Behind the Rules: XP Experiences

Andrew Jackson, Shiu Lun Tsang, Alan Gray, Cormac Driver, Siobhán Clarke
Distributed Systems Group,
Computer Science Department,
Trinity College Dublin,
Dublin 2, Ireland.
{firstname.lastname}@cs.tcd.ie

Abstract

Agile processes such as XP (eXtreme Programming) have been recognised for their potential benefits of improving software. During adoption of the XP process, teams can misapply the XP principles by following them verbatim, ignoring the context in which they are applied. In this paper we document our experiences where naive applications of XP principles were altered in recognition of context. We detail our observations of how teams "looked behind" the rules and began fitting XP to the problem rather than attempting to fit the problem to XP. We conclude by reflectively focusing on how this transformation occurred and suggest that it is buying into the XP ethos that drives this change of perspective on the XP process and principles.

1. Introduction

Agile processes such as XP (eXtreme Programming) have been recognised for their potential benefits of greatly improving software in terms of fewer bugs, early delivery of valuable functionality, closer client/programmer interaction and a lower cost/change curve [1, 2]. As with any process, factors such as experience, time pressures and project scale influence its success or failure. This paper focuses on how developer buy-in to the process can be a major determinant of the success of a project.

XP prescribes a series of principles which, when collectively followed, are the basis for an ethos of good programming practice. Although nothing fundamentally new, XP "takes a set of common sense principles and practices to extreme levels" [3]. There are twelve principles in all: Whole Team Involvement; Planning Game; Customer Tests; Simple Design; Pair Programming; Test Driven Development; Design Improvement; Continuous Integration; Collective Code Ownership; Coding Standard; Metaphor; and Sustainable Pace.

In this paper, we share experiences derived from using XP as the development process for seven different projects over the course of three years. Upon reflection, we have identified that a naive understanding and application of the XP principles can often bring problems to a project. We explain how we feel these situations arose, and give our experience and advice on how to address them. Our strongest piece of advice is to get everyone involved in the project to fully participate in the process and buy into the underlying reasoning behind it.

Section 2 describes the projects undertaken and the environment in which they were executed. Section 3 discusses factors that affect XP in universities and how we attempted to mitigate their effect. Section 4 presents our experiences of naive applications of the XP principles, how these were addressed and the results of the changes. In Section 5 we summarise and reflect on our experiences. Section 6 describes an interesting avenue for future work.

2. Background

This section describes the environmental setup for the projects and the projects themselves.

2.1. Environmental setup

Our experiences are drawn from projects that have been carried out at Trinity College Dublin using the XP process. The projects were part of a one year Masters Degree program course in Software Engineering for Distributed Systems. We include seven projects that took place during three years of the course. Project teams were made of four to six students, all of whom were from various backgrounds, different disciplines, and varying levels of experiences. The projects spanned 12 weeks and ran in parallel with a number of other projects on the Masters program.

2.2. Projects

It has been documented [4] that most projects that are undertaken in universities are trivial and lack industry-standard requirements. As [5] notes: "XP is for professionals, not students".

However, these conclusions are based on undergraduate-level projects and teaching. It is an aim of the Masters course that the students get as real an

exposure to industry-related projects as possible. Therefore, the projects assigned were non-trivial and client/end-user focused. The academic environment still imposed some restrictions (as discussed in Section 3), but the project goals aimed to provide high-quality, useful software to the end-user. Brief descriptions of the seven projects follow:

Journey time estimation system for Dublin Bus (BUS). This project was to provide an SMS based interface to a system which would plan a route for a user from an origin to a destination based on estimated journey times for Dublin Bus. It is interesting to note that in the recent past (almost nine months after this project completed), a similar system has been deployed to the public by Dublin Bus and Irish Rail that serves static timetable information via an SMS gateway.

Route planning system for Dublin Taxis (TAXI). This project was to provide a system whereby a taxi could find the most efficient route(s) to get from an origin to a destination. This included using real-time traffic information to estimate trip times.

Location aware game (LAG). This project was to provide a peer-to-peer, augmented reality game played out by users with wearable computers over the Trinity College campus, leveraging real-time GPS information. As can be seen from the Nokia N-Gage [6] and other mobile devices, location aware augmented reality games are the next generation of mobile gaming.

Mobile voice and SMS over a peer-to-peer network (TTALK). This project was to provide a completely decentralised architecture through which users could call or SMS one another over existing LAN infrastructure. TrinyTalk has evolved into a sourceforge project [7], demonstrating its applicability in the real world.

Distributed conference management tool (FL-CMT). This project implemented a conference management tool which provided a diverse set of functionality to fully organise and manage an international human rights conference, including event, attendee and accommodation management. This system is currently being used by an international human rights agency called Front Line Defenders [8].

Conference contact and publicity management tool (FL-CPL). This project ported a legacy contact management system to a new environment and extended this with an integrated publicity management subsystem and a new logging facility. The system constructed was built on the Lotus Notes/Domino Server platform which is recognised as an enterprise industrial platform. This system is also being used by Front Line Defenders.

Web enabled postgraduate application management system (AppTrack). This project was to produce a sophisticated collaborative document workflow management system, built on J2EE and the Jakarta struts framework.

3 Factors affecting university XP

Factors such as experience, time pressures and project scale can affect a project in different ways. From the outset, we identified a number of aspects that would have an effect on how the XP process would be used in our environmental setting. These factors affect most XP projects in universities, and our findings support those discussed in detail in [4, 5, 9, 10].

3.1. Factors affecting the customer

Customer is a professor/lecturer. University XP projects are proposed by members of staff. However, in many courses, the dichotomy between the roles of teacher and customer is not addressed. This disparity was avoided by our set up, where each customer was a member of staff who was not involved in the teaching of the Software Engineering course. As such, the course lecturers were not involved as customers, but were available as "agile consultants", who were called upon from time to time to help guide the process.

Limited access to the customer. XP encourages continuous customer interaction and testing. However, customers were members of academic staff with busy schedules, thus limiting their availability for dealing with XP teams. They were thus unable to sit in on every XP session. Different teams handled this in different ways. The three prevalent ways were:

- Weekly meetings with the customer to report on the project and to get customer decisions on important stories and functionality, etc.
- One team member was elected to act as a proxy that represented the customer's interest and liaised with the customer at a mutually convenient time. Some teams rotated this responsibility and others did not.
- Teams as a whole acted as a proxy and all customer decisions were made by consensus.

3.2. Factors affecting project timescale

Projects had a shorter time scale. In industry, projects have a lifecycle that is typically in the order of months. In academia, projects have a much shorter lifecycle. This means that there are no stage releases. This was offset by making each iteration into a release in itself. Iteration length varied from group to group, from one iteration per week to one iteration every three weeks.

No set working hours. Students did not have set hours such as there would be in an office environment and instead were given the freedom to assign their own working hours.

Projects done in conjunction with other work. Students did not work solely on the XP project and constantly had to juggle the work of a number of projects simultaneously. This meant that teams had to agree the number of hours per week to devote to the project. Most teams opted for three four-hour sessions per week, for a total of twelve working hours per week.

3.3. Factors affecting project teams

No XP manager/champion/coach. Typically in industry when a team switches to XP as their software development process, it is done as part of an integrated company policy. Usually, an XP "champion" or "coach" (one who has been part of an XP team previously) is brought into the team to drive the process and to ensure adherence to the guidelines. This is not true for academic projects, where team members are responsible for driving the process within their own team. The project teams handled this in one of two manners:

- One team member was appointed the XP champion, whereby all decisions pertaining to the process were made by this person.
- All team members took equal responsibility for the process with process decisions made by consensus.

Team members of varying skills and backgrounds. Ideally, teams are comprised of members that have practical experience in the technologies used on the project. The team members in our study had widely differing backgrounds and experiences, ranging from fresh graduates to team leaders from industry. Therefore, team members would not have applicable prior experience in the technology they were working with in the project, except by happy coincidence.

3.4. Factors affecting process goals

Focus on process rather than product. The success of a team using XP in industry lies in the success or failure of the end product. The same is not necessarily true in academia where students are graded. In the projects presented here, students were given a single pass/fail grade based on their use of the process in delivering a product to their customer. This helped to simulate the success/failure of a product in an industry environment.

Shrink-wrapped products. Because of the lack of availability of the customer to the XP teams and the triviality of the projects themselves, it is often the case in academia where "shrink-wrapped" products are delivered at the end of the project, with no customer interaction during development. This was not the case for the projects discussed in this paper, as teams communicated regularly with their clients.

4. Our experiences

Here we present examples from our experience of how naive application and understanding of the XP principles can slow down overall project velocity. In each case, we discuss how we feel problems arose, how each was addressed by the teams and how it affected the project. We complete each section with a short reflection on how to potentially avoid the problems.

4.1. Non-functional requirements are requirements too

XP tells us that only code that adds value to the project should be included, and that anything that does not add value should be stripped away.

The TAXI team applied the simple design and planning game principles too rigidly and naively in the initial phase of their project. For each iteration, user stories were completed and functionality added to the code-base, without looking to later iterations. However, the "functioning skeleton" [3] written to fulfil these stories was very basic and addressed the simplest interpretation of the functional requirements. The team took this approach because they believed that by following the XP principle of simplicity, their code would not be embellished and have extraneous content. This approach produced well modularised, maintainable and easily comprehendible code. About half-way through project life-cycle, it became apparent that the emerging architecture, could not meet the non-functional requirements of the project; which included fault-tolerance and scalability. The model that the TAXI team built was a standalone application, the architecture of which would be very difficult to alter.

At this stage, the team reassessed the project, and realised that the scalability and fault-tolerance that they were missing could be provided by exploiting facilities of provided by a J2EE platform. The team had been developing on the standard Java platform and thought they could easily port much of their functional code into a new architecture. This meant a complete re-design of the existing system to fit the new platform and exploit the relevant J2EE services. More resources (in the form of overtime) had to be devoted to the project in order to finish on time.

The TTALK team began their project by experimenting with the different technologies and architectures, finally settling on a peer-to-peer architecture using JXTA. Although the team chose JXTA because there is a ready architecture there (following the XP principles of simplicity), they found that the development platform provided a basis to fulfil their functional as well as non-functional requirements. These non-functional requirements included security issues, the

need for scalability and fault tolerance. The code produced used the services provided by the underlying platform to implement their non-functional requirements. In this way, the code produced met all the projects requirements, both functional and non-functional, while remaining maintainable, comprehendible and unembellished.

In our opinion, the problems encountered by the TAXI team were due to the fact that the team members valued functional requirements much more highly than non-functional ones, coupled with the naivety of their planning game. They perceived that value was only added to the project when functionality was, and simplified the design to this end. Ultimately, they (the team and the customer between them) realised that the non-functional requirements were not met and could not easily be worked into their current design. The TTALK team showed that you can avoid major re-design if your initial "functioning skeleton" supports the short and longer term product requirements.

Non-functional requirements are requirements too. Every requirement is important. Long-term non-functional requirements should not be ignored during early iterations and an architecture that can accommodate these requirements should be used.

4.2. Testing can be trying, but do try to test

In test driven development, we write a test that will fail and then make the simplest addition/alteration to the code that will make the test pass.

The members of all of the teams were new to test driven development and, as a result, some teams inadvertently applied the principle naively. The LAG team in particular decided to follow the test-driven ethos to the letter, and so wrote every conceivable test as code was added to the project. These included trivial tests to make sure simple object assignment was working. They started by writing exhaustive tests for simple functionality such as "setters" and "getters" that had a low risk rating. This level of testing was reproduced for all behaviour from high-risk complex functionality, to low-risk one line methods.

At the end of the second two-week iteration, it was clear that the project velocity was very slow. The LAG team were losing time, and the stories they had set them selves were not being completed. The team had been over-testing miniscule things and relaxed the testing regime. Project velocity picked up for two iterations and then dropped again. We observed that the LAG team had overcorrected and by stopping much of their testing to catch up with their schedule. However, as time went on we found that were losing time trying to locate and isolate bugs. Once this was realised the team then focused the testing effort on key behaviour and high-risk functionality. Consequently, a good velocity was maintained for the remainder of the project.

The TAXI team did not have such problems with finding a balance in testing. The team was lead by an experienced programmer, who introduced his team to test driven development through JUnit and HTTPUnit, using a lot of practice tests. The team identified what they were going to test and at what level at the beginning of the project. The team did not suffer from teething testing related problems as a result of this experience and foresight.

The FL-CMT and FL-CPL teams were developing for Lotus Notes/Domino Server platform and developed through writing scripts in a proprietary language that ran in the environment. There were no unit testing tools available to support test driven development on this platform. These teams found it difficult to test but adapted and wrote scripts that tested their development. We observed that a lack of support for testing forced these teams to spend a lot of time trying to write tests. In order to regain lost velocity, they reduced testing across the board, as did the LAG team. However, the FL-CMT and FL-CPL teams did not learn to focus their testing efforts, perhaps due to the inflexibility of their development environment, and were less successful as a result.

We found that teams new to XP tended to over-test during the early iterations, which led to an initial reduction in velocity. When teams realised this, they reduced the testing effort. Successful teams learned to focus their testing effort on key behaviour early in the development, whereas unsuccessful teams reduced testing effort across the board.

Testing can be trying, but do try to test. Inexperienced teams often over-test in early iterations, and in response, reduce testing effort. Although writing tests takes time, it is time well spent when it is focused on testing important behaviour.

4.3. Fail to plan and you plan to fail

Every two weeks, planning for the next two-week iteration means that plans can be rapidly adapted without wasting effort on eventualities that might never arise.

The features of the BUS project are very similar to those of the TAXI project in that they are both route planning systems that are open to public use. As such, they also shared many non-functional requirements. As discussed in Section 4.1, the TAXI team did not plan for the long term, ignoring the non-functional requirements of the project. The BUS team planned to address the non-functional requirements from the outset by developing an architecture that would provide support for fault-tolerance and scalability.

The TTALK team also suffered from focusing on the short-term and ignoring the long-term. The team began

their project by spiking to investigate the technologies that would be involved in their system. They ran into problems while spiking because of the myriad of technologies they would have to use. Without a clear and consistent view of the overall deliverable and route to take to produce that deliverable, they found it difficult to leave the spiking stage. They continually invested in the problems at hand without planning for the future. The team were in the doldrums for half of the project's life-time, until they finally realised now much time they had lost. At this point they had to work overtime and drop some of the system features that they were required to implement.

By naively following the planning game principle verbatim, teams can neglect long-term aspects of their project. This is analogous to driving a car, but only paying attention to the ten yards of road immediately in front of you and ignoring what lies on the horizon.

In comparison, teams that developed their projects whilst keeping one eye on long term goals tended to be more successful, because there was less time spent doing rework in later iterations to add new functionality to the code. In our experience, teams that planned without any view to future project milestones were less successful, because they tended to ignore long term problems.

Fail to plan and you plan to fail. Plan for the next iteration at the beginning of every iteration. However, keep an eye on long-term aims of the project so that they can be worked into the solution at a later stage

4.4. Watch this pace

Sticking to a set working week keeps your team members fresh, enthusiastic and productive. We observed three cases of how the sustainable pace principle was applied by project teams, each with varying degrees of success.

The FL-CPL team stuck stringently to their set working hours. Team members promptly stopped once their allotted XP hours were over, often leaving source code and integrations incomplete. There were two main consequences of this. Firstly, it meant that incomplete iterations had to be finished in subsequent ones, leading to a backlog of tasks that at the end of the iteration were either scrapped or remained undone. Secondly, having a degree of incompleteness in projects negatively affected the morale and enthusiasm of some team members (completely opposite to the aim of the principle), leading to decreased productivity and quality of the work being done.

Members of the AppTrack team were a lot more flexible in their approach to their working hours. The team accepted that sometimes it is necessary to do overtime in order to complete tasks and ensure that the nightly integration succeeds. If the team had to do no more than thirty minutes of overtime in order to make sure the integration tests passed, then they were prepared to do so without any fuss. We observed that the team enjoyed a more relaxed development environment, as each day the pairs could look forward to new challenges, without having to worry about having to face incomplete code hanging over them.

As mentioned above in Section 4.3, the TTALK team lost a lot of development time in early iterations. Despite having removed all nonessential stories from later iterations, they still had a great deal of work to do to meet the requirements for minimal functionality of their system. They addressed this by deciding to increase their working time by an extra three hours a week (an extra 25% on their original plan). Initially, the team had a good velocity, but the extra workload meant unhappier team members and after a while the quality of work degraded as has been observed in many workplaces.

Rigidly adhered to, the sustainable pace principle can adversely affect the overall development of some projects and we observed that teams that refused to put in extra effort from time to time did not enjoy the same success as those that did. We do agree that doing long hours of overtime does not help, but we saw that teams who did the overtime necessary in order to complete iterations and integration builds were more successful in terms of enthusiasm, code and deliverable quality, and enjoyment.

Watch this pace. It is important to stick roughly to your working week, but having the flexibility to do small amounts of overtime to complete nightly builds yields happier teams and better code.

4.5. Buy in and butt in

Pair programming enables team members to share ideas and experience, maintaining good project velocity and producing good quality code.

In the LAG team, there were two very experienced programmers and one particularly novice programmer. Whenever the weaker member was paired with one of the stronger ones, they refused to take the keyboard, claiming inexperience. The member didn't interrupt the other person to add their views, and thus was not contributing to the development effort.

Another reduction in project velocity was apparent in the FL-CMT team. Team members viewed pair programming as an opportunity to take a break when they were not at the keyboard. Obviously, this goes against the XP ethos and we believe this attitude was borne out of a lack of buy-into the XP way and lack of understanding of the reasoning behind the principle.

The TTALK team, as discussed in Section 4.3, lost a lot of development time because they devoted too much time to spiking. Even though they resolved to do a lot of overtime, they decided to abandon pair-programming to

try to gain as much development time as possible. As mentioned in Section 4.4, their initial velocity rapidly decreased.

Naively applied, pair programming can lead to a large reduction in project velocity. This is borne out of a lack of understanding of its purpose. In our experience, members new to the concept sometimes lost focus when not actively programming. Additionally, we observed that in a pairing of an inexperienced and an expert programmer, the potentially valuable opinions of the "weaker" person were often disregarded without consideration.

Buy in and butt in. When not at the keyboard, developers are not on a break and should actively contribute. They should "buy in" to the XP ethos and "butt in" to actively engage in the development process with the team member who is typing.

4.6. What is mine is yours, but arrange before you change

In any project, there can be dependencies that can slow velocity, where members must wait for a component to be complete before they can progress. Collective code ownership can circumvent some of these dependencies, and hence unnecessary down-time, by exploiting a shared code-base.

However, during the projects, we observed that the application of this principle did not always lead to the stated benefits. In each of these cases, well-meaning pairs often modified parts of the code-base in isolation from other members. While this may be a technically correct adoption of the principle, this lack of team communication had a negative impact on project progress.

The AppTrack team ran into exactly this problem during the last iterations of their project when they were adding in the last stories and minor functionalities. The shared code-base allowed pairs to do this in parallel. Once a pair had finished the story they were currently working on, they could move to the next one. However, pairs sometimes ended up working on the same component, modifying the same code in order to add new functionality. This caused some inconsistencies that introduced new bugs into the system.

In a similar, but subtly different case, the FL-CMT team were bug fixing during the last iteration. Although the CVS showed that a pair had exclusively checked out some code, that code had previously been worked on and the behaviour slightly modified, without conveying this change to the rest of the team. This lead to some confusion between team members as to the actual working of components that had previously been thought complete. The main consequence of naively applying collective code ownership is confusion among team members about the status of different parts of the system. This in turn can lead to implementation problems where occasionally behaviour already implemented was duplicated or even rendered incorrect.

We agree that collective code ownership can help reduce development time. However, it should be applied in conjunction with good communication so that all members have a good understanding of all parts of the system at any time, even if they are not currently working on it, thus eliminating the problems described above.

What is mine is yours, arrange before you change. Collective code ownership is very useful, but you should have strict control structures around the code so that everyone can see who is working on what part of the project so that there is no conflict.

4.7. Stick to standards, or things can get sticky

Conformance to coding standards ensures that the code produced during the course of the project all uses the same style and nomenclature. This code should appear to have been written by a single, very competent individual.

We observed that teams defined standards which were very rigid. Some teams, such as the BUS team, were very rigorous in their approach to the coding standards across the board. This naturally lead to code that was consistent, both in terms of style and naming systems.

In the LAG group, the team inherited some code for interfacing with the GPS receiver kits. This code did not conform to the standard that was defined by the team for the rest of the project. New code that was added to the GPS modules conformed to the old standard to ensure continuity and legibility, whereas new code for the UI and the game logic was written within the guidelines defined by the team. Additionally, the code that interfaced with JXTA for peer-to-peer communication followed an existing third style.

Contrary to what might be expected from this, the members of the LAG team did not experience any difficulties when switching between different sections of the project. We believe that this was because the LAG group frequently rotated pairs and everyone worked on all of the parts of the project at different stages, meaning that the whole team was conversant in all aspects of the project.

The FL-CMT team also had to interface with legacy architecture and technology. The developers initially accepted the agreed standards, but soon we observed that the standards were being ignored, because legacy code did not fit the standard. The team was not affected by this in the early iterations.

However, as the semi-standardised codebase for projects expanded, so too did the cost of change. Towards the end of the project, attempting to refactor code became difficult, due to naming policy clashes between

developers. Measures were put in place to halt the abandonment of the standards, including the use of code formatters and refactoring to ensure consistent naming.

We observed that teams who had to use existing code and architectures found it very difficult to maintain a single code standard, because of the differing styles between legacy code modules. Teams can address this in one of two ways – abandoning all standards, or defining different standards for subsections of the project. Although the effects of abandoning coding standards are not felt immediately, the cumulative effect hindered the project in later phases.

Stick to standards or things can get sticky. Code standards can be difficult for some developers to follow, but they do yield benefits as the project grows. Agree early on standards and enforce them.

4.8. Don't all think the same? Who's to blame?

The concept of a common metaphor in XP is analogous to that of a company mission statement. It sets out a common vision or goal that all team members (or employees) can strive for. This can ensure that workers are always working towards the same outcome and can also serve as a motivating factor for people to perform better in a bid to achieve the specified goals.

All teams started out with common metaphors. This was either a phrase, architecture, or both that all teams members strived to achieve. Although these teams started off aiming for a common goal, this goal skewed or evolved as development progressed.

In the AppTrack team, the common metaphor was closely aligned to their architecture. Although the metaphor skewed slightly as the project progressed, it was communicated throughout the team and did not lead to any misunderstandings. However, this communication was not evident in other teams, which lead to huge problems for one in particular.

The TTALK team began with a strong coherent idea of the product that they were required to build. As discussed above, the team had many problems trying to understand the technologies that they had to deal with. During this period there was no development and team members were spiking individually to assess new technologies and products. As there was no focus the team members all began to view the project deliverables differently.

When the TTALK team began development they had to realign their common metaphor. The main consequence of this was confusion among members, which slowed development and sometimes caused unrest between team members. Pairs were often programming towards different goals and arguments became more commonplace. The distortion of the metaphor caused a lot of problems for the team in terms of lost time investigating technologies not directly related to their requirements. The team also suffered a hit to team morale as the process of realigning the metaphor involved much argument and blame.

We believe that a common metaphor is useful for teams, but members must be careful that an agreed metaphor evolves as the team and the product evolves.

Don't all think the same? Who's to blame? Members should be in constant communication to ensure that for the duration of the project, a common metaphor does indeed remain common.

4.9. Champion of the court

Having an XP champion (a person who has experience using XP and drives the process from within the team) is beneficial for teams to ensure that members are adopting the XP process correctly.

As described in Section 3.3, our teams did not have any XP champions and it was the responsibility of either one member or the whole team to learn and adopt XP in a manner that they saw fit.

In the former case, it was common for other members to constantly question the individual responsible for championing, creating an undue pressure on that individual member. Doubts were always present in the minds of members as to whether principles were adopted correctly, thus affecting the overall project. We observed this scenario in the LAG team, where a team member seen as competent was elected as XP champion. In this role the champion was responsible for running the XP process. The LAG champion (new to XP himself) was faced with tough decisions which had to be justified to the team. In this situation when the champion made a bad decision, the team saw it as a reflection on the process rather than a human error. On the other hand when the decision was correct team members felt that they were excluded from the decision making process.

In contrast is the latter case where all members actively became part of the process, and were therefore more enthusiastic and motivated by the process of XP, and thus leading to better results. This situation was observed most prominently in the BUS team. The BUS team set out by appointing two champions, but each member of the team took a large interest in the process and soon each team member became a champion. In this situation decisions on testing, planning and other process related topics were agreed democratically. This approach worked very well as decisions were made in a more informed way. The only drawback was that decisions would undergo much more debate than having a single champion, occasionally proving a sticking point.

In our experience, we observed that groups who elected a single member to champion the process were less successful than those where all members became champions. Consequently, we believe that it is more

beneficial for all team members to be champions who all fully participate in driving the XP process, rather than assigning this responsibility to a single individual.

Champion of the court. It is always beneficial to have an experienced XP practitioner to drive the early stage of the XP process. However, teams will enjoy much more success if everyone buys into the XP ethos and they drive the process together, although a single voice with whom the final decision lies should be elected and listened to.

5. Reflection/Discussion

Three levels of XP maturity have been observed [11]:

1. Do everything as written.
2. After having done that, experiment with variations in the rules.
3. Eventually, don't care if you are doing XP or not.

Our experiences centre on projects where the teams are new to XP. In all of the projects over the three years, we have witnessed the transition from maturity level one to two. We have repeatedly seen teams make the same mistakes over and over again. This is generally because they tend to follow the XP principles verbatim. We have narrowed to a single factor how the teams move from XP maturity one to two and three. This factor is buy in, that is, how much the team understands the reasoning behind the process and how much they want the process to work.

We observed that at some point the participants realise that XP is an ethos and not a strict rule set. By rigidly conforming to the rules, teams experience benefits and drawbacks. The teams gradually begin to realise which XP principles work for the context they are in and those which do not. In our experience, teams who want the process to work have bought into the process and adapt the rules. They begin to understand the concepts behind the rules and bend the rules to fit their circumstances, in line with the second level on the XP maturity scale.

In Section 4 we presented many cases that illustrate this point. Each subsection describes a situation that we have encountered where one or more teams have naively applied an XP principle. In each case, the teams that overcame the problems succeeded by adapting their approach to suit the circumstance.

The vital ingredient in a successful project with an inexperienced team is to get the team members to buy into the process. Once everybody is committed and wants XP to work for them, we recommend firstly looking at the experience of others and learning from their mistakes. Then enter the XP process armed with your twelve principles. However, treat these as guidelines rather than commandments and soon the team will begin to appreciate the ethos. In our experience, the true value of XP is found through a solid understanding of the ethos and bending the rules to fit your project and environment.

6. Future Work

There has been interest expressed in the psychology behind eXtreme Programming. Current work in the area is limited, in that the studies that have been carried out fail to answer the question *why* the XP practices work or fail [12]. We have conducted psychological profiles of the participants in our studies, and continue to expand this data set. With this data we hope to gain an insight into the psychology behind XP and answer the questions of why technically proficient XP teams succeed or fail to leverage XP. Now that we have identified buy-in as a core requirement for effective application of XP we wish to understand buy in and the psychology behind it further.

7. Acknowledgements

We would like to thank the team members of all the projects that have been included in our experience. We would also like to acknowledge our shepherd, Peter Brown, for his understanding and input.

8. References

[1] Lippert, M.M., Roock, S., Wolf. "*eXtreme Programming in Action: Practical Experiences from Real World Projects*", Addison-Wesley, 2002.

[2] Jefferies R., Anderson A., Hendrickson, C. "*Extreme Programming Installed*", Addison-Wesley, 2001.

[3] Beck, K., "*Extreme Programming Explained: Embrace Change*", Addison-Wesley, 1999.

[4] Astrachan, O., Duvall, R.C., Wallingford, E., "*Bringing Extreme Programming to the Classroom*" in XP Universe, Raleigh, North Carolina, USA, 2001

[5] Schneider, J-G., Johnston, L., "*eXtreme Programming at Universities – An Educational Perspective*" in International Conference on Software Engineering, Portland, Oregon, USA, 2003, pp. 594-599.

[6] Nokia, http://www.n-gage.com/en-R1/home/home.html

[7] Olias-Sanz, A., Stamouli, I., West, D., Wu Qiu, R., http://trinytalk.jxta.org/servlets/ProjectHome

[8] Bassot, V., Vitaliev, D., http://www.frontlinedefenders.org/en

[9] Hedin, G., Bendix, L., Magnusson, B., "*Introducing Software Engineering by means of Extreme Programming*" in International conference on Software engineering, Portland, Oregon, USA, 2003, pp. 586-593.

[10] Wood, W.A., Kleb, W.L., "*Extreme Programming in a Research Environment*" in XP/Agile Universe, Chicago, Illinois, USA, 2002, pp. 89-99.

[11] Cockburn, A., "*Agile Software Development*", Addison-Wesley, 2001.

[12] Bryant, S. "*XP: Taking the psychology of programming to the eXtreme*", Psychology of Programming Interest Group Annual Workshop, Carlow, Ireland, 2004.