To appear in *Journal of Computer Science Education* **11**(1), 2001.

Experience with Work-Product Oriented Software Development Projects

Jürgen Börstler

Department of Computing Science
Umeå University, Sweden

jubo@cs.umu.se
http://www.cs.umu.se/~jubo

Abstract

In this paper, we describe our experiences with student team projects in object-oriented software development. Object-oriented development processes are not as straightforward as, for example, traditional waterfall-like models. In object-oriented development, there is no clear border between analysis, design, and implementation. Students therefore have difficulties deciding on what to do next, how to do it, and why to do it.

A work-product oriented development process provides a framework for structuring and managing object-oriented development. Development can be defined in terms of interrelated work products. Each work product is defined by its purpose and contents, the inputs needed, and the techniques used to produce it. The definition of a development process and the production of a single work product are therefore more straightforward.

Our experiences show that such an approach is very suitable for student team projects.

Key words: Team Projects, Work Products, Object-Orientation, Experiences.

1. Introduction

Object-oriented approaches are now widely accepted and practised in industry. Many software development companies and organisations have already made the transition to object technology. There are therefore increasing demands for individuals educated in object-oriented development. Industry needs people who are able to join their ongoing object-oriented projects without the necessity of extensive retraining ([GeMa 96]).

Object-oriented development is not just a matter of changing to another programming language. It requires a new way of thinking ([Guzd 95]). Object-oriented education should therefore cover all aspects of object-oriented development. To enable students to join "real-life" object-oriented projects realistic case studies and hands-on experiences are needed. In our course entitled Object-Oriented Software Engineering (OOSE), we try to simulate a real software development project as closely as possible. This means that students work in teams to develop a non-trivial software system. The teams start with informal project proposals and develop running prototypes for the proposed systems.

However, simply going through all the development phases is not enough to prepare the students for industrial projects. There are many tasks beyond those of core development, where the students need some training, as for example, project management and inter- and intra-team communication. These aspects are treated in detail in [BiTh 98] and [Leth 98]. During the OOSE course, we set up the following conditions.

- Customers and/ or users are involved in the project.
- The project must meet a tight schedule.
- Industrially significant tools are used.
- Subcontractors develop parts of the code.
- Formal project team meetings have to be attended.
- Weekly reports have to be send to "upper management."
- Tight deliverables are evaluated on a regular basis.
- The resulting product is evaluated by competitors.

The reader should note that the OOSE course is not taught as a stand-alone project course. We require basic knowledge in Software Engineering as a prerequisite. Our department has a strong profile towards Software Engineering and software engineering topics are treated explicitly in many courses.

Already in their first Computer Science course students are introduced to systematic program development. From the very beginning, we require each assignment to be delivered together with a report. These reports have to comprise at least a problem description, a solution description, the source code and the test results. The first-year courses emphasise the principles of data abstraction, information hiding, and modularization.

During their second year students take (among others) courses in systems programming, human-computer interaction, and software engineering. The Software Engineering course introduces the core concepts of Software Engineering and is oriented towards structured methodologies. The course is built around a small project, where students are exposed to non-trivial team structures.

All of our first- and most of our second-year courses are compulsory. After their second year students are free to select courses from our profiles in Software Engineering, Cognitive Science, Parallel Computing, and Scientific Computing.

The OOSE course is a central course in the Software Engineering profile. The other courses in the profile deal with the study of well-known application domains (compilers, databases, and operating systems) and software development theory (algorithm analysis, formal specifications, and semantics). Furthermore, we offer students a "conference" to gather experiences in research, technical writing, and oral presentations. The Software Engineering profile is described in more detail in [Börs 97].

The reminder of the paper is organised as follows. In Section 2 we discuss the advantages of a work-product oriented development process for the teaching of object-oriented development. Section 3 gives an overview over the structure and contents of the OOSE course. This section is followed by a short discussion of appropriate project types. In Section 5, we describe some problems that arose in the course and how we solved them. Section 6 summarises our experiences with this and similar courses. Section 7 concludes the paper and discusses some future work.

2. Advantages of a Work-Product Oriented Process

Typically object-oriented development processes are iterative and incremental see figure 1). The same languages or notations are usually used during analysis and design. This leads to a development approach that is sometimes described as "analyse a little, design a little, code a little" ([Booch 94]). The process descriptions give very little help to decide when to stop or continue one of these activities. Furthermore are these activities not unambiguously associated with specific documents. Editing a class diagram can for example be on analysis, design, or even coding level.

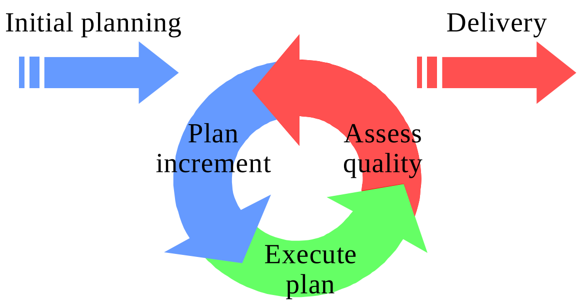


Figure 1: Iterative and incremental development.

Traditional waterfall-like development processes, on the other hand, have well distinguished development phases. In each development phase specific methods, languages, and tools are used to produce specific outcomes, associated with this phase only. Compared to traditional waterfall-like development processes, iterative and incremental processes seem therefore very fuzzy.

Table 1: Phases, Activities, and Work Products.

Project Management	After initial project planning, define development activities and allocate resources to the activities. Allocate requirements to releases and manage project schedule and issues.	Project Workbook Outline	An organised list of work products that are expected to comprise the project workbook.
		Resource Plan	Analysis of the resources required for the successful completion of the project.
		Schedule	A task time line showing dates, milestones, critical path, etc.
		Risk/Option Management Plan	Lists development options and describes the plan for minimising project risks.
		Test Plan	Outlines the project's plan for testing the application.
		Issues	A list of outstanding issues, questions, and concerns that are reviewed on a regular basis.
Requirements Gathering	Group functional requirements (into use cases) and prioritise them.	Problem Statement	Description of the problem to be solved in non-technical terms.
		Use Cases	An OO formalisation of functional requirements describing the usage of the system by external agents.
		Nonfunctional Requirements	Requirements that do not belong to user function, such as performance, platform, and quality.
		Prioritized Requirements	Defines the relative priorities of functional and nonfunctional system requirements.
UI Design	Document how users will interact with the application.	Guidelines	Describes the user interface guidelines and standards.
		Screen Flows	Documents user navigation through the application's user interface.
		Screen Layouts	Documents details of all screens.
		UI Prototype	A prototype built to show users the "look and feel."
Object-Oriented Analysis & Design	**Analysis:** Identify objects, their attributes, behaviours, and interrelationships. Develop solutions to system usage scenarios in terms of active objects that group related tasks and communicate with other objects in order to complete them. **Design:** Plan a solution to the problem examined during analysis in terms of interacting objects, within the constraints specified by the nonfunctional requirements.	Guidelines	Records the details of the analysis and design approach being followed.
		System Architecture	Description of the high-level components/structures of the system and the design principles guiding the implementation.
		Object Model	A consolidated model describing the classes of a system together with their responsibilities and static interrelationships.
		Scenarios	Descriptions of required systems behaviour. Scenarios refine use cases and are formalised in OIDs.
		OIDs (Object Interaction Diagrams)	A working out of a scenario, showing the interactions between objects to accomplish (the implementation of) a task.
		State Models	Show the life cycle of an object, i.e. its possible states and state transitions.
		Class Descriptions	Detailed descriptions of all classes.
		File Structure	The files and their structure as required by the system.
		Traceability Matrix	A cross-reference table that relates design elements to requirements.
Implementation	Systematically code the classes as specified in the class descriptions so that they can be built and installed on the target platforms.	Coding Guidelines	A description of the coding guidelines and standards.
		Source Code	The actual implementation of the product.
		User Support Materials	Documentation delivered in various forms, which support the customer's use of the product.
Testing	Insure that the application meets the requirements set forth in the problem statement and requirements gathering work products.	Test Cases	The testing work products.

A work-product oriented development process helps to structure and manage object-oriented development. A work product is “any planned, concrete result of the development process; either a final deliverable or an intermediate one” ([OOTC 97], 77). All work products are held and managed in a central depository, the project workbook. We use a WWW-based version of a workbook. All groups have their current work products on-line. This strategy supports inter- and intra-group communication as well as project tracking and control. The list of work products used in our course is adapted from IBM’s OOTC (Object-Oriented Technology Center, see [OOTC 97]) and listed in Tables 1 and 2. Figure 4 in the appendix gives an example of the structure of a group’s workbook from our Fall 1998 offering of the course.

Table 2: Further work products not bound to particular phases.

Miscellaneous	Document miscellaneous activities in appendices and add them to the project workbook.	Glossary	Definitions and terminology.
		Historical Work Products	Back-level work products.
		Meeting Minutes	The minutes of all project meetings.
		...	...
Course organisation	There are some special documents to support course organization.	Team Description	A presentation of your team and the members of the team.
		Project Proposal	A short description of a software product, where your team is the customer.
		Subcontract	An agreement stating that another team produces a certain piece of software for your team.
		Prototype Evaluation	An evaluation of another team’s prototype.
		Weekly Reports	Short reports with up-to-date project information (send to “upper management”).
		Final Report	A complete document set.

The development process is defined in terms of work products and their interdependencies. Each work product is defined in detail by, among others, its purpose, alternative methods/ techniques for its construction, how its correctness can be verified, and how it is related to other work products. Furthermore, examples are given for most of the work products and guidelines for their development. The techniques used to construct work products can be grouped into phases according to Tables 1 and 2. This approach has several advantages.

- It is language independent.
- It is method independent.
- It makes the contents and purposes of work products explicit.

- It becomes easier to decide what to do, since methods and techniques are related to work products.
- It becomes easier to decide what to do next, since work product interrelationships are clearly defined.

The situation described above is somewhat oversimplified, since there are a few cyclic dependencies between work products. However work product orientation allows a much more structured description of the development process.

3. Related Approaches

Since we adopted the work-product oriented approach in 1997 several other approaches have been described that share some similarities with the one described in above, like the OPEN Process ([GHJ 97]), the Unified Software Development Process ([JBR 99]), the Rational Unified Process (RUP, [Kru 99]), or eXtreme Programming (XP, [Beck 00]). From a pedagogical point of view, a process applicable to student team projects should fulfil the following basic requirements:

- **Comprehensibility**. The process must be described in a form that makes it easy to understand. Our experiences show that students prefer a description in terms of the things that have to be produced.
- **Accessibility**. The information must be organised in a way that makes it possible to determine what should be done next and how.
- **Scalability**. The process must be easily adaptable to small teams and small projects.
- **Instructive**. The process must teach approaches that can be (and are) applied in industry.

The OPEN and the Unified processes are on too general a level to fulfil those requirements. Their purpose is rather to define a framework for (object-oriented) development processes. To actually use them they need to be instantiated to a concrete environment.

The RUP is an instance of the Unified Software Development Process. It describes development in terms of workers (whereas we use specialist roles), workflows (phases), activities, and artefacts (work products). The RUP is not defined around a workbook, but otherwise quite similar to our approach. The RUP has the advantage that it is quite popular in industry (especially in Sweden, where it probably will replace Objectory, a very popular predecessor of the RUP). Furthermore, it is available as an on-line product including process, methods and tools guidance, as well as document templates, promising very high accessibility. A great disadvantage is that its paper version does not include any descriptions of methods or techniques to construct the various artefacts. Its on-line version on the other hand is neither method nor tool independent. Many companies are now developing own instances of the Unified Software Development Process. Some of these developments are quite close to both the RUP and our approach (see for example WoW (Ways of Working, [Sonn 00]).

XP is a “lightweight methodology” ([Beck 00]) differing considerably from the approaches described so far. XP fulfils all applicability requirements stated above, but would require extensive changes to our course. Almost all roles, phases, activities, and work products need to be reconsidered. In XP, all developers participate in all activities, which makes it appealing for student team projects. On the other hand is analysis and design done “on the fly” instead of up-front, which requires experienced developers. Further XP practices like simple design guided by metaphors, teamwork and pair programming, early and continuos testing, and collective code ownership seem difficult to handle by a team of students without project experience. XP is less mechanical than “traditional” approaches giving developers more creative freedom. How to handle this freedom though is very difficult to teach. XP demands on-site customers on the teams, which would be difficult to handle in student team projects.

4. Course Outline

The OOSE course focuses on object-oriented analysis and design and has our basic Software Engineering course as its main prerequisite. It is a team project-oriented course and is strongly oriented towards the work products the student teams have to produce. The course runs over 10 weeks with a workload of 20 hours per week. Table 3 gives an overview over the schedule of the course.

Table 3: Schedule of the Object-Oriented Software Engineering (OOSE) course.

Week	lecture topics	project deliverables due
1	L1: Project organisation; Object-oriented development L2: Phases, Activities, and Work Products; Object-oriented analysis	Team Descriptions
2		Project Proposals; Project Workbook Outline
	Oral presentations of teams and proposals	
	L3: Introduction to UML (Unified Modelling Language) and Rational Rose™	Initial Project Plans
3	L4: More object-oriented analysis	
4	L5: Object-oriented design; An example	Requirements Documents; Final Project Plans; GUI Design
5	L6: Subtyping vs. inheritance; Another example	Object-Oriented Analysis & Design (version 1)
6	L7: More inheritance; Frameworks	
7	L8: Design heuristics	Object-Oriented Analysis & Design (version 2); Test Plans; Subcontract
	Oral presentations of projects' analysis and designs	
8	L9: More design heuristics; Patterns	
9	**Prototype Demonstrations**	

		Prototype User Manuals
10		Prototype Evaluations; Final reports
Weekly		Status Reports

Each student is expected to work about 200 hours on the project. A team of 5 to 7 students can therefore manage projects of sizes 6-8 person months. Each team runs a complete project from the planning stage to the presentation of a running prototype. Each team member fulfils the roles of one or more specialists for example team manager, requirements specialist, and so on. Each role is associated with the responsibilities for one or more work products and/ or presentations. Together with peer evaluations and self-evaluations, this allows us to estimate the contributions of an individual student to a project.

During the course, each team is involved in several projects, while adopting the roles of a customer, a contractor, a subcontractor, and an evaluator.

A customer publishes a project proposal that can be selected by one or more contractors for development. Teams are not allowed to contract their own project proposals. The reasons for that are explained in more detail in Section 5. Since Fall 1996, teams have also been able to contract proposals by external customers (colleagues, other departments, and local firms).

During the project, each team must subcontract another team for a small part of the prototype implementation. To do this each team designates a well-defined part of its design and provides it with black-box test cases. These parts have then to be subcontracted by another team for implementation. The subcontracts can be freely negotiated with the other teams, but teams must not subcontract two by two. Subcontracting confronts the teams with the necessity of fixed and clear interfaces. Furthermore, it constitutes a good case study for a risk analysis. Since the teams are allowed to choose development platform, programming language, and tools/ libraries, they must be very careful in defining their subcontracts. If a team fails in finding a subcontractor, we engage in the subcontracting process. In rare cases, we have allowed teams to subcontract themselves.

Finally, we require that each team evaluate another team’s prototype to motivate the production of a user manual and to assist the course instructor in project evaluation. Figure 2 summarises the project participants, their roles, and interrelationships.

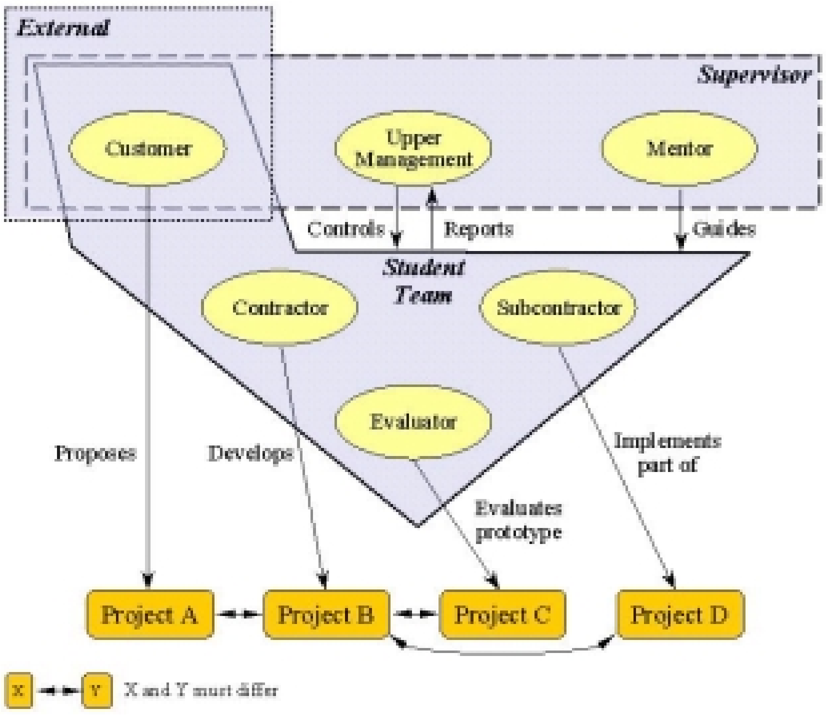


Figure 2: Project participants, roles, and interrelationships.

5. Types of Projects

Over the years we have had a wide range of projects in this and similar courses. Up to now we gained experiences with the following kinds of projects: Editors, inventory systems, accounting systems, games, and simulations. It turned out that not all types of projects are equally well suited for object-oriented development.

Usually inventory and accounting systems are proposed as typical and useful student projects. Our experiences show that such systems are not an optimal choice for experiencing object-oriented development. The architectures of inventory and accounting systems usually have two basic components. First, there is a simple database that holds some kinds of stocks or states of accounts. The other component is a user-driven interface to the database. Very often, such projects develop a form-based interface to a database without any real processing logic. That means that object behaviour modelling is not an issue in these projects, since there are no objects with non-trivial behaviour.

Modelling object behaviour is a very important aspect in all object-oriented approaches. There exists a wide range of techniques to model behaviour that cannot be taught adequately in simple inventory and accounting systems. Non-trivial interaction diagrams, state machines, etc. are usually not needed to understand the behaviour of such systems. The students try to avoid the usage of these techniques, since they seem to add complexity to the problem and development process instead of simplifying them. The potential of these techniques will therefore not be experienced.

Editors, games, and simulations typically offer many opportunities for behaviour modelling. They all have active processing components, such as a formatter in an editor, the rule base in a typical game, and the objects in a simulation that model real-world entities. Such systems are furthermore quite sensitive to changes in the requirements. It is therefore much easier for the students to experience the importance of design and the planning for change.

Examples of successful projects have been different versions of a Game of Life, where interacting life forms can be simulated. The simulation can be instantiated with different types of life forms. One team developed even a 3-dimensional graphics interface. Another team developed an on-line version of the Personal Software Process ([Hum 95]). For our Fall 1999 offering, we had a collaboration with local industry, where three teams competed in developing a browser for a PalmPilot handheld computer in Java.

Further examples of actual project proposals are available from the "Projects" Section on the course home pages at http://www.cs.umu.se/kurser/TDBC18/Ht00.

6. Problems and Course Evolution

The OOSE course evolved from the idea of integrating a traditional Software Engineering course (lectures, exercises, and assignments) with a practical course organised around a single non-trivial team project (no lectures, but student teams lead by mentors). We realised quickly that there was too much material for one course. However, we did not want to give up the idea of a team project in either of the courses. Thus, we ended up with a compulsory basic Software Engineering course and the elective OOSE course. The basic Software Engineering course combines a traditional track comprising lectures, exercises, and assignments with a small team project. The students who take the OOSE course are therefore well prepared for a more advanced team project.

When we started in Spring 1994, the structure of the course was simpler than described in Section 3. In the remainder of this section, we will discuss some of our problems and experiences and how these effected the course's design.

6.1. Poor Requirements Documents

Students usually have serious problems documenting the requirements for projects that they themselves have proposed. Requirements documents are usually incomplete and difficult to understand by others, since obvious or "crystal clear" requirements are ignored. They argue that these things need not be written down, because all team members know what the result will be like. Since there are no users of the requirements documents, other than the teams themselves, it seems to be a waste of time to produce such a document. It is therefore difficult to give them an idea of some of the problems than can arise when requirements have to be gathered from "real" customers.

It was interesting to see that the teams performed much better on projects that were proposed by others. For games, for example, the teams produced extensive and detailed descriptions of the functional requirements and the user interface, which could be derived from the rules of the games. For their own proposals, the emphasis often was on the non-functional requirements. It is therefore very important to make clear that the teams do not write the requirements documents for their own pleasure, but that they are part of their contract with the customer.

Since we introduced the rule that *no team must choose its own proposal*, we observed a significant improvement in the quality of the requirements documents. A side-effect of this rule has been that the students now realise that requirements are not "invented" by developers, but must be gathered and evaluated in collaboration with customers.

6.2. Process Fuzziness

As described in Section 2, object-oriented development processes are not prescribed step-by-step procedures. Object-oriented analysis and design (OOA&D) is comprised of several techniques, for example use case analysis, scenario modelling, state modelling, etc. Several of these techniques can be used in the analysis and the design phases as well. It can therefore be very difficult to decide what to do next, especially when the border between analysis and design is not well defined. Most textbooks on OOA&D implicitly assume that some form of requirements engineering precedes OOA&D. Examples often start with a short problem statement and requirements are presented and evaluated "on-the-fly," during object-oriented analysis. This gives the misleading impression that object-oriented analysis is the first phase in a development project.

We have reacted to this problem in two ways. First we emphasise that object-oriented analysis cannot replace traditional requirements engineering completely. The definition of a separate requirements gathering phase, that precedes object-oriented analysis, helps to clarify their differences. Second we require a requirements document to be produced, which is then used as the input to OOA&D. This prevents student teams from delving into OOA&D before the requirements are clearly defined.

6.3. Structure and Contents of Deliverables

From the beginning, we gave only vague descriptions plus a few recommendations on the formats and contents of the required deliverables. We did this for two reasons. First, students should think about the purpose of each deliverable and find a suitable structure and format. Second, we hoped to get students interested in standards. This worked quite well, but had the disadvantage that students tended to spend too much time on these issues. When the concept of work products was introduced in addition to deliverables, we decided to provide more detailed descriptions, so that students could concentrate on the important aspects of the course.

We therefore reworked our original list of deliverables and defined the purposes and contents of all deliverables in more detail. All descriptions are available on-line (see Figure 5 in the appendix for an example). All deliverables are comprised by sets of work products and relationships between work products are made explicit (see Figure 3).

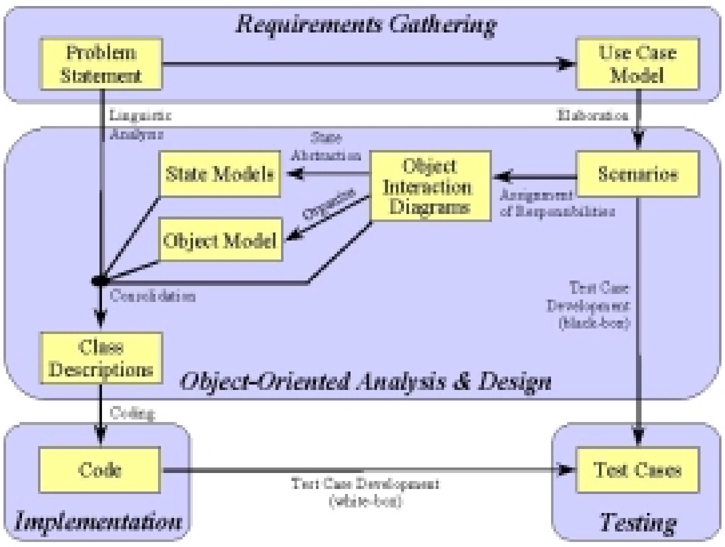


Figure 3: Work product interrelationships.

6.4. Teamwork

Teamwork, communication, and management skills are demanded by industry ([Leth 96]), so they need to be addressed in project courses. Our course syllabus therefore prescribes deliverables, oral presentations, and informal meetings. In the beginning, we experienced some problems with teams that strictly distributed their work among team according to their specialist roles. Synchronisation was done just before a presentation and for the final report. This had the advantage that team members could work independently, but made it very difficult for the team managers to keep track of the project's progress. Some problems, such as very uneven workloads and unsynchronised documents of very different formats, were much more common in such teams than in more collaborative teams. Strict distribution together with missing intra-team communication can also jeopardise whole projects, since team members who do not fulfil their duties are first detected when the next one will take over or a deadline has passed.

A negative side-effect of the strict distribution strategy from the educational point of view was that it was possible to pass the course with only superficial knowledge of the techniques used in OOA&D (except for the specialists responsible for OOA&D).

To avoid these problems we first introduced a second review for the project plan, since the initial project schedules tended to be too superficial. This allows us to check the assignment of the team members to tasks in more detail. Furthermore, we now require that all meeting minutes be archived and a list of issues is kept. Third we require weekly reports on the status of the project and opened/ closed issues. These actions had the effect that students now take planning and communication more seriously.

Since Fall 1995, each team has to set up WWW-pages with information on the team and its project(s). For the Fall 1997 offering, we extended this requirement to a complete on-line project workbook with up-to-date versions of all work products of a project. Some teams use their WWW-pages to support team communication. See Figure 4 in the appendix for an example page showing the workbook main page and communication facilities (on top, below the header).

6.5. Subcontracting

Students do not like subcontracting. As described in Section 3, the problems are mainly due to the freedom we give students in their choice of development platforms, programming languages, and the usage of libraries and tools. In most course evaluations students actually suggested skipping the subcontracts, since they are seen as a waste of time and energy. Nevertheless, we are convinced that there is an important lesson to learn for the students. In industry, it is quite common that different teams work together in the same project. Students need to be aware of problems associated with inter-team work. Subcontracts are useful case studies to clarify the importance of clearly understood interfaces.

A further factor that contributes to the subcontracting problem is that students do not like to give away control over their projects. They are not comfortable with being dependent on others, and they do not trust the work done by their subcontractors (“not invented here” syndrome). Some teams actually reimplemented their subcontracts.

In Fall 1997 we introduced formal contracts that must be signed by the contractor and the subcontractor. Each team must then evaluate its contractor and subcontractor and include this evaluation into the final project report. Furthermore, we highlighted the different roles of a team and the fact that several teams contribute to the success of a project.

After some start-up difficulties, the situation has improved considerably since Fall 1998.

6.6. Grading

Course grading is on an individual basis. The grade is determined by means of team grading forms (see figure 6 in the appendix). The forms are used to keep track of the status and quality of all group performances. The entries in the grading forms can be traced to individuals by means of their specialist roles. Each team must furthermore evaluate the performance of the team as a whole, as well as the relative performance of its team members. These self-evaluations are delivered as part of the final report.

It is important to note that the final prototype is only a small part of the project and contributes as such to the final grade. The main goal of the course is to run a project properly, not to code a fancy product.

From the very beginning (spring 1993), all team members received the same grade (passed or not passed). This made grading easy, but had several disadvantages.

- No motivation to do more than “just enough” to pass the course.

- No rewards for teams or individuals performing especially well.
- Very difficult to “catch” passive team members.

Since teams’ as well as individuals’ performances varied widely, we introduced a levelled grading scheme for the Fall 1999. However, individual grading still depends to a high degree on team performance. To make grading as objective (and open) as possible we have developed a kind of credit/ penalty system.

Each team earns or looses “credits” depending on its performance. All work products and presentations are subject to this system. Criteria for evaluation include timeliness, completeness, comprehensibility, consistency, etc. At the end of the course, each team can freely distribute all credits earned among its team members. Individual grades are then computed from the individual’s number of credits. We give immediate feedback after each performance evaluation; i.e. the teams can calculate preliminary grades at any point in time.

The results so far were quite impressive. The course results have improved considerably. Details about the current grading scheme are available from the course’s web pages (http://www.cs.umu.se/kurser/TDBC18/local/Grading.html).

7. Experiences from Object-Oriented Team Project Courses

Our experiences with the OOSE and similar courses are very positive. Some recommendations for a successful object-oriented development course are summarised below (see [BaBö 97] for more details):

- **Select the project carefully.** As described in Section 4, the projects should be complex enough that students can experience the whole spectrum of object-oriented techniques, but also so simple that a reasonable prototype can be developed in time.
- **Produce a traditional requirements document first.** In our experience, the biggest single obstacle in object-oriented development is identifying the objects. Once an object or a class is brought up, it is easy to convince students of its usefulness. However, how objects are discovered seems to be a kind of magic to them.

  As described in the section above, we solve this problem by first developing a traditional requirements document, which is used as input for object-oriented analysis. Checklists and linguistic analysis work quite well in most cases to detect objects and classes, their properties, and relationships.
- **Define a clear process.** Beginners need a well-defined process to organise their work. We recommend following a work-product oriented process, as described in Section 2.
- **Apply design heuristics and patterns.** We have developed a series of examples to explore alternative designs. These examples are used to explain the differences between (multiple) inheritance and aggregation, and to motivate some design guidelines.

In the Fall 1996 offering we successfully introduced design heuristics ([Riel 96]) and patterns ([Gam+ 95]). Students find this form of concrete advice much more useful than abstract guidelines.

- **Provide mentors.** Mentoring ([Laza 95], [Lilly 96]) is a very effective way of providing the right information at the right time. We use students who took this course in previous years as coaches for student teams. The coaches keep themselves up-to-date on the status and progress of a project and evaluate the work products. They must not actively participate in the development process to avoid conflicts in interest. Reviewing work products and giving timely feed-back is quite time consuming and no mentor can manage coaching more than three teams. The mentors do not get any educational credits for their work, but are employed as teaching assistants by the department.
- **Do not use hybrid languages.** Hybrid languages like Turbo Pascal and C++ do not enforce encapsulation, and allow for functionally decomposed solutions in object-oriented disguise. If students are not forced to apply the new paradigm, they might be reluctant to do so ([GrBi 93]).

  Since Fall 1997, many of our teams have used Java very successfully.

8. Conclusions and Future Work

During Fall 1999, the course was given for its 7th time. Up to now 38 student teams completed 34 of 38 projects successfully and on time. The remaining four projects were finished successfully after giving the students some extra time.

Although the course lasts only 10 weeks, students are able to implement prototypes of significant size. Our experiences show that a work-product oriented approach helps students to organise and carry out their work.

The course evaluations show that students like the course. The design of the course gives worthwhile insights into many new aspects of their future work. Many students comment that teamwork was more difficult than they assumed, but that this was an important lesson to learn. Students often complain about the heavy workload and the amount of programming involved in the project, but we think that a prototype must be built to validate analysis and design.

For future courses we have planned for the following changes:

- **Offer more "real" projects with external customers.** Since our Fall 1997 offering, we offered occasional project proposals from outside the department. The students appreciate this and it is our goal to increase the number of external projects.
- **Allocate more time.** We are currently planning to extend the time scale of the project to 20 weeks to be able to include an iteration in the project.

- **Rearrange subcontracting.** When more time is allocated to the OOSE course, it overlaps partly with our basic Software Engineering course. This would allow for a new way of subcontracting. Teams from the basic Software Engineering course could then act as subcontractors for the teams of the OOSE course.
- **Provide more freedom in project planning.** Currently, all deliverables have fixed dates common for all projects. This can be a problem for teams with many team members who have to do some other work during parts of the project. We have planned to allow for some rescheduling that has to be "signed off" by "upper management" (the supervisor and/ or customer).
- **Track working time.** Currently students do not track their time. It is therefore very difficult to determine the current status of a project and to reschedule tasks if necessary in a fair way, so that the workload is distributed equitably.

  Time tracking would also be a useful tool for the supervisors to detect critical projects/ tasks early, and take action if necessary.

  Most students are already familiar with the Software Process (PSP^{SM}, [Hum 95]) from the preceding Software Engineering course. Time tracking could therefore be introduced without high start-up costs.
- **Collect metrics.** Metrics are an important tool to control the progress of a project. Apart from effort (time, see above) it would be very useful to collect size and quality metrics to be able to evaluate and compare projects.

References

[BaBö 97] V. Bacvanski, J. Börstler: Doing Your First Project—Discussion Paper, *Educator's Symposium of OOPSLA'97,* Atlanta, GA, Oct, 1997.

[Beck 00] K. Beck: eXtreme Programming eXplained, Addison-Wesley, 2000.

[BiTh 98] S. Biffl, G. Thomas: Preparing students for industrial teamwork: a seasoned software engineering curriculum, *IEE Proceedings Software* **145** (1), Feb 1998, 1-11.

[Booch 94] G. Booch, *Object-Oriented Analysis and Design, with Applications,* 2nd ed., Benjamin/ Cummings, 1994.

[Börs 97] J. Börstler: The Software Engineering Profile at Umeå University, *Proceedings of the International Symposium on Software Engineering in Universities (ISSEU'97),* Rovaniemi, Finland, Mar 1997, 60-69.

[Gam$^+$ 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Object-Oriented Architecture*, Addison-Wesley, 1995.

[GeMa 96] E. F. Gehringer, M. L. Manns: OOA/ OOD/ OOP: What programmers and managers believe we should teach, *Journal of Object-Oriented Programming* **9**(6), Oct 1996, 52-60.

[GHJ 97] I. Graham, B. Henderson-Sellers, H. Younessi: *The OPEN Process Specification*, Addison-Wesley, 1997.

[GrBi 93] T. Grechenig, S. Biffl: The Challenge of Introducing the Object-Oriented Paradigm, An Empirical Investigation of a Software-Engineering Course, *Structured Programming* **14**(4), 1993, 187-198.

[Guzd 95] M. Guzdial: Centralized Mindset: A Student Problem with Object-Oriented Programming, *Proceedings of the 26th Conference on Computer Science Education (SIGCSE'95)*, USA, Mar 1995, 182-185.

[Hum 95] W. Humphrey: *A Discipline for Software Engineering*, Addison-Wesley, 1995.

[JBR 99] I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*, Addison-Wesley, 1999.

[Kru 99] P. Kruchten: *The Rational Unified Process: An Introduction*, Addison-Wesley, 1999.

[Laza 95] E. Lazarus: Toward object-oriented mentoring methodology, *Journal of Object-Oriented Programming* **8** (6), Oct 1995, 64-69, 72.

[Leth 96] T.C. Lethbridge: The Relevance of Software Education: A Survey and Some Recommendations, *Annals of Software Engineering* **6**, 1998.

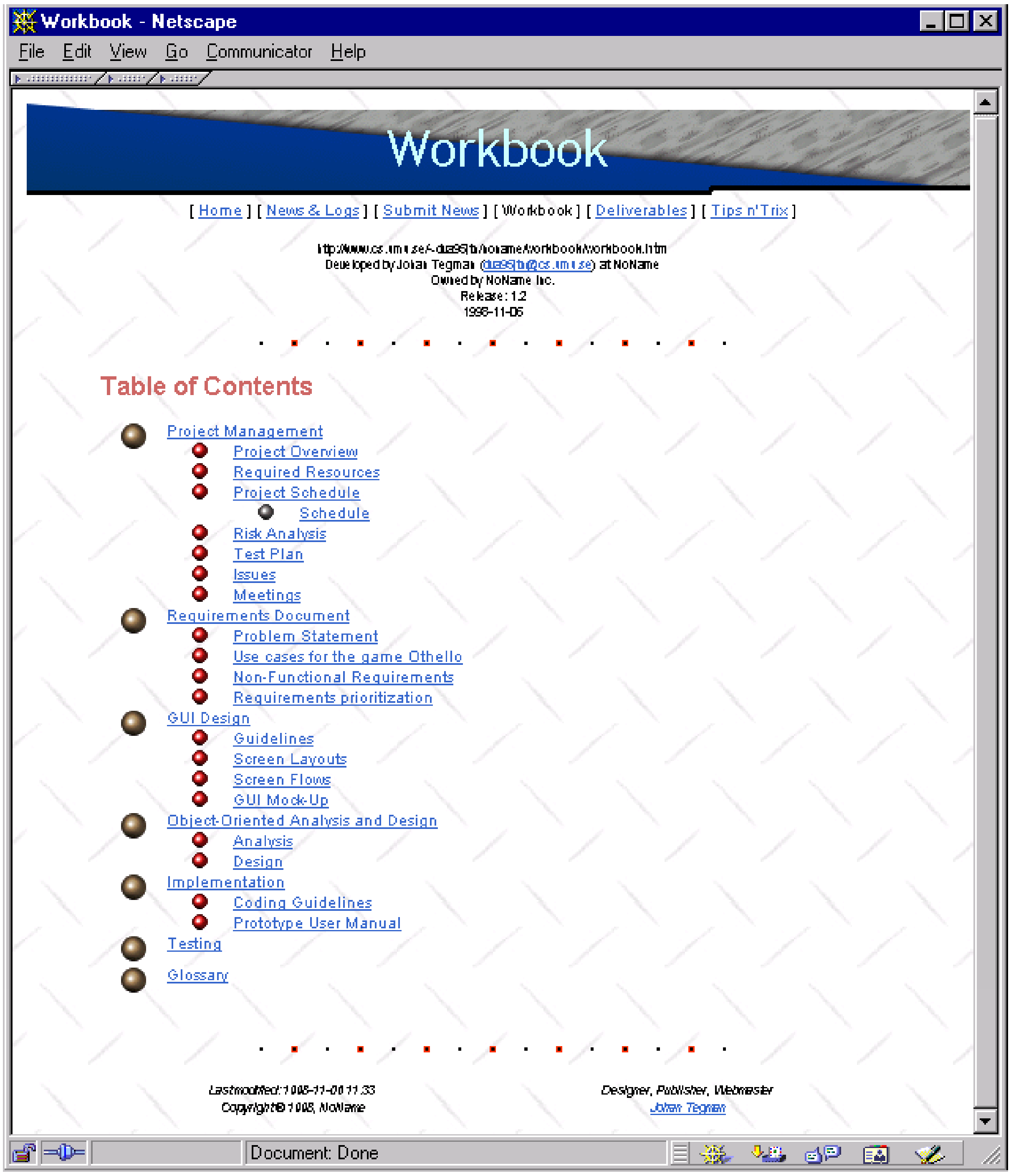
[Lilly 96] S. Lilly: Case Studies in the Classroom, *Object Magazine* **6** (8), Oct 1996, 81-83.

[OOTC 97] Object-Oriented Technology Center (IBM): *Developing Object-Oriented Software*, Prentice Hall, 1997.

[Riel 96] A. Riel: *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

[Sonn 00] T. Sonning: WoW, Ways of Working, A Software Development Process, Revision 1.0, Report EPL/T/D 99:039, Ericsson Erisoft AB, Umeå, Sweden, 2000.

Appendix



Workbook - Netscape

File Edit View Go Communicator Help

Workbook

[Home] [News & Logs] [Submit News] [Workbook] [Deliverables] [Tips n'Trix]

http://www.cs.umu.se/~dua95jtn/noname/workbook/workbook.htm
Developed by Johan Tegman (dua95jtn@cs.umu.se) at NoName
Owned by NoName Inc.
Release: 1.2
1998-11-06

Table of Contents

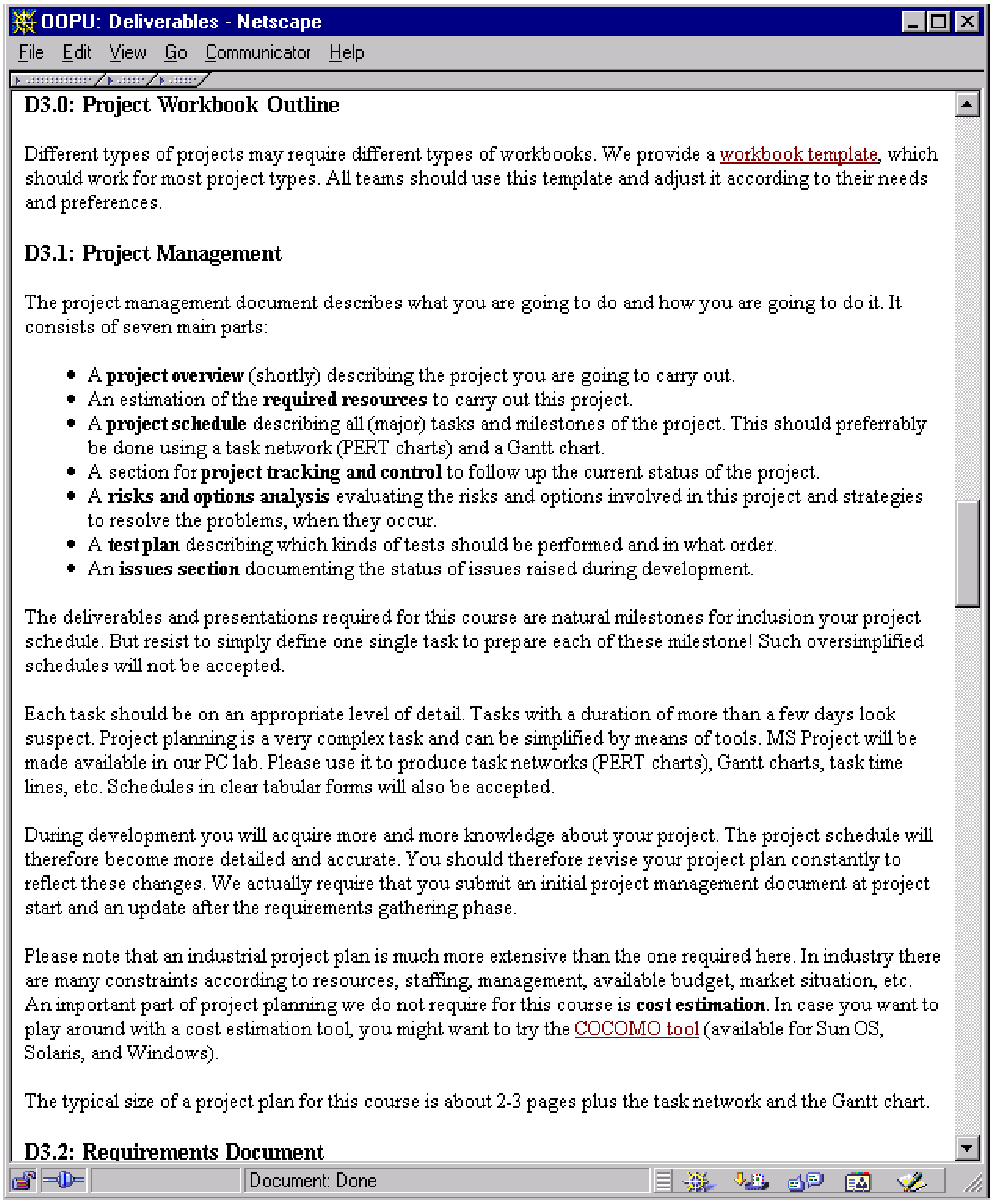
- Project Management
  - Project Overview
  - Required Resources
  - Project Schedule
    - Schedule
  - Risk Analysis
  - Test Plan
  - Issues
  - Meetings
- Requirements Document
  - Problem Statement
  - Use cases for the game Othello
  - Non-Functional Requirements
  - Requirements prioritization
- GUI Design
  - Guidelines
  - Screen Layouts
  - Screen Flows
  - GUI Mock-Up
- Object-Oriented Analysis and Design
  - Analysis
  - Design
- Implementation
  - Coding Guidelines
  - Prototype User Manual
- Testing
- Glossary

Last modified: 1998-11-06 11:33
Copyright © 1998, NoName

Designer, Publisher, Webmaster
Johan Tegman

Document: Done

Figure 4: Example of a student team's workbook.


OOPU: Deliverables - Netscape
File Edit View Go Communicator Help
Document: Done

D3.0: Project Workbook Outline

Different types of projects may require different types of workbooks. We provide a workbook template, which should work for most project types. All teams should use this template and adjust it according to their needs and preferences.

D3.1: Project Management

The project management document describes what you are going to do and how you are going to do it. It consists of seven main parts:

- A **project overview** (shortly) describing the project you are going to carry out.
- An estimation of the **required resources** to carry out this project.
- A **project schedule** describing all (major) tasks and milestones of the project. This should preferrably be done using a task network (PERT charts) and a Gantt chart.
- A section for **project tracking and control** to follow up the current status of the project.
- A **risks and options analysis** evaluating the risks and options involved in this project and strategies to resolve the problems, when they occur.
- A **test plan** describing which kinds of tests should be performed and in what order.
- An **issues section** documenting the status of issues raised during development.

The deliverables and presentations required for this course are natural milestones for inclusion your project schedule. But resist to simply define one single task to prepare each of these milestone! Such oversimplified schedules will not be accepted.

Each task should be on an appropriate level of detail. Tasks with a duration of more than a few days look suspect. Project planning is a very complex task and can be simplified by means of tools. MS Project will be made available in our PC lab. Please use it to produce task networks (PERT charts), Gantt charts, task time lines, etc. Schedules in clear tabular forms will also be accepted.

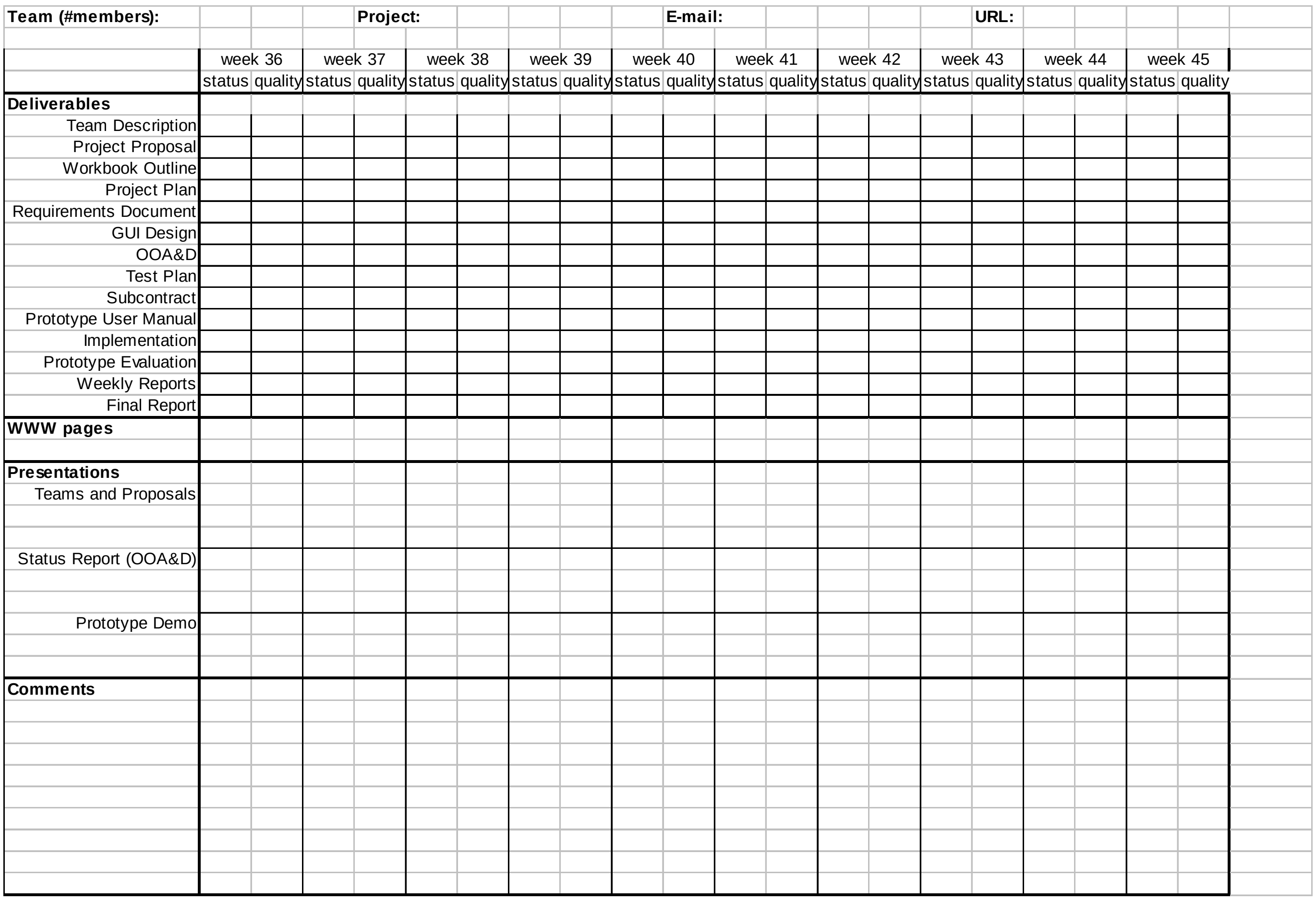
During development you will acquire more and more knowledge about your project. The project schedule will therefore become more detailed and accurate. You should therefore revise your project plan constantly to reflect these changes. We actually require that you submit an initial project management document at project start and an update after the requirements gathering phase.

Please note that an industrial project plan is much more extensive than the one required here. In industry there are many constraints according to resources, staffing, management, available budget, market situation, etc. An important part of project planning we do not require for this course is **cost estimation**. In case you want to play around with a cost estimation tool, you might want to try the COCOMO tool (available for Sun OS, Solaris, and Windows).

The typical size of a project plan for this course is about 2-3 pages plus the task network and the Gantt chart.

D3.2: Requirements Document

Figure 5: Excerpt from the Description of Deliverables for the course.



Team (#members): **Project:** **E-mail:** **URL:**

	week 36		week 37		week 38		week 39		week 40		week 41		week 42		week 43		week 44		week 45	
	status	quality	status	quality	status	quality	status	quality	status	quality	status	quality	status	quality	status	quality	status	quality	status	quality
Deliverables																				
Team Description																				
Project Proposal																				
Workbook Outline																				
Project Plan																				
Requirements Document																				
GUI Design																				
OOA&D																				
Test Plan																				
Subcontract																				
Prototype User Manual																				
Implementation																				
Prototype Evaluation																				
Weekly Reports																				
Final Report																				
WWW pages																				
Presentations																				
Teams and Proposals																				
Status Report (OOA&D)																				
Prototype Demo																				
Comments																				

Figure 6: Grading form.