

Failproof Team Projects in Software Engineering Courses

A. T. Berztiss
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
alpha@cs.pitt.edu
and
SYSLAB, University of Stockholm

Abstract

The computer science department of the University of Pittsburgh offers two undergraduate and two graduate courses in software engineering in which we emphasize the importance of general engineering principles for software development. For the last ten years the undergraduate courses have been based on team projects. This structure has advantages: students see immediately the relevance of what they learn, and the team setting leads to a better understanding of what they learn. The projects in the two courses are of different types. In one course the result is the formal specification and design of a software system. In the other, the teams implement such a system, but emphasis is on testing rather than on the implementation itself. The success of each project is guaranteed by making it open-ended. A team establishes a list of priorities that is to ensure that a useful product will have been built by the time the term ends. We discuss the nature of team projects, and our evaluation scheme.

1. Introduction

Although there is still some discussion of whether the development of software is in fact engineering, we hold that it is. We first argued our case in [1], and then in [2]. Here we shall merely summarize the main argument, based on what engineers do. We find that they define requirements and make development plans; resolve conflicts in requirements and set up priorities; use standards; anticipate change and allow for modifications; provide for the unexpected; transfer theory into practice; scale up from models or prototypes; control quality; select tools; cooperate with other specialists; manage all of the above; and, when needed, engage in technological adventure. In [1, 2] we show that this is also what software developers do.

Applications cover many different domains. Therefore, despite the engineering orientation of software development, it may be best to continue to teach software engineering as part of a computer science curriculum in a BS program so that students get a broad education. However, students taking software engineering courses must come to an understanding of engineering principles, and, more importantly, of how these principles are to affect their professional work after they graduate. To this end, our software engineering courses have always contained a team project component in which we have tried to create as realistic a setting as possible within the limits set by the constraints under which we have to operate.

Our projects are designed to be failproof, in the sense that a team always produces a product. Sometimes the product does not fully meet initial expectations, but the outcome is still something that works. It has been suggested to us that projects that fail can teach important lessons. This may be so, but we consider that our approach teaches the most important lesson, which is that outright failure is nearly always avoidable by careful planning.

In Section 2 we discuss teamwork, and in Section 3 describe the project components of our two undergraduate software engineering courses. Section 4 deals with evaluation of the projects. Section 5 presents our conclusions. Throughout we take a rather broad interpretation of software – we define it as anything that can be represented in binary form and assists in the utilization of computers.

2. Teams and teamwork

The literature on team projects in software engineering is quite extensive, published for the most part in the proceedings of annual conferences on software engineering education, organized by the Software Engineering Institute of Carnegie Mellon University. We shall refer to just two articles here. The authors of [3] discuss failures of projects under the headings of management, and per-

sonal and technical problems, and emphasize the need for sound team management. In their view team effectiveness depends on how the team is built, the environment in which it operates, what motivation there is, the degree of empowerment of the team, its productivity, and how it is evaluated.

The authors of [4], who are at the Norwegian Institute of Technology (NTH), give a list of what needs to be considered for a project to be realistic: literature review, feasibility study, planning, requirements analysis, design, coding, quality assurance, configuration management, reuse aspects, operation and maintenance manuals, and installation and training. They conclude that no project course can deal with all these aspects, and discuss what makes project courses effective at NTH. An NTH project starts with an initial exploration of requirements with a real client and ends with the demonstration of a prototype. Stopping there avoids extensive testing, which saves much time. The goal of the project course is to expose students to problems that are only partially defined, to working as a team, and to management of the software development process by the team itself. The NTH teams consist of 6-8 students, and each student works on the project for 19 hours per week over 13 weeks. This translates into 1700 or so person hours.

Compared to this, we are much more restricted in what we can demand of our students. Our courses consist of 27-28 80-minute class periods. We have no separate project course, and very few of the class periods can be devoted to the project. The best we can ask of our students is that they put in 7-8 hours per week over, say, 9 weeks on a project. This period of 9 weeks cannot be extended because our university has a very liberal policy for adding and dropping courses. This means that the class roster does not stabilize until well into the term. Our team size is normally four – with larger teams too much time would go into establishing a communication structure. Hence the total time spent on a project is only about 270 person hours, which is less than a sixth of the time available at NTH. Our task, then, has been to institute a project component in a regular course in such a way that something useful is accomplished, while preventing failures, those listed in [3] and any others. Our goals are the same as at NTH – exposure to incompletely defined problems, understanding of the benefits of teamwork, and practice at self-management in a team. To these goals we add another: exposure to "industrial-strength" software verification and validation.

We have observed that software engineering projects tend to differ from projects in other engineering disciplines by putting much emphasis on actual construction, i.e., on coding. It is as if a civil engineer, after having designed a building, would get on a bulldozer to prepare the ground, would pour the foundations, and so forth. To

some extent implementation is justified with software projects. Whereas the civil engineer can in general guarantee the soundness of a building by a thorough analysis of a design and the use of appropriate safety factors, software soundness is mostly established by testing of the finished product, and it is not unusual for half the total system development cost to go into testing. But, in order to test the code, it must be there. Hence coding is needed, but it should not be emphasized.

The software engineering component of our undergraduate curriculum consists of two courses. The first (CS1530) is a general introduction to software engineering. The second (CS1631) specializes in specification and design. The project for the first course is the implementation of a software system from a specification, and extensive testing of this system. Testing establishes that the system is consistent with its specification. This is verification. The project for the second course is the specification and design of a system. This includes validation, which is the determination that the design corresponds to the requirements for the system.

The two courses appear to be back to front in that we put implementation before design. Indeed, for a long time the order CS1631-CS1530 was followed. However, earlier computer science courses leave some students with the misconception that software development is just coding and some very superficial testing. This attitude needs to be corrected, and we found it best to do so in stages. Thus, in CS1530 there is still some coding, but emphasis is on testing; in CS1631 there is no coding at all.

In the past ten years the author has taught the design course to 254 students, and the testing course to 206 students. The projects have included information systems for a car rental business, banking machines for the year 2000 (in 1993), a supermarket chain, student registration, an urban public transportation system, a cultural center, and an editorial office for a technical journal. All projects were completed, which was achieved by making them open-ended, and by carefully designing the material distributed to the teams.

Team membership is established randomly in the hope that when students of different ability make up a team, better students will help the weaker students not just with the project, but with their work in general. This hope has been justified. An exception to the rule is made when random selection puts too many students with limited knowledge of English in the same team. Teams decide on their internal mode of operation themselves – there may be an elected leader (rare), a de facto leader emerges without a formal selection, or the team works without a leader (most common). Students are told what problems can arise with teamwork, but this is not given much emphasis. Reported personality clashes within a team have arisen in three instances, but they were not

sufficiently serious to have a major effect on the work of the teams.

3. Software engineering projects

Our first software engineering course gives a general overview of software engineering, but emphasis is on testing. We have used three approaches in the initiation of a project in this course, and all three have worked equally well. For a comparatively small project, an ATM (Automatic Teller Machine) for the year 2000, students were given a fairly complete design of a conventional ATM, and additional requirements were defined by means of brainstorming. For a somewhat larger system, such as a single-location car rental business, requirements were presented as capsule summaries. In other instances the teams received a complete design expressed in a formal specification language. We shall now discuss the three approaches in detail.

The initial ATM design was adapted from [5]. Then we ran a class session in a brainstorming mode (on brainstorming see, e.g., [6]). In this session, besides the essential ATM services, 57 additional services or features were suggested, including voice activation, access to 900-code telephone numbers, gambling, handcuffing of a person caught using a stolen bank card, and the use of ATMs as voting machines in elections. A list of the 57 items was distributed in the next class, and students were to allocate 15 points, in any way they liked, to the items they liked best. The ten most popular items were selected for implementation. They included bill paying, location indication of next nearest ATM in service, airline ticketing and seat selection, and sale of travel checks.

For the car-rental project, students were given a list of capsule summaries of the tasks that define rental processes in general, such as reservation, cancellation, changed reservation, wait line, handover, return, lease-purchase, damage assessment, overdue item, loss, bumping, payment, inventory management, distributed sites, and transaction trends. Rentals extend to cars, formal wear, video tapes, library books, etc. The occupation of an airline seat or of a hotel room can also be regarded as a rental (of the seat or room). So can the long-term leasing of an apartment or a car. Not all of the tasks listed above relate to all rental situation, so some selection has to occur with specialization. We show here the capsule summary for reservations.

Reservation. A customer makes a reservation of a rental object for a length of time starting at an indicated date and/or time. The credit-worthiness of a customer may have to be established as part of this task. Variants of the basic pattern include (a) group reservations, (b) indication of just the desired starting point and no indication of the length of rental, (c) no indication of a starting point, as in the case of a library book that is currently

checked out to some other borrower, (d) confirmed reservation, (e) overbooking, in anticipation of a cancellation, (f) no prior reservation, with the arrival of a customer interpreted as a reservation, (g) in case of shortage of rental objects, a customer may be put into a wait line, and allowed to make a reservation if the shortage is no longer in effect, (h) for some applications, the rental site and the return site of a rental object may differ.

A specialization follows a fixed format, made up of five components. They define the mode of initiation of a task, steps that comprise the task, changes to the information base caused by the task, other tasks that may be initiated by this task, and (optionally) notes – they refer to special aspects of the task that do not fit in under the other headings. Students were given specializations of the reservation and cancellation tasks, and were to specialize the other tasks. It is easy to code directly from such specializations. This methodology is quite new; it was first published as [7].

When students were working from formal specifications, these had either been developed specifically for the course, or they had been written by students in a graduate course. An example of such a specification, for the editorial office of a technical journal, can be found in [2].

Each team decides itself what programming language it will use. In recent years C has been the favorite. The code that the teams produce has to be thoroughly tested. This is done in stages. Initially the code is subjected to technical reviews (see, e.g., [8]). In addition, the teams are required to execute their code with test data. Here there is a fair degree of flexibility. All projects have been so designed that coding and documentation of the code can be completed with ample time left for testing. Testing continues until time runs out. The aim is to achieve statement coverage, i.e., that every statement be executed at least once. Although we would wish to be able to apply the techniques of software reliability engineering [9], the projects are too small for this.

The main purpose of the design course is to build the students' self-confidence, and to give them further exposure to work in teams. While in the verification course the teams start with a detailed definition of the product they are to implement, in the design course they have to find their own way. This used to be our first software engineering course, in the belief that the natural order of design before implementation should be followed in the sequencing of the courses. However, our students were not ready for this much independence, and the order of the courses has now been reversed.

A design project can take different forms. One is to find a real client, and to design a system that satisfies the client's requirements. In a variant of this the course instructor acts the part of the client. A third variant is to

design a so-called product software system, i.e., a system that is to be marketed at large, and for which the developers themselves write the requirements. A fourth variant is to "copy" an existing system, which means that requirements are generated by observing the operation of the existing system. These requirements are then to lead to a design. Under our time constraints real clients are not a realistic option. Also, when there is a client, be the client real or be it the instructor, there is a tendency to regard a project as having failed if there is not enough time to allow in the design for all the requirements supplied by the client. A feeling of failure is very bad for morale. On the job, unless a company is incompetent, there will be successes to compensate for the occasional failures. No such compensation is possible when a student takes just a single project course. Hence projects must not be allowed to fail.

Both the remaining variants are failproof because the scope of a system is defined by the team itself. The product system approach has been used with systems for inventory and sales control by a supermarket chain, for the running of a multi-purpose cultural center, and the control of the vehicle fleet of an urban transportation system. The "copy" approach has been used in the design of a student registration system – we intend to use it in the design of an information system for a library.

The designs are expressed in the author's formal specification language SF (short for Sets-Functions; an introduction to SF can be found in [2]). In principle SF can be easily turned into a prototyping language. Although it has not been implemented as such, the ability to do so implies that the designs are detailed. Each team is required to carry out a technical review of the design it produces.

4. Evaluation of the project work

In CS1530 the team submits the code, a user manual, a maintenance manual, a test plan and the test cases used, a group log, and personal logs. In CS1631 the list of deliverables is the same except that there is no test plan and no test cases. Most submissions are impressive. The author is again and again pleasantly surprised by the high professional standard of the work and its presentation. Students are advised to make individual copies, and to take them to job interviews. There has been good feedback regarding this practice.

An implemented system definitely needs a user manual, but there can be benefits when a user manual is produced early, as part of a design project – in such cases the user manual serves as a requirements document. Also, simultaneous development of a system and its user manual is one way of introducing the practices of concurrent engineering into the software development process. Maintenance manuals explain the structure and pur-

pose of different software components and are to aid in the understanding of the software if it should be in need of modification at a later date. We encourage the liberal use of diagrams in maintenance manuals. Test cases should be preserved so that retesting the system after modification will incur little additional cost. Also, with regard to evaluation of a course project, the list of test cases shows how thoroughly the system has been tested, and this is used in grading.

The group log shows when and for how long the team has met, who attended, and what tasks were allocated to specific members of the team. When the team meets for technical reviews, the log shows what suggestions for improvement were made. Each member of a team keeps a personal log – it gives the dates and times spent on the tasks performed by this student, as well as details on exchange of e-mail with other members of the team.

In addition, each student submits a confidential evaluation of other team members. The form of the evaluation is not prescribed, but we suggest that a score of 0-10 is given to each team member in each of the following categories: good citizenship in project, overall effort, punctuality and preparation for team meetings, timeliness of deliverables, quality of deliverables, understanding of concepts. However, more important than such scores are the comments that accompany them.

These confidential peer evaluations have been found to be reliable and useful. In the early years the components worked on by an individual were identified by reference to the logs, and the effort put in by this individual was estimated by examination of the components. This demanded much time, and was found to be superfluous because the confidential evaluations give equivalent information. Indeed, peer evaluations are more reliable. Because students lack experience in the estimation of effort, they find it impossible to distribute the total project workload equitably. Consequently students often help each other out, particularly when the project deadline approaches. Without the peer evaluations it would be impossible to determine who did what.

Over the years the proportion of under-performers has been more or less constant, at about 10%. The under-performers have always been identified by others in a team. There has not been a single instance of having two under-performers in the same team. It may be that a team will tolerate one slacker, but that peer pressure builds up when there is danger of collapse of a team due to more than one team member not doing adequate work.

Grading in CS1530 is based on general quality of the work (10 points), presentation of the project deliverables (10 points), thoroughness of testing (30 points). In CS1631 it is based on quality of requirements gathering (15 points), quality of the design (15 points), presentation of the project deliverables (10 points), validation

(10 points). Project grades are basically team grades. Every member of a team is initially allocated the same score, based on the evaluation of the project as a whole, say 42 out of 50 points. In most cases this is the final project grade for all students. However, if somebody has not put in enough of an effort, points are taken away from this student, and these points are allocated to the other members of the team in such a way that the aggregate score for the team does not change. Project grades are not released to students to avoid compromising the confidentiality of the peer evaluations.

5. Conclusions

We have three main objectives for the project courses: experience of teamwork, exposure to verification and validation techniques, and the realization that design is more important than coding. We believe that we have fully succeeded with the first two objectives, but not with the third. Students still do not fully realize that conversion of a thoroughly reviewed detailed design into code is trivial compared to the design process. We now believe that it was a mistake to put the design course first. Under this approach students were learning quite well to work as a team and make their own design decisions, but they were not happy doing so. One reason for this was that they were denied the satisfaction of seeing their system actually running. The break with past experience was too sharp. We hope that the new ordering of the courses will improve matters.

One major problem the teams faced was the coordination of meeting schedules. However, this problem is being resolved by increased electronic communication. Teams are beginning to set up web pages, and effecting communication via the web.

Another problem, which relates to educational philosophy, is that the projects are not realistic enough. For example, most of the budget of an information systems department is spent on the maintenance of existing software. Also, development costs can be greatly reduced by reuse, i.e., by incorporating existing software components into new products. In at least some of the offerings of CS1631 we will try to make the project different: a team will be given an existing software system, and will be required to modify it to fit new requirements.

It is interesting to consider how relevant our approaches are for the established engineering disciplines. Testing in the way software is tested has no counterpart in these branches – having a machine bang a car door 10,000 times is very different from software testing. So we should consider our design course alone. It may seem that only software design projects can easily be made open-ended, but this is not so. The ATM we discussed in Section 3 consists of both hardware and software. We looked at just the software functions, and each software function

could be implemented or not implemented. But quite a few of the 57 features suggested in the brainstorming session were hardware features, or at least they required close interaction between software and a sensor or a mechanism. These features could also be built or not built so that open-endedness is not restricted to software. Indeed, all our suggestions for the design course can be applied in other fields. We can go further. Codesign of host devices and the software embedded in them is becoming an important practical concern. This may be the time to start considering joint design courses with teams made up of students from software engineering, mechanical engineering, and electrical and electronic engineering.

6. References

- [1] Berztiss, A., "Engineering principles and software engineering," *Software Engineering Education* (LNCS No. 640), Springer-Verlag, 1992, pp. 437-451.
- [2] Berztiss, A., *Software Methods for Business Reengineering*, Springer-Verlag, 1996.
- [3] Kantipudi, M., Collier, K., Collofello, J., and Madeiros, S., "Software engineering course projects: failures and recommendations," *Software Engineering Education* (LNCS No. 640), Springer-Verlag, 1992, pp. 324-338.
- [4] Andersen, R., Conradi, R., Krogstie, J., Sindre, G., and Solvberg, A., "Project courses at NTH: 20 years of experience," *Software Engineering Education* (LNCS No. 750), Springer-Verlag, 1994, pp. 177-188.
- [5] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W., *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
- [6] Gause, D., and Weinberg, G., *Exploring Requirements: Quality Before Design*, Dorset House, 1989.
- [7] Berztiss, A., "Domains, patterns, reuse, and the software process," *Domain Knowledge for Interactive System Design*, Chapman & Hall, 1996, pp. 78-89.
- [8] Freedman D., and Weinberg, G., *Handbook of Walkthroughs, Inspections, and Technical Reviews, 3rd Ed.*, Little, Brown, 1982 [Reprinted: Dorset House, 1990].
- [9] Lyu, M. (Ed.), *Handbook of Software Reliability Engineering*, McGraw-Hill, 1996.