Fst—a small firm with approximately 160 employees—designs, builds, and outsources Internet services. Our areas of expertise are building complex portals with multichannel access and strict security requirements, supporting digital signatures, and handling various kinds of financial transactions. Fst is not a software house; it aims at building partnerships with its customers by providing solutions to specific needs.

Fst has made large investments in Java technology for Internet development. We base our project management process on the Rational Unified Process (RUP) and use the Rational tools (Rose, Requisite Pro, and so on) extensively. Our quality system is lean; it comprises 19 procedures and guidelines and more than 30 templates for managing the software development process.

For the past two years, we have studied Extreme Programming as an improvement to our current practices. We hoped XP could address bloated documentation and low software quality. Our experience brought us to look at XP as a way of planning and managing projects and as a new risk management philosophy.

Two pilot projects
The Fst research laboratory has used XP since February 2001 and has completed two pilot projects. In the first project, we developed a set of cryptographic components in C++. We used a pared-down XP process, leaving out the approach's so-called planning game and coding standards. The results confirmed the 20-80 rule that Kent Beck proposed: If you follow 80 percent of the process, you get just 20 percent of the benefits. The first project lacked overall process control and coordination.

So, in the second project we decided to ap-
ply XP by the book, with two-week iterations and half-hour tracking units. We tried to release a set of user stories every two iterations, each time adding new functionalities that the customer found valuable. The project objective was to develop a demo of an e-procurement portal that supports digital signatures and time stamp functionality according to Italian law. As a necessary complement, we also developed a time stamp server compliant with RFC3161, and a few clients for requesting and managing time stamps. Our goal was to demonstrate the use of our legally compliant cryptographic components in a real-world scenario that our customers could understand.

Each project lasted approximately three months and employed three people. The research and development lab’s director played the customer role. The development team didn’t have much experience: five team members had been programming for less than two years each. The team worked in an open workspace, sharing workstations.

Table 1 summarizes the degree to which we adopted XP practices in the two pilot projects. But the table can’t tell the whole story. Today, even when we aren’t using XP (in projects that last less than two months), we can apply XP’s principles and are happy with the results. Differences from traditional methodologies can be striking. Recently, on a small project, we had XP-trained programmers working alongside programmers with no XP experience. The first specification proved inadequate. The XP-trained programmers concluded that too much effort went into up-front analysis. The programmers who were not trained in XP thought the original specification was inadequate and regretted not having done better. The two viewpoints were clearly incompatible and reflected totally different attitudes.

Strangely enough, the non-XP users claimed that they had “always done” XP and didn’t see anything new in the methodology. They didn’t understand how XP shifts the way programmers think about software development. XP is definitely not a new presentation of old concepts.

### The programmers’ perspective

XP provides a detailed, workable way to implement good but generic software engineering principles. Many people have noted the similarities between XP and other, more entrenched methodologies. However, it takes an experienced team to tailor the right subset of a methodology such as RUP. XP practices are limited in scope but well characterized. Al-

---

**Table 1**

<table>
<thead>
<tr>
<th>Practice</th>
<th>Degree of adoption</th>
<th>Project 1</th>
<th>Project 2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planning game</td>
<td></td>
<td>None</td>
<td>Full</td>
<td>Starting with the second project, we applied the planning game by the book. We allocated tasks based on velocity, recorded progress once a day, and recorded time to finish twice per iteration.</td>
</tr>
<tr>
<td>Small releases</td>
<td>Full</td>
<td>Full</td>
<td></td>
<td>We used two-week iterations and two-iteration releases.</td>
</tr>
<tr>
<td>Metaphor</td>
<td>None</td>
<td>None</td>
<td></td>
<td>We had simple metaphors for the systems but found them of limited usefulness. We evolved high-level diagrams of system architecture and UML diagrams on boards that the team used in stand-up meetings.</td>
</tr>
<tr>
<td>Simple design</td>
<td>Partial</td>
<td>Full</td>
<td></td>
<td>We added system functionalities only as needed.</td>
</tr>
<tr>
<td>Test-driven development</td>
<td>Full</td>
<td>Full</td>
<td></td>
<td>We used the test-first method a lot in C++ development, but had some development problems unit-testing Java Web GUIs. We used CPP-Unit and JUnit together with Rational Robot.</td>
</tr>
<tr>
<td>Refactoring</td>
<td>Full</td>
<td>Full</td>
<td></td>
<td>We refactored regularly. Refactoring was about 5% of total time in the first iterations and grew to about 20% in the final iterations.</td>
</tr>
<tr>
<td>Pair programming</td>
<td>Full</td>
<td>Full</td>
<td></td>
<td>This worked very smoothly. Pairs rotated regularly, but the C++ and Java camps tended to stay separate.</td>
</tr>
<tr>
<td>Collective ownership</td>
<td>Full</td>
<td>Full</td>
<td></td>
<td>We used version control to facilitate shared code ownership.</td>
</tr>
<tr>
<td>Continuous integration</td>
<td>Partial</td>
<td>Partial</td>
<td></td>
<td>We integrated at least once a week on a staging system.</td>
</tr>
<tr>
<td>Sustainable pace</td>
<td>Full</td>
<td>Full</td>
<td></td>
<td>Team members never worked more than nine hours per day.</td>
</tr>
<tr>
<td>On-site customer</td>
<td>Full</td>
<td>Full</td>
<td></td>
<td>A manager played the customer role.</td>
</tr>
<tr>
<td>Coding standards</td>
<td>Partial</td>
<td>Partial</td>
<td></td>
<td>We wrote C++ and Java coding standards (two pages) but did not retrofit existing code.</td>
</tr>
</tbody>
</table>
though they leave unspecified much of the machinery necessary for software development, they cover many parts of the process in great detail and in practical terms. Moreover, they are very helpful in assigning the appropriate priorities to the various steps in the software development process.

Planning game

In the second project and in every project thereafter, we used the planning game. We didn’t encounter any severe problems, but the planning sessions often became analysis and design meetings. We were able to complete stories at a satisfactory pace, as Figure 1 indicates.

Without performing a lengthy evaluation, we can’t say whether the planning game works. However, we offer the following considerations.

The XP coach must actively foster involvement in the planning game. In our experience, people new to XP assume a rather passive attitude during iteration-planning sessions. Task cards and user cards do not travel around but are traded between the two or three most senior programmers. Senior programmers write most of the stories and are eager to advance estimates and suggest possible solutions. Other team members do not actively sign up for tasks and are prone to accept tasks with little discussion of the estimates. The coach must make sure everyone feels part of the decision-making process and ready to assume ownership of the task description and estimates.

Never compare estimates and actuals. Fst used to rank its programmers according to their estimates’ precision. XP taught us that ranking is not only meaningless but also potentially harmful, because it introduces adversarial relationships in the workplace. Estimates are just a tool to gauge progress, signal the need for corrective action, and guarantee the best possible outcome. Emphasizing the initial estimates’ accuracy creates obstacles to corrective action and thus can lead to broken deadlines, bloated costs, poor quality, or a mixture of these factors.

XP minimizes the need for Gantt charts. We now use Gantt charts only to communicate the overall plan outside the team, not as a planning tool. This has made life easier. Although we tried before to apply wave planning, iterative and incremental planning, and so on, Gantt charts usually contained lots of unnecessary details and tasks whose start, finish, and percentage-complete points were meaningless. For example, what does it mean to be at 35 percent of a design phase? XP taught us that many dependencies in a chart are artifacts that the tool produces and that few dependencies are “real.”

XP enhances programmers’ sense of project control. XP is all about maximizing communication. Programmers that had RUP experience said that XP’s planning game gave them a stronger feeling of control than traditional planning did. They knew where their project was going and whether it was delayed. Furthermore, constant involvement with acceptance testing made the programmers more aware of how well the code was meeting its expected functionalities and keeping the project’s strategic goals in focus. This knowledge improved the programmers’ motivation.

Metaphor and simple design

Beck writes in *Extreme Programming Explained* that XP’s metaphor should play much of the role that architecture plays in traditional methodologies.1 We couldn’t make the metaphor concept work in any of our projects, however. After a few iterations, we found that we lacked a clear overall vision of the system, perhaps because of the team’s relative inexperience or our misunderstanding of the metaphor concept. Eventually we decided to put up posters that showed sketches of the architecture or checklists of important points to remember. We also started to write the documents that we felt could help improve our understanding of the overall system. We decided to hold dedicated design sessions at the end of each iteration. This way we managed to keep iteration-planning

---

sessions focused on planning. The problem with mixing planning and design is that this leads to discussions too detailed to have the whole group participate. We had to schedule design sessions in advance because inexperienced programmers are likely to be late in realizing that some design (not just plain conditional code) is called for. We feel that XP has no clear recipe for developing an architecture (see ISO/IEC 12207).  

**Pair programming**  
Pair programming is a powerful way to discipline programmers by averaging individual idiosyncrasies. Most people in both pilot projects had trouble accepting XP practices such as simple design, coding standards, and test-and-code. However, pairs followed the practices—following coding standards, practicing test-driver development, and so on—more consistently than individuals, because when one member is more rigorous than the other, the pair usually performs to the higher standard. Furthermore, when adopting XP by the book, the groups’ focus shifted to implementing the whole process. Everyone became less critical of project aspects that they would find objectionable when considered alone.

**Test-driven development**  
The team was able to follow XP’s test-driven development practice beyond our expectations because of pair programming, which helped discipline the development work. We found it difficult to assess whether we were doing enough unit testing because each pair had to decide independently how much testing was enough. Halfway through the project, to help developers assess the degree of test coverage, the tracker started to compute simple metrics correlating number of unit tests with number of lines, methods, and parameters.

**Refactoring**  
Refactoring relies on an aesthetic awareness that programmers can only develop with time. The group must have enough senior programmers so that most of the time each pair has the experience to do the necessary refactoring. The team also must agree on software quality. Inexperienced programmers often lack a common culture on which to judge quality, and the lack of consensus leads to minimally productive open-ended discussions.

**The nonprogrammers’ perspective**  
Introducing XP in a firm affects more people than just programmers. To evaluate the feasibility of adopting XP for most of our software development projects, we held a series of interviews with people from Fst’s marketing, project management, and quality assurance areas.

We informally interviewed nine people, each with some prior knowledge about XP, although none of them had participated in an XP project. The interviewers were familiar with the interviewees and their problems and were able to establish good communication.

We organized each interview in three sections. First we asked about the interviewees’ main problems, then we recalled or explained XP practices that could apply to those problems. Finally, we discussed the interviewees’ objections and observations. The questions we asked the interviewees depended on their roles in the firm.

The interviews revealed that the main objections to using XP don’t come from persons actively involved in building software but rather from marketing people and personnel managers. The issues that arose reflect the particular situations of an Italian firm based in Sardinia and might not be representative of other situations.

**Marketing representatives**  
The marketing people had two central observations.

*Customers do not readily accept requirements’ unpredictability.* The people who deal directly with customers feel strongly that they would have a hard time getting contracts signed without formal sign-off of requirements, fixed time and cost provisions, and penalties in case the supplier doesn’t honor some of the contract’s terms.

XP requires constant guidance from a customer or an empowered proxy to lead software development. The customer must work closely with the development team for the duration of the project to select priorities, clarify and redefine the project scope, and, as a last resort, move deadlines. If the customer requires all these elements to be firmly established at the project’s onset, the customer is implicitly rejecting the XP process’s adaptive nature.

Most of our customers want to buy software using the same kind of contractual agreements
they use to buy other products. In contrast, XP and other Agile methodologies claim that the software development process is inherently unpredictable, mainly because of the unpredictability of requirements. However, we believe that the case for the latter view is weak. Claiming that “in software development requirement changes are the norm” will not move customers who have little or no experience with software and software projects, especially if they have allocated considerable resources to requirements analysis. The concept of software requirements’ unpredictability is hard to sell because people might see it as an excuse for sloppiness, lack of vision, or whining. Every methodology recognizes that customers’ disinterest in risk management is a source of problems. Our marketing people are well aware of this, and they think that XP’s reliance on the customer as an active partner in project management is likely to exacerbate the problems that a “bad” customer would pose. Our marketing suggests that XP might suffer from a catch-22: It requires a high level of trust in the customer–developer relationship, especially in the initial project phase, but it eliminates activities (requirements sign-up, architecture drafting, and so on) that help increase the trust level during startup because they are of little use later in the project and will likely introduce rigidity in the process.

**Our customers do not own the project’s requirements.** Our customers are large corporations and public administrations, and Fst is often a subcontractor or a (smallish) partner in a group of large firms that won a bid. The people we speak to who are in charge of a project rarely own the requirements. Often they must play a difficult political role to gather requirements in the face of unclear legislation and competing requests from their superiors. Moreover, our customers often lack emotional involvement in projects. Their preoccupation with avoiding mistakes for which they might have to shoulder the blame overshadows their desire to reach project goals at optimal costs. Often this preoccupation couples with an incomplete grasp of the technological issues. These factors make our customers approach software engineering conservatively; they are not eager to assign priorities and don’t like discussing “plan B” and project adjustments. Instead, they prefer the kind of written “promises” that Beck criticizes: requirement sign-offs, detailed GANTT charts, and contracts that heap all risk on the supplier, who must deliver the “complete” system on a hard deadline for a fixed cost.

**Project managers**

The project managers made the following observations.

**XP overexposes a firm’s inner workings.** Our project managers feared that opening the development process to our customers would drive them to shop for specific programmers and try to build their project team themselves. This is happening to a certain extent in our firm. Firms sometimes oversell their ability in certain areas, trusting that their programmers will be able to catch up or that they’ll be able to find timely, competent third-party help. Having the customer on-site would make this more difficult or more risky.

**XP allows programmers to grow and maintain their core abilities.** Fst’s software process makes a strong distinction between programmers and analysts. There is a monotonic increase in prestige, salary, and visibility going from programmer to analyst and then to manager. Only programmers actually program. Analysts, who are usually team managers, spend all their time coordinating, planning, coaching, designing, supervising, reviewing, and interfacing with the firm’s other departments. These activities make analysts some of the busiest employees in the firm. Unfortunately, this organizational model is not sustainable in the long run. Software engineers can’t improve their competencies by only thinking, speaking, or reading. They must compile code and make it run. By isolating analysts from menial programming tasks, we guarantee that the technical competence implicit in the position rapidly becomes obsolete. Our firm mainly uses server-side Java programming on application servers. Suppose that in a couple years a significant number of our customers start requiring .NET on Win64 servers. How will the present crop of analysts adapt to change? In an interview, a manager explained to us some ideas for flattening the team hierarchy and involving analysts more closely in the teams’ workings. All the manager’s ideas are compat-
ible with XP, which addresses these issues systematically by means of concepts such as pair programming, collective code ownership, and the planning game.

Quality team

To manage our processes, Fst has developed a quality system compatible with ISO 9000 requirements, which define requisites for quality in production processes. We are also obtaining ISO 9001 certification, which is often required in Europe, especially to do business with the public sector. In our interviews with the quality team, we discussed the XP process’s compliance with the ISO 9001 requirements.2,6

The initial response was positive; we received feedback that XP can greatly improve ISO 9001 quality systems. In fact, XP might be as much about management as it is about software engineering. The methodology is specific about roles and responsibilities in customer–developer relationships. XP also details the planning game’s required input-artifact characteristics (tracking data, prioritized user stories, and exploratory prototypes). It then suggests procedures to conduct the planning sessions and a practical method to check progress. Moreover, XP is agile: it concentrates on a few of the process’s key aspects and gives programmers the freedom to customize the rest. Therefore, the development team can adapt XP to different needs of formal validation, configuration management, and so on. This helps adapt a true XP process to an ISO 9001 framework. This is a step forward from other software development methodologies. Many methodologies are clear about the artifacts that each development phase must produce, and provide templates for various documents. However, they often fail to detail the steps required to produce the documentation and the artifacts’ quality characteristics.

We are enthusiastic about XP and have difficulty imagining a software project in which we should not try to use it, at least in the Internet development domain. Currently, about 15 employees are using XP, half of them for external customers. The main project using XP is developing the security components of the e-procurement portal of the Italian central government.

XP is a novel methodology for software development. We found that “customizing” XP’s core practices is unwise because of the 80-20 effect. However, nothing in XP prevents the practitioner from integrating it with verification teams, design documentation, configuration management, or anything else that might be useful for a project. In a sense, XP requires these add-ons because it is so focused on the core practices that it leaves some necessary processes and tools out of the spotlight.

Our experience shows that XP is not magic, but it can work well. If a development team lacks programming experience, the results using XP will be at best marginally better than what would result from any other methodology. An inexperienced team will likely be poor at applying the XP process. Furthermore, XP relies on the team checking itself constantly for simple design, code quality, and progress rate, and these self-checks require experience. However, XP is a good choice even with an inexperienced team because it is a robust and flexible methodology. We have staffed a current project with five people who have one year of XP practice and two years of programming experience each. The coach has to travel a good deal and is available only for iteration planning. However, the team is functioning effectively, producing at the rate we expected. If we had adopted the normal RUP-based life cycle, the lead’s inability to work on the start-up documents (vision, architecture, and so on) would have stalled the project. It would likely have degenerated into an unstructured effort. Even without supervision and coaching, the XP programmers tend to stick to the rules, which are easy to follow because they have an immediate, perceptible value for the programmers.

XP’s most problematic feature is the amount of on-site customer involvement it requires. However, we would never worry about a project in which the customer is in charge, knowledgeable, flexible, available, and risk-conscious, as XP requires. Assuming a fairly competent development team, such a project would be bound for success. We often practice a split process, using XP to manage the development process and having limited direct interaction with the customer. The coach acts as a customer proxy, relaying requirements, priorities, and deadlines.
and interacting with the customer through documents and plans. We are trying to educate our partners and customers to the values of XP, but it looks like we have a long way to go.

References

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.