Extreme Group Projects

Mike Joy and Ananda Amatya and Andrew D H Chow and Julz Friedman

July 30, 2003

Abstract

Software engineering group projects are used in many Computer Science degree programmes. Recently, the "agile programming" style of software development, and the XP methodology have become popular. In this paper, we report a case study where the use of both classical and agile software engineering methodologies were used concurrently on an undergraduate group project, and discuss the implications for the use of agile programming in such projects. We argue that an adaptation of the XP and RUP methodologies can be successfully employed in the undergraduate curriculum, and we suggest a novel modification to the process which facilitates its use within a student environment.

1 Introduction

As part of an undergraduate degree programme in Computer Science, students usually take part in a group project in which they put into practice those software engineering skills they have learned. Not only is the exercise in group working pedagogically desirable [13], but it also prepares students for their future employment in the IT industry where group working is normal practice. Indeed, professional bodies such as the ACM, IEEE and BCS require the inclusion of such a project as a prerequisite to accreditation [1, 3].

In the last couple of years, there has been a fundamental change of emphasis by a significant number of software practitioners in the type of methodology that should be used within the software engineering discipline. No longer is the classical "software life-cycle" with its accompanying documentation and attendant methodology necessarily the process of choice. The advent of Agile Programming [15], and the XP methodology [2], have transformed our view of what constitutes "best practice" in software engineering.

In this paper we consider the implications the change has on the delivery of a group software engineering project within a computer science programme. We consider in depth the results of a case-study in which groups using "classical" software engineering techniques are compared against a group using XP under carefully monitored conditions. Finally, we suggest modifications to XP and RUP which yield a new methodology suitable for incorporation in the Computer Science curriculum.

2 Software Engineering

Software engineering is "a systematic approach to the analysis, design, implementation and maintenance of software" [7]. Ghezzi et al. [4] describe it as the "building of software systems which are so large or so complex that they are built by a group or groups of engineers".

2.1 Classical Methodology

Although there are many approaches to systematising the process [17], the standard view of the process is a variant of the "waterfall model", where the ordered sequence of requirement analysis, specification, design, implementation and testing forms the skeleton of the model. Typically the development process is done in phases [10]:

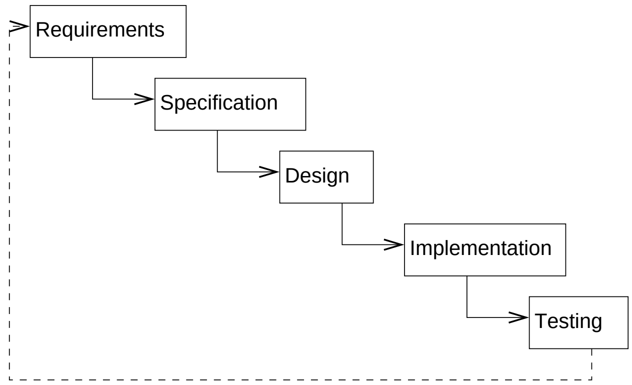

Figure 1: The classical Waterfall model

Requirements Gathering Phase: User requirements are decided in consultation with the customer. These generate a specification document which acts as a contract between the customer and the software developer group. Various prototypes provide the basis for such a document. Use case models express the software features, and system boundaries are made precise.

Specification Phase: This next phase analyses the domain specific features of the specification to provide the static (structural) and dynamic (behavioural) models of the system under consideration.

Design Phase: During this phase the static and dynamic models are refined by incorporating design decisions. Both high level design (base line architecture), and detailed design are arrived at.

Implementation and Testing Phases: Finally coding starts by converting the design documents into source programs in a high level language like Java. Usually, the implementation code is programmed incrementally, use case by use case, accompanied by unit testing. These units are then integrated into components and modules forming the system. Reuse of components and libraries is encouraged wherever possible.

These days object-oriented methods are invariably used. UML diagrammatic models support all the stages in the development process. One popular process *framework* is the Rational Unified Process (RUP) [14]. RUP advocates iterative phased development from *Inception* through *Elaboration* to *Construction* and finally to *Transition*. During the iterations within the phases various activities (work-flows) are performed to varying degrees. Much of the *Requirements modelling* is done during the *Inception* and *Elaboration* phase iterations. At the end of the *Inception* phase the *product Life-cycle objective* is decided. Much of the *Analysis* activity takes place during the *Elaboration* phase iterations. At the end of the *Elaboration* phase the *product base line architecture* is finalised. Much of the *Design* and *Implementation* activities take place during the *Construction* phase iterations. At the end of the *Construction* phase a *Beta release version* of the product emerges. The final *Transition* phase iterations deal mainly with the *Implementation* and the bulk of *Testing* activities. At the end of the *Transition* phase a release quality product is completed. Additionally, other activities that take place to varying extent in the various iterations are *Business Modelling* (mainly at *Inception*), *Deployment* (mainly at *Transition*), *Configuration and Change Management* (mainly at *Construction* and *Transition*) and *Project Management* (throughout the development process). One difficult issue with RUP is the tailoring of the framework to a process appropriate to an application environment.

2.2 Agile Programming

Classical methodologies are inflexible for modern day applications where requirements are too fluid to be fixed before implementation and where changes are made before the development process can be completed. This is due to the potentially large time-gap between the requirements specification phase and the delivery of the finished product. Agile Programming follows an iterative approach whereby the system evolves incrementally from one iteration to another, and not just from one release to the next. One important enabling factor for this is the customer involvement throughout the life-cycle of the software. Customer feedback is actively sought during each iteration to ensure that the product conforms to the needs and expectations of the customer as encapsulated in acceptance tests. One popular variant of agile methodologies which has found widespread use is Extreme Programming (XP). It has been particularly successful in web-based applications where requirements change rapidly, especially in small development groups. Since its initial success there have been studies on its use in many domains, and in larger development groups. In addition to *on-site customer*, XP uses other enabling practices, including *pair-programming*, *refactoring*, *test-first*, *coding standards*, *simple design* and *iterative development*. These promote high quality coding and customer satisfaction. Other practices that enable the management of a project using XP are *planning game*, *continuous integration*, *collective code ownership* and the *use of metaphor*. Such practices flatten the “Cost of Change Curve” [2].

3 Undergraduate Group Projects

The use of group work in higher education has many benefits, and there is a substantial body of knowledge on its effective deployment [9, 11, 13, 16]. Henry [11] identifies five principal reasons for students to undertake group work.

- Application of knowledge – students are able to put into practice the knowledge and theory they have assimilated in previous modules.
- Motivation – a suitably chosen project is likely to be of direct relevance to the student.
- Higher cognitive skills – students develop a deep understanding of the material they are working on, and develop corresponding deep learning skills.
- Autonomy – students have control over what they learn and how they learn it.
- Assessment – projects are effective at separating the "sheep from the goats".

In the context of Computer Science, there is a cultural factor. Hughes and Cotterell [12] note that "A problem with major software projects is that they always involve working in groups, and many people attracted to software development find this difficult". This is reflected in the the Professional Practice chapter of the ACM/IEEE Computer Science Curriculum[1]: "Learning how to work in groups is not a natural process for many students, but it is nonetheless extremely important." For this reason we feel it appropriate to add a sixth reason for the inclusion of group work in the Computer Science curriculum:

- Group skills – students should learn to work effectively as part of a group, rather than as lone programmers.

Not only is group work educationally desirable, its inclusion in the curriculum is mandated for the purpose of course accreditation. In the UK, the British Computer Society in its Exemption and Accreditation Guidelines [3] stipulates that each accredited course must contain a "substantial group project" which must fulfil (inter alia) the following criteria:

- "it should exhibit a structured approach to information systems practice, involving a number of stages in the life-cycle;
- each student must clearly identify their contribution to the overall project; ... and the assessment must clearly identify each individual's personal contribution;
- the product must exhibit the attributes of quality, reliability, timeliness and maintainability."

In the United States, the ACM/IEEE curriculum [1] refers to "group-work with individual accountability", and requires the incorporation of a "significant group project that encompasses both design and implementation".

4 Case-Study

"Introduction to Software Engineering", is a module offered to second year Computer Science undergraduates at the *University of Warwick*. The syllabus includes the fundamentals of software engineering, the software life-cycle, the UML notations, testing and maintenance, and overviews of software engineering methodologies such as Fusion [5].

As part of this module the students are required to work in groups of 5 or 6 to develop and submit a piece of fully documented software at the end of *Term 2*. The software is specified by the tutor, and the specification document is distributed to the students midway through *Term 1*. Groups are finalised before the end of *Term 1*, and once formed groups remain unchanged until the software is submitted. Formal documentation is required at all stages of the software development process, and groups are assessed on the quality of the software, the quality of the documentation, and the effective working together of the group members as a group. It is intended that the project would be completed by each group in *Term 2* (ten weeks), although groups could commence work earlier if they wished.

The usual process adopted is a classical software life-cycle. During the 2002/03 academic year, one group of six students elected to use a combination of RUP and XP methodologies to develop their project, and this was monitored closely. This section describes the process which they used, and outlines the major issues affecting their deployment of the methodology.

As young Computer Science students, members of the group had little experience of working in a group on a large software development project. However, they realised from the onset that the project could only succeed if they allowed a degree of flexibility in the features to be implemented.

The size of the project did not warrant using the full breadth of the RUP, which would also have meant missing out on the advantages of many agile methodologies. On the other hand, XP could not be fully implemented as it was impractical to collocate programmers to allow for pair-programming and constant communication.

4.1 The Development Life-Cycle

The members of the group had organised themselves together soon after the software specification had been released, and they began the Inception Phase of the development process late during *Term 1*. Much of the planning and initial architectural design was done in this period before the actual development work in *Term 2*.

Because of the time-scale of the project, only one formal release was possible. Planning involved an initial Blue Sky meeting, in which the group discussed possible changes to be planned for in the system architecture. This was necessary since the original specification was open-ended as to the exact range of the required features. It provided a brainstorming opportunity where members of the group contributed ideas based on their interpretation of the given specification. Each idea was then accepted or discarded depending on its relevance and feasibility within the time-frame. In this way, the group came up with a small set of essential features and another set of optional features consistent with the assessment criteria of the project. This meeting effectively replaced the initial release meeting with the customer and the XP "planning game".

These features were converted into user stories according to XP practices. These user stories were in turn split into tasks, which were undertaken by members on a voluntary basis according to personal preferences and expertise. During the Inception Phase, developers were encouraged to experiment with various algorithms. This resulted in high quality implementation of the areas for which spike solutions were constructed in the next phase of the development life-cycle (*see later*).

XP advocates just-in-time planning, but since the final program specification was restricted, and (approximately) known at the start of the release, most of the user stories were specified up-front, rather than at the start of each iteration. This was perceived as an important factor affecting the project, and one over which neither the students nor the tutors had control.

Once the features for the system had been decided, the architecture and the accompanying UML models were developed. The software architecture evolved into a variation of the standard Model-View-Controller pattern [8]. Much of this work was undertaken by a member of the group who acted as the "coach", and who ensured consistency of the whole system with this architecture.

The system was developed in four two-weekly iterations. The first iteration was taken as the Elaboration Phase, which finalised the evolution of the architecture. It was a zero-feature iteration in which spike solutions were developed for areas that were deemed risky in the project (mainly, the complicated algorithms). The Construction Phase of the development life-cycle was made up of the second and third iterations in which the bulk of the system was developed. Whilst the system was developed under the XP principle of "collective code ownership", the group found it efficient to have each developer concentrate on his area of expertise. This is perhaps a reflection on the different skills offered by the individual group members.

In order to ensure every developer had an insight into the whole system, pair-programming took place where the lead coder of a particular area worked with another developer. Group Programming (*see below*) was used, and some rotation did take place. However, key algorithms were designed by a single person with some pair programming.

It was not considered workable in practice to use pair programming for the majority of the code, since the programmers could not be collocated for sufficiently long periods of time and no on-line system (such as "instant messaging") seemed to work effectively for that purpose. To overcome this a kind of "Group Programming" was used instead. It involved, after each weekly meeting, the group dividing into pairs to work on any problematic areas. Also, near the end of each iteration, a Group Programming session took place. This involved the entire group gathering round two computers to work on the problematic areas, designing new code or redesigning and refactoring the offending code. The lack of pair programming also necessitated more up-front design than most agile methodologies recommend. However, it was felt that the combination of Group Programming with frequent instant messaging was a good compromise between traditional methodologies and Extreme Programming outside of a collocated environment. Whilst its concept is novel, the idea of "Group Programming" is very much in the spirit of XP involving group members being collocated and communicating freely and effectively to maintain a group vision of the code and its architecture.

The final iteration was used as the Transition Phase where the small remaining number of features were implemented. The system was then beta-tested to make sure that the user interface worked as specified, and that the acceptance tests did not specify incorrectly what was wanted. Most things the beta testing found were problems between the user interface (no tests) and the command interface (tests). Where bugs were found during the beta-testing phase, they were relatively easy to fix, thanks to the XP practice of Test-First development employed by the group. After the conventional means of testing were exhausted, the group put the system through extensive user testing, which discovered bugs that had been missed by the Unit, Acceptance and Smoke Tests.

4.2 Communication

To maintain communication the group attached a great deal of importance to the meetings that took place throughout the development life-cycle. Formal meetings were held at the end and in the middle of each iteration to discuss the progress of the iteration, and to resolve problems encountered by the developers. Minutes were taken at every formal meeting to record the decisions made and the solutions to the problems reported. In addition to these meetings, members arranged meetings amongst themselves in smaller groups whenever they encountered problems or needed help, and the outcomes of these meetings were circulated by e-mail. This was usually followed by instant messaging discussion with those members not present but interested.

Instant messaging was extremely useful in that it allowed developers to ask questions to each other while writing code - a necessity when code is shared. However, the group realised that it could not be an alternative to pair-programming.

To improve communication at coding level, the group used the Eclipse IDE [6] which provides the facility for certain comments to be defined as "additional compiler markers". This feature causes any comment preceded by certain predefined keywords to appear as a task in the shared task list, providing a clear overview of the project's progress. A further use of comments was for non-urgent discussion of areas of code. A developer struggling with a minor problem, or unsure as to which of several options to use, would leave a comment and move on to the next task. Later, another member of the group would come across this comment, and either fix the problem it referred to or leave an explanation. On occasions, this extended to four or five levels of discussion from different people, debating the relative merits of two approaches, followed by an explanation of why a third method had actually been used.

4.3 Code development and testing

As stated before, the group adopted the principle of Collective Code Ownership from XP, which theoretically allowed every member of the group to modify each other's code. This was reinforced by a set of Coding Standards containing naming and formatting conventions, which were strictly adhered to by all the developers, giving the code a uniform format. The source code of the system resided on a dedicated CVS server, allowing developers to share tasks throughout the development process. The CVS server, whilst acting as a central repository for the most up-to-date version of the source code, also allowed developers to work concurrently on a particular file. Because of the shortness of the project duration, Collective Code Ownership did not fully work, each unit bearing to some extent the mark of its creator. However, the developers did gain a wider insight into the whole system.

As already mentioned, the group tacitly adopted the XP practice of Test-First Development. Unit tests ensured correctness of the unit codes, and acceptance tests validated that the code covered the user stories. A rather major last-minute bug found was a case of having gone ahead and written some code by writing test-first the simplest thing which could work in order to get another programmer started on writing the full thing. Because no acceptance test specified any more rigorous requirement, no-one spotted that although necessary code had been written to pass the unit tests, the unit tests did not specify sufficiently the requirements to be met.

It was intended that acceptance tests would be written at the beginning of the iteration that implemented a user story. This was to ensure that the code conformed to the specifications and integrated fully as a system. In practice, too many acceptance tests were written after the relevant user story had been completed; this happened because the group did not have a specific planning meeting, and there was no on-site customer to perform the acceptance tests.

4.4 Management

The project was managed according to XP principles, involving a *project manager*, a *coach* and a *tracker*. The tracker ensured that the project was on track, although he was also responsible for development work. The coach was responsible for putting together the set of coding standards, the design of the system architecture and providing general assistance to developers when they faced problems. The project manager played essentially a supporting role, being solely responsible for arranging the best possible working environment for the group programming sessions, organising all the meetings, taking minutes, reporting, and communicating outside the group.

5 Evaluation

The lessons learnt during the project form the basis for a methodology, "Effective Development Process (EDP)", that is well suited to undergraduate group work.

5.1 The Elaboration Phase

5.1.1 Some Architecture Up-front

A basic architecture of the system was in place right from the beginning, with the possibility of some change as the system evolved. This compensated for the absence of pair-programming and the comparatively limited group communication. The"blue-sky" meeting allowed the group to identify and plan for the likely areas of change to this architecture. This was an effective compromise between no architecture and over-architecture.

Suggested guidelines

- "Blue-Sky" meeting to identify areas of likely change and to provide group input into the architecture of the system

- Initial skeleton architecture followed by evolutionary change in the construction phase.

5.2 The Construction Phase

5.2.1 Differing skill levels of participating students

Pair-programming was used only as a special tool to solve difficult problems and during the switch-over when people moved between modules. Some coding was done by only one member, while others were developed jointly. Code with no architectural basis at the elaboration phase needed refactoring.

Suggested guidelines

- All new code should begin by being programmed by at least one *experienced* programmer.
- A certain core base of knowledge should be codified as appendices to the coding standards: unit testing (with the proper use of mock objects etc., not simply use of JUnit), standard patterns (so as to understand areas where the architecture uses it quickly), and cohesiveness and coupling (to avoid common errors requiring refactoring).
- At each Group Programming session, a pair of programmers should be responsible for refactoring code, and nothing else.

5.2.2 Allowing code to be written before acceptance tests

Writing code for user stories without acceptance tests was not expedient, instead errors became more difficult to correct.

Suggested guidelines

- No story may be added to the story pile without a set of pre-written acceptance tests.
- The Lead Programmer is responsible for implementing the tests at the start of the iteration.
- At the start of the iteration the number of failing tests should be recorded, as a psychological incentive to have written the tests.

5.2.3 Student programmers don't understand refactoring

Delaying, or omitting altogether, refactoring resulted from the lack of pair-programming opportunity.

Suggested guidelines

- It is probably not possible to assume or ensure that refactoring will always be done at the end of a story without pair programming, but it is possible to have a pair of programmers, or sometimes the entire group, work on refactoring exclusively at each Group Programming session.

5.2.4 Testing the edges of the system

Almost all non-trivial bugs found in the system during use were attributable to cases where tests were hard to write, e.g., the UI and the IO sub-systems. In the case of the UI, this occurred because coupling had been introduced between it and the command layer, meaning it was not using the command as specified, and therefore the command's tests did not spot problems. In the case of IO, a complex system was left untested because it was difficult to make assertions about the state of the file-system without duplicating the file reading code which was being tested.

Suggested guidelines

- The UI should be constructed using an automatic JavaBean based RAD framework (or something similar), preventing any logic being added to it. It should simply be setting and reading properties on beans, and bouncing button presses to a Command object's `execute()` method.
- The IO system should be replaced by mock objects for acceptance testing. A simple helper class should be written to aid Unit tests that write files to disk (rows to database) and assert that correct objects are being returned.

5.2.5 Code Conversations

Code tags such as `Todo:`, `Fixme:` and `BUG:` were a major accidental discovery in the project. These were supplemented by `note:`, `question:` and `URGENT:` tags, the former two for situations where programmers wished to clarify issues other than bugs, the latter for issues which needed to be sorted out immediately. The use of code tags as a communication tool turned out to be very effective.

Suggested guidelines

- Use code tags such as `note:`, `question:` and `URGENT:`.

5.2.6 Group Programming sessions

Group Programming sessions before the end of each iteration worked well. They avoided the need for members of the group working late to meet the deadline, and probably abandoning many good practices in the process. If something had been left late it was worked on by the whole group and proper practices could be continued.

Suggested guidelines

- Group Programming sessions must be used the day before the end of every iteration.
- Group Programming sessions should be used at the mid-iteration meeting; this will fish out the case of "99% done meaning 0% done" and ensure that bad decisions are corrected before the end of the iteration after which they would have been too difficult to change.

5.3 The Transition Phase

Not enough time was initially allocated to the transition phase. Bugs were found at the last minute. Further refactoring were clearly needed. This led to the conclusion that the transition phase should not have been combined with the final construction iteration of a release. The transition phase should have been used only for final quality assurance. The story pile (as opposed to the code) should have been frozen at the end of the penultimate iteration of a release, any uncompleted stories at the option of the customer should have either been moved to the next release, or the deadline for the release moved to allow further extra construction iterations. Although it is a valid ideal to have code of production quality at all stages, the constraints of an academic project make it virtually unattainable.

Suggested guidelines

- Transition phase and code phases should be kept strictly separate.
- Story pile frozen at 0 at start of last iteration.

6 Conclusion

A case study on the use of both classical and agile methodologies in an undergraduate group project was undertaken.

It became clear that the constraints of the educational context (such as the project size and the difficulty of collocating group members) precluded the use of XP, RUP or other current agile methodologies. In response to the challenge of articulating a workable methodology appropriate to the educational environment, an adaptation of the XP and RUP methodologies was developed and successfully applied. Evaluation of the exercise suggested further enhancements to the methodology.

The new methodology was evaluated on a module where other groups used a traditional methodology for software development. Its successful use without the need to make any significant changes to the module's structure or assessment process indicates that adoption of agile methodologies within the Computer Science curriculum is feasible.

The methodology appears pedagogically sound, satisfying Henry's five motivations for group work [11]. Students were able to apply their knowledge throughout the project on the tasks assigned to them, and were able to work autonomously during each iteration. This allowed assessment of each student's individual contribution whilst retaining the group ethos and the agile style of software development. The exposure of each group member to a variety of tasks requiring different skills, the use of pair-programming at critical stages of the process, and the individual responsibility to contribute at group meetings appears to enhance the students' deep learning and their effective group skills.

The requirements of the professional bodies appear to be met. Since EDP is a variant of XP and RUP, it exhibits a similar structure, and the testing and management disciplines contribute to the quality, reliability, timeliness and maintainability of the project deliverable. Although group and pair programming are present in the methodology, they are not ubiquitous, and the contribution of individual students can be clearly identified.

References

[1] ACM/IEEE-CS Joint Curriculum Task Force. Computing curricula 2001. www.acm.org/sigcse/cc2001/, 2001.

[2] Kent Beck. *Extreme Programming Explained - Embrace Change*. Addison-Wesley, 2000.

[3] British Computer Society. *Guidelines on Course Exemption and Accreditation*. Swindon: British Computer Society, 2001.

[4] Mehdi Jazayeri Carlo Ghezzi and Dino Mandrioli. *Fundamentals of Software Engineering (2nd edition)*. Prentice Hall, 2003.

[5] Derek Coleman et al. *Object-oriented Development: The Fusion Method*. Prentice-Hall, 1993.

[6] Eclipse.org. Eclipse. www.eclipse.org, 2003.

[7] FOLDOC. Free online dictionary of computing. www.foldoc.org, 1994.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1997.

[9] Graham Gibbs. *Learning in Teams: a Tutor Guide*. Oxford Centre for Staff Development, 1995.

[10] Hassan Gomma. *Designing Concurrent, Distributed, and Real-time Applications with UML*. Addison-Wesley, 2000.

[11] Jane Henry. *Teaching Through Projects*. Kogan Page, 1994.

[12] Bob Hughes and Mike Cotterell. *Software Project Management (3rd edition)*. McGraw-Hill, 2002.

[13] David Jaques. *Learning in Groups: A Handbook for Improving Group Work*. Kogan Page, 2000.

[14] Philippe Kruchten. *The Rational Unified Process - An Introduction, Second, Edition*. Addison-Wesley, 2000.

[15] Robert C. Martin. *Agile Software Development: Principles, Patterns and Practices*. Prentice Hall, 2003.

[16] Michael Reynolds. *Groupwork in Education and Training*. Kogan Page, 1994.

[17] Schach Stephen R. *Object-Oriented and Classical Software Engineering*. McGraw Hill, 2002.