An Immersion Model for Software Engineering Projects

Neville Churcher and **Andy Cockburn**
Department of Computer Science
University of Canterbury
Christchurch, New Zealand
neville, andy@cosc.canterbury.ac.nz

April 12, 1996

Abstract

Software development projects are an essential component of software engineering courses. They provide the opportunity for students to apply theoretical material and to gain valuable experience in an environment typical of the workplace. These benefits, however, are difficult to realise. We discuss strategies for managing final-year software engineering projects in order to optimise the balance between pedagogy, course administration, and time constraints. In particular, we advocate an "immersion" model for software engineering projects. The immersion model emphasises the commercial realities of software development including activities such as reverse-engineering of existing systems, extensive code re-use, team work, user-interface development, meetings with management, and oral presentations. Our experiences with the immersion model have been extremely encouraging with significant improvements in the quality of student projects.

1 Introduction

Teaching text-book theory of software engineering and user interface design, though valuable, provides students with a shallow understanding of the underlying issues.

We believe it is important for students to experience at first hand the application of theory to a 'real world' software system. Experience gained in group dynamics, teamwork, project management, presentation and technical writing provides students with skills that are valuable in the job market and in further study. The opportunities for imparting such experience within the confines of an undergraduate course are limited [4, 6].

The third year software engineering course we teach in the Department of Computer Science at the University of Canterbury is typical in that it has, as a major component, a software development project. In 1995 we substantially modified both the project structure and our management procedures. Successful results were achieved in terms of the quality of the work produced, the range of skills learned, and the administrative burden of management and assessment.

Three major factors motivated our reassessment of the software engineering project. First, we were concerned that standard assignment submissions, accepted as normal in most academic courses, do not encourage students to carry out iterative improvement of their work based on feedback from the lecturers. Second, we were dissatisfied by the amount of staff time spent on activities that provided little academic worth to the students. Marking the submissions associated with large team projects is a substantial burden on the academic staff. If students do not revisit the marked work there is little academic value in the staff's work. Third, the course time constraints place pressure on the scope of the software project. To reinforce the theoretical course content a software development project has to be sufficiently large and complex to require the management (and other) strategies described in the lectures, yet the time constraints limit the depth of student involvement. These three factors, if allowed, can form a vicious circle that results in frenetic work by students and lecturers, but without commensurate academic gains.

In this paper we discuss an "immersion model" for software engineering projects in which the academic staff play the role of project managers. Most formal assessment in the project is replaced by regular (approximately fortnightly) meetings with management. We outline some of our educational strategies and experiences, describe our current approach and compare it with some of the alternatives we have explored.

The paper is structured as follows. The next section outlines our current approach to coordinating the project, and briefly describes the 1995 project tasks. Section 3 summarises our primary learning objectives and identifies some of the advantages and difficulties of running a project in this manner. Our experiences with two previous models of project course structure are compared in section 4. Conclusions are presented in section 5.

2 The Immersion Model

This section describes the structure and management of our current project model which aims to immerse students in a realistic software development scenario. The 1995 project is briefly reviewed to provide an example of the model's application. The educational (and other) rationale behind our use of the immersion model is presented in the section 3.

2.1 Structure and Management of the Project

Approximately forty students work in self-selected small groups (normally three per team) throughout the year to produce a functioning and documented piece of software. The project consists of four overlapping tasks, as follows:

1. Analyse and document an existing software system. The focus is on "reverse-engineering" the design of the system.

2. Modify and extend the functionality of the system. Students are required to make design decisions, re-use software components, and implement extensions.

3. Design an improved user interface for the system. This stage focuses on design rationale and rapid prototyping.

4. Implement the interface. The teams package the entire system and its documentation for release on the Internet.

The students are informed of this four-stage outline at the start of the year, but the detailed specifications of each stage are revealed individually throughout the year.

There is a single formal submission at the end of the year: the delivery of the complete extended software, interface, and companion documentation. On-going feedback, motivation, and direction is given to the teams at fortnightly meetings at which the current course lecturer plays the rôle of project manager.

At each meeting the team presents a progress report and discusses progress made, problems encountered, alternative solutions, design rationale, and so on. Groups are responsible for developing their own formats for documenting meetings, design decisions, and other records.

2.2 Applying the Immersion Model

The 1995 project involved the `jgraph` package [7] which according to its `man` page

> "...takes the description of a graph or graphs in the standard input, and produces a postscript file on the standard output. Jgraph is ideal for plotting any mixture of scatter point graphs, line graphs, and/or bar graphs, and embedding the output into LaTeX, or any other text processing system which can read postscript."

Figure 1 provides an example of `jgraph` instructions and the resultant graph.

Task 1: Technical documentation

The first project task required the students to produce technical documentation for `jgraph`. To provide context for their work, the students were informed that they would be extending the functionality of `jgraph`, and that they would ultimately design and implement a graphical interface to their extended system. The regular meetings with 'management' allow the course supervisors to ensure that the groups remain focused on appropriate tasks.

```
newgraph
title : Sample Graph Title
xaxis min 0 max 10
label : The X Axis Label
yaxis label : The Y Axis Label
newcurve
marktype circle
fill 0
linetype solid
pts 0 0 2 4 3 9 4 16
label : Line 1
newcurve
marktype xbar
fill 0
linetype none
pts 1 5 9 2 5 5
label : Line 2
legend on
```

Figure 1a: `jgraph` instructions.

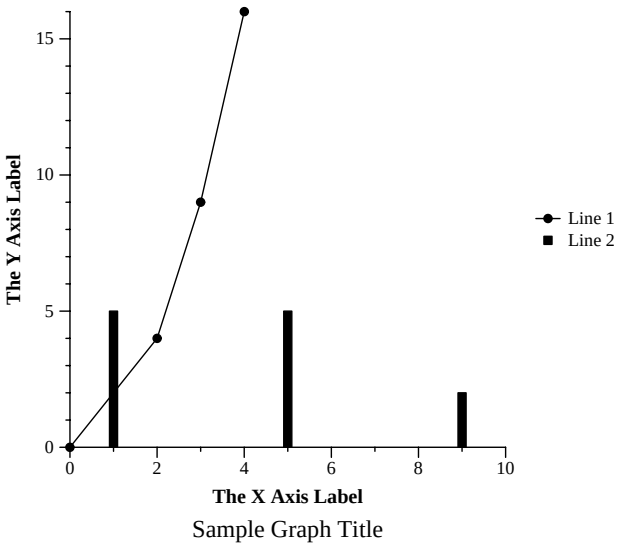

Figure 1b: The resultant graph.

Figure 1: Preparing and viewing graphs with `jgraph`.

This task tests a variety of skills based on the ability to read, understand and abstract important elements from C code. The code is undocumented and uncommented, packaged into eleven `.c` files and three `.h` files. Ultimately, the test of the students' documentation is its usefulness in subsequent tasks involving modification of the software. We avoid precise specification of document structure but state that the object is to add 'value' by using the system representation techniques covered in lectures (such as those of the standard structured techniques) to convey the recovered design. Students are advised that discussion of algorithms, data structures, use-cases etc. should be appropriately cross-referenced both within the documentation and to the code itself, and students are guided towards appropriate support tools.

Task 2: Design and implement `jshell`

Task two required students to develop an interactive shell, `jshell`, which would allow exploratory and incremental development of graphs through a command line interface. Without this extension `jgraph` requires an entire graph description (such as that shown in figure 1a) to be input without any interaction. Consequently, each modification to a standard `jgraph` graph requires a time consuming cycle of edit-compile-view.

The task requirements are deliberately imprecise, offering no instructions on how the modifications are to be made. The aim is to encourage students to detect, confront, and overcome issues requiring design and coding decisions.

The teams rapidly diverged in the focus of their designs, and we encouraged the diversity provided that the proposed solutions were documented with appropriate design rationale, and that they were achievable in the available time. The regular meetings provided an opportunity to moderate progress and design directions.

Some of the major categories of design issues addressed by different groups included the following:

Architecture Some groups opted to modify the `jgraph` code directly. Others constructed a shell by implementing a front-end which communicated with the original `jgraph` via pipes.

Compatibility Some groups felt it important to ensure that their shell could process `jgraph` scripts directly. Others sacrificed backwards compatibility for increased functionality.

Error handling Those who assembled and passed entire `jgraph` scripts to `jshell` faced a variety of problems detecting and reporting errors in the script, broken pipes

and so on. Those who processed individual commands in the shell command loop also faced challenging problems.

Task 3: Design a GUI for `jshell`

Task 3 required the design of a non-functional prototype user interface.

Before considering any graphical interface elements, students were required to identify a system context, stating who the users of their system would be, and identifying typical usage scenarios. Depending on their intended user base, the emphasis on various usability properties (such as those captured by the principles in [1]) varied substantially across the project teams.

Students were also required to produce a rough paper and pencil sketch (or "storyboard") of their design, including cut-outs for system components such as pull-down menus or transient dialogue-boxes [8]. The emphasis was on rapid modifications to the interface. "Smart" teams were admonished if they produced screen-dumps of executable code. At the project meetings the storyboards were used to identify potential usability problems, and to discuss design alternatives.

Task 4: Implement the GUI and Full System

Students were then required to implement the system designed in the previous task. The GUI was implemented using the Tool Command Language (Tcl) and the Tk widget set [5] and was used to drive the character based `jshell`. A sample session with one of the best completed applications is shown in figure 2.

Tcl is an interpreted scripting language which is powerful and easy to learn. It is also possible to embed Tcl interpreters in C or C++ programs.

Tcl/Tk is an extremely effective addition to our software engineering project. Previously the size and complexity of interface toolkits (such as Xt and Motif, SUIT, InterViews, and so on) have almost prohibited the inclusion of GUI development within the time constraints of the project. In past projects we have experimented with the SUIT graphical toolkit and with the character-based terminal-independent package `curses`. Both experiences were disappointing: the students struggled to overcome the complexity of the SUIT toolkit, and the crude interfaces produced by `curses` failed to motivate the students. In contrast, Tcl/Tk abstracts most of the complexity of GUI development, allowing students to quickly build high-quality interfaces. The polished look and feel of their systems is also a strong motivator for the students.

3 Learning Objectives and Rationale

This section describes some of our learning objectives across four, somewhat overlapping, arenas of system development: software issues, user-interface design, team-management, and information presentation. Rather than present a complete list of our educational goals, only those pertinent to the immersion project approach are discussed.

In general, students benefit in that they have a chance to receive feedback early enough to avoid wasting time on 'red herrings' and to learn from small mistakes rather than suffer the consequences of large ones. The frequent meetings have led to a closer relationship with staff, encouraging students to view meetings as an opportunity to discuss ideas rather than an oral examination. Administering the meetings is demanding on staff time, but these costs are offset in two ways. First, there is no formal marking of submissions other than the final submission at the end of the year. Second, as the student teams have an allocated appointment time with the managers, they tend to 'drop-in' with problems less frequently than they have with previous models of project coordination.

3.1 Software Issues

A primary objective of the project is to disabuse students of the impression, reinforced by several years of throw-away programming assignments, that software is a disposable item. Our decision to base the entire project on a piece of existing software (such as `jgraph`) is motived by several important educational objectives.

- Examining and modifying code written by others is a common task in the workplace—particularly for entry level employees.
- The difficulty of comprehending code written by others is a powerful way to reinforce

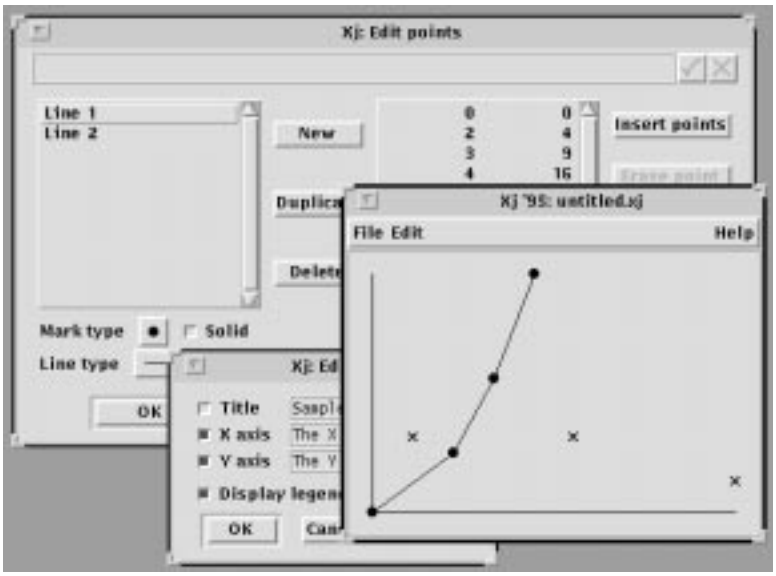

Figure 2: One team's final system.

the importance of code quality and documentation.

- We believe that studying existing code nurtures empathies for those who will modify the student's own code. Students are advised that their code may be evaluated by peer teams and that it may form part of the project exercise in subsequent years.

- Modifying existing code fosters an awareness of code evolution.

Additionally, experience with graphical user interface design is a common demand in the workplace. By including a GUI development component in the project, students gain experience in programming in an event-driven style.

3.2 User Interface Design

Our Software Engineering course includes a major lecture component on Human-Computer Interaction (HCI). Like many theoretical Software Engineering topics, theoretical HCI is often viewed as "obvious" by students until they experience it for themselves. We contend that the immersion approach is particularly appropriate for a project with a GUI development component. Two of our arguments are given below.

First, iterative design is a cornerstone of good user interface development [2, 3]. Continuous project meetings allow several design iterations with techniques such as storyboards [8]—students delay commitment [9] to one particular interface solution, and have the opportunity to consider and discuss alternatives. Rough sketches of the students' proposed solutions allow rapid improvement in the designs long before coding starts. The meetings use the rough storyboards as conversational props, with a focus on good design. In contrast, our past experience has shown that, when asked to for a formal submission of a "rough" storyboard, students produce polished computer-generated storyboards that focus on presentation rather than on design: years of assessments have stressed the importance of neat presentation for all submissions.

Second, once the project groups' storyboards are finalised, the students can begin coding the interface. All deviations from the storyboard must be documented. In our experience, when students do not have a concrete target for their user-interface design (such as a storyboard), their interfaces tend to be *caused* by work on the functionality rather than *designed*.

3.3 Team Management

The educational value of team-projects and peer-learning are well known [10]. There are many standard requirements in team-projects such as our requirement that the overall task be sufficiently large to demand continuous work over the year. Our concern here, however, is how the immersion approach fosters effective group-work within project teams.

It is essential that the students are, from the beginning, made aware that the project is not amenable to last minute assaults. A team size of three ensures that communication issues must be addressed, while being small enough to avoid large communication overheads. Regular meetings between the team and their 'manager' ensure that groups are continually accountable for their progress: an analogy between pay and grades is powerfully received by students.

Groups are encouraged to discuss internal problems such as personality clashes before they escalate to unmanageable proportions. Such situations have been particularly problematic under previous project coordination styles, but are nipped at the bud when raised at the fortnightly meetings. The meetings also provide staff coordinating the course with an early warning system for the detection of potential drop-outs, personality clashes, and unequal individual contribution. The increased student contact helps us balance the conflicting aspects of individual assessment for group work.

Team selection is normally negotiated by the students themselves, but some 'matchmaking' is usually required. The resulting improvements in productivity and group dynamics outweigh the slightly more realistic situation of assigned groups. The fact that groups consisting of academically undistinguished but highly motivated and organised students often out-perform groups who look better 'on paper' provides a valuable lesson for all concerned.

To further reduce team problems such as inequitable group work, in 1996 we have introduced progress reports which all team members must submit independently to the course supervisor by email. The students' responses are requested, through standardised email templates, prior to each round of project meetings. While easy to administer, these messages provide additional information about individual contributions and potential disharmony.

3.4 Information Presentation

Few courses in Computer Science provide the opportunity for iterative improvement of written material. Normally, a submission is made, and it is graded. Although the student may attend to the marker's comments, it is rare that another pass is made through the same material.

We strongly believe in the educational value of modifying written work from an editor's comments. In our project the fortnightly meetings provide an opportunity to review the latest draft of the documentation. Additionally, all groups are required to make a copy of their documentation available on-line for the manager's perusal.

Oral presentation skills are also promoted. The meetings are intended to provide a relatively stress-free environment for gaining experience in technical discussion. Additionally, each project group makes a formal presentation, or 'sales pitch,' of their system to their peers and managers at the end of the year.

4 Other Models of Project Organisation

In this section we briefly outline two models for project course structure that were used at Canterbury prior to our experiment with immersion. All three approaches make the assumption that re-use is an important component. We have not considered models where the products are predominantly built from scratch.

4.1 Big bang

In the Big-bang approach tasks are introduced at the beginning of the course. Students work in groups which are entirely responsible for their own management. The project deliverables are submitted for assessment at the end of the course.

There are many problems associated with this approach, but one major difficulty is the failure of some groups to begin work early enough to achieve satisfactory results. To some extent this makes the project self-assessing as those who lack managerial skills are penalised (often severely). Although this may appear convenient, our educational objectives do not include punishing those students who do not currently possess particular skills and rewarding those who do. Rather, we should provide all

students with opportunities to learn, improve and demonstrate these skills.

4.2 Milestones

To ease the problems of the big bang approach we introduced several milestones within the project. The idea is to 'buy' time-slices from students who work in a climate of inexorable internal assessment. Generally, milestone points correspond to the availability of one or more deliverables for assessment. Individual tasks may also be timed so that they overlap, allowing for the possibility of introducing changes in requirements during the course of the project.

While we have found that this technique ensures that some progress is made earlier in the year, an equally undesirable side-effect is introduced. Students perceive the project as a series of small assignments rather than as a large piece of work—leading once again to last-minute poor quality work. Consequently, the milestones have a tendency to become 'inch pebbles' to the students while the assessment load generated makes them appear millstones round the necks of staff! The considerable effort invested in marking is largely wasted as students have moved on to other tasks by the time marking is completed and little opportunity is available for them to learn from their mistakes.

5 Conclusions

The sample project described in section 2.2 emphasises our message that software is constantly evolving. Tasks involving design recovery, documentation for change, and adding a GUI to an existing application are natural companions to the more traditional design, coding, and testing activities.

It is important to choose a topic that can capture student interest, and much suitable software such as `jgraph` is available via the Internet. The Tcl/Tk language has many features which make it ideal for use in projects. It is easy to learn, powerful, very good for controlling other programs, and—most important of all—fun! This makes it feasible to attempt to develop X11 GUI software within the tight timeframe typical of software engineering courses.

We have been encouraged by our results. The quality of the end products—the systems and documentation that the students produce—has increased. Our claims of success are supported by results of an evaluation exercise conducted in 1995 by the University of Canterbury's Educational Research and Advisory Unit (ERAU) in addition to the normal course surveys. Although the project management demands on the staff remain about the same, we feel that our time is much better spent. Additionally, the incidence of intra-group problems has fallen, primarily because problems were detected earlier.

We are continuing to use and refine the immersion approach, and we encourage comments from others involved in project-oriented courses. The appendix below directs interested readers to an archive of information on our 1995 project.

Appendix

The full set of task requirements for the 1995 `jgraph` project and tar files containing some of the student submissions are also available from the same address can be accessed from `http://www.cosc.canterbury.ac.nz/~neville/project314.html`

References

[1] Dix, A., Finlay, J., Abowd, G., and Beale, R. *Human-Computer Interaction.* Prentice Hall, 1993.

[2] Gould, J., Boies, S., and Lewis, C. Making usable, useful, productivity-enhancing applications. *Communications of the ACM 34*, 1 (1991), 74–85.

[3] Gould, J., and Lewis, C. Designing for usability: Key principles and what designers think. *Communications of the ACM 28*, 3 (1985), 300–309.

[4] Ibrahim, R., Ed. *Proc. 8th SEI CSEE Conference* (New Orleans, LA, Mar. 1995), vol. 895 of *Lecture Notes in Computer Science*, Springer-Verlag.

[5] Ousterhout, J. K. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Massachusetts, 1994.

[6] Pierce, K. Rethinking academia's conventional wisdom. *IEEE Software March* (1993), 94–99.

[7] Planck, J. Jgraph — a filter for graph plotting to postscript. Anonymous ftp to princeton.edu pub/jgraph.Z, 1992.

[8] Rettig, M. Prototyping for tiny fingers. *Communications of the ACM 37*, 4 (1994), 21–27.

[9] Thimbleby, H. *User Interface Design*. ACM Press, Addison-Wesley, 1990.

[10] Thorley, L., and Gregory, R., Eds. *Using Group Based Learning in Higher Education*. Kogan-Page, 1994.