Software-Oriented Exercises for Learning Teamwork

Debora Weber-Wulff
Technische Fachhochschule Berlin
Fachbereich Informatik
Luxemburger Str. 10
D-13353 Berlin, Germany
weberwu@tfh-berlin.de

Abstract

In this paper, a selection of software-oriented exercises is presented. They can be used in college-level programming language and software engineering instruction in order to teach communication skills necessary for learning to work as part of a team.

1 Introduction

One of the most difficult skills for software engineering students to learn is how to work together with other individuals as a team to create a software system or to solve a problem. People who are interested in computers tend to be individualists who believe themselves to be fully capable of solving any problem on their own. They often do not like to accept the work of others as being correct, and also have difficulties explaining their own results to others. However, to implement the complex systems needed in industry today, many people must work together as a team in order for the systems to be completed correctly and in a reasonable amount of time.

Our students do not learn teamwork in the schools – there, more often than not, only the results of individual effort are rewarded. How can we teach working together in a team as part of of software engineering education? In this article I will present some of the exercises I use at the Technische Fachhochschule Berlin to help students learn teamwork skills.

2 What is a Team?

A team is a group of two or more persons who work together to achieve a common goal. The group must communicate effectively in order to do this. They must be able to communicate in writing as well as verbally and non-verbally.

The capability for exchanging information is often not well developed – people are accustomed to receiving information through the media, but are not well-versed in organizing their thoughts for presentation. So curricula for computer science must not just be concerned with listing the technical contents to be taught, but must also offer the students the opportunity to practice communicating through writing, giving presentations, learning discussion skills, making compromises and learning to work together.

Luckily, we do not need to invent new methods for teaching such skills – many ideas can be found in didactical works on how to encourage group formation or to teach effective communication. I have adapted a number of these ideas for use in teaching students how to construct software.

3 First Programming Course

I begin training my students for teamwork during the two-semester programming language instruction that precedes our software engineering project course. They work by themselves in the first semester only, but are already confronted with the work of others by having to read and test software components constructed by other students.

The following is the exercise framework I use in the first programming course (our first language is Ada):

First exercise: In order to learn to use the rather quirky editor in our environment, the first exercise makes the students write a one page specification in German using complete sentences for a simple program (computing a table of function values for a complex function, computing a currency conversion table, or taking statistics on a text - longest line, shortest line, number of 'E's, etc.[1]).

Second and Third exercise: A basic *Hello world* program in order to learn the use of the compiler, and then the design and implementation of a simple program using a <u>loop</u> and an <u>if</u>. For the design the students may use structograms, pseudo-code or flow charts[2].

[1] Since our student union collects exercises and examinations with answers for all professors, one must make up a new application for each semester.

[2] Yes, I know, these are felt to be are outdated. But since

Fourth exercise: Design a program and develop a test plan for the specification from Exercise 1.

I choose the best specification as the basis for all students in this exercise so that they all have the same starting point. This exercise frustrates many of the students who can already program a little, as they are anxious to get to programming "real" systems and don't want to waste time on this trivial stuff. But this is an extremely important exercise, as it builds the foundation for all that is to come: they must learn to put their expectations in writing, so that they can be checked by others.

Fifth exercise: Implement the design of the person sitting to your right. This will cause uproar: "I can't read their writing", "It's been designed wrong", "This is too fancy". To give the anger an outlet they first have to write a half page to a page praising points in the design (and they must find something!) and then discussing the problem areas. When these are handed in I just check them off and hand them to the design authors to read, who are often shocked that their fantastic design was not met with the appropriate enthusiasm. The teacher must be prepared to approve design changes for both ends of the spectrum: the whiz-kid who wanted a menu-driven system just for printing a simple currency exchange table, to the foreign student who didn't understand the exercise or who did the design using Indonesian variable names.

Sixth exercise: I collect the previous exercises at the begin of the hour, give everybody a perfect score, mix up the implementations and hand them out to be tested. Each person is to do a white-box test of the pile of paper and the diskette given to them. A test report pinpointing errors is to be handed in. I take points off both for errors not found and for wrongly reporting an error where there is none.

Correcting these exercises is a nightmare, it takes between 30-45 minutes per exercise. But the results are tremendous. The students have learned the importance of testing, and they have now been exposed to a design and an implementation from another person, and will now program as if someone might have to read what they have written. Also, each student now knows the name of at least two other students in the group, they can now often be observed sitting together in the cafeteria and talking together outside class – they are beginning to regard their classmates as fellow humans and not as competitors, an important basis for being able to work well together with others in a team.

Seventh exercise: Specify, design and implement a package for an abstract data type *date*. Include functions to set a date, to read and write a date, to find the next date, to determine the week day of a date, and three other interesting functions of their own choice such as adding a number of days to a date or subtracting two dates to find the number of days between them. The date package must be valid from the date change in 1583. Since a package in Ada is not directly executable, a test plan must be developed and a test harness written to ensure that the implementation is correct.

Calendars are an important cultural artifact and spark interest in the students, leap year calculation and differing month sizes are of just the right complexity for the current student knowledge level. They begin to work together in searching for weekday conversion algorithms. One student, wanting to implement a function to calculate Easter and not knowing about algorithm collections, called his priest and then the local observatory before finding the algorithm!

Eighth exercise: Do a black-box test of another student's implementation using your test harness. This exercise demonstrates the usefulness of packages and the sanctity of interfaces – if an implementation has renamed the required functions, the test harness won't work.

Further exercises: The rest of the exercises in the semester are "traditional" and done working alone, although when they ask me if the exercise is going to be tested by someone else I smile mysteriously and say "Maybe!", to keep them on their toes.

4 Second Programming Course

In the second semester, working in pairs is introduced. The group – at our college it will more or less be the same group of people, as they go through the program as a class, although a few failures from previous courses will join in – is now accustomed to having looked at the work of others. They must now learn how to divide up and integrate the work between two people. The exercises given in this semester are either too large to be comfortably solved by just one student, or there is the need for a test driver to properly test the solution, which provides an obvious division of labor.

In addition, the pair will have to defend their work orally two or three times during the semester. This

there is a German DIN Standard for them, many companies still use them, so we at least expose our students to them.

gives me an opportunity to see if each understands the work that the other has done. It also makes the students speak about their programs as they explain it to me. In addition they suddenly see why I am so nasty and take points off for programs that are not properly commented: even a week after handing in an exercise, they do not remember why they chose the data representation they did or what on earth the tricky statement sequence really accomplishes!

A "Test Rally" in which the pairs compete against the other groups using their own test drivers to try and find errors in the solutions of other groups has been very successful. One of the first exercises in the second semester involves implementing an abstract data type for variable length strings (VSTRINGS exercise from [1]). The specification, including the signatures for all the operations in the ADT and a specific implementation data structure, is given. Each group must make a test harness to test their package, which consists of many functions such as substring slicing, searching in a string, concatenation, and such.

The test rally, which is only 90 minutes long, involves first obtaining copies of all the other implementations in the group, as well as my own which contains many typical and a number of very subtle errors. Each implementation is linked to the test harness and a test report is made. Scoring is based on the following scheme:

Submission of test report	+ 10	points
Each fault found by others in own package	- 1	point
Each fault found in another group's package (where n is the number of groups to find this error)	+ $1/n$	points

This normalizes the points – an error found causes a point to be taken from the group that made the error and divided up between all the groups that found the error. I set a maximum of 20 points and a minimum of 1, although in actual use over a number of semesters the maximum has been 18 and the minimum 4. The students quite enjoy the competition this exercise offers, and tend to have a concept of themselves as a team after this point.

The teams also get so used to exchanging work, that it has happened that they test each other's packages, even if that is not part of an exercise. I observed this in the past semester during our first recursion exercise. They were to use recursion to find a path through a labyrinth. They did not really believe me that the algorithm would solve any labyrinth that has a path from the beginning to the end, and spent extra hours in the lab designing intricate mazes for each other's programs, and discussed amongst themselves quite energetically how this recursion "magic" really worked.

The maze exercise, which was posed just after the test rally, was a turning point in the semester. For the rest of the semester the students tended to work more together, discussing how to solve the exercises, with the better students helping the weaker ones. They are now in a good position to work cooperatively as team members in the coming semesters.

5 Software Engineering Project

In the third and fourth semesters a team consisting of four students forms to work together on a project that will span both semesters. Often two "pairs" from the previous semester will form a team for this project work, but even if a team is made up of people that do not know one another, a number of exercises can help them learn to work together.

5.1 A Group is Formed

The "House Building" exercise adapted from [2] will often predict how well a group can work together, so that the teacher can see where to concentrate effort on encouraging team structures. This exercise can easily be done together with an round of introductions at a first 90-minute course meeting. Each team introduces itself to the rest of the teams in a section (we have 4 teams per section) by having team member 1 introduce member 2, 2 introduce 3, 3 introduce 4 and 4 introduce 1. They first interview each other for about 5 minutes, and then tell the entire group who the person is, where they come from, what they did before beginning to study, and why they decided to study computer science.

Then the teacher hands out a construction set to each group: 1 large sheet of paper (old conference announcement posters are great), 1 pair of scissors, 1 pencil, and 1 ruler. The teacher has a pencil sharpener for emergencies, but there is to be no tape, glue, staplers or other materials available. The teacher announces the exercise:

> "You are to build a house in the next 40 minutes. Your house must have a roof, a door, two windows, and at least three walls. You may not use any other tools than the ones I've given you. There will be no time-limit extensions, after 40 minutes you must present your house to the group."

Do not give any hints or help – each team must decide on whether to build a flat-roofed house or a pointed one; they must decide how to fasten the house parts; they have to agree on what kind of a house to build; and they have to share the tools to get done in time – one person working alone can't get it done in time. The teacher can also apply extra pressure by announcing how much time is left every five minutes.

Observe how the groups interact! Do they brainstorm? How do they decide what to do? Is there a leader who tells the others what to do? How do they divide up the work? What happens when something goes wrong? I had one group put their roof on the house in the last few seconds, whereupon the house collapsed under the weight of the roof. The finger pointing began immediately: "I told you so!" "You made that wrong!". It took some time to get the group to discuss what had happened and to realize that it didn't matter whose fault it was – they had failed as a team, and all are responsible for the work of the other team members.

Not only does this exercise give the teacher insight into how the persons interact in each group, this shared experience will tend to pull them together – they will discuss what happened amongst themselves and their colleagues, and this should start the concept of "us" as opposed to the others in their minds. And there are many good analogies to be found between the process of building a house and building software – especially in the area of what goes wrong during the process.

5.2 Role-Playing in the Project

I have successfully used an adaption of Tomayko's Roles in Software Engineering Projects [3] a number of times to encourage team members to take responsibility for specific parts of the project. Each team member selects two roles from the responsibility list. Assuming a role does not mean that the person must actually do the work, only that they are responsible for it. This means that their individual grade will suffer or profit doubly from the grade given for each document.

The roles serve to reinforce the desired behavior, that is, working together towards one goal. If student A refuses to help student B work up the documentation and it is given a bad grade, then B will not be inclined to help A do the tests. But if each team member cooperates with the others, then all will profit.

Since we try and keep the teams at four persons, the following eight roles are assigned:

- The *main architect* is responsible for the entire product and specifically for the requirements document and for the analysis documents.
- The *project manager* is responsible for the division of the tasks among the team members and is responsible for collecting the time sheets from the members and issuing weekly cost reports.
- The *configuration manager* is responsible for developing the configuration plan and for controlling changes in the latter stages of the project. This person prepares the product for release (i.e. presentation to the entire group).
- The *quality control engineer* is responsible for the quality of the end product, must develop a quality control plan and a test plan.
- The *design engineer* is responsible for the design documents.
- The *implementation engineer* develops the team programming style sheet and is responsible for the code production from the design documents.
- The *documentation specialist* is responsible for the user handbooks and the program documentation. The responsibilities include spelling and grammar checks in all documents, as well as enforcing layout consistency.
- The *verification and validation engineer* is responsible for conducting the module and system tests and for making sure that the requirements specification has been completely implemented.

Many students enthusiastically assume their roles, even going so far as developing team change control forms, or a "team identity" – including having T-shirts printed with the team logo –, or even the configuration manager distributing the official source code daily – if you didn't register your changes with him, they're gone. It also gives the team members a bit of security to be perfectly clear about the deliverables for which they are responsible. And there are some roles that are less programming-oriented, for example, the documentation specialist, and often students will be able to be a productive and respected member of the team, even if they are not strong programmers.

Most of the groups successfully complete the course – the ones that don't tend to self-destruct along the role lines with a lot of finger pointing ("That was your responsibility!"). The roles can be used to help pass team members that are seen to be working diligently, even if there is no product produced. For example, I was able to justify passing a team member that put together an excellent quality assurance and test plan, even though the team essentially produced extremely bad work and did not deliver a functioning product.

Usually, however, all members of the team receive the same grade for each part of the process. The only differentiation in the project grade is through the responsibilities. An oral exam at the end of the fourth semester gives the students a chance to practice such an examination, which they will encounter at the close of their studies[3], and the teacher a chance to differentiate the grading more strongly.

6 Conclusions

The exercises presented here have been used a number of times in our polytechnic college, and I feel that their use has been successful in teaching many of the students how to work in a team. It is, however, impossible to measure a property such as "able to work well in a team", so there cannot be before and after measurements that would enable us to state that the exercises did in fact make a difference. I can only offer my observations, which are also based on discussions with students and on the anonymous evaluation forms that they fill out every semester, that the use of the exercises has been beneficial in promoting teamwork skills.

Acknowledgements

Some of the exercises are adapted from general didactical works, others are own creations that have been used a number of times by myself and by colleagues. I am grateful for all the ideas and feedback I have received, especially from my students.

References

[1] Michael B. Feldman and Elliot B. Koffman. *Ada : Problem Solving and Program Design.* Addison-Wesley, Reading, MA, 1993.

[2] Claus-Jürgen Höper, Ulrike Kurzleb, Alke Stobbe, and Bertram Weber. *Die spielende Gruppe. 115 Vorschläge für soziales Lernen in Gruppen.* Jugenddienst-Verlag, Wuppertal, 1974.

[3] J. Tomayko. Teaching a project-intensive introduction to software engineering. Technical Report CMU/SEI-87-TR-20, AD-A200603, SEI, Carnegie-Mellon University, 1987.

[3] In Germany, most courses of study have oral examinations which take place after the written thesis has been graded. In our college this is a 30-minute oral exam, in which questions from the entire course of study may be asked.