

Student projects  
SOFTWARE ENGINEERING: Principles and Practice  
Second Edition

Hans van Vliet  
Division of Mathematics and Computer Science  
Vrije Universiteit  
De Boelelaan 1081a, 1081 HV Amsterdam  
Email: hans@cs.vu.nl

October 26, 2000

## Introduction

An often-cited problem in software education is that many of the relevant topics are difficult to address in a university setting. Though project work is a useful supplement to the lectures, it is very difficult to make the project truly realistic. In our environment, we therefore do not try to mimic all aspects of reality. Rather, we try to concentrate on a few aspects which can be realistically dealt with in a university environment. Section 1 describes our experience with a project which concentrates on maintenance activities. Independent evaluations by both students and project instructors reveal that these maintenance activities were most seriously hampered by inappropriate documentation at the component level. Section 1 is based on [van Vliet, 1989].

Section 2 gives some additional possible foci for software engineering student projects, as well as additional examples of lab projects.

## 1 A Software Maintenance Project

The project that is being dealt with in this section is a companion to a second-year undergraduate course in software engineering. Before attending the software engineering course, students have followed an introductory computer science course, a programming course (using Modula-2 or a similar high-level programming language), a course on data structures (emphasizing abstract data types; see e.g. [Weiss, 1998]), and a course on computer organization. All in all, it is fair to say that most students of this software engineering course have not been exposed yet to very advanced topics in CS.

A recurring problem in software engineering education is phrased by Brooks: "you cannot teach it, you can only preach it" [Brooks, 1995]. Lacking a solid theory, the instructor is left with good advice. It is easy to say that discipline is important, or that proper documentation is a critical success factor. Students will readily believe so. They will even think they practice those principles. Practicing good discipline or making proper documentation is hard to realize, though.

Courses in software engineering are therefore usually supplemented by some sort of project work. These projects are intended to give students a feeling of how software is really being developed. Several authors have stressed the importance of this project work, and advocate to bring in projects from outside the university itself.

Such is not feasible in many a university setting. Nor is the level of expertise of second-year undergraduate students such that realistic problems can be tackled successfully. We therefore took another approach, one which does not try to mimic all aspects of reality. Rather, we try to expose students sufficiently to some of the important aspects of real software development projects. The central objectives of the project were to expose students to:

- problems incurred by working in a team;
- problems incurred by imposing strict deadlines for various milestones;
- maintenance problems.

The degree in which the first two types of problems can be addressed realistically is seriously hampered by the university environment. The project can only give students a feeling of the importance of these aspects.

We expected to be able to realistically address maintenance problems, though. In one of the first lectures, we stressed the importance of having a clear and precise requirements specification. Through a careful analysis, one may hope to build a sound perspective of user requirements and anticipate future changes. It was also noted that, no matter how much effort is spent in a dialogue with the prospective users, future changes remain hard to foresee. In this respect, specifying requirements has much in common with weather-forecasting: there is a limit as to how far the future can be predicted.

In the project discussed below, students were first asked to develop a certain application. Next, they had to maintain a version of that same application, developed by another group of students. Section 1.1 gives a short description of this project.

Both project instructors and students gave an evaluation of the software that had to be maintained in the course of this project. Section 1.2 discusses these evaluations. The difference in rankings given by project instructors and students yields valuable insight into some of the major causes of maintenance problems. It turned out that the quality of the documentation at the component level was particularly troublesome, much more so than the (technical) quality of the design and code.

## 1.1 The project

The course consisted of 12 weekly lectures, of 2 hours each. The project started in week 1 and ended by the end of week 12. Students were grouped into teams of 4 during week 1. Students were allowed to choose their own partners. Assistance with respect to this team organization was only given on request. During week 1 too, a one-page problem description was handed out. A summary of this problem description is given in figure 1. Each team was supposed to finish this project within 6 weeks. The following milestones were identified:

1. At the end of week 2, the requirements specification had to be completed and agreed upon by the project instructor;
2. At the end of week 4, the design had to be completed and agreed upon;

3. At the end of week 6, the program and documentation had to be completed. (A number of test files was provided by the instructor.)

---

You are to develop an interactive program which allows the user to move a cursor around in a maze. The maze to be used is specified on a file. {Follows a description of the input format.} When the program is started, part of the maze is shown on the screen. The part initially shown contains the entrance. The maze only contains horizontal and vertical corridors. Note that the maze need not entirely fit on the screen.

When the program is started, the cursor is positioned at the entrance. The cursor can be moved in four directions (left, right, up, down). The program stops if an exit is reached.

---

Figure 1: Problem description 1

For both the requirements specification and the design documentation, students had to adhere to a strict scheme. We follow IEEE Standards 830 and 1016, respectively [IEEE830, 1993], [IEEE1016, 1987]. Students were supposed to write their documentation in SGML-style. SGML is an ISO-standard for marking up structured documents [Bradley, 1997]. In SGML, one writes a context-free grammar for some specific document type. Subsequently, documents of that type are marked up according to that grammar, and a parser may then check these documents for consistency with the corresponding grammar. We provided the students with templates for both the requirements specification and design documentation, and parsers for those document types. These parsers were generated, using an SGML parser-generator, from simple grammars derived from the IEEE-standards. Moreover, simple backends to produce formatted text were supplied.

In this way, students were forced to pay attention to a number of aspects in the documentation. They had no need to bother about layout problems; every aspect hereof was taken care of by the tools provided. To the project instructors, this way of doing had the added advantage that all documents had the same appearance and the same order of treating things.

Students were allowed (and even urged) to have weekly meetings with their project instructor. They could reserve a slot in the instructor's agenda to discuss problems or the documentation delivered. For each milestone, the corresponding documentation had to be approved of by the instructor before the next phase could start. (We could not really enforce those rules. For instance, we have the suspicion that some teams wrote the program first, and made the design afterwards.)

During the second half of the project, each team had to maintain a program written by some other group. To this end, the results of the first stage were, largely at random, divided amongst the groups again. The teams were given some freedom with respect to the exact maintenance activities to be performed. For the types of maintenance activities allowed, see figure 2. The schedule for week 7-12 was the same as that for week 1-6.

## 1.2 Evaluation

At the beginning of week 7, each group was given both the source files and all documentation of the program they had to maintain. In week 11, they were given an evaluation sheet that contained a number of questions addressing various quality aspects of the documentation and program received. These evaluation sheets were, independently, also filled in by the

---

You are to maintain the maze program handed out to your group. You have to incorporate at least three changes, one out of each of the categories given below.

Category 1:

- Commands to move the cursor should be extended such that one may move ‘nr’ positions in any direction. It should also be possible to indicate that a corridor should be traversed up to its end.

Category 2:

- The maze should be allowed to contain corridors with an angle of 45 degrees to the horizontal axis.
- The screen should only show the part of the maze that has already been traversed.
- Each corridor has a certain width. The cursor has a certain width too. Thus, it may not be possible to enter some corridor.
- One may allocate penalties with certain locations within the maze. Each play starts with the player having a predefined number of points. Each time a location with a non-zero penalty is visited, that penalty is subtracted from the running number of points. If the number of points reaches zero, the play ends and the player has lost.

Category 3:

- The screen shows two windows, each with its own cursor. Both windows show a, possibly different, part of the same maze. Thus, two persons may play in one and the same maze via their own window.
- One or more mazes are placed above each other. The mazes are connected by one or more stairs.

---

Figure 2: Problem description 2

instructors that assisted during the first six-week period. The rankings of both the instructors and students are summarized in figures 3 and 4 (some 140 students took the course on which these data are based). The data in the average-column are computed by counting the hopeless, bad, reasonable, good and excellent markings as  $-2, -1, 0, +1, +2$ , respectively.

The discussion below concentrates on the difference in rankings between instructors and students, and possible explanations therefore. These differences give us some valuable clues.

A first observation seems to reveal that, on the average, project instructors gave higher rankings than students. To see whether this pattern is consistent, we have to take a closer look at the individual rankings. We therefore counted, for each question, the number of times the instructors gave higher rankings than the students for the same program and documentation, and vice versa. These figures are given in figure 5.

The data in figure 5 show that the trend is not all that consistent. For some questions, notably 2a, 3a, 3b, 3c, 4a, 5a, 7, the pattern is rather chaotic. For others, it is much more consistent. The results of applying the two-sided sign test with null hypothesis ( $H_0$ ) that instructors and students give equal rankings, is given in the last column of figure 5.

question	answer					average
	hopeless	bad	reasonable	good	excellent	
Requirements specification						
1. Completeness reqs spec		3	12	17	1	.48
Design						
2. Modularization:						
a. Cohesion of components		3	17	11	3	.41
b. Coupling between components		5	17	9	3	.29
c. Information hiding		2	11	18	3	.65
3. Generality/adaptability of modules						
a. More than 1 maze possible		9	3	20	2	.44
b. More than 1 window possible		14	8	11	1	.03
c. More than 1 cursor possible	1	13	11	8	1	.15
4. Design documentation:						
a. Global structure	1	2	14	13	3	.45
b. Individual components			12	17	4	.76
Program						
5. Modularization:						
a. Cohesion of components		4	17	10	3	.35
b. Coupling between components		6	19	8	1	.12
c. Information hiding	1	2	17	11	3	.38
6. Individual components:						
a. Neat, understandable code	1	1	12	17	2	.55
b. Documentation/comment	1	5	10	12	6	.50
7. Result of testing		7	12	14	1	.26

Figure 3: Assessment by project instructors (a few answers are missing, so not all rows add up to the same number)

Questions 3a–c asked for an assessment of the maintainability of the program received. These questions address the changes mentioned in Category 3 of Problem description 2 (these changes were considered to be more difficult to realize than those in Categories 1 and 2). Both instructors and students have similar opinions about the effort needed to accommodate these changes. Somewhat to our surprise, the actual changes decided upon by the students do not agree with their assessment. A large majority (24 teams) decided to implement multiple windows, though they generally thought this change more difficult to realize.

A possible explanation is that they very well knew the difficulties incurred by the change actually realized, while they only had a global opinion about the change not realized. Thus, they may have overseen intricacies that would have cropped up had they decided to actually incorporate that change. Though this is in line with some other observations discussed below, it is somewhat unsatisfactory, since the project instructors gave a similar assessment of the difficulties involved in these changes.

A further analysis of free-form evaluations written by the students reveals that the actual effort required to augment the program differs widely. For some groups, the changes were easy

question	answer					average
	hopeless	bad	reasonable	good	excellent	
Requirements specification						
1. Completeness reqs spec	1	8	16	9		.03
Design						
2. Modularization:						
a. Cohesion of components		1	15	18		.50
b. Coupling between components		12	17	5		.21
c. Information hiding	1	8	15	10		.0
3. Generality/adaptability of modules						
a. More than 1 maze possible	1	7	6	17	3	.41
b. More than 1 window possible	2	12	9	11		.15
c. More than 1 cursor possible	4	7	7	15	1	.06
4. Design documentation:						
a. Global structure	4	5	9	15	1	.12
b. Individual components	1	8	18	7		.08
Program						
5. Modularization:						
a. Cohesion of components		4	17	13		.26
b. Coupling between components		14	16	4		.29
c. Information hiding	3	8	11	12		.06
6. Individual components:						
a. Neat, understandable code	1	10	13	9	1	.03
b. Documentation/comment	6	14	6	6	2	.47
7. Result of testing		9	11	14		.15

Figure 4: Assessment by students

to accomplish, for others it turned out to be a major undertaking. Fairly often also, some of the changes were easy, while others required a substantial re-design and/or re-implementation of part of the program.

Before the course started, a pilot implementation of the maze program was written by one of the instructors. This person was not involved in the planning of the changes contained in Problem description 2. It turned out that some changes were easy to realize in this pilot implementation, while others were very hard to realize.

These observations confirm our earlier statements about the unpredictability of change requests. The project carried out was a success in this respect. Project instructors could only assess project results fairly globally. Each instructor had to supervise 8 to 9 teams. Since the schedule was fairly tight, they often had less than one day to evaluate the next set of milestone reports. Conversely, the students had to really work with the end product of some other group. They got hit by all the details that were missing, simply wrong, or ill-documented.

Such is reflected in the difference in rankings. A further analysis of free-form evaluations written by students confirms this opinion. Quite a few of them contain phrases of the form "at

question	instructor>students	instructor<students	$H_0$
1	18	4	0.004
2a	9	13	0.52
2b	17	6	0.035
2c	17	3	0.003
3a	8	8	1.0
3b	8	9	1.0
3c	10	13	0.68
4a	12	8	0.50
4b	21	2	0.0000
5a	13	11	0.84
5b	14	4	0.031
5c	16	7	0.093
6a	19	6	0.015
6b	23	4	0.0004
7	14	8	0.29

Figure 5: Difference in rankings between instructors and students

first sight, the requirements specification/design/program looked OK, but on closer inspection we encountered the following problems: ... ”.

The most prominent difference in assessment concerns questions 4b and 6b. Students have a rather more negative opinion about the quality of the documentation at the component level. (Note that the global structure of the design documentation was fixed in advance, and adherence to it was taken care of by the tools provided.) Analysis of the free-form evaluations gives us some insights into possible reasons for this negative assessment. The comments given can be roughly categorized into two classes:

1. Documentation is wrong or inconsistent;
2. Documentation is incomplete, or not clear.

Documentation is wrong or inconsistent if a description at one level does not agree with the corresponding description at another level. Most often, the requirements specification or design documentation tells one story, while the program tells another. Examples vary from different names for one and the same variable (like InputMaze versus ReadMaze) to cases where the program has been changed and the corresponding documentation has not been updated.

It is difficult to separate this type of problem from the one where documentation is incomplete or not clear. In most cases, people seem to have really done their best to provide clear documentation, but it just does not seem to be adequate. Some programs, for instance, contain abundant documentation where it is not needed, while important facts remain hidden in the code. A particular class of such problems is exemplified by the following reactions:

- ”Quite unexpected to us, the Main module checks for Signals”
- ”It is strange that this module updates Position, rather than TerminalCursor”

- "The key-function NewScreen was not documented, while it turned out to be very important to know that it calls ChangeMaze"
- "The close cooperation between NewScreen and Maze was not documented"
- "The postcondition of routine X does not specify that the routine checks for negative values. This is important to know, since it is explicitly used by some other routine"
- "In other parts of the program too, the fact that connections have length 1, was used"

From these comments, we conjecture that the negative opinion of students is partly caused by the occurrence of 'delocalized plans' (see [Soloway *et al.*, 1988], and/or chapter 14 of the textbook). According to Soloway, "Basically, the idea is that experts have and use knowledge structures that are a schematic representation for stereotypic behavior patterns [ . . . ] Correspondingly, plans in programming are stereotypic action structures [ . . . ] In a delocalized plan, pieces of code that are conceptually related are physically located in non-contiguous parts of a program." The study by Soloway et al. indicates that the occurrence of delocalized plans is one particular source of maintenance problems.

Our students are no experts. They do not have a vast body of stereotypic program plans. Their conception of what the various components do is largely inferred from the documentation they get. One could argue that their ideas are based on them having implemented the same problem before. Their evaluations, however, do not indicate serious problems in mismatch between those preconceived ideas and the solution they had to work on.

Thus, the comments given by our students reveal that hidden, i.e., undocumented, relations between non-contiguous parts of a program are insidious. Questions which indirectly relate to the same phenomenon, such as those concerning the coupling between components (2b, 5b) and information hiding (2c, 5c) have relatively low scores as well, certainly when compared with those given by the project instructors.

In setting up this project, we anticipated problems with the documentation. The results show that the maintenance activities asked for were indeed seriously hampered by the inadequacy of the documentation. The analysis further revealed some particularly insidious problem areas.

### 1.3 Conclusion

The project described here aimed at exposing students to some important aspects of real-life software development projects, and software maintenance in particular. The project showed students that even a careful requirements analysis and design phase is no guarantee that maintenance will be easy.

An analysis of evaluations by both project instructors and students shows that, in particular, the importance of documentation presented itself in this project. When the students were asked to maintain a program written by others, they almost invariably found problems with the documentation at the component level. In particular, inconsistencies in the documentation at different levels (requirements specification, design, program) and improperly documented relations between components, showed itself as major causes of trouble.

By presenting the above analysis to the students at the end of the project, they were made aware of the significance of proper documentation for maintenance activities. They could similarly reflect on their own performance in this respect during the first phase of the

project. To us, the exercise was helpful in that it gave further insight into problem areas that can be realistically tackled in our environment.

## 2 Other Examples Of Student Projects

As an instructor, you have to think about the educational goals of student projects in relation to a course in software engineering. The goal of student projects in software engineering is **not** to have them write yet another big program. Such has a very meager learning effect with respect to relevant aspects of this eclectic field. In my opinion, it is much better to focus on one or a few particular aspects. The maintenance orientation discussed in section 1 is one example of such a focal point. Other possible foci are discussed in the next subsections.

### 2.1 The Importance of Structured Documentation

In the course of a software development project, several pieces of structured documentation are developed, such as a requirements specification, design specification, test plan, etc. A software engineering (team) project might concentrate on the contents of one or a few of such documents.

We have experimented with this model in the following way:

1. student teams prepare a requirements document, design specification and implementation of a given problem in a period of 8 weeks. Figures 6 and 7 give example problems we used for this type of project. Teams are judged on the quality of the documentation they produce (clarity of presentation, traceability of decisions, adequacy of the decisions w.r.t. the problem statement), and the consistency between successive documents (requirements specification – design specification – actual running software).
2. After this period of 8 weeks, the complete piles of documentation are redistributed amongst the teams. So each team gets the requirements, design and software written by another team. Each team is now asked to evaluate the set of documents, test the software, and write an evaluation report. Moreover, they are asked to implement a simple adaptive maintenance request. The time for this is 3 weeks.
3. At the end of the project, each team gives a presentation of its findings.

The major lesson students learn from this type of project is that, no matter how hard they try, it is difficult if not impossible for them to produce documentation that is easy to use, is well-structured, whose meaning is obvious to others, etc.

### 2.2 The Virtues of RAD

A potential risk of the type of project as discussed in section 2.1 is that students severely overestimate their capabilities to finish a project on time. They generally have little experience yet in software development projects. They are not able to relate specific requirements to effort. They tend to plan too optimistically. All this results in a project where time pressure builds up considerably near the end of the project. We feel this is largely caused by the waterfall-like, Big Bang approach in the first part of the projects as discussed in section 2.1.

---

A Geographic Information System (GIS) offers the user the possibility to interactively ask for information about (some part of) a region. For Greater London, e.g. a GIS could give information per district about the number of inhabitants, location of shopping areas and, say, the density of population per square mile.

For this assignment, take some area of your own choice, split the area into subareas (at least seven) and include a number of categories of information (at least five) per subarea. The choice for those categories is up to you.

It is optional to use aggregated areas, such as:

Greater LONDON  
  London  
    The City  
    Westminster  
    Saint Marylebone  
    ...  
  Enfield  
  Harrow  
  ...

The user must be able to interactively request information about a subarea through a graphical interface. The system should keep data about which subareas have been queried, and how often (per information category).

---

Figure 6: Problem description for a Geographic Information System

---

An Interior Design Assistant (IDA) is a tool that helps an interior designer in his work. When designing an interior, the architect starts from a given floor-plan and an inventory of available furniture. The functionality of IDA includes the presentation of a space and the positioning of pieces of furniture in that space. The description of the space may be written from a file. When positioning a piece of furniture, the system should check whether this is possible. For instance, a chair should not be positioned on top of a sidetable. It must in addition be possible to request information about the available pieces of furniture, such as the price and terms of delivery. It must also be possible to obtain the total price of a designed interior, and information on how to order it.

Except for designing an interior, IDA must also be able to act as a *showroom*, by presenting a series of available designs.

---

Figure 7: Problem description for an Interior Design Assistant

We therefore decided to experiment with a RAD-type scheme. In RAD, several incremental prototypes are developed. Each cycle has a fixed duration, and the number of requirements realized in this timebox is not fixed in advance (see also section 3.4 of the textbook).

We have experimented with a scheme involving two cycles:

1. A first cycle of 4 weeks. We do not expect student teams to achieve all that much in this cycle. It is really meant as a learning period. They learn how RAD works, what might and might not be realizable in a given period of time, etc. It is a complete RAD

cycle, though, and they have to deliver a first running prototype at the end of week 4.

2. A second cycle of 8 weeks, which results in the actual product.
3. Optionally, an evaluation cycle such as discussed in section 2.1 may be added.

### 2.3 Emphasis on OO

Everything is object-oriented, nowadays. It is quite easy to combine the type of project as discussed in sections 2.1 and 2.2 with object-oriented modeling techniques. For instance, we may require the design documentation to include a (UML-) object model which includes all classes and their inheritance- and uses-relations. Yet another example which we used in an object-oriented RAD-type project is given in figure 8.

---

An *Agenda Support System* assists the user in maintaining a record of important events, dates and appointments. It moreover offers the user various ways of inspecting her agenda, by giving an overview of important dates, an indication of important dates on a calendar, and (more advanced) timely notification.

A *Multi-user Agenda Support System* extends a simple Agenda Support System by providing facilities for scheduling a meeting, taking into account various constraints imposed by the agendas of the participants, such as a special event for which a participant already has an entry in her agenda.

A minimal Multi-user Agenda Support System must provide facilities for registering important dates for an arbitrary number of users. It must, moreover, be able to give an overview of important dates for an individual user, and it must be possible to schedule a meeting between an arbitrary subset of users that satisfies the time constraints for each individual in that particular group.

This minimal specification may be extended with input facilities, gadgets for presenting overviews and the possibility of adding additional constraints. Nevertheless, as an advice, when developing a Multi-user Agenda Support System, foll the KISS principle: Keep It Simple ...

---

Figure 8: Problem description for a Multi-user Agenda Support System

### 2.4 A ‘Fail-Safe’ Real Life Project

As noted before, real life projects are difficult to deal with in an educational setting. However, by taking a few careful measures, a ‘close-to-real-life’ project setting is possible. In this section, we describe our experiences with this type of project. More information can be found in [Luden, 1995] and [Gordijn and Niessink, 1998].

In these projects, a real customer with a real problem is the starting point. In one of the projects, for example, the students were required to study the possibilities of electronic commerce for a customer firm. The type of project we have experimented with has three phases:

1. the contract phase, in which the student team negotiates an agreement with the customer. At the end, a sound project plan must be made. This phase takes 4-5 weeks.

2. during the subsequent project phase, the deliverables of the contract are produced. These deliverables include one or more designs. The students do not implement these designs. The project phase takes 8 weeks.
3. Finally, the students deliver and present their results to the customer.

This project offers the student a real life setting. However, care must be taken to not let them drown. For example, in the e-com example mentioned above, the student team opted for a waterfall-like approach in the project plan, while the customer had very little knowledge of the Web and its possibilities.

For this reason, the student team is coached and guided by mentors (university staff members, available throughout the week), and a steering committee (university staff members too, available for a few meetings). The students are required to regularly submit written reports. These are then discussed with their mentors and/or the steering committee. In the above case, both the mentors and the steering committee coerced the team to follow a prototyping kind of approach. The general idea behind this scheme is that we do not intervene, unless this is deemed necessary by the mentors or steering committee to reach the project goals.

Another example of this type of project is given in [Dawson and Newsham, 1997]. Dawson reports about a university course with teams of 8–9 students. Over the years, elements of an industry course were introduced into the university course: specifications were left vague and ambiguous, requirements and priorities changed in the course of the project. etc. This line of thought is carried further in [Dawson, 2000], where a number of ‘dirty tricks’ is proposed to simulate the real world.

## 2.5 Other project foci

Yet other possible project foci include:

- teamwork aspects. This may involve having students play certain team member roles, such as project manager, tester, designer, and so on. Guarding this type of project tends to be very time-consuming for the instructor.
- the use of tools to support software development. You then have to supply the students with one or a few tools. The project may aim at system development using a toolset, assessment of tools by doing, say, a design using different design tools, or assessment of different drawing techniques offered by a single toolset.
- software reuse/component-based development. By making available some reuse library, one may envisage projects trying to reuse the knowledge embodied in that library. Such a software-development-with-reuse project often emphasizes the object-oriented approach.
- user-interface issues. Projects may concentrate on interface aspects, e.g., by having students develop mock-ups or prototypes, or having them engaged in user testing of an application.
- ”pure” maintenance or system enhancement aspects. The project may take an existing system as starting point, and let students do maintenance or enhancement tasks.

Besides these possible focal points, there are at least two other "dimensions" along which student projects may differ. The first dimension has to do with the organizational setting of the project. In [Shaw and Tomayko, 1991], four project styles are recognized:

1. The toy project, in which teams of 3-5 students get the same task, pre-specified by the instructor. The project discussed in section 1 fits this model.
2. The mix-and-match project, in which the product requires several components. Each student team develops one or a few components, and several versions of the product may result from integrating collections of components. The now famous "Software Hut" style of project ([Horning, 1976]) fits this model.
3. The project for an external client, in which students develop components of a product for an external client, integrate them, and pass an acceptance test set by the client. This type of project typically emphasizes teamwork aspects, project organization, client relations and the like. It tends to require much work from the instructor (see also section 2.4).
4. The individual project, in which each student team has a separate project. Besides the fact that the projects are independent, this type of project has many of the characteristics of the toy-type project.

The final dimension concerns the timing of the project. Projects may either run concurrently with class lectures, or be scheduled in a separate semester. I personally prefer to run the student lab concurrent with the lectures. Many topics treated in the lectures sound obvious and all too easy to students. Everyone knows that the design and code should correspond to one another, that systems must be documented properly, and so on. Following such practices in a project is much harder, though. By actually confronting students with these issues in the lab project, lectures will have a greater impact. The benefits of this confrontation get lost if lectures and projects are scheduled in different semesters.

Running lab projects concurrent with lectures does impose some constraints on the order in which topics are treated. For example, if the project starts with a requirements analysis, that topic has to be treated in one of the first lectures.

Given any of the above focal points for student projects, there are a great many example applications that can be used. Sample (toy-type) projects are:

- a simple interactive editor;
- a simple graphical browser for object-oriented systems;
- a graphical editor for, say, dataflow diagrams;
- a pretty printer;
- a system to compute complexity metrics, such as McCabe's;
- an implementation of a cost estimation model like COCOMO;
- a syntax checker for VDM (this is a rather complex assignment);
- a Web-based interactive system for a travel agency;

- an interactive information retrieval system;
- and so on, and so on.

The Shaw & Tomayko report cited above is a useful source for further information on possible types of lab projects for a software engineering course. Other sources include The Forum for Advancing Software Engineering Education (FASE) volume 9, no 8 (1999) – <http://www.cs.ttu.edu/fase/v9no8.txt>, [JSS, 1999], [Annals, 1998] and the yearly IEEE Conference on Software Engineering Education and Training (CSEE&T).

## References

- [Annals, 1998] Special Issue on Software Engineering Education. *Annals of Software Engineering*, 6, 1998.
- [Bradley, 1997] N. Bradley. *The Concise SGML Companion*. Addison-Wesley, 1997.
- [Brooks, 1995] F.P. Brooks. *The Mythical Man-Month*. Addison-Wesley, second edition, 1995.
- [Dawson and Newsham, 1997] R. Dawson and R. Newsham. Introducing Software Engineers to the Real World. *IEEE Software*, 14(6):37–43, 1997.
- [Dawson, 2000] R. Dawson. Twenty Dirty Tricks to Train Software Engineers. In *Proceedings 22nd International Conference on Software Engineering (ICSE22)*, pages 209–218. IEEE, 2000.
- [Gordijn and Niessink, 1998] J. Gordijn and F. Niessink. Balancing Freedom and Supervision in a Real-Life Educational Project. In P. Klint and J.R. Nawrocki, editors, *Proceedings Software Engineering Education Symposium (SEES '98)*, pages 77–84, 1998.
- [Horning, 1976] J.J. Horning. The Software Project as a Serious Game. In *Software Engineering Education Needs and Objectives, Proceedings of an Interface Workshop, New York*, pages 71–77. 1976.
- [IEEE1016, 1987] *IEEE Recommended Practice for Software Design Descriptions*. IEEE Std 1016, 1987.
- [IEEE830, 1993] *IEEE Recommended Practice for Software Requirements Specifications*. IEEE Std 830, 1993. Figures 9.6 and 9.7 and Appendix C on pages 665-668 reprinted with permission from IEEE Std 830-1990, ©1993 by IEEE. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner.
- [JSS, 1999] Special Issue on Software Engineering Education and Training for the Next Millennium. *Journal of Systems and Software*, 49(2/3), 1999.
- [Luden, 1995] H. Luden. Practicum Informations-Systems Development. In *Proceedings of the Second International Conference on Software Engineering in Higher Education (SEHE '95)*, pages 337–344, 1995.
- [Shaw and Tomayko, 1991] M. Shaw and J.E. Tomayko. Models for Undergraduate Project Courses in Software Engineering. Technical report, Technical Report CMU/SEI-91-TR-10, Software Engineering Institute, 1991.

- [Soloway *et al.*, 1988] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM*, 31(11):1259–1267, 1988.
- [van Vliet, 1989] J.C. van Vliet. Teaching Software Maintenance. In N.E. Gibbs, editor, *Software Engineering Education*, volume 376 of *Lecture Notes in Computer Science*, pages 80–89. Springer Verlag, 1989.
- [Weiss, 1998] M.A. Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley, 1998.