structured Design for CS1. *Proceedings of the 26th Technical Symposium on Computer Science Education (SigCSE 1995), Nashville, TN,* 258-262.

Parvez, S. (2008). A Pedagogical Framework for Integrating Individual Learning Style into an Intelligent Tutoring System, PhD dissertation. Bethlehem, PA: Lehigh University.

Popyack, J., Shokoufandeh, A., & Zoski, P. (2000). Software Design and Implementation in the Introductory CS Course: JavaScript and Virtual Pests. *Proceedings of the Fifth Annual Consortium for Computing Sciences in Colleges Northeastern Conference, Mahway, NJ*, 166-177.

Pugh, J., LaLonde, W., & Thomas, D. (1987). Introducing Object-Oriented Programming into the Computer Science Curriculum. *Proceedings of the 18th Technical Symposium on Computer Science Education (SigCSE 1987), St. Louis, MO*, 98-102.

Rasala, R., Raab, J., & Proulx, V. (2001). Java Power Tools: Model Software for Teaching Object-Oriented Design. *Proceedings of the 32nd Technical Symposium on Computer Science Education (SigCSE 2001), Charlotte, NC*, 297-301.

Ratcliffe, M. & Thomas, L. (2002). Improving the Teaching of Introductory Programming by Assisting Strugglers. A Birds of a Feather Session. *Proceedings of the 33rd ACM SIGCSE Technical Symposium on Computer Science Education, Cincinnati, OH.*

Reges S. (2000). Conservatively Radical Java in CS 1. *Proceedings of the 31st Technical Symposium on Computer Science Education (SigCSE 2000), Austin, TX*, 85-89.

Reid, R. (1991). Object-Oriented Programming in C++. *SIGCSE Bulletin 23(2)*, 9-15.

Reid, R. (1993). The object oriented paradigm in CS1. *Proceedings of the 24th Technical Symposium on Computer Science Education (SigCSE 1993), Indianapolis, IN*, 265-269.

Reis, C., & Cartwright, R. (2004). Taming a Professional IDE for the Classroom. *Proceedings of the 35th Technical Symposium on Computer Science Education (SigCSE 2004), Norfolk, VA,* 156-160.

Reiser, Brian, Kimberg, Daniel, Lovett, Marcia & Ranney, Michael (1992). Knowledge Representation and Explanation in GIL, An Intelligent Tutor for Programming. In J.H. Larkin & R.W. Chabay (Eds.), *Computer-assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches* (pp. 111 – 149). Hillsdale, NJ: Erlbaum.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-Oriented Modeling and Design.* New Jersey: Prentice Hall.

Santorini, Beatrice (1990). *Part-of-Speech Tagging Guidelines for the Penn Treebank Project*. Available at: http://ftp.cis.upenn.edu/pub/treebank/doc/tagguide.ps.gz

Shute, V. & Psotka J. (1996). Intelligent Tutoring Systems: Past, Present, and Future. In D. Jonassen (Ed.), *Handbook of Research on Educational Communications and Technology* (pp. 570 – 600). New York: Simon & Schuster Macmillan.

Skublicks, S. & White, P. (1991). Teaching Smalltalk as a First Programming Language. *SigCSE Bulletin, Vol. 23, No. 1*, 231-235.

Soloway, E., Rubin, E., Woolf, B., Bonar, J., & Johnson, W. (1983). MENO-II: An AI-based Programming Tutor. *Journal of Computer-based Instruction, 10(1),* 20-34.

Soloway, E., Woolf, B., Rubin, E. & Barth P. (1981). Meno-II: An Intelligent Tutoring System for Novice Programmers. *Proceedings of International Joint Conference in Artificial Intelligence, Vancouver, British Columbia.*

Stroustrup, B. (1994). *The C++ Programming Language (2nd Edition)*. Reading, MA: Addison Wesley.

Sykes, E. R. & Franek, F. (2004). Inside the Java Intelligent Tutoring System Prototype: Parsing Student Code Submissions with Intent Recognition. *Proceeding of the IASTED International Conference on Web-Based Education, Innsbruck, Austria,* 613-618.

Sykes, E. R. & Franek, F. (2004). A Prototype for an Intelligent Tutoring System for Students Learning to Program in Java. *Advanced Technology for Learning, Vol. 1, No. 1*, 35-43.

Tabrizi, M., Collins, C., Ozan, E., & Li, K. (2004). Implementation of Object-Orientation using UML in Entry-Level Software Development Courses. *Proceedings of the 5th Conference on Information Technology Education, Salt Lake City, UT*, 128-131.

Temte, M. (1991). Let's Begin Introducing the Object-Oriented Paradigm. *SigCSE Bulletin, 1991, Vol. 23, No. 1,* 73-78.

VanLehn, Kurt, Lynch, Collin, Schulze, Kay, Shapiro, Joel, Shelby, Robert, Taylor, Lynwood, Treacy, Don, Weinstein, Anders, & Wintersgill, Mary (2005). The Andes Physics Tutoring System: Lessons Learned. *International Journal of Artificial Intelligence in Education, 15(3),* 1-47.

Weber, Gerhard & Brusilovsky, Peter (2001). ELM-ART: An Adaptive Versatile System for Web-based Instruction. *International Journal of Artificial Intelligence in Education, Vol 12*, 351-384.

Weber, Gerhard & Schult, Thomas (1998). CBR for Tutoring and Help Systems. In *Case-Based Reasoning Technology: From Foundations to Applications (Lecture Notes in Computer Science Vol. 1400)* (pp. 255-272). Heidelberg: Springer Verlag.

Wei, F. (2007). A Student Model for an Intelligent Tutoring System Helping Novices Learn Object-Oriented Design, PhD dissertation. Bethlehem, PA: Lehigh University.

Williams, L., Wiebe, E., Yang, K., Ferzli, M., & Miller, C. (2002). In Support of Pair Programming in the Introductory Computer Science Course. *Computer Science Education, Vol. 12, No. 3*, 197-212.

Appendix A
A Design-First Java Curriculum for a One-Semester Course
Version 2; Uses Java 5.0

Unit One

Lesson 1a: Introduction to Software Engineering

Analogies:

1. Building a house (What planning/design is done before building begins? What pieces are put together? How many people are involved and what parts does each work on?)

2. Building a car (What pre-made "off the shelf" parts are put into it? How many people are involved in the process? What planning/design is done up front?)

Discuss how these processes are similar to building a software system. The developer must understand the requirements (exactly what the program is to do), then must design the program (decide how it will be structured), and last, translate the design into code.

Talk about the different types of software development jobs: systems analysts, who gather and document user requirements, and design how a system should be built, and programmers who write code. Talk about different levels of programming: application programming, writing the code for high level uses like businesses (much of which is being outsourced, or even automated); systems programming, writing lower-level code for operating systems, controlling hardware of all types (less likely to be outsourced)

Instructor Notes: Lesson2.doc
Handout: Lesson2Handout.doc

Lesson 1b: Object Orientation

Everything in the real world is an object (a house, a car, a person)

How are individual objects identified?

- Objects may have different characteristics (a car and a house have different components, and "do" different things)
- Objects that share the same characteristics have different values for those characteristics (everyone in this class is a person, but each person has a different name)

- Objects that share the same characteristics belong to the same category, or class. Individual objects (each person in the room, for example) are also called instances of the class.

Use the Person class as an example, and have the students identify the information about a person (attributes) and the behaviors of a person (methods).

Another example would also be helpful here: perhaps a Car class, where attributes include the Parts of a car (engine, body, wheels, tires), which are also objects.

Lesson 1c: Looking at an example in Java

What is a compiler?

Introduce the IDE (Eclipse/DrJava/UML)

Student Exploration #1: Students follow handout to open the Shapes project, then write and execute statements to create shapes and manipulate them using the classes provided (through the Interactions Pane).

Handout: ObjectsEclipse-1.doc

Student Exploration #2: Following handout, students learn to create objects and call methods within a method inside a Java class. Students modify an existing method that uses the shapes classes to draw a picture.

Handout: ObjectsEclipse-2.doc

Student Exploration #3: An extra, advanced activity for students who finish the first two exercises early.

Handout: ObjectsEclipse-3.doc

Unit 1 Quiz Review

Handout: ObjectsQuizReview.doc

Unit Two - The Development Process: Gathering requirements, designing a class, implementing it and testing

Lesson 2a: Gathering requirements

Use the Automated Teller Machine example (problem description is included in UseCasesHandout.doc).

As a class, come up with the requirements of a Automated Teller Machine (What does it do? Most students have used an ATM card, and are familiar with entering a Personal Identification Number, and withdrawing or depositing money and checking their balance.)

Introduce Actors and Use Cases:
Identify and list all actors.
Write use cases for the Automated Teller Machine (together).

Instructor Notes: UseCaseInstructorNotes.doc

Handout: UseCasesHandout.doc

Lesson 2b: Designing the Automated Teller Machine Class

Introduce simple UML class diagrams

Draw a class diagram for the ATMMachine class (Use the Eclipse UML plug-in. It automatically generates the code stubs, which students can modify later.)

Create a list of tests for the ATMMachine class.

Handouts: ClassDesignLesson.doc and CreateProject.doc

Lesson 2c: Variables, assignment and arithmetic

Handouts: VariablesAndAssignment.doc, VariablesExercise.doc and AssignmentExercise.doc

Lesson 2d: Writing the ATMMachine methods

The return statement

Handouts: returnStatement.doc

Write the code for the constructor, checkBalance, and deposit methods. Introduce the use of comments (Use Javadoc format. When code is complete, generate the Javadoc for the Automated Teller Machine.)

Variables and Assignment Quiz Review

Handouts: VariablesAssignmentReview.doc, VariablesAssignmentReview2.doc

Lesson 2e: Writing the verifyPin method

Communicating with the Bank Database
AcctDB: a class to access the bank "database".

Handout: DBAccessorClass.doc
Code and test data files: AcctDB.java, Account.java, accounts.dat

Lesson 2f: Testing the ATMMachine class so far

Emphasize when tests are developed! (They can be written before creating the code for the class!)

Write a test plan for the ATMMachine class. Execute the test code in the Interactions Pane in Eclipse.

Unit Three

Lesson 3a: Character-based Printing and if statements

System.out.print, println
if-else statements
{} blocks
Relational operators < > <= >= == !=
Nested if statements

Handouts: Printing.doc, PrintExercises.doc, IfStatements.doc, IfStatementExercises.doc

Lesson 3b: Completing the ATMMachine methods

Write printReceipt method
Write withdraw method, including alternatives (not enough money in account, not enough money in machine).

Lesson 3c: Completing and testing methods for ATMMachine

Review all methods. Students write and execute final test plan, using Interactions Pane.

Unit Four

Lesson 4a: Creating a Character-based User Interface

Designing the look of the character-based interface.

Creating a ATMInterface class and main to display a customer menu.

A class for reading character input: Scanner

Coding and testing the main method to handle just one operation for one customer.

Handout: AddingUIJava15.doc

Printing, If and Scanner Quiz Review

Handout: QuizifPrintingReadingReview.doc

Lesson 4b: Loops

do-while loop

Handout: loops.doc

Students write the code to keep displaying the customer menu until 4 is entered to exit. Then, after testing the first loop, they add a second loop to allow multiple customers to use the ATM.

Lesson 4c: Scope of variables
Use example from ATMInterface. Students had been declaring variables where they are used. Discuss pros and cons of declaring variables at the top of a method, outside a block, etc.

Handouts: scope.doc scopeExercises.doc

Loops and Scope Quiz Review

Handout: loopExercises.doc

THIS MARKS THE END OF THE AUTOMATED TELLER MACHINE PROJECT. STUDENTS NOW COMPLETE ALL CODING AND TESTING OF THE FINAL VERSION BEFORE MOVING ON TO THE GROUP PROJECT.

Unit Five: Tying It All Together

Group project: An arithmetic drill program for elementary students.

Handout: MathPracticeProgrammingProject.doc

Deliverable #1: Use cases.
Deliverable #2: Class diagram (including attributes and method definitions) in Eclipse.
Deliverable #3: All methods (except main) coded and unit tested (testing demonstrated to instructor).
Deliverable #4: Character-based Interface.

Unit Six: Graphical User Interface

Lesson 6a: Creating a frame with text objects

JFrame and Container classes
JTextField and commonly used methods
JLabel
JButton
JRadioButton and ButtonGroup

Group Project Activity: Students redesign the Math Drill interface to use text fields and radio buttons. Students create new MathDrillGUI class to display it.

Handout: GUI1.doc

Lesson 6b: Tying action to text fields and buttons

ActionListener interface, building your own Listener class
actionPerformed method
Connecting Listener class to a field
Converting String data to int or double

Exercise: Implement a listener for a simple program that adds two numbers.

Handouts: GUI2.doc, DataConversion2.doc, GuiSum.java

Group Project Activity: Students create a MathListener class, including attributes and constructor method. Students tie MathListener to the text fields in the MathDrillGui class.

Lesson 6c: Tying a listener to a button

Exercise: Expand GUISum example program to add and subtract.

Handout: GUI3.doc

Group Project Activity: Tie MathListener to radio buttons in MathDrillGUI.

Lesson 6d: Simple Dialog Boxes using JOptionPane

Displaying message boxes and simple dialog boxes for input of one field.

Handout: GUI4.doc

Group Project: Students complete their graphical MathDrill program by planning and coding the actionPerformed method.

Deliverable #5: Math Drill application with a graphical interface.

Lesson 6e: Converting an application to an applet

Demonstrate GUISum sample program as an applet.

Handouts: applet1.doc, applet2.doc

Sample html that invokes an applet: MathDrill.htm

Deliverable #6: Math Drill graphical program run as an applet.

Unit Seven: Sequential Files

Lesson 7a: Reading sequential files

Accessing disk files and reading using Scanner

Handouts: readingFiles1.doc, readingFiles2.doc

An introduction to the Sun Java documentation

Handout: sunJavaDoc.doc

Lesson 7b: Writing sequential files

Handout: writingFiles.doc

Lesson 7c: Applying use of files to add functionality to the Math Drill program

Group Project: Add code to save the last average scores on addition and subtraction problems in a file, so that the program can praise students who've improved their scores or encourage students who did not do as well as before.

Create a new class that handles all management of the prior scores.

Handout: MathDrillFiles.doc

Lesson 7d: Creating a jar file

Handout: MathDrillJar.doc

Unit Eight: Arrays and Searching

Lesson 8a: Arrays and For Loops

Creating arrays and reading in data. Sequential search.

Handout: Arrays1.doc

Data file for program assignment: students.dat

Group Project: Create a graphical program that allows searching student data by any of four fields.

Handouts: StudentListGUI.doc, StudentSearchDesign.doc

Lesson 8b: Binary Search

Handout: BinarySearch.doc

Arrays and Searching Quiz Review

Finding the largest or smallest value in an array

Handout: ArraysReview.doc

Author: Sally Moritz
Last Modified: June 1, 2006.

Appendix B
Sample Exam Questions

Object-Oriented Questions:

Here is the definition for a class that represents a Student. In the code, circle and label an example of each one of the following:

1. variable	2. datatype	3. method	4. parameter
5. constructor	6. comment	7. assignment statement	

```
public class Student
{
   private String firstName;
   private String lastName;
   private int gradYear;
   private double gradePointAvg;
  /**
  * Create a new Student with a first name and last name
  */
   public Student(String fname, String lname)
   {
         firstName = fname;
         lastName = lname;
   }
  /**
   *  Set graduation year
   */
   public void setGradYear(int year)
   {
         gradYear = year;
   }
    /**
     *      Set grade point average
     */
   public void setgradePointAvg(double gpa)
   {
         gradePointAvg = gpa;
   }
   /**
    * Return the student's current class
    */
   public String studentClass(int currentYear)
   {
         if (gradYear == currentYear)
```

```
            return "Senior";
        else
            if (gradYear == currentYear + 1)
                return "Junior";
            else
                if (gradYear == currentYear + 2)
                        return "Sophomore";
                else
                        return "Freshman";
    }
}
```

8. List the attributes of Student.

9. Write a statement that creates an instance of Student with the name "Tamara Roberts"

10. Write a statement that sets Tamara's graduation year to 2006.

11. Write a statement that prints Tamara's class.

Procedural Questions:

12. What is the scope of the variable "firstName"?

13. What is the scope of the variable "year" (defined in setGradYear)?

14. Suppose we want to add a new method called getGrade. getGrade has no parameters, and returns a char. If the student's gradePointAvg is greater than or equal to 3.5, getGrade should return 'A'. If gradePointAvg is greater than or equal to 2.5 (but less than 3.5), return 'B '. 1.5 to 2.5 should return a 'C'. 0.5 to 1.5 is 'D' and below 0.5 is 'F'. Write the section of code to return the correct letter grade.

15. Write a segment of code to read in a series of test scores and print the average. The scores should all be non-negative whole numbers. If a score is less than zero, stop reading scores and calculate and print the average. You may use either a while loop or a do while (whichever you prefer). Include all variable declarations. HINT: Your logic should be:
    - Read in a score
    - If the score is greater than or equal to zero, add it to a running total of all scores, and count it. Read in the next score.
    - ELSE, if the score is less than zero, don't read in any more scores. Calculate the average of the scores you read in, and print the result.

The average is calculated by dividing the sum of all the scores by the number of scores.

Appendix C
DesignFirst-ITS Instructor Tool
User Manual

The Instructor Tool allows instructors to enter problems into DesignFirst-ITS through a process that creates a solution for a textual problem description and allows for revision and annotation by the instructor. The end result is a solution description that is used by the Expert Evaluator module of the DesignFirst-ITS to evaluate student work and to provide input to the Student Model and the Pedagogical Agent. The Instructor Tool provides the benefit of automatically-generated solution suggestions that follow basic object-oriented design rules used by students, while also allowing for instructor variation and interpretation. Additionally, it is a learning tool for the instructor. Its generated solutions provide an evaluation of the clarity and completeness of the problem description, and offer insight into possible solutions which may not have been apparent to the instructor. Indeed, the latter has been my own experience with the tool; some of the solutions it generated initially surprised me and inspired new ways of thinking about design.

This document explains how to use the Instructor Tool by presenting guidelines on writing an appropriate problem description, then by stepping through two examples.

Problem Description Guidelines

DesignFirst-ITS is intended to teach beginners how to identify basic design elements. It accurately generates solutions for problem descriptions that are appropriately structured for human students. Thus, the same requirements an instructor

should follow in creating descriptions intended solely for human students apply to the Instructor Tool.

First, a good problem description for beginners must provide all the details necessary for a complete object-oriented design consisting of classes and their attributes and methods. An explicit listing of all requirements clarifies the instructor's expectations and acknowledges that all students cannot be assumed to possess knowledge about any given problem domain.

Second, a good description does not provide details beyond those required. While developers in the real world must contend with identifying what data is pertinent to a problem, a student learns this skill through experience. Too much information early in a student's learning process can cause frustration and detract from the basic learning goals.

Third, problem details relating to program implementation (coding) should be excluded. The Design-First curriculum emphasizes solution design *before* coding, and DesignFirst-ITS teaches object-oriented design. The focus is on high-level functionality rather than step-by-step details. Distinguishing between tasks that require a method versus a single line of code is a necessary skill that requires knowledge of procedural programming, and is improved primarily through experience. The Instructor Tool will generate superfluous methods if coding details are part of the problem description. The instructor can either revise the description and generate a new solution, or annotate them using the Instructor Tool, which would provide appropriate feedback (and a learning opportunity) to a student who makes the same error.

Finally, use simple declarative sentences, with basic subject/verb/object structure and active voice. Clarity rather than literary style is the goal. Long sentences with multiple clauses, use of passive voice and other complex grammatical forms can be confusing to both human students and to the Instructor Tool.

MontyLingua is the natural language processor used by the solution generation software within the Instructor Tool. It tags parts of speech, provides semantic interpretation of names, dates and other proper nouns, and parses sentences into (verb, subject, object) tuples. It works best with simple sentence structure of the form subject, verb, object. It does, however, have the following limitations:

1. Appositives are not recognized: "An automated teller machine, or ATM, dispenses money." Do not include "or ATM." In many cases, specifying synonyms is not necessary because they are added automatically by the Instructor Tool. If a desired synonym is not included, it can be added manually.
2. Compound sentences are only partially processed: "The ticket machine prints tickets and returns the customer's change." This should be broken into two sentences.
3. Dependent clauses are not recognized: "If the tickets are not sold out, the machine prints tickets for the customer." This should be split into two sentences. Another example is "The machine counts the number of tickets sold." "...tickets sold" implies "tickets that are sold," a dependent clause that describes tickets. This also should be reworded to avoid more than one verb in the sentence.

In summary, follow these grammatical rules in writing problem descriptions:

1. Use declarative sentences with simple subject/verb/object structure.
2. Use the active voice ("The ATM dispenses money.") rather than the passive voice ("Money is dispensed by the ATM.").
3. Add synonyms through the Instructor Tool rather than as appositives in the problem text.
4. Write out numbers as words ("three") rather than numerals ("3").

The problem descriptions in the two examples that follow illustrate the appropriate style. While it may at first seem overly simplistic, its meaning is clear to

the human reader. Future enhancements to the MontyLingua software may allow more grammatical variety and a more natural style.

Entering Problems into DesignFirst-ITS

Example 1: Movie Ticket Machine

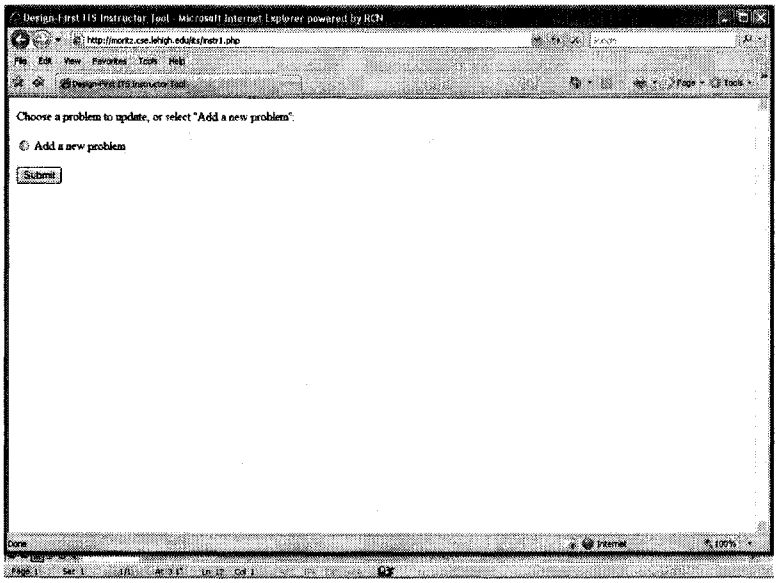


Figure 1: Initial Instructor Tool Screen

1. The initial Instructor Tool screen presents a list of existing problems in the database, plus an option to add a new problem. Click on "Add a new problem" and press "Submit."

2. Enter the title and description in the fields provided. The user interface option will generate the classes needed for either a character-based or graphical interface.

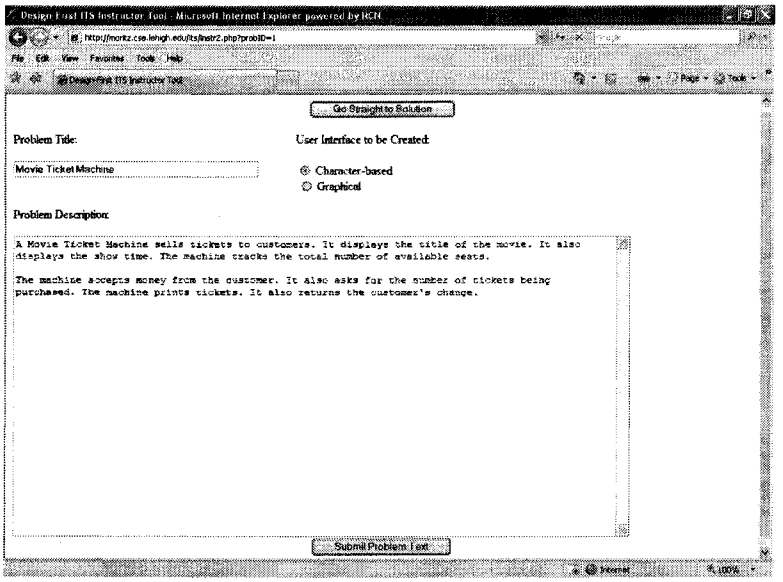


Figure 2: Problem Description Entry

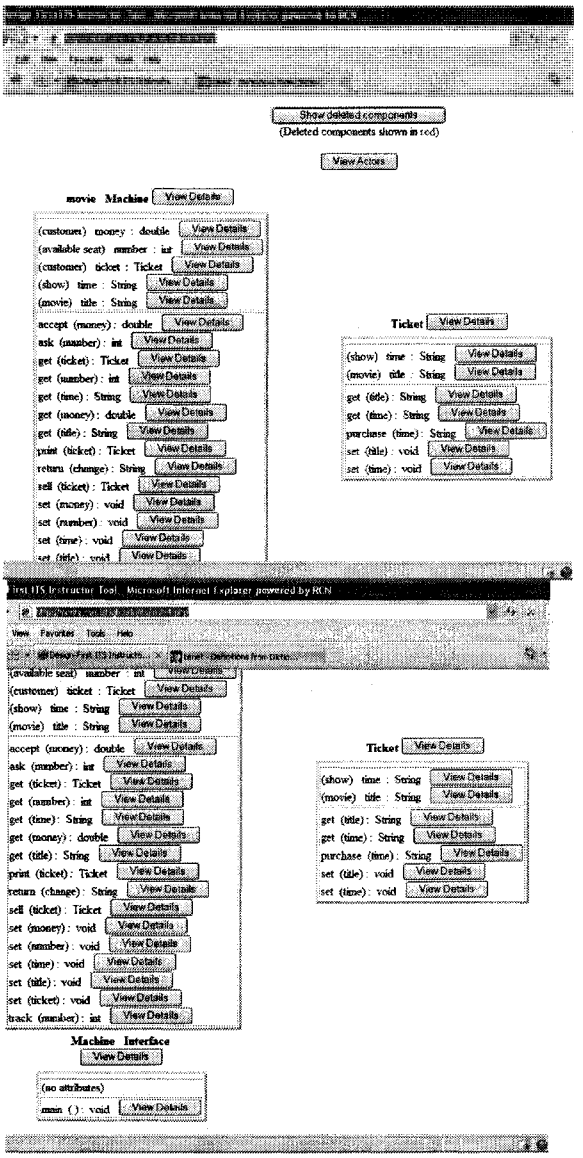
The problem description is:

> *A Movie Ticket Machine sells tickets to customers. It displays the title of the movie. It also displays the show time. The machine tracks the total number of available seats.*
>
> *The machine accepts money from the customer. It also asks for the number of tickets being purchased. The machine prints tickets. It also returns the customer's change.*

Notice that the problem description is concise. It includes the necessary functions of a machine that sells movie tickets and nothing else. It contains simple sentences, all in the active voice.

3. Press the "Submit Problem Text" button. This deletes any existing solution components for this problem, and generates a solution from the text. If the process completes within 30 seconds, a class diagram is displayed. However, the analysis usually takes longer, in which case a message is displayed asking the user to view the result later.

4. To view the class diagram, go to the problem description screen and click on "Go Straight to Solution". Do not click on the "Submit" button unless you want to start from scratch; it will delete all instructor annotations as well as the existing solution.



Figures 3a, 3b: Generated solution to Movie Ticket Machine

5. The instructor can view the details about any class, attribute or method by clicking on the "View Details" button next to the component name. Before proceeding to view and change components, let us examine the design that was generated.

   Three classes have been created: Machine, Ticket and Interface. The Interface class was generated for the character-based user interface option that was selected; the other two classes come from nouns found in the text. These nouns are either subjects of sentences, or were objects that were found to have some characteristics of their own and were not identified as simple data items. If a noun was used with adjective(s), one of those adjectives appears in parentheses before the class name.

6. Click on "View Details" next to (movie) Machine. The following screen appears:

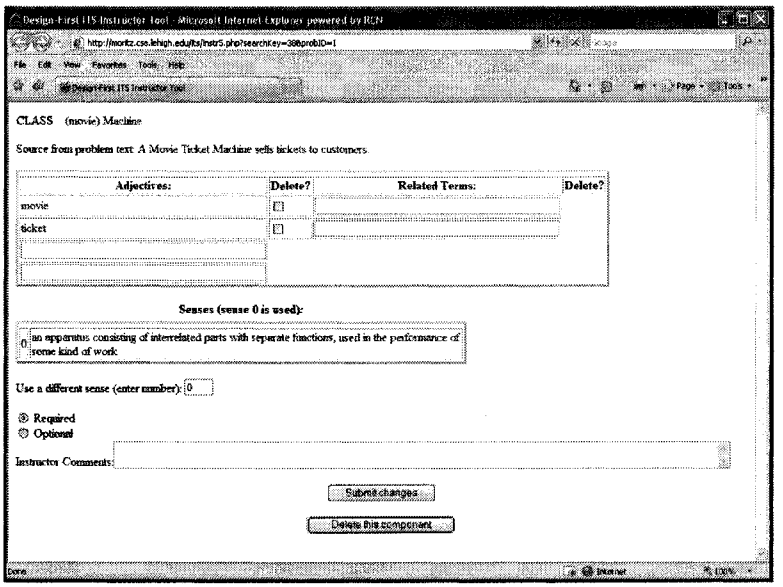


Figure 4: Details of Machine class

Fields on this screen are:

- Source from problem text - This is the first sentence from which this component was derived.
- Adjectives - These are all adjectives for the component found in the problem description. Adjectives may be added by typing into the blue-outlined fields; they may be deleted by checking the box next to the name.
- Related terms - Synonyms for the class name. They come from WordNet, which is also the source of "Senses." Related terms may also be added and deleted manually.
- Senses – "Sense" is WordNet's term for definition. Many words have multiple senses, listed in order of frequency of use. The Instructor Tool chooses the first sense (number 0) by default, but the instructor may choose a different sense. Doing so invokes the Instructor Tool to replace the related terms based on the new definition.
- Required/Optional - Required is the default for all components except get and set methods. The instructor may mark any component optional, or change an optional component to required.
- Instructor Comments – Comments are used to provide additional feedback to the student explaining the purpose of the component, or why it may be optional. Comments are available to the Pedagogical Agent for display in the student interface of Design-First ITS.

- "Submit changes" button – Any changes entered in the above fields will be processed when this button is pressed.
- "Delete this component" button – Used to mark the component deleted. The instructor must enter a reason for deleting a component; it is entered on a separate screen that is displayed automatically.

7. Close the Machine detail screen and return to the class diagram. We will now delete the Ticket class. Click on "View Details" next to the Ticket class name.

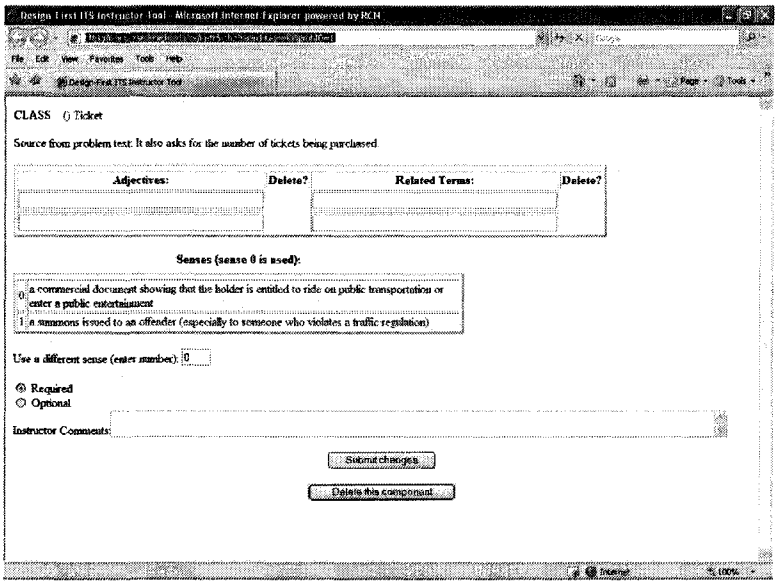


Figure 5: Details of Ticket class

8. Click on "Delete this component" and enter "*Although ticket is an object in the problem, it is only printed as needed, not saved between customer transactions. It is not a "persistent" object in the problem.*" into the Instructor Comments field on the next screen, then click "Submit." You will receive a confirmation of the delete.

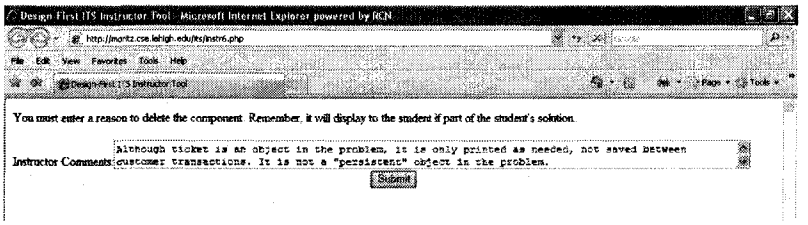


Figure 6: Reason for deleting class

9. Back on the class diagram, click on "Show deleted components" at the top. The screen will be redisplayed, with deleted components in red. All components of the class have been marked deleted.

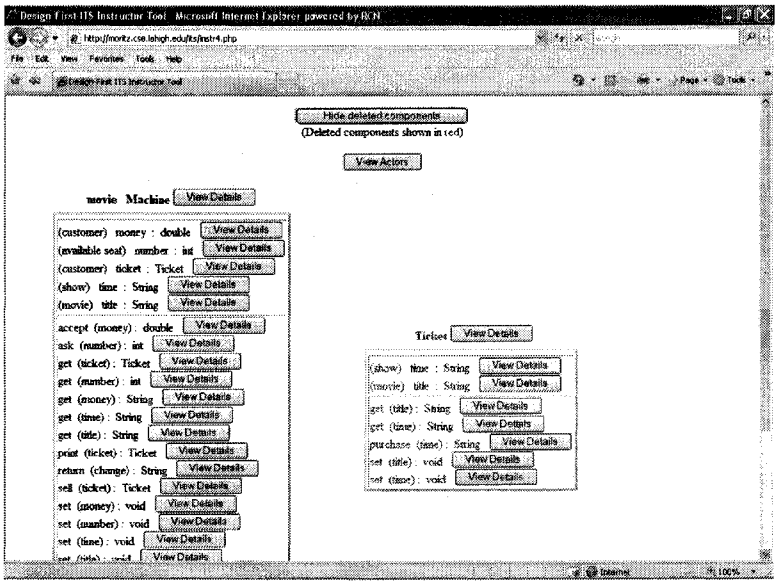


Figure 7: Class diagram with deleted components

10. Notice that there is an attribute "ticket" of type "Ticket" in the Machine class. Delete this attribute through the same process: click on "View Details," "Delete this component" and enter a reason. The attribute and its get and set methods will be marked deleted.

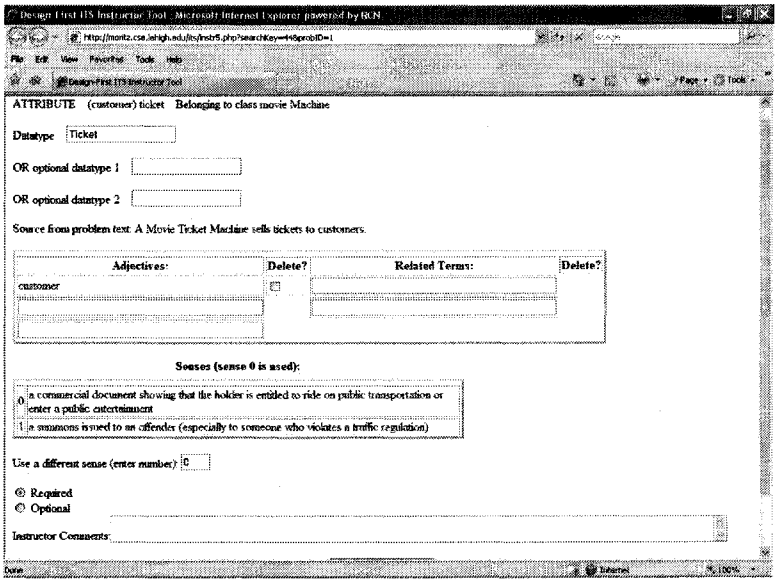
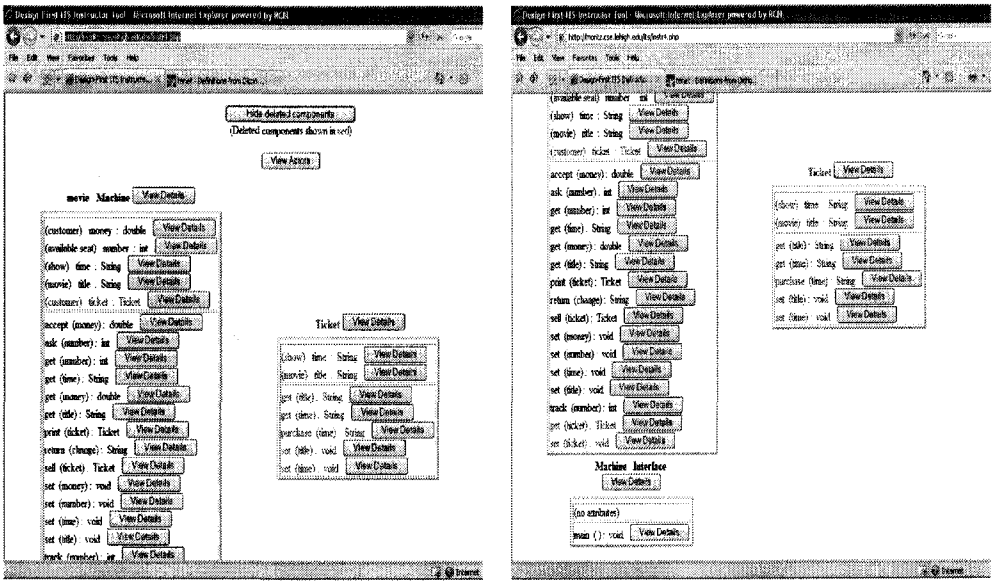


Figure 8: Details of attribute ticket

The attribute detail screen looks like the class detail screen, except that the datatype is shown. The user may change the automatically-selected datatype, and may also add up to two alternate acceptable datatypes.

11. The ticket attribute and its get/set methods are shown after deletion.



Figures 9a, 9b: Result of deleting attribute ticket

12. There are two methods remaining related to the deleted class Ticket: print (ticket) and sell (ticket). We view the details of print (ticket) first.

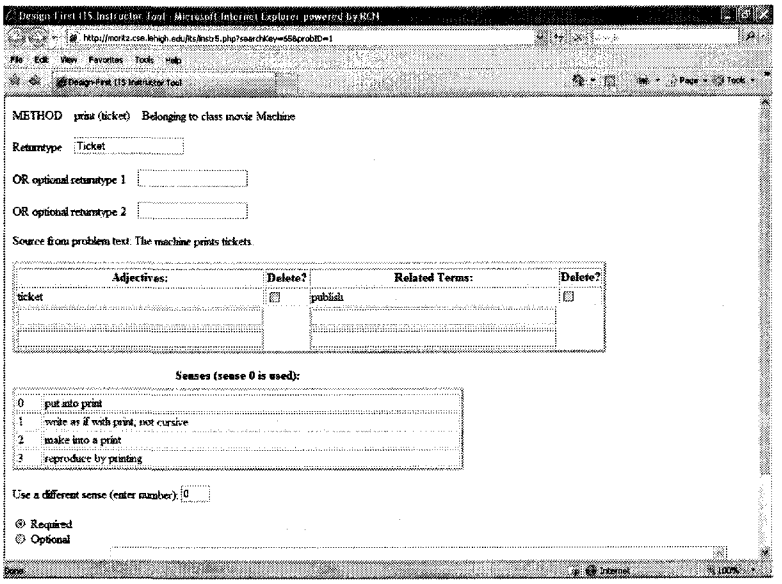


Figure 10: Detail of method print (ticket)

The method detail screen is similar to the attribute detail screen. Instead of datatype, the returntype of the method is shown. Alternate returntypes may be entered. The adjectives listed for a method are really the direct object(s) of the verb (which is the method name). This print method prints tickets. If more than one adjective were displayed, they would all be synonyms of the word ticket.

Change the Returntype from Ticket to void. Also enter boolean as an optional return type. Click "Submit" to save the changes. A screen verifying the update is displayed; click the button to redisplay the updated method.

13. We now address the method "sell (ticket)." The problem text from which this method came refers to the entire process. It does not represent a step or a discrete function the Machine performs. Delete this method using the same process by which an attribute is deleted.

14. Figure 11 shows the class diagram after our changes (deleted components are hidden). The instructor should review the details of every remaining component to make sure the correct sense has been chosen, all acceptable data/return types are correct, all adjectives and related terms have been entered, and each is correctly marked as required or optional. Additional instructor comments should be added to enrich the feedback provided to students.

    The next few steps illustrate some common changes.

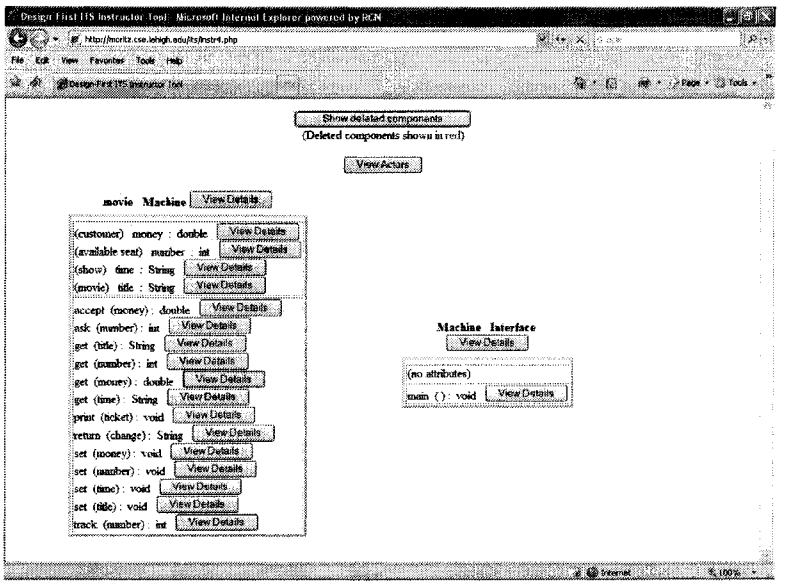
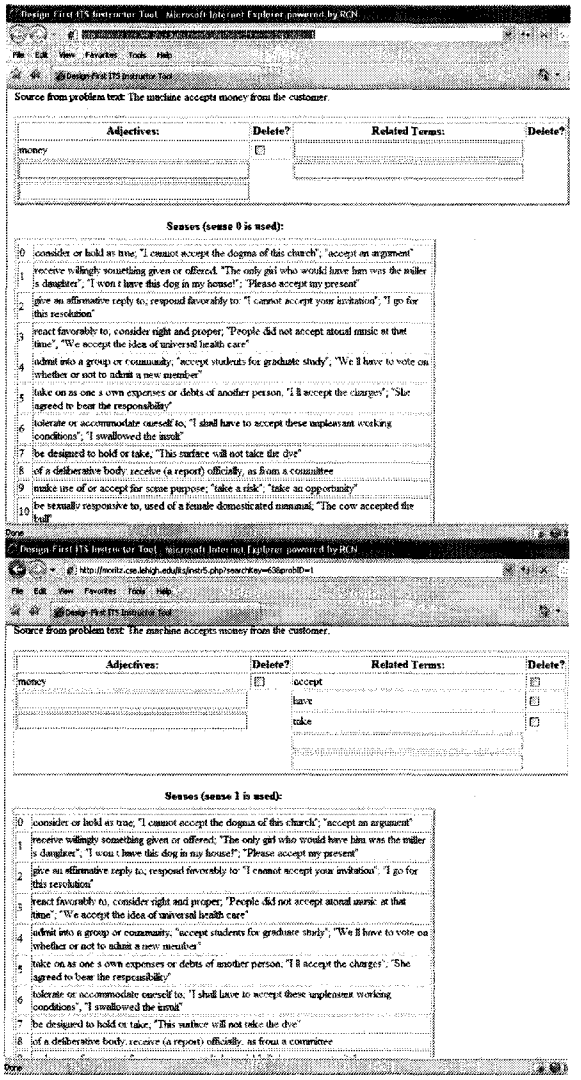


Figure 11: Class diagram after changes

15. View details for accept (money). The sense that was chosen to define accept is not appropriate. The second choice, "*receive willingly something given or offered,*" is a better meaning for the problem context. Enter its sense number (1) in the "Use a different sense" field. A portion of the original and resulting details screen for accept (money) is shown in Figures 12a and 12b. Notice how new related terms appropriate to the new definition have been added.

16. Method track (number) was added for tracking the number of seats left in the theater. This doesn't really need its own method – it could be accomplished with a single line of code. An instructor could choose to delete it, or accept it as appropriate in some designs (perhaps as a method that returns the number of

tickets remaining). Figure 13 shows the revisions an instructor could enter to support it as an optional component.



Figures 12a, 12b: Before and after changing sense of method accept (money)

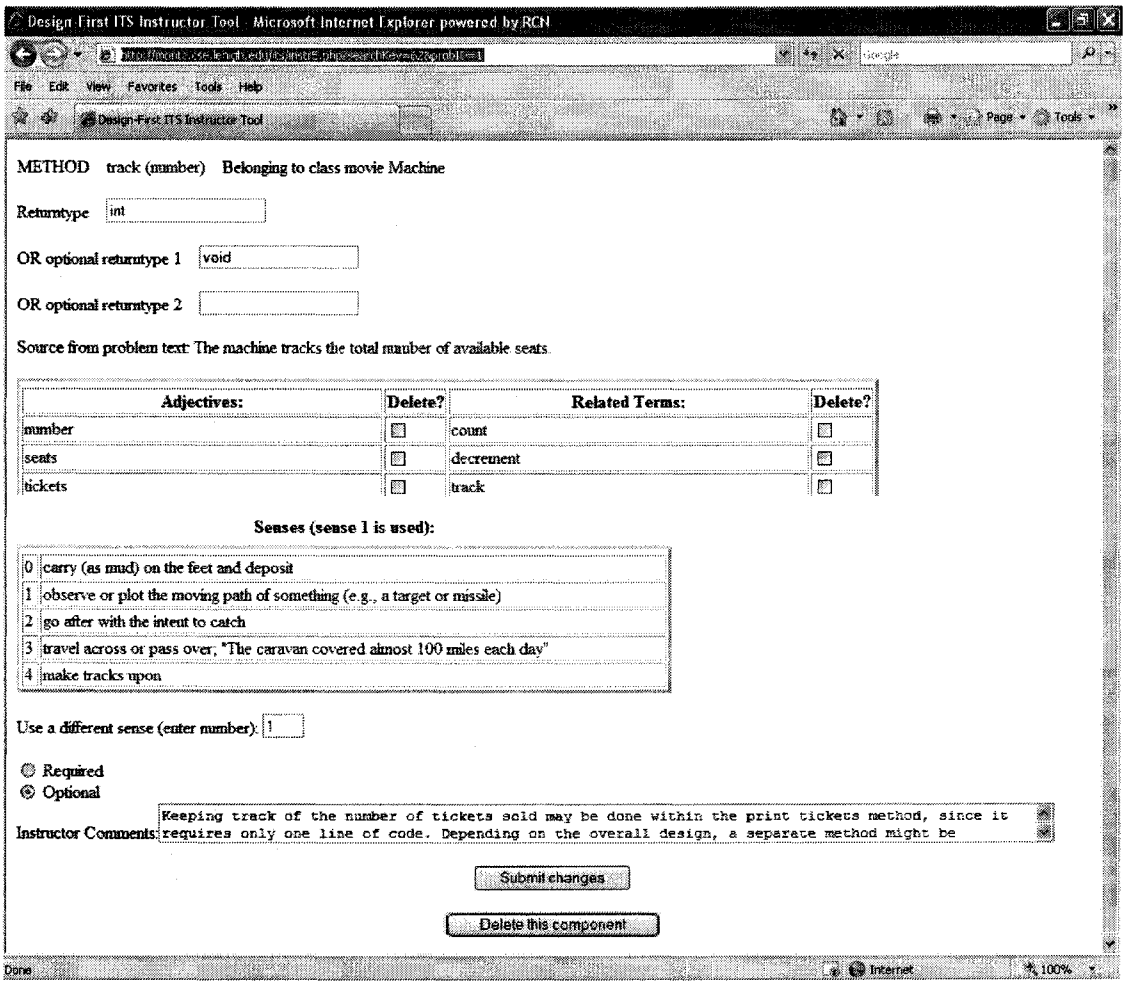


Figure 13: Method track (number) after revisions

17. The "View Actors" button on the class diagram screen displays a list of all the actors extracted from the problem description. The "Update/Delete" button next to the actor allows entry of instructor comments or deletion of the actor. Deletion requires a comment, just as for all other components.

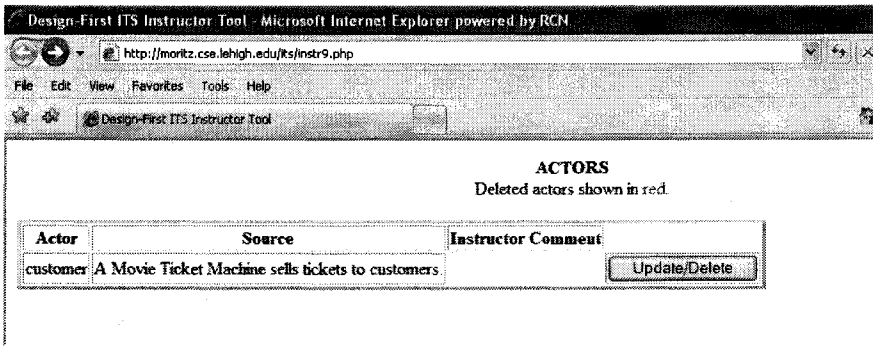


Figure 14: Actors screen

This completes Example 1. By following the steps outlined for all solution components, a complete solution with optional variations was built. All deleted components include an explanation of why they are not suitable for a correct solution. All variations allowed through optional components are identified by the instructor, allowing the instructor's judgment to determine what DesignFirst-ITS evaluates as acceptable. Instructor comments allow the instructor's own thoughts to be conveyed to the student by the ITS' Pedagogical Advisor.

A notable omission in the Instructor Tool is the ability to add components directly to the solution. One of the principles of DesignFirst-ITS' problem solving approach is that all required knowledge for a problem be explicitly stated in the problem description. If a desired component does not appear in the automatically-generated solution, the problem text should be revised to include that component and a description of its role in the overall problem. A new solution may then be generated from the text.

Example 2: Automated Teller Machine

Example 2 is included to give the user a deeper sense of how the Instructor Tool interprets problem descriptions. An increasing familiarity with this aspect of the tool will improve the instructor's ability to create descriptions that are more concise and accurate, and yield good generated solutions. We also show a process which the instructor can follow in evaluating and revising a generated solution.

1. Select "Add a new problem" on the initial Instructor Tool screen, then enter "ATM Machine" for the problem name and the following text for the problem description:

   An ATM dispenses money from an internal supply. It reads a customer's card number. It also reads the customer's password. It verifies that the card number and password are correct. The ATM contacts the bank's central database for the verification and customer balance.

   The ATM can display the customer's balance. It can deposit money into the customer's account. It also withdraws money from the account. The ATM prints a receipt showing the type of transaction and account number. The receipt also shows the new account balance.

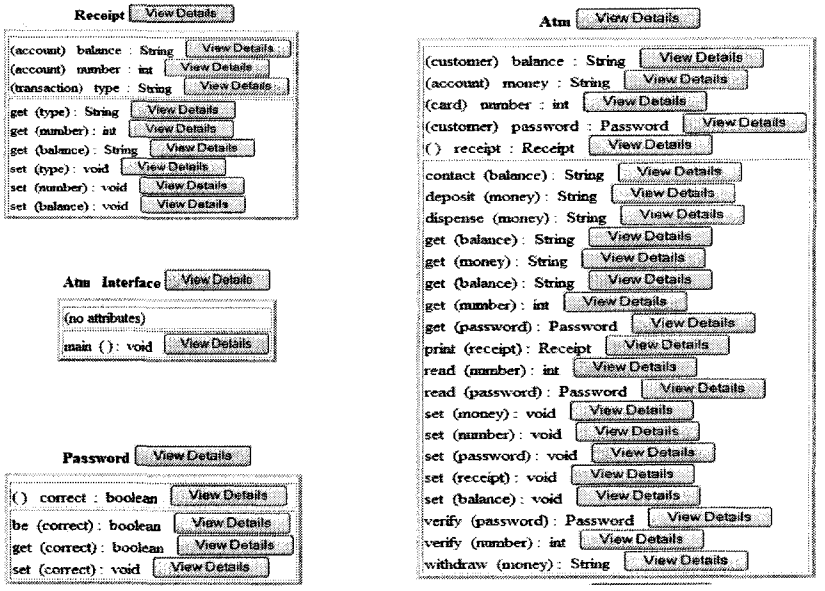


Figure 15: Class diagram generated for ATM problem

ACTORS

Deleted actors shown in red.

Actor	Source	Instructor Comment	
(bank) database	The ATM contacts the bank's central database for the verification and customer balance.		Update/Delete
customer	It reads a customer's card number.		Update/Delete

Figure 16: Actors generated for ATM problem

2. Classes should be examined first.
   a. The ATM class is required.
   b. A separate class for Receipt is questionable. We could treat it like the Ticket class in the Ticket Machine problem, by considering it a short-lived object that is printed by a print receipt method. But a valid design could also treat it as a class, and create a Receipt object when one needs to be printed. This would also require moving the print (receipt) method to the Receipt class. Copying a method from one class to another is not currently supported in the tool, but may be recommended as a future enhancement. Even so, the instructor may mark this class as optional, with an explanation in the comments field.
   c. Password should not be a class. Typically, a password is just a character string that does not have any behaviors.
   d. The Atm Interface class is needed as a driver and to control the user interface.

   Make note of which classes need revisions or deletion, but don't enter the changes until after completing step 3.

3. Look for missing classes. If an expected class is not present, review the problem description. Remember that every component of the problem must be mentioned in the text, along with a description of actions and data that it includes. Attributes and methods will also not be part of the design if they are not included in the description.

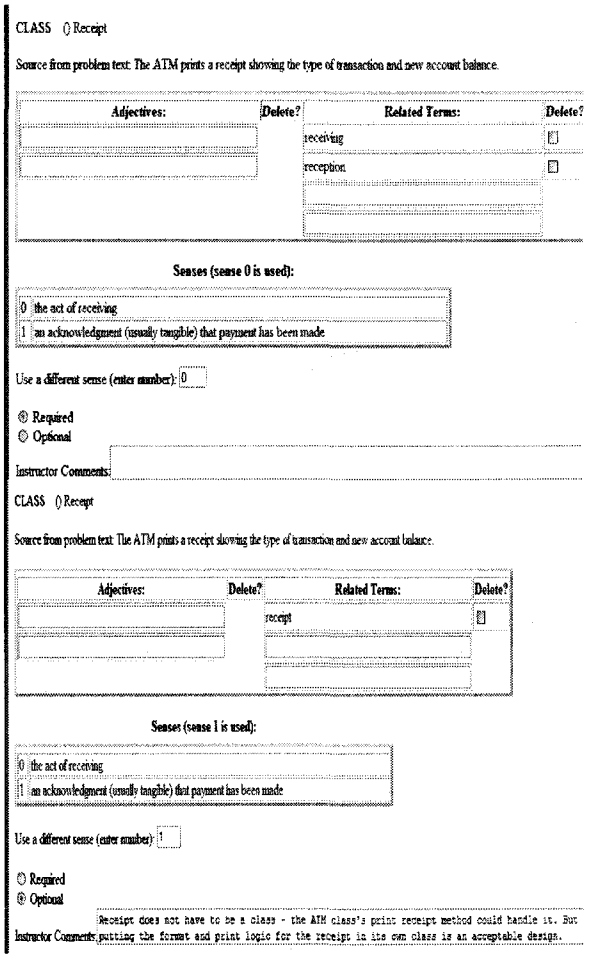
   Missing attributes and methods in the classes that are a valid part of the solution should also be considered at this time, since revising the problem description requires regeneration of the entire solution and wipes out annotations already made to other components.

   If a missing component is included in the description, try rewording the text that pertains to it. Were all sentences simple subject/verb/object structures, in the active voice, with no appositives? Split multiple clauses into separate sentences.

   Text revision and solution regeneration is expected to be an iterative process. Experience with the tool will result in more thorough initial descriptions and

Text revision and solution regeneration is expected to be an iterative process. Experience with the tool will result in more thorough initial descriptions and fewer iterations needed to create a complete solution, as well as fewer misunderstandings by students.

4. After text revisions produce all expected components, enter the changes to the classes noted in step 2. Also review all classes generated as a result of the revisions to the description.

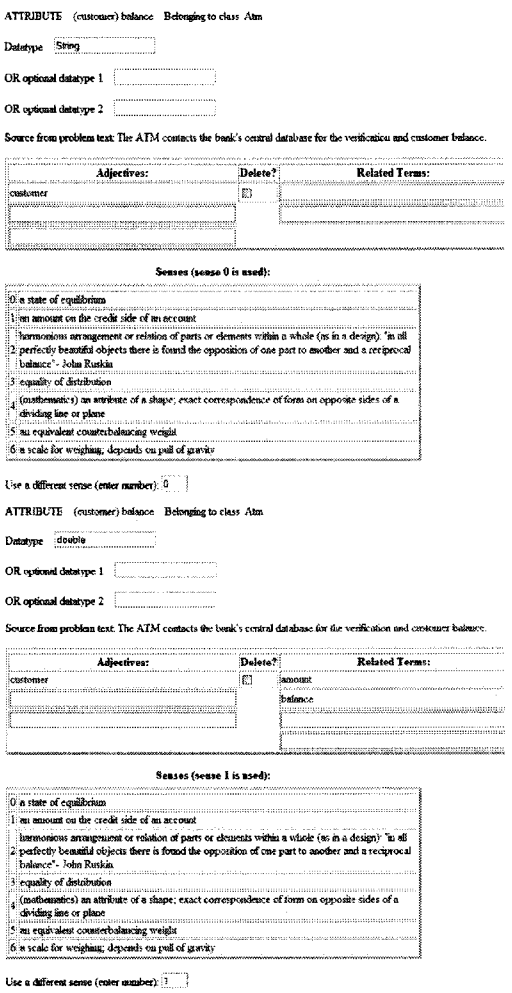


Figures 17a,17b: Receipt class before and after revision

5. Now carefully review the attributes and methods of all valid classes. We describe changes to only a few of the components of the ATM class, to illustrate some common situations.

   - (customer) balance – Notice the datatype is incorrectly set to String. On viewing the details for the attribute, we see that an inappropriate sense was chosen as the definition. We change the sense to "*an amount on the credit*

side of an account." The datatype is automatically corrected to double, and related terms for the new definition are added.



Figures 18a,18b: (customer) balance before and after revision

- (account) money – The source of this attribute is "*An ATM dispenses money from an internal supply.*" That means this attribute keeps track of the amount of money in the ATM for distribution to customers. *Account* is not a suitable adjective for this attribute. We delete it and adjective *customer account. Internal* is retained as a valid adjective, and we add *atm* and *machine* as adjectives. We also add related terms *balance* and *cash*, and correct the datatype (it was String, but should be double).
- (customer) password – Should be deleted, not because the Password class was deleted, but because there is no need to store the password. It is needed only to verify that it is correct, which is handled in a single method *verify (password)*. It can be passed to that method as a parameter.
- receipt – Also not needed as an attribute. Whether created in a Receipt class or the print receipt method, it exists only for the current transaction.

- contact (balance) – This is an example of an extraneous method created due to a flaw in MontyLingua's processing of the sentence containing this verb. It can be deleted with a notation that it is a solution generation error.
- read (password) – Reading data should be delegated to the user interface portion of a system. That allows for separation of logic and interface, and simplifies coding changes needed to replace the user interface. Additionally, password is not an attribute, so there is no "get" method needed. Thus, method should be deleted.
- verify (number) - This method is redundant with *verify (password)*. When the ATM checks the customer's password, it needs the card number as well. So verifying that the card number is valid is part of the process of *verify (password)*.

The solution will be complete when a review of every attribute and method in each class done, with changes and comment entered as required.

This example is intentionally more complicated than the first to illustrate the thought process by which an instructor constructs a problem for her students. The Instructor Tool was designed to aid this process. The intended results are clear, well-defined problems with no misunderstandings in student interpretation or surprises in valid solutions. Additionally, the Tool provides sufficient customizations and commenting of the generated solution to reflect the instructor's preferred design techniques and teaching style. The final solution set can be as restrictive or permissive of variation as the instructor desires.

Appendix D
DesignFirst-ITS Instructor Tool Evaluation Questionnaire

Thank you for agreeing to evaluate the DesignFirst-ITS Instructor Tool. I hope that you will find it a useful tool for developing new assignments for beginning Java students, for evaluating and refining solutions to problems you may already be using, or for other non-instructional uses, such as a software engineering aide. Below are some recommendations for getting started, followed by specific questions about your experience with the tool.

Getting Started

1. Access the Instructor Tool at moritz.cse.lehigh.edu/its/instr1.php. Currently it must be accessed from within Lehigh's network (via the VPN if off-campus).

2. Read the User Manual (a link is provided on the first page of the Instructor Tool). Step through the two sample problems. You may also view the problems already entered in the tool.

3. Please keep in mind the goal of the tool as stated in the User Manual: to create problems that support the Design First curriculum, in which students learn how to apply object-oriented principles to design solutions before coding.

4. When entering your first problem, start small: enter two or three sentences that describe part of the problem, then generate a solution. Experiment with rewording the text. Add new pieces of the problem once you are satisfied with what is generated so far. Keep in mind that regenerating a solution from the text removes all edits made to the class diagram (as explained in the User Manual), so complete all text revisions before editing the solution.

5. After entering a complete problem, you are encouraged to try different versions of the same problem. They may be added as new problems so as not to overwrite any complete solutions you want to save.

6. If the solution generation does not complete within a few minutes, please email me at sgh2@lehigh.edu. Postpone additional testing until receiving a response from me.

7. Add one or two problems before answering the questions below.

8. I am also available to receive your feedback in person. Please contact me if you'd like to meet to discuss your impressions of the tool at a more detailed level.

Thank you for your time and expertise!
Sally Moritz

Evaluation Questions

A. Solution Generation

1. Rate the quality of the solution(s) generated on a scale of 1 to 5 (1=poor, 5=excellent).

2. Did the solution generation:
   a. fall short of your expectations?
   b. meet your expectations?
   c. exceed your expectations?

   If you answered a or c, please give example(s).

3. List any bugs you found in the solution generation.

4. Please include any other comments about the solution generation.

5. Was the class diagram presented clearly? Please list any recommendations for improving the display of the solution.

B. Solution Revision/Annotation

1. Rate the ability to update the generated solution on a scale of 1 to 5 (1=poor, 5=excellent), for each of the following aspects:
   a. ease of use
   b. completeness of functionality

2. List any functions not incorporated in the tool that you feel should be added.

3. List any bugs you found in the user interface.

C. General Observations

1. For what uses would you find this tool useful?

2. Please include any additional comments about the tool.

Vita

Sally Moritz has more than 20 years experience in Information Technology in a variety of areas, including software development, database design and administration and networking. She has also held leadership roles, including Director of Information Technology for Times Mirror/Tribune CoOpportunity Center, a financial services provider for more than a dozen newspaper and magazine publishers. Dr. Moritz was an Adjunct Instructor in Computer Science at Muhlenberg College from 1986 to 2002, and Visiting Instructor at Lafayette College in 2003. Courses she has taught include Computer Science I and II in C++ and Java, Data Structures, Software Engineering and Relational Database Theory and Practice.

Dr. Moritz was a National Science Foundation Teaching Fellow from 2003 to 2006 while working on this dissertation's research. As a Teaching Fellow she taught Computer Science classes at Broughal Middle School in Bethlehem, PA and Dieruff High School in Allentown, PA. She is currently a Sr. Developer at Travel Impressions, a subsidiary of American Express in Bethlehem.