

INTEGRATING A FORMAL SPECIFICATION COURSE WITH A SOFTWARE PROJECTS COURSE VIA AN EDITING TOOL

R.F. Gamble

Department of Mathematical and Computer Sciences

The University of Tulsa

Tulsa, OK 74104

email: gamble@tara.mcs.utulsa.edu

Abstract

This paper reports on a two-course sequence for undergraduate students that provides them with an intensive course in formal specification methods and a traditional software design course. The specification course provides an appreciation for the use of rigorous specification methods within the software lifecycle. The manual nature of developing a formal specification provides the appropriate justification for the projects in the software design course in which the students create useful tools to aid the development of a formal specification. Within the project course, the students developed the first prototype of a graphical editor for building Z specifications. We discuss the building of the editing tool and its integration into the software engineering curriculum.

1. Introduction

Formal methods are mathematical techniques used to describe properties of computer systems [13]. These methods provide a framework within which systems can be specified, developed, and verified. One of the main uses of formal methods is to develop a formal specification for the contract between the client and the software engineer. A formal specification provides an unambiguous description that is more precise and usually more concise than an information specification. In addition, formal methods can be used to prove the system complies with the specification.

Experience with formal methods is necessary for software engineering students to fully understand the complexities of creating correct software for critical applications. The undergraduate software engineering

curriculum at the University of Tulsa currently includes two courses: a formal specification course taught with the language Z [12] and a traditional software projects course. In this paper, we present our experience in tying together the two courses by building software tools that can be utilized by both classes.

Limited functionality and complexity of a graphical editor for Z specifications was specified abstractly, but formally in Z, by the specification class. This specification was then used by the projects class to design and implement the editor. By limiting the functionality and complexity initially, both classes could create a final product, which becomes a prototype for the next sequence of classes. In addition to using the editor, the subsequent specification class will analyze it by constructing proofs that the editor meets its specification and where it fails. New functionality will also be specified to augment and perhaps modify the original specification. The current editor, its specification, and analysis will be passed to the projects for design, implementation, and maintenance. We envision that in some cases, only ideas from previous prototypes will be used as redesign and reimplementations occur. Such changes are viewed as necessary in order for each class to have some anonymity from other classes and to participate fully in the software lifecycle. When the editor is deemed complete with respect to the requirements, design and implementation of a new tool that complements the editor, such as a proof tool or automated assistant, will begin.

It is often said that formal methods are the only way in which trustworthy software can be built [3,5,8]. However, due to the tedious nature of manipulating formal specification languages and the inexperience with applying mathematical techniques to software development, formal methods fall by the wayside in industrial software development. Unless we introduce students to rigorous approaches to software development and encourage them to use their formal techniques, we propagate the development of poor quality software. Thus, we have incorporated a course on the Z specification language as part of the software

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

SIGCSE '95 3/95 Nashville, TN USA

© 1995 ACM 0-89791-693-x/95/0003....\$3.50

engineering curriculum. Z is easy to learn provided the student has a basic understanding of discrete mathematics.

In an effort link the specification course to the project course and vice versa, the projects in both courses are intertwined. A formal specification of non-trivial software benefits the students in the specification course by illustrating the applicability of Z for large products. Developing tools to satisfy the specification provides the students in the software design class with a formal vs. informal comparison. Also, developing a tool for review and use by peers motivates the students to take more pride in their work. Finally, by interleaving the developed project through the specification course, we increase the amount of time in which new concepts in specification refinement, reusability, and maintenance can be introduced.

Many students will find entry level programming positions upon graduation. Often these positions are on an ongoing project. These positions do not provide people immediate design and development responsibilities. Instead, these positions may require a person to complete, port, or maintain a particular piece of existing software. These difficult tasks are not readily taught in a traditional software engineering class. While they could be taught using the project the class develops in that semester, often the time limitations and team personnel problems lead to trouble even completing the assigned project. By passing a prototype project from one class to the next, more industrial aspects of entry level software engineering jobs can be brought into place.

We do not purport to simulate a real-world environment in our classroom because we are missing many key components, such as senior personnel and a full-time work schedule [9]. In addition, the motivation for completing the course successfully is not based on eventual promotion or other career-oriented aspects. However, we do present a non-trivial project which with to teach the students the fundamentals principles of software engineering that they can apply to many of the issues they will face with industrial software design.

2. Software Specification Course

The goal of the course is to teach students various methods of rigorous software specification, allowing them to experiment with and compare various specification methods from diagramming to algebraic specifications. The majority of the course is spent on Z:

learning the language, the proof process, and refinement techniques. The final project of the course is to create an abstract initial specification for some aspect of the software project used in the second course of the sequence. This task is non-trivial in that it involves a great deal of complexity that must fit into Z notation.

The specification class is structured using design teams. Each team has a designated leader (often a graduate student) and a librarian. The librarian is responsible for maintaining definitions and documentation. The remaining team members are responsible for the majority of the specification development, including maintaining consistency, though not necessarily equality of representation, with what is being developed by the other teams.

2.1 Initial Editor Specification

Formulating requirements is taught using the available material from the Software Engineering Institute [6]. Each team develops a requirements document. This document is compared with the instructor's document which covers the entire tool. A portion of the functional requirements are chosen for specification. This functionality is limited in order for the class to be able to complete a specification. In class instruction on Z begins with continued focus on the final specification project through the use of team assignments. Once each team has determined the appropriate component schemas, the class as a whole determines a single state schema with which to continue. Each team is responsible for declaring what operations are needed to meet the requirements. These operations are divided among the teams for completion. A final specification document is produced by the end of the class.

The most simplistic operation specification of the editor was the *SaveAs* operation (Figure 1). This specification requires the state schema for the editor, *EditorState*, and the schema for a *File*. For brevity, we present only a portion of the complete schemas. A schema in Z is separated into two parts: the declaration (upper half) and the predicate (lower half). The declaration holds the variables and their types. Z uses the standard types and allows user-define types, which are marked with a UD in Figure 1. The predicate holds constraints. In a schemas like *Editor State* and *File*, the predicate (not shown in Figure 1) can be thought of as describing program

EditorState		File	
InFile: File	<i>file being edited, if saved</i>	Name: String	<i>file name</i>
CurrentContents: P Symbols	<i>contents of file in editor (UD)</i>	Location: Ptr	<i>storage pointer (UD)</i>
		Size: N	<i>size of file</i>
		Contents: P	<i>stored contents</i>
		Symbols	

SaveAs	
Delta EditorState	<i>provides before and after state variables of editor</i>
OutputFile!: File	<i>new file for storage of editor contents (output)</i>
FileName?: String	<i>new file name (input)</i>
NewLocation?: Ptr	<i>new location of file (input)</i>
OutputFile!.Name = FileName?	
OutputFile!.Location = NewLocation?	
OutputFile!.Size = ComputeSize (CurrentContents)	ComputeSize is a function described in another schema
OutputFile!.Contents = CurrentContents	
<i>All other variables remain in the same state.</i>	

Figure 1: A Sample Z Specification

invariants. In a state transition schema, such as *SaveAs*, the declaration may hold input (decorated with ?) and output (decorated with !) variables. When schemas are embedded, the dot notation is used to access embedded variables.

2.2 Subsequent Tool Refinements

At the start of each new tool to be developed, the abstract specification is provided to the project class to build an initial prototype. This specification is refined by subsequent formal specification classes as the tool is developed. For example, the next Z class will utilize the editing tool developed by the preceding software design class for quicker schema development and type checking. In addition, the class will verify that the tool satisfies the previous formal specification and where ambiguities reside in the specification. This specification will be further refined as the class project continues to be developed and analyzed. By continually evolving the projects in both classes, the students learn a valuable lessons in prototyping, reuse, and reengineering, where software engineers make development mistakes, the impact of changing the specification of a product, and maintenance issues.

3. Software Projects Course

The second course in the sequence is a traditional project course required by all graduating students in Computer Science and Computer Information Systems. Along with discussing the full software engineering lifecycle, the students are introduced (some reintroduced) to the very basics of

Z in order to design the product from the specifications. The final project of the course was to design the graphical editor specified by the previous specification class.

The structure of the software projects course is similar to many courses found in the literature [1,2,11]. Teams were chosen by the instructor using an evaluation questionnaire based on courses taken and student confidence in their abilities of programming, writing, organization, and speaking to a group. To emphasize the strength of each student and to force the students to develop skills sometimes crucial for a good software engineering, team roles were assigned according to the student's strength, with the requirement that each student to participate in some way in all other roles.

All teams were given a basic set of requirements from which they wrote a requirements document and informal specification with each team using a different specification mechanism. The resulting team specifications were compared with the Z specification and discussed with the students. Team presentations allowed for exposure and discussion of the different methods of specification, from informal to informal, as they related to the project. Additional class discussion brought the teams together for quality assurance as each team editor was developed. As a whole, the class took great pride in creating a tool they knew would be used and evaluated by their peers.

The resulting tool was completed as defined. The specifications were restricted so that the prototype could be built, and the students could

experience the whole life cycle. By utilizing the editor in future classes, the entire life cycle can be experienced, including practical aspects of maintenance which can be sometimes overlooked in the typical projects course.

4. Editing Tool

One of the most immediate needs to incorporate formal methods into software development is the use of tools that aid practitioners in developing a formal specification of an eventual implementation. In addition, automated tools make formal methods more convenient and practical to use by reducing the drudgery and redundancy sometimes associated with software specification. Since formal specifications can become large and mathematically unwieldy, a tool that makes available names of specification components and their dependencies is very useful. For the beginner, it is important that the tool restricts the user to the particular constructs of the language being specified. Finally, the availability of the appropriate fonts is important for expedient development. If the specification language is strongly typed, then there is need for type checking within the developed specification. These characteristics form the core requirements of the editing tool.

The actual editing tool designed and developed by the software projects allows the input and general editing of Z specifications. Graphical templates are used to set up Z schemas. These templates restrict the user to the expected format of the specification to force syntactic correctness, e.g., what can be present in the declaration and predicate sections of a particular schema type. For example, declaring variables can only appear in the declaration section of a schema. Thus, the editor guides the user to a reasonable specification using templates for each major component of the specification. Easy access to symbols, including the non-standard ones, used in Z is provided by the availability of font symbol buttons. The editor provides a list of system defined and user-defined types. Finally, the editor converts the document into L^AT_EX [7]. This conversion allows for printing of the specification and its inclusion into documents. For type checking, the editor is interfaced with the ZTC type checker [14], which takes the L^AT_EX format of the schemas.

The first editing tool was restricted to state and state transition schemas. The templates that guided the user did not provide aid in constructing many of the complex types allowed in Z. Additional limitations included the maintenance of schema names, cutting and pasting large portions of text within the document and across documents. Figure 2 shows the some of the schemas from the Wing's

Library Problem [4]. The specification class compared the editor with MicroSoft Word 6.0® using this problem. More schemas and edits were tested than what is shown in Figure 2. All of the class was highly experienced in MicroSoft Word 6.0® and not experienced in the use of the Z editor which runs in Motif on HP-UX. The times to complete the entering of the schema were similar. However, the editor uses templates to guide the user through the correct syntax. In addition, the fonts were readily available. This was seen as a benefit when creating a new schema from scratch. The absence of cut and paste facilities increased the time and much of the hassle associated with the Z editor. These facilities will be specified and implemented in subsequent classes.

For the specification class, the idea of incorporating an automated tool, such as an editor, provides a mechanism by which students can increase their skills in formal specification on a non-trivial task. In addition, since the majority of these students specifying the tool must also design it to meet the specification in their project class, they are instilled with a larger goal than just completing the class. For the project class, the tool serves as a practical piece of software to be used by their peers, providing students with a sense of pride beyond that of obtaining a grade.

When the editing tool is complete (or nearly complete), students can continue to build other CASE tool components that integrate the editing tool. We note that there are other tools available to build Z specifications. However, the our goal is to have the students build a complex CASE tool (over several semesters), in which the editor is only a part. Since every student is familiar with the use of an editor, we believed that this tool would be the easiest one to use for our integration experiment. As the courses mature in this manner, we will bring in the free tools available to the students to evaluate with respect to what has been done in class and the future additions to the tool.

5. Discussion and Conclusion

There is an abundance of literature on the structure of software engineering project courses, e.g., [1,2,9,11]. Most advocate the instruction of principles of software engineering along with the development of a non-trivial project. However, the use of formal methods is not fully considered, though these methods are believed to be the key in training a professional engineer [5]. Saiedian [10] states that formal methods are the foundational tools lacking in an undergraduate software engineering curriculum.

Overall, we have found that the ability to work with such a language fosters a level of

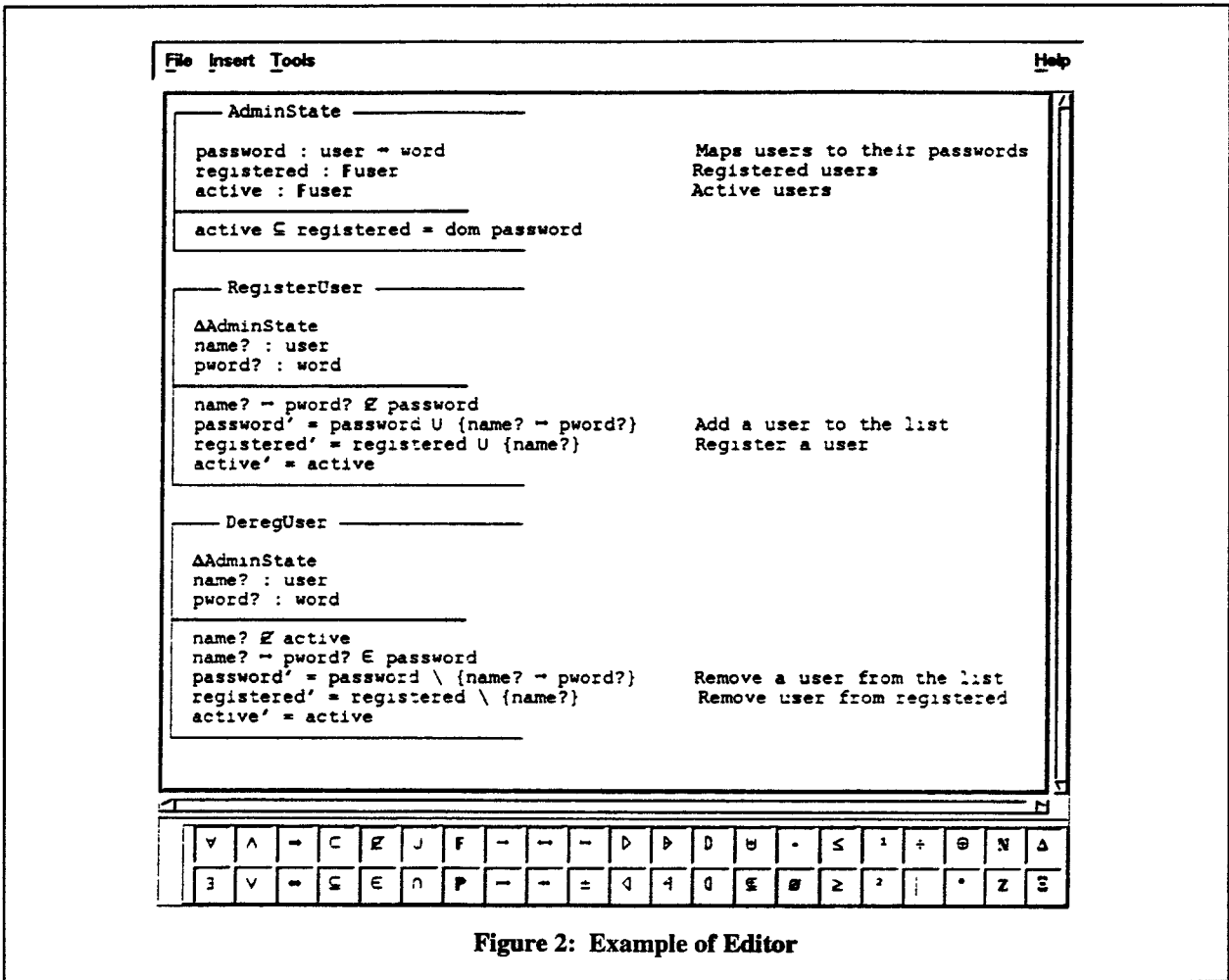


Figure 2: Example of Editor

confidence in the creative problem solving abilities of the students. Coupling this with a practical design and development course provides them with a final challenge of bringing together their mathematical and programming knowledge.

References

- [1] E.J. Adams, A project-intensive software design course, SIGCSE Bulletin, V. 25, N. 1, March 1993.
- [2] J.M. Clifton, An industry approach to the software engineering course, SIGCSE Bulletin, V. 23, N. 1, 1991.
- [3] E.D. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ, 1976.
- [4] A. Diller, *Z: An Introduction to Formal Methods*, J. Wiley & Sons, New York, 1991.
- [5] D. Gries, *The Science of Programming*, Pitman Publishing, London, England, 1987.
- [6] P.C. Jorgensen, Requirements Specification Overview, SEI-CM-1, Software Engineering Institute, Carnegie-Mellon University.
- [7] L. Lamport, *L^AT_EX User's Guide and Reference Manual*, Addison-Wesley, Reading, MA, 1994.
- [8] M.R. Lowry and R.D. McCartney, *Automating Program Design*, AAI Press, Menlo Park, CA, 1991.
- [9] P.N. Robillard and D. LeBlanc, the simulated working environment in a project-based software engineering course, *Computers & Education*, V. 12, N. 4, pp. 471-477, 1988.
- [10] H. Saiedian, Towards more formalism in software engineering education, SIGCSE Bulletin, V. 25, N. 1, March 1993.
- [11] T.J. Scott, L.H. Tichenor, R.B. Bisland, Jr., and J.H. Cross, "Team dynamics in student programming projects," SIGCSE Bulletin, Vol. 26, No. 1, March 1994.
- [12] J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, 1989.
- [13] J.M. Wing, A specifier's introduction to formal methods, *IEEE Computer*, pp. 8-22, Sept. 1990.
- [14] J. Xiaoping, ZTC type checker, Depaul University.