Introducing Formality into the Software Development Process: Experience with a Software Project Course

Robin A. Nicholl
Department of Computer Science, University of Western Ontario, London, Canada

Abstract

The past five years have witnessed an enormous interest in the use of more rigorous methods of software engineering. Despite numerous published formal specifications of quite complex software systems and occasional reports on the use of formal methods in advanced software development organisations, these methods remain controversial.

One testbed for examining these methods is a software project course. This paper describes how the software process model used for such a course is now at the point where students are required to adopt an entirely formal approach to the development of their software projects. The software process model also strictly separates the responsibility for building a software system from the responsibility for specifying it; a separation that is facilitated by the use of formal notation to achieve precise communication.

This paper describes this software development process and assesses it both from an engineering and from an educational perspective. It also identifies areas where the effectiveness of formal methods must be further increased.

1 Introduction

More than 20 years have passed since the term software engineering was introduced, yet it is still comparatively rare for the software development process to be directly compared with the development process of other engineering products. For much of this time, but particularly over the past 5 years, there has been enormous interest among members of the software engineering research community into the use of formal mathematical methods in software development. This interest extends beyond the so-called "ivory-tower" environments of universities and research laboratories to a number of commercially interested software development organisations. Among these the IBM experience with Cleanroom software engineering is considerable; Mills [4] gives a very readable overview of this work. It is sometimes argued that this is simply a normal part of the development of programming into an engineering profession; after all engineers always use the relevant mathematics to help ensure the correctness of their designs. Nonetheless the use of formal methods in software engineering remains extremely controversial, particularly since there is still little detailed information available on their practical application. Of particular concern is the lack of information regarding the impact of greater formality on the process of software development.

When we compare the development of software with the development of other engineering products we find another big difference. Software is usually specified, designed, coded and tested by a single organization. A more common engineering practice is for one organization to prepare the product specification and for a different organization then to do the detailed design and construction. The significant impact of this simple separation of responsibilities is discussed below.

The traditional model of the software development process is the waterfall model. Despite the evident weaknesses of this model it remains the standard against which other process models must be compared. The waterfall model has the characteristic that once each stage of the project has been completed and checked it will not be repeated (at least not until after the software is in use and some changes become necessary). This refusal to repeat a previous stage can lead to one of two possible difficulties.

1. A good quality implementation is developed that takes advantage of information gained during later stages of a project, even when this conflicts with decisions made during earlier stages. However, the documentation recording development of the software is no longer consistent, and this is likely to become a problem during maintenance. In extreme cases this may even cause such severe management problems that the development project itself may fail.
2. A poorer quality implementation is developed that refuses to take advantage of information gained during later stages of a project if this conflicts with earlier decisions. Although the documentation is now consistent, the poor software quality is likely to become a problem during maintenance and will probably also be a problem while the software is being used.

On the other hand, it is obvious that a too ready willingness to repeat previous stages can also render a development project unmanageable, particularly with respect to meeting scheduling constraints.

This paper describes a software development process that is designed around the use of formality. It is inspired by Hoare’s view of programming as an engineering profession \[1\] and by the common engineering practice of separating the responsibilities of specification and construction. This new process is compared with the waterfall model in the light of the author’s experience with both models in six years teaching (and hence, in part, also managing) software project courses. For reasons explained below, this process model is fairly rigid in that it discourages the repetition of previous development steps. However iterations can still occur within each step of development, as the developers attempt to gain a complete understanding of the system and its required behaviour.

2 Influences on the Process

Though this section is largely anecdotal, it provides important background information for understanding how the proposed process model has come into being. The model has been influenced by three different sources, in addition to the standard process models \[3\].

1. Discussions with engineers familiar with other engineering disciplines, particularly civil engineering.

2. Literature on the use of formality in software development, particularly \[1\] for project organization and \[5, 9\] for technical details.

3. Past experience in using and teaching the waterfall model.

The discussions with the engineers largely confirmed the basic project organization suggested by Hoare \[1\], but described a different division of responsibilities among the individuals involved. Past experience was used to guide the assignment of milestones and deadlines, to increase the likelihood that each person would be involved in a successful implementation and at the same time allow document quality to be assessed critically. This reflects a balance between the various educational and engineering factors involved.

It should be noted that characteristics of student programmers differ considerably from professionals, so that the factors presented here may not all apply directly to commercial practice. Furthermore useful lessons have sometimes been learned as the result of following procedures that would clearly be unacceptable in commercial projects.

Adaptation of the waterfall model

This model was chosen to suit the type of deadlines normally associated with course work: (1) the work must be submitted (and preferably completed) by a fixed date and time; (2) the quality of the work submitted will be assessed; (3) after the deadline there will be no opportunity to improve the quality of the work (nor is there any incentive or reward for doing so).

For the first few years of the course students were given module specifications to be used in doing module design, implementation, testing and integration. For the past 2 years however, the projects have been made much more straightforward and instead students have done program specification, design, implementation, testing and some maintenance. In these circumstances all integration problems are handled within a single group. Four separate groups of people work on each of several projects: on the program's specification (c. 2 people), design and coding (c. 3 people), testing (c. 3 people) and maintenance (c. 3 people). To ensure that each group is assessed only on the quality of their own work, the assessment of one group's work is made without reference to the work done by other groups on earlier or later stages of the same project. An attempt to separate design and coding was unsuccessful, so instead there is a deadline for a design review to be conducted before coding can begin. It is attended by the leaders of all the other groups involved in that project.

This format has not caused too many problems: some of these problems and attempted solutions are discussed below. Most problems have arisen with projects that were not familiar to the students. For such projects adherence to the waterfall model had the predictable effect that students were prevented from recovering from mistakes made in the early stages of the project. Where the problem domain was more familiar, the waterfall model proved quite satisfactory. I believe this is in agreement with experience in industry.
Constant changes of personnel

High staff turnover is known to be a genuine problem experienced by software organizations. My preference for having students change the membership of groups is not an attempt to simulate this unpleasant reality, but rather to encourage greater awareness of the need for documentation to be well-written and comprehensive. When none of the members of a design group were involved in writing the specification document for their design, they inevitably become more dependent on what it actually says. They quickly become aware of omissions, ambiguities and inconsistencies. In previous projects, when group membership did not change, less attention was paid to document quality. Groups who were competent programmers expected to be able to build reasonable software even if they were not too careful about recording their design (and even specification) decisions accurately. Thus even a badly written specification might give rise to a good design if the authors of the specification also built the software and understood well enough how it should operate.

This approach of forming completely new groups for each stage of the project is now being used for the third year. It came as a great surprise to me that in both previous years students have expressed an almost unanimous preference for changing group memberships, rather than remaining in the same groups throughout the project.

Separate quality assurance groups

It is quite clear that the implementers of a program will have some difficulty adjusting their mentality to really check the program thoroughly for errors (as reported, for example, by [7]), but it is only by comparing the alternatives that I have come to appreciate just how much difference this makes. I suspect that this difference is more marked among students than among professionals.

In my experience, implementers do a much less thorough job of checking for errors in their own code than do a separate group of people who are given the responsibility of finding errors in the code. I have seen separate testing groups do extremely thorough checking of others' code and thus find errors that I cannot imagine the implementers finding for themselves. In some cases there have been confrontations between the 2 groups, and the implementers do indeed become defensive and protective of their code. The implementers' "escape hatch" discussed below bears evidence to the effectiveness of the separate testing groups.

Groups must face the problem

Leaving aside the familiar arguments over its ambiguity and imprecision, natural language has one characteristic that lessens its usefulness for writing software documentation: it is possible to write a quite good informal description of a piece of software, making that software sound attractive, and still not really understand what the software has to do! When all documents (other than the code itself) are written informally, and even if a "structured" natural language such as pseudo-code is used, it is possible that no one ever faces the problem squarely before the implementers try to start writing code. The hard parts of the problem can be left hidden in the informal descriptions. It is noticeable, but perhaps not surprising, that this problem has occurred most often in "hard" projects; however these are the projects where it is most important to face the problem early.

This is a disastrous state of affairs, which has occurred in my course despite my efforts to prevent it. It leaves the implementers with far too much work to do, much more than simply coding. Although there are other ways to mitigate this problem, by stricter review and inspection processes for example, the need for very strict deadlines imposed by course requirements (and the resulting limited opportunities to use feedback) leads me to conclude that it is essential to prevent the problem by reducing dependence on informal text and increasing dependence on formal notations. It is much more difficult to avoid facing a problem when the results of your analysis of the problem must be expressed in a formal notation and even more difficult to hide the fact that you have done so.

The implementers' escape hatch

Implementers can be required to develop code of good quality and to suffer fairly severe penalties for failing to do so. A thorough testing process can make it quite likely that many of the errors present will be discovered. This combination creates a situation in which the implementers feel considerable pressure. Although this pressure is intended to ensure the quality of the implementation, there is another escape. Informal specifications are not precise, especially when written by people of limited experience. It is frequently possible for the implementers to argue that a particular eccentric output from their program, considered by the testers to be an error, is not actually forbidden by the specification and thus needs no correction. ("It's not a mistake, it's a feature!"). Although such an argument is not at all appealing, it has a certain validity since it shows a clear error (imprecision) in the specification for which the implementers probably bear no responsibility. Thus while we might hope for a more reasonable attitude on the part of the implementers, yet we must also work much harder to ensure the adequacy of our specifications.
Summary
Whenever projects have involved problems that are sufficiently straightforward, the waterfall model has proved quite successful. However, it is not always easy to predict how straightforward a project will be, and the ease with which problems can remain hidden in informal text means that the waterfall model can lead to severe project failures on complex problems that are not faced squarely. Document and code quality is improved by insisting that each stage of the project (including error checking) is handled by a separate group of people.

3 A Process Model Based on Engineering Practice
This process model is chosen to support the use of greater formality in software development. This is expected to be consistent with the goals of improved quality in software documentation and of operating under strict scheduling constraints with only minimal opportunity for making changes after each stage is completed.

A major distinctive characteristic is the view that there are 3 (not just 2) types of party involved in the technical aspects of a software development project:

1. **Client(s).** Clients are the people who commission the project. They probably also pay for the project and they either are the intended users of the software or they represent the intended users. The success of the project is largely determined by the satisfaction of the clients with the delivered software.

2. **Project Engineers.** Project engineers are the people with primary responsibility for making sure the project is successful. They analyse the clients' requirements to prepare a detailed specification and they advise clients when certain requirements appear overly elaborate, expensive or risky. They represent the clients' interests in dealing with the builders of the software, ensuring that the implementation has the required quality characteristics, is consistent with the specification and is proceeding on schedule. Their approval is required during all stages of the implementation, and they are responsible for everything they approve.

3. **Software Builders.** The builders of the software have two tasks: submission of designs for approval and implementation of approved designs. Designs must be demonstrably consistent with the system specifications. Implementation is done to an agreed schedule, with scheduled (and unscheduled) inspections carried out by project engineers. The details and methods of design are approved before implementation begins.

Variations on this basic division of responsibilities include using a fourth party to inspect the implementation and having project engineers do the preliminary design themselves.

Following normal engineering practice, the project engineers and software builders should belong to separate organizations, to avoid any conflicts of interest that might otherwise arise. The builders act to efficiently construct the software system to specifications and the engineers ensure that the clients' requirements are captured by the specification and that the software has the desired quality characteristics. This keeps development costs down without compromising design quality.

Proposed development process
Let us now consider the details of the development process that has been outlined in the discussion above.

1. **Project engineering.** This is the first stage in developing a software system, and may require construction of prototypes to obtain all the necessary information.
   - Analyze requirements.
   - Write specification, including initial drafts.
   - Write (initial) user manual.

2. **Bidding - preliminary design.** This is the second stage in developing a software system, carried out by one or more groups separate from those who wrote the specification.
   - Distribute specification.
   - Submit designs.
   - Select design.
   - Approve final design.

3. **Building.** This is one side of the third stage in developing a software system, in which the final approved design is implemented.
   - Design detailed algorithms and data structures (by refinement of preliminary design).
   - Code in a particular programming language.
   - Analyze execution behaviour: performance and error checking.
   - Add implementation specific details to user documentation.
4. Inspection. This is the other side of the third stage in developing a software system, in which the implementation is monitored.
   - check that refinements are done properly.
   - approve refinements and code for correctness, speed, space, user friendliness, etc.
   - ensure that work is done to approved standards and by agreed methods e.g. statistical testing.

5. Delivery. This is the fourth and final stage in developing a software system, in which the software is delivered to the client.
   - provide final user documentation and code.
   - participate in acceptance tests.

Project engineering

The project engineering task is to analyse the clients' requirements, to advise the client on the consequences of certain decisions and to prepare a detailed and comprehensive specification. This specification must include (see [3] for example):

1. statement of required functional behaviour.
2. statement of external interfaces: user interface, device and file interfaces, interface to other software, e.g. operating system.
3. statement of required performance characteristics.
4. constraints on the system and system goals.

Since the project engineers must take responsibility for the software delivered, it is essential that the specification be of the highest quality and that the client understand that it will be both difficult and costly to make changes once building begins. Not only must the engineer understand the constraints on the system and system goals, but also the problems of design, implementation and quality control. It is essential to be realistic when deciding which requirements will appear in the specification. (See [2] for a frank discussion of a project failure due to overambition.) Finally, to ensure that the specification addresses the real problems of the client, it is essential that the specification should be both detailed and precise. The engineer must ensure that the client understands the consequences of decisions made about the software. Thus it appears inevitable that project engineers will tend to be conservative in specifying software requirements: it is their responsibility to ensure that the software will operate to the client's satisfaction.

For the reasons discussed in the previous section, it appears to be essential for both functional behaviour and external interfaces to be defined as precisely as possible, preferably in a formal notation. Some of the burden of preparing this formal notation is relieved when the same definitions are reused. Such reuse is quite straightforward when dealing with interfaces to devices and to other pieces of software - simply insert a reference to the document specifying the relevant interface. The same method can also be used in situations where a user interface standard is being used and where some aspects of the problem domain already have a standard definition.

Bidding - preliminary design

A major feature of this process model is the separate responsibilities of project engineers and program builders. The detailed process model identifies a competitive bidding process to select a design for a specification. Naturally, there may be circumstances in which only a single group is invited to submit a design, but the conflicting interests of project engineers ("meet the clients' expectations") and program builders ("get the program finished quickly") dictate that these two groups must be different.

I have used the following approximation to the traditional tendering process for bidding for the specification:

1. distribute copies of the specification
2. submit designs
3. select design
4. give final approval to design and specification

For this process to work the specification must be comprehensive, precise and clear. The use of suitable formal notation, reinforced by well-written explanations can help considerably. This is the basis for the use of the Z notation to write specifications such as [6], but the argument itself is independent of any particular formal notation.

One essential characteristic of the design is the explicit statement of connections between portions of the design and portions of the specification. Obviously this should include at least the fact that a particular section of the design is there to address a particular set of sections of the specification. Informal documents may not allow us to demonstrate any more precise connection than this between the two forms. Formality, on the other hand, allows us to state some precise mathematical relationship between parts of the two documents, e.g. that a particular logical property in the specification is implied by the postcondition of a particular procedure present in the design.

In principle we can imagine checking that every formally expressed property of the specification can be formally shown to follow from some formally expressed properties of
the design. In practice, the more properties we can check in this way, the more confidence we will have that the design is consistent with the specification. This provides a great incentive for ensuring that a significant degree of formality is used in specifications and in designs. This should also be considered when making the selection of which design is to be implemented.

Following the selection of a particular design to be implemented, it is necessary to make any final corrections or other changes to the specification and/or design documents that the bidding process has shown to be necessary. The result of the bidding process is thus an approved design for the specification. Naturally it is the client's final responsibility to approve the selected design, but it should normally be the case that the client would agree to the selection.

Building the software
Following approval of a design we would expect the process of implementing that design to be fairly straightforward. The building plan will indicate who will build the software and following what procedures. The steps to implement the design are predictable:

1. refine the design to code.
2. express the code in a particular programming notation.
3. run the program: check performance and look for errors.

Current practice shows that step 3, under the name "testing" (but really including also "debugging"), is enormously costly and time-consuming. We expect more emphasis on steps 1 and 2 to result in a reduced cost in step 3. The terminology "checking for errors" more accurately describes the role of step 3 and more strongly suggests that it is inappropriate to have to devote significant resources to this step. Concentration should thus be directed to step 1 in particular, where the process of refinement must be checked carefully. This checking is done most convincingly if the designs are recorded formally and if the correctness arguments used during the refinement have been recorded. This familiar argument carries greater weight in this situation, where the bidding process has been used to ensure the effective use of formal notations in the design to be refined.

This process has the additional distinctive characteristic that it is subject to periodic inspections by an independent party - a project engineer acting on behalf of the client. This inspection is done in addition to any steps normally taken by the builders of the software to control software quality. Note that the process of checking for errors in the implementation remains, though it is less prominent and different in nature from testing as practiced in the waterfall model.

Inspecting the construction
The confidence that the client will be happy with the delivered software comes from the attention to detail paid during preparation of the specification and from the careful scrutiny of the submitted designs. However, the process of building the software is crucial to ensuring that the earlier work pays off.

Inspection of any engineering project is concerned with the construction process as well as with the results of that process; inspection of a software project is no different.

1. Inspectors must be able to examine calculations done in refining "portions" of the design. Deadlines should be established for completing certain refinements, and inspectors must approve these refinements when they have been completed satisfactorily.
2. Coding should not begin until the inspectors have approved the refinement from selected design to detailed algorithm. Inspectors should also approve the methods used to encode detailed designs in a programming notation and to check that this encoding is done correctly.
3. Inspectors monitor the results of error checking done on each piece of code, possibly submitting pieces of code for independent quality assessment. If some piece of code is found to contain any (or more than a certain number of) errors, the inspectors should insist that it be re-designed, re-coded and re-checked. The inspecting organization will be held responsible for the effects of any errors not eliminated.

Although it is clear that conflicts may arise between the inspectors and the people building the software, they have a common interest in ensuring the construction of a good quality piece of software. Even though project engineers and inspectors have responsibility for guaranteeing the quality of the software, the reputation of the software builders also depends on the quality. This fact may sometimes lead the builders of the software to suggest improvements to a flawed design or even to a flawed specification.

Delivery
Once built, the project engineers will deliver the finished software and its documentation to the client. The client may then choose to perform additional "acceptance tests" on the software, possibly with external assistance. It should be rare to find errors at this stage, and if errors
Some compensation should (probably) be made to the client. This should also happen if errors are discovered while the software is in use.

Documentation available to the client before delivery includes the specification and the approved design. It may also include a preliminary users' manual. On delivery, additional documentation will include final user's documentation (one or more manuals) and instructions for installation and operation. It may also include the program (source) code and a warranty setting out any restrictions on the use of the software and the liability of the engineers in the event of failure: failures including both deviations from the specification and damage caused by the software in normal use. The engineers have some professional responsibility for ensuring that the software will still respond reasonably safely even if not being used in accordance with the restrictions. Such obligations have generally been set out in law (or otherwise codified) for other engineering professions.

4 Engineering Concerns and Educational Concerns

As described in this paper, formality has been introduced into the software development process for a mixture of goals: some of these goals are educational while others can reasonably be described as engineering goals. The purpose of this course is to introduce students to a more professional attitude to software. They must meet the (fairly vaguely stated) requirements of a "client", each step of their work is open to review and evaluation, they must cooperate with other people and they must write and depend on documents to provide answers to detailed questions that they might otherwise prefer to answer for themselves.

Some of the factors considered in preparing a course to fulfill this purpose were:

1. Realism. Students should understand that these are important practical problems, even though the way we approach these problems in a university course is not all that realistic.

2. Effort. This is just one of many courses the students are taking, it is inappropriate to demand a disproportionate amount of effort to be spent on the project. Parnas [8] specifically suggests that "significant" projects should really be done outside an educational setting, perhaps in a co-op or intern year.

3. Judgement. This is not a subject where all the answers are clear-cut. Students must learn when it is appropriate simply to find some technical solution to a problem and when (and how) they should make trade-offs between different solutions. In many cases there is no "right answer"; many students find this difficult to adjust to.

4. Group interaction. Some individuals have very bad experiences with particular group members. This can happen when a group includes members with particularly strong or weak personalities or members who are unwilling or otherwise unable to do their fair share of the group's work. While some of this can be justified as being realistic, there is no educational advantage in letting it go too far.

5. Deadlines vs. quality. The university calendar imposes fixed bounds on the duration of the course and hence also on the project. Departmental policy insists that assignment due dates be announced at the start of the course and strictly adhered to. This takes away a lot of flexibility as far as project deadlines are concerned. Since there is thus little opportunity to fit feedback into the development process, it is all the more important that documents be of the highest quality, and there is considerable educational value in having the students aim for this. Although this means that projects are smaller, it still allows interesting problems to be investigated.

6. Individual evaluation. This problem is not really dealt with adequately. At present evaluation is entirely on the basis of work submitted by groups - all members of a group are given identical evaluations. One step in the direction of individual evaluations is to ask for the names of individual authors to be attached to documents and portions of documents. My experience is that students will simply attach the names of all group members - with a very few exceptions, where there were major problems within the group. I have resisted suggestions to ask students to assess their partners' efforts as too subjective and inconsistent across the class as a whole.

7. Specific training. The aim is less to provide training in a specific software development method and more to demonstrate the types of activities that go on during development. Nonetheless, a project must use a specific method. The method chosen aims to demonstrate how quality can be checked in principle rather than how it is done in current practice. This is based on the assumption that it will be easier for the students later to reduce the level of formality to that of current practice than to increase it beyond that level.
5 An Assessment of the Proposed Development Process

This section discusses the perceived strengths and weaknesses of the proposed life-cycle. These are considered both from an engineering and from an educational point of view.

The nature of the process

This process has a number of prominent characteristics:

1. separation of responsibility between project engineers and program builders
2. absence of iteration or feedback between stages
3. dependence on formal notations

The separation of responsibility is based on current practice in other engineering professions, and follows from observing that the interests of the client and the builder may sometimes be in conflict. Lacking the technical expertise (and probably the time) to direct the construction process, the client needs the representation of technically knowledgeable professionals.

The absence of iteration or feedback between stages is generally recognized as a weakness of the waterfall model. In the case of this development process the absence of iteration is more a necessity and arguably a strength (as has been argued for Cleanroom [4]). The project engineers are held responsible for program quality, even though they do not build the software. If this is to be a true separation of responsibilities, there must be considerable predictability over scheduling, budgetting, functionality, performance and so on. Obviously there must be enough flexibility in the process to allow errors to be corrected, but there must also be enough confidence at the end of each stage that no significant time will be required to repeat or correct work already done.

The use of formal notations provides the foundation for confidence in the process. It allows precise communication between project engineers and designers. It is necessary if the design process is to be done rigorously. Finally, it is important when deciding if the construction has been completed satisfactorily.

Project engineering

In this development process, project engineers have a very heavy responsibility. To fulfill this responsibility they must be able to assess the client's true needs and be able to define very clearly and precisely the behaviour of a software system. This involves considerable interaction with the client and will frequently require the development of models (or prototypes) as a means of discovering the true requirements of the system. It also requires a real mastery in preparing formal definitions and their explanations. The fact that project engineers do not build the software themselves means that they must understand the system in considerable detail by the time the specification is complete. This is a major obstacle to overambition on the part of the project engineers and can thus help avoid the associated dangers [2]. The costs of having specifications prepared in this way are seen in the time required to prepare specifications in such detail and the fact that some features desired by the client are not implemented because of their complexity or cost. The chief benefits are the improved understanding of the behaviour of the software being specified and the increased confidence that a reliable software product will be delivered.

The specifications written by my students have these characteristics to a certain degree. The documents specify behaviour in considerable detail and limit functionality to the near the minimum acceptable. The more ambitious specifications were easily identified and avoided by the students working on design. Some students did write specifications without really facing the hard problems of the project, but this was very obvious in the formal specification - certain names appeared with no (or at best incomplete) definitions.

Naturally students also made mistakes in writing specifications, since they were unfamiliar both with the process of specification and with the notation they were required to use, and also because each group had fairly limited time to interact with the client. Most of the basic conceptual errors were detected when a draft specification was reviewed by the client and were corrected in the final document. A good proportion of the errors that I found later in assessing the specifications could have been eliminated by preparation and review of a second draft specification, but instead were found by the students who prepared designs. In the most serious case the student designers had to entirely ignore one system function because of errors in its specification. However most errors were concerned with differences in the treatment of special cases between the formal definitions and either the explanation given in the specification or the designers' understanding of the system. In such cases the designers were usually able to guess the correct definition. All errors detected in this way were corrected during the bidding process.

The specifications prepared in this way have been much more effective than the informal specifications prepared in past years. Given the students' unfamiliarity with both process and notation I found the results very impressive. Naturally there is room for improvement, if the schedule left time for a second draft specification and if students
had more experience before they started.

**Inspection**

The project engineers also have the responsibility of inspecting the construction of the software. The software builders will have their own quality controls in place, so inspection as carried out by project engineers is an activity that is simply not present in the waterfall model. It was introduced into this process model to follow engineering practice, but does in fact seem to be an inevitable feature of this process model. Since it is primarily the project engineers who communicate with the client, they are in the best position to check that the software is being built in accordance with the client's requirements. Moreover their independence from the builders and their technical expertise puts them in an excellent position to accurately and critically assess the software from the client's point of view.

Inspection has been the most difficult role for my students to adapt to. They feel too close to their fellow students to want to criticize their work and dislike the idea that they should report on errors in their friends' work. In fact their criticism should be constructive, with proposals for methods to overcome any errors uncovered. This difficulty would be less serious if inspectors and builders were more independent and more experienced.

**Program design and construction**

The bidding process is effective in identifying weak and overly complex designs as well as overly complex specifications. Several student specifications were not implemented at all, since their authors lacked confidence that the proposed designs could actually be implemented and meet the client's needs.

In some respects software builders get off lightly in this process model. They do not have to analyse the client's requirements nor prepare the specification, they are not even held responsible if the delivered software does not satisfy the client's real needs. All they have to do is build a piece of software to satisfy the specification.

However, since the task of the software builders is made simpler, there are correspondingly higher expectations as to how well it will be carried out. The process used to build the software is open to inspection by the project engineers and thus must be carefully and rigorously documented. Designs must be proven to satisfy specifications and must also be formally refined to code. These stages must be performed and checked very carefully, lest subsequent testing reveal the presence of too many errors and thus require the refinement process to be repeated. Mills [4] provides quantitative data on the effectiveness of a similar process used for building software in Cleanroom.

This is the part of the process where my students have experienced most technical problems. In particular the refinement process, based on Morgan's refinement calculus [5], is too unwieldy to be done entirely formally. Students experience confirms Morgan's own assessment in this regard. More research appears to be necessary before a formal refinement process can be used, although formal refinement of parts of a program can certainly be done using this method.

**6 Conclusions and Future Work**

Introducing formality into the software development process as described in this paper has many educational attractions. Remaining disadvantages, such as the difficulty of evaluating individual performance, are common to all projects done in groups. It must be admitted however that the role of program testing (or error checking) is sufficiently different from current practice that students may be misled into thinking that the effort typically devoted to testing programs today is unnecessary. Checking for execution errors is minimized in the proposed process model only because of the extra effort devoted to formalizing the specification and design documents, to the use of a formal refinement process, and because of the care taken to inspect refinement and coding before execution.

One cause for concern in the course projects is the frequency with which different groups must repeat or duplicate similar definitions. This effort can (and should) be reduced by making sure that such definitions have been prepared in advance. This applies to definitions used in specification and design documents, and even in code.

From a more general software engineering perspective, this development process must be considered quite demanding on the people involved in a software project. This is likely to make people less ambitious in the types of project they are willing to undertake, for they must first assure themselves that they can meet its demands and live up to their responsibilities. However, this conservatism is probably a key distinction between an engineering project and a research project; most software projects should not be treated as research projects!

The growing experience in stating specifications formally [4], reinforced by my own experience with student projects, shows that it is both possible and practical to write formal specifications and that doing so can uncover potential pitfalls and force developers to face the real problems earlier in development. However problems still remain in the process of refining specifications to detailed designs. These problems are practical rather than theoretical: the refinements
can be done, but they require too many small steps and an unreasonably long time. Until researchers have developed a way to speed up this process, it is probably more practical to mix two modes of working: do some refinements in small steps and develop some algorithms in ad hoc fashion before checking that they have the desired properties.

Although a quantitative assessment of the proposed process model would be very valuable, a teaching environment does not lend itself very well to such experimentation. My own assessment is that more qualitative, anecdotal and experiential results are needed to define a detailed life-cycle model that would make quantitative data meaningful.

Acknowledgements

The author would like to thank the 175 students who have taken his software engineering courses at the University of Western Ontario and the University of Ulster over the past 6 years. It has been a challenge and a pleasure to teach them and to learn from them. It is also a pleasure to acknowledge the advice, assistance and insight given by Michael Mak and Gary Lythe into how other engineering professions operate. The financial assistance of the Natural Science and Engineering Research Council (NSERC) of Canada is gratefully acknowledged.

References


