Using Extreme Programming to Manage High-Risk Projects Successfully

MARTIN LIPPERT AND HEINZ ZÜLLIGHOVEN
it - Workplace Solutions GmbH & University of Hamburg (Germany)

Abstract: Today, software development and its process management is a demanding task. The development company must reach the project goals on time and on budget. On the other hand, requirements change daily, developers are not domain experts and customers want to maintain close control of the project. Web application development has increased this challenge as new project-management issues have to be met. Extreme Programming (XP), a lightweight development process, is designed to meet the challenge. We have used XP successfully in a number of projects and attained the goals mentioned above. Using XP enabled us to deliver software on time and on budget, while supporting close communication between the (potential) customer and the development team as requirements changed daily. One to two releases per customer per week (two per day at the peak) indicate the flexibility and risk-minimizing capabilities of the process. This allowed optimal control and planning of the projects by the development company. We illustrate our experiences using one particular time-critical project.

Keywords: Extreme Programming, Project Management, Agile Processes, Risk Minimization

1 Flexible Processes for High-Risk Projects

Everyone of the software community is talking about Extreme Programming (XP, see [1]), but few people actually apply it in their projects. Most developers, managers and project leaders see XP as a process without a plan, without documentation and, most important, without control. They look on XP as the tribal rites of a group of hackers. A more serious view accepts XP as suitable for very small application programs where there is no need for major planning or specification efforts. Critiques point to unfamiliar things like “lack of upfront design” or “pair programming”, viewing them as contradictory to the principles of software engineering. There seems to be a general consensus that XP will never work for large systems and high-risk projects where one needs to be in control of what is going on.

We will try to unriddle some of the myths about XP and show that it does not mean working without plans, design or control. These fundamentals are simply used and real-

ized differently. And, as with every methodology or technique, XP ideas must be adapted to meet the user's specific needs.

We have used XP in a number of successful projects and will demonstrate how to face the risks of today's software development projects. We have delivered high-quality software to our customers in cycles within weeks. This is crucial for web applications. And we were in control of the projects and able to react to changing requirements on a daily basis. Problems that caused other projects to fail or run out of funds hardly affected us at all.

To attain these goals, we adapted Extreme Programming to our needs and combined it with other project-planning techniques. Today, we have elaborated our planning techniques to enable us to achieve optimal control and planning for high-risk projects within a flexible process.

1.1 Risk

Development projects often involve a lot of risk. Reducing and managing risks is one of the major challenges of project management. But what is the nature of these risks? What are the main problems causing risks? Based on a number of high-risk projects, we have collected the following list of risk causes in software projects:

- Often, the team has to develop a software system for an application domain they are not familiar with. Such application domains are usually complex and hard to understand. The chance of making mistakes due to misunderstandings is high.
- New technologies are emerging every day. Frequently, they must be used for a software project, though the developers have little or no experience with their use. Sometimes it is the combination of new and untried technologies that is troublesome, but in most cases the impact of these new technologies is unclear to both the developers and the client. Thus the risk of having to replace an unmanageable technology late on in a project is high.
- User requirements change daily. The reasons are manifold: the client learns more about the possibilities of the new system during the development process; the client's organization changes; and the customer nearly always has new ideas after seeing the first part of the system running.

These are some of the causes. There may be more risks arising in a development project, but the above list should do for the moment. The crucial question we face is: How can we reduce major risks?

Analyzing the list of causes helped us to draw up some ideas and guidelines. These we have successfully used in a number of projects to reduce risks. Here, then, is our list of “first-aid” measures to tackle project risks:

- Deliver small, usable systems as early as possible. Get feedback from the users by trying out the small system. A short “author-critic cycle” helps avoid building unsuitable or unusable systems. And it reduces the risk of dissatisfied customers (who are not normally willing to pay for poor work).
- Present the future development of the system as a sequence of small, comprehensive iterations and discuss them with your customers. Plan only the next small release in detail and explain it. It is essential to react swiftly to changes.
- Provide for a flexible development process – crucial in most settings – but make sure that you will be able to plan and control it. Remember that resources and money are limited, even in new-economy companies.

Obviously, traditional software engineering methodologies and processes are unable to meet these requirements. The key question is: What kind of processes do we need to attain these goals?

1.2 Extreme Programming

Extreme Programming is an example of an agile process. Agile processes are based on a set of principles (see [5]):

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late on in development. Agile processes harness change to the customer's competitive advantage.
- Deliver working software frequently, every couple of weeks or couple of months, with a preference for the shorter time scale.
- Business experts and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective way of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.

- Continuous attention to technical excellence and good design enhances agility.
- Simplicity – the art of maximizing the amount of work that doesn't need doing – is essential.
- The best architectures, requirements and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Extreme Programming itself is based on a four values: Simplicity, Communication, Feedback and Courage. They are all used in combination with one another, allowing us to form a development team that is able to deal with risks. Using the twelve techniques contained in the Extreme Programming concept (see [1]), we can create a development process that is geared to low risk.

But why is XP a feasible solution? An important feature of the XP idea is the value of feedback. If a team gets feedback early on, project management can react to that feedback and steer the project in the right direction. In order to get early feedback, the team must develop valuable software within a short period of time. This does not mean six months or more: we are talking about release cycles of one month or less. With very short release cycles, we get feedback from the customer and become aware of problems and misunderstandings very early on. This is invaluable.

However, it is not easy to create a workable release every two weeks. The small-releases technique can only work in conjunction with the other features: continuous integration, refactoring, testing, simple design, pair programming, a 40-hour week, coding conventions, on-site customer, collective ownership, metaphors and the planning game. All these techniques help to build a system of interconnected elements. Using these techniques in combination is not a simple job for "hackers" that can be done at will. A team needs a lot of discipline to learn the XP way of software development. But once they have the concepts, they will be a powerful team. They will produce high-quality software at high speed on time and on budget. And customers can change their requirements every day. What else do we want?

This short section on XP can only serve as a quick overview. We do not attempt to describe every technique in detail in this article. It is largely devoted to the planning and steering aspects of XP. Changing requirements every day sounds great, but it suggests not being able to plan anything. Let us take a closer look, then, at how we can plan a project and how we can stay in firm control of the costs. Then we will see how we can adapt XP to meet our needs.

2 Steering XP Projects

The planning and steering aspect of pure XP is fairly simple. We will summarize the main points in the following sections. They should demonstrate how realistic and iterative project planning can be done.

2.1 The Planning Game

Based on so-called story cards, the development team and the customer can plan their project together. The customer writes stories about what the system should do. These stories are not detailed specifications of the system. Figure 1 shows some examples taken from [2]. They provide informal information. Therefore they are backed up with one of the most important elements of XP: the dialogue between the customer and the developers. The developers' first job is to understand the stories. The second step is to assess each story and to give the customer an estimate of how much they can get done within the next iteration. The customer then prioritizes the stories, deciding what is to be built first and what is to be realized in a later iteration.
User stories are written on cards. That is why we call them story cards. It is very convenient to have them written on cards when discussing them in the planning of the next iteration.

Sort available flights by convenience When you're showing the flights, sort them by convenience: time of journey, number of changes, closeness to desired departure,

Print immigration paperwork Print paperwork required to leave and arrive at a planet – only for the easier planets (e.g., not Vogon)

Figure 1: Example user stories. They can be enhanced with fields for date, planned iteration, estimation, etc.[1]

This planning game not only *sounds* simple and straightforward, it *is* simple and straightforward. The most demanding task in this game is assessing the stories. Based on their experience, the developers should give the customer an idea of how much can be done in the next iteration. This is not a

[1] We use these examples from the XP planning book to give you an impression of what user stories are like. The stories we used for the project (see below) were somewhat more detailed, but unfortunately we are not allowed to publish them.

trivial task. But the developers don't have to assess the whole system as they do in traditional projects with bulky requirement documents. They only have to deal with a small part of the future system. And they have the chance to ask the customer when something is unclear. Sometimes the stories are too long. The customers must then split them into smaller parts, the developers assessing each part individually. Developers learn from assessments they have made in the past. They can look at a similar story they have realized in the past to see what will be a realistic time span. Using this kind of assessment helps the team to be increasingly precise in estimating projects. After a few planning-game cycles, the time and resource plan will be fairly accurate. This is usually the case after a few weeks if the release cycles are short enough. (More about story assessment can be found in [2]).

2.2 Small and Frequent Releases

We have already said that early feedback is essential to reduce risks in a software project. And we have also discussed how to get early feedback from the customer. The key is to deliver small releases of valuable software to the customer at the end of nearly every period of short iterations. Each iteration might take one week and a first release should be delivered after two weeks of development. This is a good basis for continuous feedback.

At first, this might sound strange. You might say that this kind of release cycle would be wonderful to obtain early feedback, but a release after two or four weeks? Never! This objection is justified if you think in terms of current project settings with separate software components that are realized by independent programmers or individual subteams. Given this kind of division of labor, software integration is a major and time-consuming task. So how can we make small releases possible without subjecting the software team to round-the-clock stress? The answer is clear: Continuous Integration.

The idea of continuous integration is to make small changes to the system and integrate these as often as possible, at least once a day. This provides an excellent basis for the team to share code and benefit from one another's development activities.

And then there is the testing strategy. You write the test first, and then the new program feature. If you are not that rigorous, you at least do extensive unit testing.

In combination with the frequent testing, the small-releases strategy is easy to pursue, so easy that it does require thinking about, because if you make small changes to the system and continuous integration is combined with unit testing, you at least have the previous integration as a running system. Then you might deliver this version as a release to your customer.

Why not? Maybe it does not contain all the features you would like to present to the customer, but if there are fixed dates for the release, you will have a running system that is as close as possible to the customer's requirements. Your customer will see that the system can do something valuable. Often, this is much better than delaying a release merely in order to achieve the complete set of features.

2.3 Project Planning

We have discussed small releases and risk reduction with early feedback. This is important and forms the foundation for project planning. Without this sound foundation, no planning will be possible.

The simple XP planning game can be used not only to plan the next one or two iterations, but also for planing the major steps of a project. The participants then play the planning game on a larger scale, making a rough plan for the entire project. This is sufficient for smaller or less complex projects. If our projects are on a major scale, we would like to reify the whole planning process by adding artifacts (see also [6], [7]).

Martin Fowler and Kent Beck propose a simplified version of a release plan to do the rough long-term planning (see [2]). We have extended this idea. The large-scale planning process described in the next section has proved useful in various projects over the past ten years and fits in well with the planning ideas of XP.

2.3.1 Kernel System with Extensions and Special Systems

We always begin by clarifying which part of the system is to be realized in the project. In many cases, we come to the conclusion that an application system with the proposed functionality does not need to be built in a single project cycle. We then break down the target system into subsystems. We call this concept a *kernel system* with *extensions* and *special systems*.

In this way, we can decompose a complex system into parts that are meaningful for the application domain and can be developed at different times. In most cases, the *kernel system* has to be realized first. It provides the basic services that are needed by nearly all the other parts of the application. In domain terms, the kernel system meets urgent requirements. As the kernel system itself is often still too big to be built “at one go”, we break it down into *extensions*. These extensions are defined in terms of logical dependencies between the system components from the domain perspective. The components on which all other components depend are the first to be built. Figure 2 shows a kernel system – designed to be built in five extensions – of process

automation software for hot rolling mills. The software components of Extension 1 are needed for the components to be built in Extension 2, i.e. the Primary Data Handler builds on the Telegram Handler. The component for Model Computation to be developed in Extension 4 cannot be developed without the Extension-3 components Measured Value Processing, Model Control and Material Tracking.

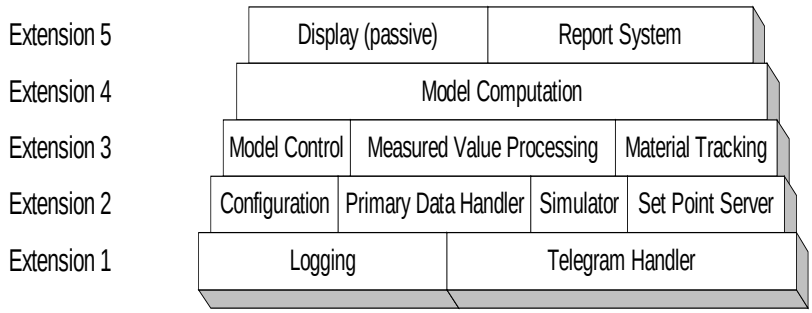

Figure 2: Example of a kernel system with extensions

For the kernel system and extensions concept, it is important that the system is decomposed along domain lines and in a user-comprehensible manner. This enables users and specialist departments in the application domain to be involved in the planning process and to adjust to this planning horizon. We select the extensions such that they can be delivered to the users (or at least some of the future users) as runnable versions or components. An extension should, at the very least, always be realized as an evaluable prototype. This is very important, as an operative system or evaluable prototype usually has a strong impact on the application domain. It gives both users and developers the necessary experience to decide what functionality the subsequent extensions should actually contain. So, the initial specification of kernel system and extensions serves as a kind of commonly accepted blueprint for the overall project, while each extension adds more insight into how this system should be built in detail.

Special systems can be added to the kernel system. These should be designed to interface with the kernel system, but should otherwise be independent of one another. This also enables them to be developed concurrently in subprojects. In the case of special systems, it is often worth looking for finished, off-the-shelf solutions.

2.3.2 Stages and Reference Lines

If we are planning a large-scale project in iterations, we must often take into account external schedules and domain constraints. This leads to a rough planning in what we call *project stages*, which, in turn, are planned individually, in terms of XP, using *reference lines*.

A *stage* defines a – from the domain perspective – relevant state of the project at a specific point in time. A project stage is reached when the substantive goal is realized and a runnable system version, i.e. a prototype, an extension or a component is ready. Stages thus become synchronization points for developers and users. The necessity of integrating executable systems in the project planning at manageably short intervals enables the customer to keep track of the project status. This motivates developers, in each case, to make for a domain-related subgoal in a way that is comprehensible in terms of the project as a whole. Evaluation of the subgoal based on the running system version may lead to the overall goal being redefined in consultation with the users.

We view a stage as a complete miniproject in which all essential development tasks are run through. In the case of very large projects or projects running for a long period of time, it will be worthwhile adding to the XP planning mechanisms a further planning level with reference lines.

Reference lines define work packets quantitatively, and in particular qualitatively, on the basis of verifiable document or system states and specify responsibilities. To this end, they contain a rough assessment of the planned effort. Reference lines describe work packets specified and implemented using XP techniques.

2.3.3 Selective Prototyping

Early feedback from users and customers is, for us, inextricably linked with prototyping. Different tasks can be supported by prototypes: project initiation, analysis of the application domain and design and construction of the application system. For each of these tasks, one or more *kinds of prototype* are suitable:

Prototype	Purpose
Presentation Prototype	A presentation prototype supports project initiation: it is designed to show the client what the application system might in principle look like. It is intended to give developers and users a basic idea of the system's handling and application context. Presentation prototypes are mostly evaluated by management.
Functional Prototype	A functional prototype helps to clarify the task and answer design questions. It shows parts of the use interface and a section of the functionality, i.e. relevant parts of the use model. In most cases, it already has the architecture of the application system and thus of the technical design. Functional prototypes are evaluated by all groups.
Laboratory Prototype	Laboratory prototypes are designed to clarify software engineering questions arising during development of the application system. For this purpose, they realize a section of the technical and the implementation model. This kind of prototype is also found in traditional development projects. Laboratory prototypes are evaluated by the developers.
Pilot System	A pilot system is deployed and evaluated in the application domain. It is a technically »mature« prototype validated in both domain and technical terms. An initial pilot system often corresponds to the kernel system of the future application. It is added to in an evolutionary manner with extensions. A pilot system offers convenient and reliable operability and minimal use documentation.

There should be a specific question that each prototype helps answer. It is important that this question be clearly formulated before the prototype is built. This helps to prevent the prototype from being evaluated according to criteria for which it was not developed. Without a clearly formulated question, there is a danger of "muddling through", i.e. of arbitrary executable software components being built, which are sold as a success if they appeal or rejected if they fail to appeal on the grounds that "it was only a prototype after all."

Prototypes are produced throughout the development process. This is why, in large projects, we use the whole range of prototype kinds depending on the question in hand. But this also means that prototyping is not reduced to a stage in the development process, e.g. to support requirements analysis.

Incidentally: if you have read this section carefully, you will have recognized that our prototypes are closely related to the Spike Solutions described in [4].

3 Experiences: Facts and Numbers

Now that we have introduced the XP concept, you may feel that this is just a lot of theory. People often have the impression that XP will never work in practice or for their own projects. But here is the good news: there is a body of experience to the contrary, as we will show in the following sections.

We conducted a number of projects using XP for software development. To illustrate our point, however, we will focus on one particular project[2]. This was rather time-critical and therefore a good example for this kind of project.

The project was based on a two-month prototyping period at the end of 2000. Three developers participated in the prototype development, all of them part-time. We thus had 31 programming days to realize the different prototypes[3].

The development process of the actual product began in January and ended in late March 2001. The main deadline was March 22nd. This date was fixed because of a world exhibition at which our customer wished to present the product. During this period, the development team consisted of seven or eight people, which included the project management. Most of them worked for the project only part-time. The real number of programming days per week was 20 during the first month and 25 during the next two months. We reached 30 programming days at a peak.

The requirements could not have been unclearer. The customer wished to create something totally new for e-processing. The users of the future system were to be able to create, for example, a web-based shopping application. It was to combine a web front end with back-office workplaces, using the new product without programming a single line of source code.

This description looks fairly vague, but in fact there was little more to go on. How would you assess this project? We couldn't.

So we started by building a prototype to enable us to get an overall idea of the product. This not only helped us to reduce risks, it was a great help for our customer as well. We were able to create a common metaphor for the product and demonstrate what the product would look like.

At the end of the two-month period, the prototype exhibited the core functionality of the future product at a very basic level. That was a great improvement on what we had before prototyping. The prototype suggested that we might be able to build the actual product within the remaining twelve weeks. The main technical questions were clarified, leaving only a few technical problems to be resolved. That was a great help. We were able to demonstrate to the customer that we would meet the deadline with a presentable product. The user interface would not be as neat as it might be, but the core functionality would be ready for demonstration and use in an in-house project. The customer was happy with these constraints and all parties committed to the plan.

---

[2] We have since conducted other projects with equal success.

[3] We actually built one major prototype (a combination of a presentation and a functional prototype) and several small laboratory prototypes. In what follows we will talk mainly about the major prototype.

However, this was obviously not a traditional project plan. No detailed requirements were specified, nor did we have a list of all the requested features. There were still many high-risk issues left.

The next step in the process was actual development of the product. And the first serious requirement changes surfaced after only 15 days. Our customer began to attach more importance to a nice graphical system interface. How did we deal with this new requirement? We established the process described in the next section.

3.1 Small Releases

We used continuous integration and small changes to ensure a running system every day. That gave us good control over the process, i.e. high reliability and flexibility. We had the earliest running system at the end of the first day. Does that sound impossible? The prototype's architecture (using the JWAM framework, see [3]) was sound enough to be used as basis for the product implementation.

It took two and a half weeks, however, to produce the first release. This delay was due to minor difficulties in the start-up phase of the project. First, we had to establish the software team. A number of developers were new to the team and had to be integrated. Then we had to establish communication and cooperation with our customer with a view to product development, which was not as easy as we thought because of Christmas and seasonal holidays and vacations.

The following table gives a detailed overview of the small release cycles over the 12-week development period. And remember: every release was a running system suitable for demonstration purposes. It was also used for planning the next release.

Week	**Releases**
01/01 – 01/05	--
01/08 – 01/12	--
01/15 – 01/19	Release 2001-01-18
01/22 – 01/26	Release 2001-01-24
01/29 – 02/02	Release 2001-01-29 Release 2001-02-01
02/05 – 02/09	Release 2001-02-08 Release 2001-02-09
02/12 – 02/16	Release 2001-02-12

	Release 2001-02-16
02/19 – 02/23	Release 2001-02-20 Release 2001-02-22 Release 2001-02-23
02/26 – 03/02	Release 2001-02-26 Release 2001-03-01
03/05 – 03/10 (includes Saturday)	Release 2001-03-05 Release 2001-03-10
03/12 – 03/16	Release 2001-03-12 Release 2001-03-13 Release 2001-03-15
03/19 – 03/21 (03/22 fixed deadline)	Release 2001-03-19 Release 2001-03-20 Release 2001-03-21
Total	21 releases

Table 1: Releases per week

As you can imagine, weekly releases and daily builds are not for free. The quality of the system must be kept at a constant high level. We realized that by using constructive quality assurance. In detail, it was realized by unit testing, refactoring, continuous reviewing (with pair programming), coding standards and design by contract. More details can be found in [1] and [8]. One of the most important factors in making daily builds and weekly releases both possible and smooth is the Continuous Integration facility of Extreme Programming. In fact, very short release cycles cannot do without Continuous Integration[4]. Table 2: shows the number of integrations in the project.

Week	**Number of Integrations**	**Ø Integrations per day**
1st week	24	4.8
2nd week	19	3.8
3rd week	25	5
4th week	43	8.6
5th week	45	9
6th week	93	18.6
7th week	91	18.2

[4] The different XP techniques (12 in all) benefit from each other. Continuous Integration is, then, not the only prerequisite for small and frequent releases, but here we focus on this technique.

8th week	105	21
9th week	67	13.4
10th week	115	23
11th week	77	15.4
12th week (only 3 days)	44	14.7
Total	748	12.9

Table 2: Integrations per week

This kind of development provided us with a running system every day. And that was not the only benefit. Each release also meant feedback from our customer and a platform for continuous planning. This is an important key to reducing risk within a software project (cf. Section 1.1).

The process benefited in another way, too: by using this kind of iterative development, the user was able to change priorities whenever he/she wanted. We will take a closer look at this in the next sections.

3.2 Internal Daily Planning

The main planning was done for every new release. Thus, the scheduling of development activities was done for one week at a time. Our customer presented his priorities for the next release, which we then used to adjust our plans. Sometimes it was not just the priorities the customer wanted to change. Over the twelve weeks, the product requirements changed, but at the same time they became clearer. We observed a very common phenomenon: every new product feature in a new release raised new questions.

These questions led to new feature requests, new story cards or changes to realized functionality. These results of mutual learning and communication are not merely consequences of the iterative process. If development is not done in short release cycles, all these problems typically surface at the end of the project, causing major problems with the customer. Our short release cycles enabled us to deal with these problems, which are common to every development project.

But what about project planning? Is planning possible with all these moving targets?

Yes, it is. But you need to plan from a different perspective. We used daily activity planning for the project, which involved a stand-up meeting every morning (see [4]). At the stand-up meeting, each developer explained what he/she had done the day before and what he/she planned to do that day. Then we updated the daily schedule on the whiteboard.

Figure 3 gives an idea of what a schedule on the whiteboard looks like. Each row is for one team member, each column for one development day. Daily cells contain the numbers of the story cards to be processed by a team member. Project previews for the following week were also possible using this daily schedule.

We deliberately chose a whiteboard for daily scheduling of activities, instead of an excel spreadsheet or other planning software. One of the most important aspects of daily planning is communication within the team. This can be improved by using a whiteboard that everyone can see and point to. A software product would be inconvenient at planning meetings because everyone would have to sit in front of a computer screen.

Of course, the project manager can transfer the whiteboard image to project-planning software or a simple spreadsheet in order to present the plan to top management.

In addition to the activities, each release day was marked on the whiteboard with an “R”. Below the daily plan, we drew a small table showing the story-card estimates and the time taken to realize story cards. By working with story cards and the daily planning schedule, we were able to respond very quickly to wrong estimates.

How did we respond? First, the project manager checks whether the other story cards are on time. If some cards are likely to be done in less time, resources are shifted to enable the late story card to be finished within the iteration. If not, the manager notifies the customer. They discuss the situation frankly and decide what to do. The options are clear: the customer has at his disposal all the story cards that are still unfinished and can thus rearrange

Figure 3: Daily planning on the white board

priorities for the current release. Our customer realized very early on in the project that a wise choice on the available design options was vital to the success of the project. So the story cards for the next release were chosen with great care. Once the customer had made the decision, the developers could be

sure they would deliver the most valuable system possible to the customer with the next release.

This is the way to respond to changed requirements and wrong estimates. And, we found, it is another key to reducing project risks.

3.3 Estimates and Forecasts

One of the main tasks of project management is dealing with estimates and forecasts. We did initial planning based on the prototype and estimated that we could build the system within twelve weeks. Obviously, these estimates were rough for all participants. But how could we get better estimates and ensure planning safety?

Every time we worked on a story card, we learned more about the efficiency and capabilities of the team. We wrote down the time needed for each story card every day. This helped improve our ability to estimate. Kent Beck and Martin Fowler call this the Yesterday's Weather technique (see [2]).

Within the project, we had to assess about 30 story cards with a wide range of issues. Table 3 shows how the estimates evolved: the first 26 finished story cards with the estimates, reduced to four releases within the twelve-week period. The next column shows the real size needed to finish the set of stories. The last column is the most important. It shows the difference between the estimated and real sizes of the stories. As can be seen, we had a difference of about 10% at the end of the project and a fairly good development over the twelve- week period.

Release	**No. of story cards finished**	**Estimated size**	**Real finished size**	**Difference between estimated and real values**
2001-01-08	5	20	25	+25%
2001-02-01	8	39	43.5	+10%
2001-02-22	17	109	124.5	+14%
2001-03-13	26	169	156.25	-8%

Table 3: Estimates and real values.

This helps to provide the customer with more accurate feedback on changes to requirements or priorities. And we demonstrated that we were able to improve the estimates within this fairly short project, achieving a fairly acceptable size. Using this kind of estimate enabled us to make minor project corrections smoothly. Kent Beck demonstrates this idea of project steering

using the car-driving metaphor (see [1]). Many small corrections are better then a few big ones.

3.4 Lessons Learned

What are the main lessons we learned using this software development approach for professional high-risk projects?

- *The steering capabilities of this approach are excellent*. We were in control of the project throughout the entire development process. Projects risks were reduced to a very acceptable level.
- *The process is easy to communicate and handle*. After initial skepticism, our customer responded very positively to the flexible process and the fairly good project estimates. The team was able to respond quite smoothly and quickly to changes without extra cost.
- *The process has to be adapted to the individual project settings*. It is not a ready-to-use process with an "instruction manual". Users must learn how to adapt it to their specific needs.
- *Unit Testing is essential*: The general Extreme Programming description recommends test-first programming. It guides the developer to simpler interfaces and provides major support for constructive quality assurance. We made the mistake of not incorporating this element of XP as an essential in our daily programming. We had unit tests, but not enough of them. This caused problems that could have been avoided with a better set of unit tests.

4 Summary and Perspective

Nearly all major software projects are high-risk. Many of them are not completed successfully. We used the techniques known from Extreme Programming and adapted them to our project settings. This paper presented and discussed some additional techniques to improve the planning capabilities of XP.

In addition to the essentials for dealing with high-risk projects, we presented some material based on projects we successfully completed using this approach.

Our experience with this approach has been very positive and we are currently using similar techniques – with success – for a number of other projects.

5 References

[1] Beck K (2000) Extreme Programming Explained – Embrace Change, Addison-Wesley, Reading

[2] Beck K, Fowler M (2000) Planning Extreme Programming, Addison-Wesley, Reading

[3] The JWAM Framework: http://www.jwam.org

[4] Jeffries R, Anderson A, Hendrickson C (2000) Extreme Programming Installed, Addison-Wesley, Reading

[5] The Agile Alliance: http://www.agilealliance.org

[6] Lippert M, Roock S, Tunkel R, Wolf H (2001) Stabilizing the XP Process Using Specialized Tools, Proceedings of XP 2001 Conference, Villasimius, Sardinia, Italy

[7] Lippert M, Roock S, Wolf H, Züllighoven H (2001) XP in Complex Project Settings: Some Extensions, Proceedings of XP 2001 Conference, Villasimius, Sardinia, Italy

[8] Hunt A, Thomas D (1999) The Pragmatic Programmer – from journeyman to master, Addison Wesley Longman, Reading