# The Journal of Computing in Small Colleges

## Proceedings of the Third Annual CCSC Midwestern Conference

September 27-28, 1996, DePauw University
Greencastle, Indiana

## Proceedings of the Fifth Annual CCSC Rocky Mountain Conference

October 18-19, 1996, Metropolitan State College of Denver
Denver, Colorado

Volume 12, Number 2 November 1996




# CS II: AN APPLIED SOFTWARE ENGINEERING PROJECT

*Marcin Paprzycki and Janusz Zalewski*
*Department of Mathematics and Computer Science*
*University of Texas of the Permian Basin*
*Odessa, TX 79762*

**ABSTRACT**

Recent publications suggest that there is a gap between computer science curricula and industry needs. It is shown how a software design project introduced in Computer Science II can help narrow the gap.

## 1. INTRODUCTION

It was recently suggested (Lawlis & Adams, 1995; later denoted as L&A) that current CS curricula do not address the real needs of the industry:

> " ...industry needs software practitioners who understand the dynamics of developing and reengineering large software systems, adeptly employing techniques such as analysis and reuse." (p. 5)

They also criticize the current CS curricula for putting too much attention on creating small pieces of software from scratch. It is even suggested that in the modern CS curriculum there is no place for creation of software artifacts from scratch. Analogous criticism can be also found in Jones (1995). In addition, there is evidence that specialists with software engineering background are sought by the industry (see Epp 1993, Paprzycki et. al. 1996).

The aim of this paper is to introduce a semester long software development project that is currently used in the CS II course at the University of Texas of the Permian Basin (UTPB) - Section 2. We will discuss how this project addresses industrial needs as defined by L&A - Section 3. Finally, the current experiences from the project will be summarized - Section 4.

## 2. PROJECT DESCRIPTION

At UTPB, CS I is offered not only to prospective CS majors, but also to students who plan to major in other sciences - biology, chemistry, geology and mathematics students use it to satisfy their programming requirement. CS II caters primarily to prospective CS majors. Both courses use the textbook "Structures and Abstractions" by W. Salmon (1994). The first semester usually covers the first thirteen chapters of the book thoroughly. Chapter 14 (on arrays) is covered near the end of the semester without in-depth exercises. CS II is a relatively small course, consisting typically of one section of approximately 15-20 students. Material-wise it covers the remaining chapters of the book.

Two years ago our introductory CS curriculum (freshman and sophomore level courses) has been updated by introducing Software Engineering elements to it (Paprzycki et. al 1994, 1995b). In CS I, emphasis has been put on the concepts of software design and testing. A principle of program design in CS I is that the students can start coding only after their program design, which follows established rules, has been approved by the instructor. The design is language independent but very detailed, so that another person would be able to write the

straight code based on it (which is often the case – students exchange designs before coding). Along with the design, a test plan is produced based on it, for use after the code is ready.

A semester long software development project was introduced to CS II in an attempt to confront the future computer scientists with a larger software artifact. The project proposed here is a natural extension of the ideas expressed by Andrianoff et. al. (1991) and Epp (1993). The primary difference is the size and length of the project. Their assignment(s) deal with relatively small project(s) which take only a couple of class periods to complete.

The starting point for the project is Conway's "Game of Life" (Gardner, 1970). This relatively simple simulation game is extended to a larger artificial life simulation project during the course of the semester. The project is based on group work, where groups consist of 2-4 students - depending on the class arrangements and attrition rate. To mimic the real-life software development cycle the project proceeds in phases. In Phase I the original game of life is being implemented. As the initial assignment we use the Programming Project 14 (Salmon, 1994, 466-467). It should be pointed out that there is an important discrepancy between Conway's rules of the Game of Life and the rules specified in the assignment. To be able to observe Conway's patterns, the rule for survival (rule b.) needs to be changed from "a creature survives when there are at least two creatures in the neighborhood" to "a creature survives when there are exactly two creatures." The software development process consists of the following steps:

I Formulation of the software requirements
II Writing the design
III Instructor's comments on the design
IV Implementation
V Testing and preparation of the User Manual
VI Across-the-class comparisons
VII Instructor's comments on the artifacts

At the beginning of the semester groups are formed and students study the Programming Project as well as the paper by Gardner. Then they are asked to specify and discuss the basic requirements of the project (with a very strong emphasis being put on the proper development of human-computer interface). This step is followed by each group writing a design (in the form that they were introduced to in the CS I course – see above). These designs are then commented on by the instructor.

After receiving feedback on the designs, students work on improving them and implementing the code, testing the code and writing the user manual. At this moment, groups exchange the artifacts: design, user manual, and code. Each group receives the artifacts prepared by two or three other groups. The groups have the following tasks:

a) commenting on the design and user manual,
b) verifying the code against the design,
c) running the code and testing it,
d) commenting on the user interface.

All tasks lead in a natural way to introduction of the concepts of independent verification and validation. Students are instructed to concentrate their attention on particular verification criteria, such as consistency and clarity. They are also informed that they should read the user manual from the point of view of prospective users, so that they concentrate their attention on all things that are unclear or insufficiently explained. The improved documents are then commented on by the instructor.





Phase II, the first modification of the system, is almost a repetition of the previous one (steps I-VII), with new requirements added to the problem specification. A typical example of such a new requirement is a change in the shape of the domain from a square to the doughnut (based on the Programming Project 15, p. 467). This time students work not on the new code, but introduce modifications to their own, earlier developed, programming artifacts. They need, first, to reformulate the software requirements and rewrite the design. Then, after instructor's feedback, they implement the new system by changing the existing code. This step is again followed by testing, changes in the user manual, and across-the-class verification.

This phase is then repeated as many times as there is time during the semester. Typically, the new parameters are introduced to the system in the following sequence: gender, movement, aging and food (more information and a detailed description of the produced packages can be found in Bailey et. al. 1996).

## 3. ADDRESSING SOFTWARE ENGINEERING NEEDS

The project described above matches well the software engineering education needs as specified in L&A:

A. software development based on the software engineering principles,
B. software reuse,
C. group work,
D. increase of communication skills,
E. increase of the size of the project.

Let us address each one of these points separately.

### A. Software Engineering in Action

It is clear from the above description that only a simplified set of software engineering principles is being addressed. We must remember though that this is only the students' second semester of the freshman year. Students will take a separate advanced Software Engineering course during the first semester of their junior year. This will be the time when the full scope of software engineering principles will be introduced to them. At the same time, thanks to an early start, these students will be well prepared to take the advanced course(s) (examples of attempts at broadening the usage software engineering principles in the curriculum can be found in Aman (1996), Aman et. al. (1996) as well as above referenced papers by the authors). In addition, this gives a chance to extend the material that is to be covered in the advanced course.

### B. Software Reuse

After the first phase, students stop programming from scratch, as they introduce changes to the existing programming artifacts: the design, the code, and the user manual. This is an even more realistic model of what happens in real life than the small changes in relatively simple programming artifacts which were proposed in L&A. Here the developers are faced with a situation where the entire set of software related documents needs to be re-engineered. There are at least two additional possibilities that could make the project even more realistic and interesting. First, to use existing software artifacts (from the previous semester) as a starting point. This way the software reuse and analysis of existing artifacts can be utilized to its fullest. Second, it is possible that students from one group introduce changes to the products generated by another group. We have experimented with these possibilities during the Spring'96 semester (see below).

### C&D. Group Work and Communication Skills

Since the project proceeds as a group project, as such it matches the natural situation in industry. Students from one group take active part not only in developing their own software, but also act as analysts, verifiers, and users of the products of other groups. This allows them, first, to see the issues of software quality from all sides and, second, to evaluate their own product in view of what the other groups were proposing. Group work leads naturally to the expansion of communication skills. In addition, since the groups actively deal with the products prepared by other groups, they are involved in a group-to-group communication.

### E. Increasing the Size of the Project

Due to the introduction of this project, students have a chance, already in CS II, to be confronted with the management of a program which is between 2000 and 3000 lines long (see Bailey et. al. 1996). This experience is very unique when, during the final stages of the project, they are introducing changes/additions to the code of this size. Only then do they really begin to appreciate the important issues such as quality of the documentation, code modularity, the appropriateness and number of comments in the code.

It should be also pointed out that these requirements partially match some of the missing curricular objectives as identified by Jones (1995): maintenance and enhancement of aging software, software quality control, change management and configuration control, software reusability, software requirements and specification, user documentation. Even though Jones suggests that most of these topics should be covered by special (upper level) courses, it is also important to introduce these subjects in CS II so that they can provide a valuable background for these upper level courses.

We believe that a project of this type should be introduced at the CS II level, even if this means that some of the material typically covered in such a course will not be covered (we were able to cover all the material presented in Salmon's book in all cases, but some may consider it not sufficient). The fact that students actively develop and maintain a software system (relatively large in comparison to their former experience) is extremely important as far as meeting the needs of future employers is concerned. Also, the fact that students are introduced to group work relatively early will definitely be beneficial to them.

## 4. PRACTICAL OBSERVATIONS

### 4.1 Initial Observations

During the first two semesters of using the project in CS II a number of observations have been made. In both semesters a different number of software cycles have been reached. Also, since the students themselves decide how to proceed with the project, different types of modifications have been proposed. Typical changes include the following: introduction of a random or a goal-oriented movement (with a number of possible goals and conflict resolution strategies), introduction of two genders (leading to the modification of survival rules), introduction of food on the field (which requires the introduction of consumption and regeneration rules), introduction of the aging process. In each group a different resolution of interactions between various parameters has been observed. The initial code -- for the Project completed in Phase I – was up to 300 lines long (depending on the number of comments and the programming style). The final product was no less than 2500 lines of code and usually more than 3000 lines of working code complemented by a full design and a User's Manual.

The primary problems were related to group work (for additional information about usage of group work in the CS curriculum see for instance Fitzgerald et. al. 1992, Tennenberg 1995). It was observed that as long as group members work hard and as a unit, even if the group consists of relatively weak students, they are able to reach satisfactory results. However, any kind of group disfunctionality leads to serious problems with project completion, timeliness and quality. Typical problems are related to some of the group members not doing the required work on time, group members not being ready to cooperate with others (for various reasons), groups falling apart due to attrition.

### 4.2 Recent Observations

During the Spring'96 semester the project has been modified. Following the suggestions of L&A, the work from scratch component has been abandoned. Instead a pre-existing code has been modified. All the elements related to the Game of Life simulation have been removed and only the user interface has been left -- making this approximately 2000 lines of non-working code. After students have been divided into groups they have been confronted with this code and given a week to make it to compile, link and run. To be able to do this they needed to find all the missing procedures and create appropriate dummy modules. This situation was designed to match even more closely the real life situation when the programmer is confronted with a large software artifact without documentation and is expected to fix it. Students' reactions were mixed. At first they were shocked by a printout of the size they have not confronted before and the expectation to go through it and analyze, but after some encouragement they started working on it and were able to complete the task. Needless to say they became very proud of themselves after mastering the "monster."

From there the project proceeded similarly to what was described above with two major modifications. First, we have increased the amount of inter-group interaction. The additional interactions have been introduced in two forms. Groups were asked to take other groups' description of modifications and use it to modify their own code, as well as, take their own description of modifications and introduce it to other groups' code. Second, we have introduced an end-of-semester presentation. During this presentation each group introduced their final product, and discussed the most interesting results of the artificial life simulations they were able to generate. This step (and the fact that during the course a large number of written documents is created) is in line with the recent trend suggesting that writing and speaking are important parts of computer science education (see for instance Jackowitz et. al. 1990, Paprzycki et. al. 1995, Pesante 1990, Tureman et. al. 1992).

Overall, using iterative refinements technique students were able to produce the final working program, to reenginner the design and to prepare the user's manual. The experiences gathered did not differ from those during the earlier two semesters.

## 5. CONCLUDING REMARKS

We plan to continue with offering this project at UTPB. Most recent experiences confirm that we should use the existing code for the original Game of Life as a starting point. The next step will be to devote more attention to the group work as this seems to be the most important skill that the students are missing.

## ACKNOWLEDGEMENT

Work supported in part by a grant from ARPA, via USAF Phillips Laboratory, Solicitation No. F29601-94-K-0046

## REFERENCES

Aman, J.R. (1996). Making Connections: Software Engineering and the Compiler Course Project, Journal of Computing in Small Colleges, 11(4), 228-237

Aman, J.R. and M. Towhidnejad (1996). Applying Software Engineering Theory in Advanced Courses, Journal of Computing in Small Colleges, 11(4), 29-37

Andrianoff, S.K. and Hunkins, D.R. (1991). A Software Engineering Design Method in CS1 and CS2, Journal of Computing in Small Colleges, 7(2), 33-38

Bailey, D., Cheek, A. and Paprzycki, M. (1996). Developing an Artificial Life Simulation Package, Proceedings of the 12th Annual Conference on Applied Mathematics, Edmond, February, 1996, to appear.

Epp, E.C. (1993). Laying a Software Engineering Foundation in Early Courses, Journal of Computing in Small Colleges, 8(4), 122-126.

Fitzgerald S. and Caulfield, J. (1992). Cooperative Group Learning: A Viable Strategy for Computer-Based Laboratories?, Journal of Computing in Small Colleges, 7(5), 111-121.

Gardner, M. (1970). The Fantastic Combination of John Conway's new solitaire game "life," Scientific American, October 1970, 120-123.

Jackowitz, P.M., Plishka, R.M. and Sidbury, J.R. (1990). Teaching Writing and research Skills in the Computer Science Curriculum, ACM SIGCSE Bulletin, 1990, 212-215.

Jones, C. (1995). Gaps in Programming Education, Computer, 28(4), 70-71.

Lawlis, P.K. and Adams, K.A. (1995). Computing Curricula vs. Industry Needs: A Mismatch, Proc. 9th Annual ASEET Symposium, Morgantown, 5-19.

Paprzycki, M. and J. Zalewski (1994). Teaching Parallel Computing without a Separate Course, in: Nevison, C.H. (ed.), Proceedings of the Conference on Parallel Computing for Undergraduates, Colgate University, June, 1994, 18/1-19

Paprzycki M. and J. Zalewski (1995a). Should Computer Science Majors Know How to Write and Speak?, Journal of Computing in Small Colleges, 10(5), 142-151.

Paprzycki, M., R. Wasniowski and J. Zalewski (1995b). Parallel and Distributed Computing Education: a Software Engineering Approach, in: R.L. Ibrahim (ed.), Proceedings of the 8th CSEE'95 Conference, Springer-Verlag, Berlin, 187-204

Paprzycki, M. and Zalewski, J. (1996). Shaping the Focus of the Undergraduate CS Curriculum, ACM SIGCSE Bulletin (to appear).

Pesante, L.H. (1990). Integrating Writing into Computer Science Courses, ACM SIGCSE Bulletin, 205-209.

Salmon, W.I. (1994). Structures and Abstractions. An Introduction to Computer Science with Pascal, Irwin, Burr Ridge, Illinois.

Tennenberg, J.D. (1995). Using Cooperative Learning in the Undergraduate Computer Science Classroom, Journal of Computing in Small Colleges, 11(2), 38-49.

Tureman, R.L. and Camp, P.D. (1992). Infusion of Writing in the CIS Curriculum Using Student Communication Skills to Promote Critical Thinking, Journal of Computing in Small Colleges, 7(3), 129-132.