The New XP

Michele Marchesi

> "**Even programmers can be whole people in the real world. XP is an opportunity to test yourself, to be yourself, to realize that maybe you've been fine all along and just hanging with the wrong crowd.**"
>
> *Kent Beck in Extreme Programming Explained — Second Edition*

In October, 1999, Kent Beck's book "Extreme Programming Explained — Embrace Change" was published. Well, you may love or hate XP, but for sure you cannot say it was irrelevant. By publishing his book, sold in hundreds of thousand copies and translated in tens of languages, Beck started the tidal wave of agile methodologies. These methodologies can be fully adopted, partially adopted or not adopted at all, but no serious software engineer may overlook them. All new editions of software engineering textbooks now have a section on agile methodologies and XP.

Five years after that book, Kent again took the stage and published — with Cynthia Andres — the second edition. And you can bet that it is not just a polishing or some addition. No, this second edition critically reviews what happened in these five years, and almost completely refunds XP, albeit sticking to the original principles, or just to be able to be faithful to them.

In the first edition, XP was defined with four values, fifteen basic principles, and twelve practices. Also, Kent clearly wrote that to be XP, you had to fully apply all these famous twelve practices. Now, the twelve practices have disappeared! In the new XP, there are five values, fourteen principles, thirteen primary practices and eleven corollary practices. The new 24 practices only partially reflect the original 12 ones. Two of them, i.e. *metaphor* and *coding standards*, are abandoned.

In this report, I will briefly describe the new concepts introduced by Kent in the second edition, leaving to the interested reader the task and the pleasure to buy and read the original book — Kent Beck, with Cynthia Andres, "Extreme Programming Explained — Embrace Change", Second Edition, Addison Wesley 2005.

XP values

In the new version, XP is founded on five values, the basic root elements deemed by Beck to be those really important for a successful software development. These values are the guidance for the development itself and the inspiration of the whole methodology. Four of them are the same as original XP, and *respect* is added as the fifth value. The values for XP second edition are:

Communication: most problems and errors are caused by lack of communication. For this reason, communication among team members and between the team and the customers must be maximized. Moreover, the most effective communication is direct, interpersonal communication. Another kind of communication is between the artifacts and people who read them: they must be easily readable and up-to-date.

Simplicity: this is the most intellectual of the XP values. It says: "Do the simplest thing that could possibly work". However, being simple — not simplistic — is very difficult. It requires experience, ideas and hard work. Simplicity favors communication, reduces the amount of code and improves the quality. The main guideline is: do not plan future possible reuse — new features, when actually needed, will be easily added to a simple system.

Feedback: you should always be able to measure the system and to know how far it is from the needed features. The fundamental feedback tools are close contact with the customer and availability of a set of automated tests, which are developed with the system itself. Feedback is an important component of communication — you need to communicate to get feedback, and on the other hand, feedback results must be ready for further communication. It also contributes to simplicity. Simplicity is often obtained with a try-and-error approach. Moreover, the simpler a system, the easier it is to get feedback about it.

Courage: all methodologies and processes are tools to handle and reduce our fears. The more fear we have for a software project, the bigger and heavier the methodologies we need. Communication, simplicity and feedback allow to tackle with courage even big changes in requirements and substantial refactoring of the system. Courage alone is dangerous, but with other values it is a powerful tool to tackle changes.

Respect: the previous four values imply a fifth: respect. If members of a team don't care about each other and their work, no methodology can work. You must be respectful to your colleagues and their contributions, to your organization, and to persons whose life is touched by the system you are writing.

The five values do not give specific advice on how to manage a project, or how to write software. To this purpose, you need practices, and before practices, you need principles.

XP principles

In the new version of XP, the principles are the bridge between the values, which is synthetic and abstract, and the practices, which tell how to actually develop software. The fourteen principles of XP are:

Humanity: software is developed by people, so human factors are the main key to deliver quality software. XP aims to address the goals of the people and their organizations, so it may benefit to both. Of course, we need to find a balance among the goals. If we overestimate people's needs, they will not work properly, with consequent low productivity and business losses. This in turn could shut down the firm and heavily harm the people working for the company. If you overestimate the needs of the organization, there will be anxiety, overwork and conflicts. This isn't good for business either. In this book, people's needs are:

- basic safety — the need to maintain the job;
- accomplishment — the sense of usefulness to their own work;
- belonging — the ability to identify with a group;
- growth — the opportunity to expand their skills and perspectives;
- intimacy — the ability to understand and be understood by others.

Economics: if you produce software, you must also produce business value. Two aspects of economics are key to XP: present value and the value of options. The first says that a dollar today is worth more than a dollar tomorrow, so the sooner software development earns money and the later it spends money, the more profit the software creates. This is linked with the value of options. If you can defer design investments until they are obvious, it is very valuable for business. XP practices like incremental design, focus on customer business value, and pay-per-use ease deferring decisions.

Mutual benefit: every activity should benefit all people and organizations concerned. This is perhaps the most important XP principle, and the most difficult to adhere to. There are always easy solutions to any problem, where somebody gains and others lose. Often, these solutions

are tempting shortcuts. However, they are always a net loss, because they tear down relationships and deteriorate the working environment. You need to solve more problems than you create. So, you need practices that benefit both you and your customer, now and in the future.

Self-Similarity: nature continuously uses fractal structures, which are similar to themselves but at various scales. The same principle should be applied to software development: we should be able to reuse similar solutions, in different contexts. For instance, a basic pattern of XP is to write tests which fails, and then to write code which passes the tests. This is true on various time scales: in a quarter, you list the issues to solve, and then write the stories which describe them; in a week, you list the stories to implement, write the acceptance tests and eventually write the code to make the tests work; in a few hours, you write unit tests, and then the code able to make them work.

Improvement: continuous improvement is key to XP. Perfection does not exists, but you should continuously strive for perfection. Every day you must strive to act at your best, also to think on what to do to perform even better tomorrow. In practice, at each iteration the system is improved both in quality and functionalities, using feedback from the customer, from automated tests and from the team itself.

Diversity: teams where everyone is alike are comfortable, but aren't effective. Teams should include different knowledges, skills and characters, to be able to find out and solve problems. Of course, being different leads to potential conflicts, which must be managed and solved. Having different opinions and proposing different solutions is very useful in software development, provided you are able to manage conflicts and to choose the best alternative.

Reflection: an effective team does not just do their work. They ask themselves *how* they are working, and *why* they are working in that way. They need to analyze the reasons behind success — or failure — without hiding any errors, and make them explicit and trying to learn from them. During quarterly and weekly iterations, take time to reflect about how the project is running, and what are possible improvements. However, you should not think too much. Software engineering has a long tradition of people so busy thinking about process improvement, that they are not able to write software anymore! Reflection comes after action, and before the next action.

Flow: flow means to steadily develop useful software, performing together all development activities. XP practices assume a continuous flow of activities, not a sequence of different phases, with software release only after the last one. Only continuous flow allows the feedback to ensure that the system is evolving toward the right direction, and avoids the problems related to final integration "big-bang."

Opportunity: problems must be seen as an opportunity for improvement. You will always experiment problems, but to obtain excellence, you cannot simply correct the problems. You need to turn them into opportunities of learning and improvement. For instance, are you unable to make long-term plans? Well, use quarterly planning cycles, and revise your long-term plans every three months. Does a single developer make too many mistakes? If so, program in pairs all the time! XP practices are effective precisely because they address old problems ever present during software development.

Redundancy: critical and difficult problems should be solved in several different ways. Thus, if a solution fails, the others will prevent a disaster. The cost of redundancy is easily repaid in these cases. Software defects must be looked for, found and fixed in many ways (pair programming, automated testing, sit together, real customer involvement, etc.). This is redundant, because many defects will be found many times. However, quality is priceless. Of course, adding a practice which systematically finds defects already found by other practices is a useless redundancy, which must be avoided.

Failure: if you are not able to be successful, fail. Don't know how to implement a story? Try to implement it in three or four different ways. Even if they all fail, you have learned a lot. Is failure useful? Yes, if they teach you something. So, don't be afraid to fail. It is better to try out something and fail, rather than to delay too long for an action, trying to make it right at the beginning.

Quality: quality must be always at maximum. Accepting a lower quality does not yield neither savings, nor faster development. On the contrary, improving quality necessarily makes an improvement of other system features, like productivity and efficiency. Moreover, quality is not only an economic factor. Team members must be proud of their work because it improves team self-esteem and effectiveness. You should not confound quality with perfectionism, however. If you delay action, striving for ultimate quality, you don't promote quality really. It is much better to try and fail, and then refine the imperfect solutions found.

Baby Steps: big changes, prepared in a long period of time and made in a big shot, are dangerous. It is much better to proceed iteratively in baby steps — the shortest step that can be appreciated in the right direction. Again, baby steps do not mean proceeding slowly. A team able to proceed using baby steps can take a lot of them in a short time, thus being fast and responsive. One of the reasons behind baby steps is that a baby step in the wrong direction yields small damages, while a big step which fails can severely impair the project.

Accepted Responsibility: responsibility can only be accepted. It is easy to order to developers "Do this", or "Do that", but it does not work. Unavoidably, you ask less than what could be achieved or, more likely, more than that can be accomplished. Anyway, the person receiving the order will actually decide whether to be responsible and accept the order, or not to be responsible and start passing the buck.

XP practices

The new XP is based on thirteen primary practices, and eleven corollary practices. Primary practices must be applied first, and each of them may yield an improvement in software development process. Corollary practices, on the other hand, require expertise in primary practices, and are difficult to apply without the primary ones.

All 24 practices are very important, and should be applied in their entirety in order to obtain all possible benefits due to XP.

Kent Beck does not classify the practices. I prefer to attribute them to four categories:

- Requirement Analysis and Planning;
- Team and Human Factors;
- Design;
- Software Coding and Releasing.

Note that many practices could be assigned to more than one category. For instance, Pair Programming is classified under "team and human factors", but could also belong to "software coding and releasing". In the followings, I classified each practice under the category I deemed most appropriate.

Primary practices

Requirement Analysis and Planning:

- **Stories**: the functionalities of the system are described using *stories*, short descriptions of customer-visible functionalities. Stories also drive system development.
- **Weekly Cycle**: software development is performed a week at a time. At the beginning of every week there is a meeting where the stories to develop in the week are chosen by the customer.
- **Quarterly Cycle**: on a lager time scale, development is planned a quarter at a time.
  This is made up of reflections on the team, the project and the progress.
- **Slack**: avoid to make promises you cannot fulfill. In any plan, include some tasks that can be dropped if you get behind. In this way, you will keep a security margin, to be used in the case of un-forecasted problems.

Team and Human Factors:

- **Sit Together**: development teams should work in an open space, able to host the whole team, to maximize communication.
- **Whole Team**: the team should be composed of members with all the skills and the perspectives needed for the project to succeed. They must have a strong sense of belonging, and must help each others.
- **Informative Workspace**: the workspace should be provided with informative posters and other stuff, giving information on the project status and on the tasks to be performed.
- **Energized Work**:developers must be refreshed, so that they can focus on their job and be productive. Consequently, limit overtime working so everyone can spend time for his or her own private life. This practice in the old version of XP was called "s*ustainable pace*"
- **Pair Programming**: the code is always written by two programmers at one machine. This practice exits already in the original XP.

Design:

- **Incremental Design**: XP opposes producing a complete design up front. The development team produces the code as soon as possible in order to obtain feedback and improve the system continuously. Design is indispensable to obtain good code. The question is *when* to design. XP suggests to do it incrementally during coding. The way helpful to obtain this is to eliminate duplications in the code.
- **Test-First Programming**: before updating and adding code, it is necessary to write tests in order to verify the code. This solves four problems:
  - Cowboy coding: It is easy to get carried away to program quickly and put everything in mind in the code. If we write tests and you have to run them, the tests help us focus on the problem at hand, and can prove that our design is correct.
  - Coupling and cohesion: if it isn't easy to write a test, this means that you have a problem of design, not of testing or coding. If your code is loosely coupled and highly cohesive, you can test it easily.
  - Trust: if you write code that works and you document it with automated tests, your teammates will trust you.
  - Rhythm: it is easy to get lost and wander for hours when you are coding. If you accustom yourself to the rhythm: test, code, refactor, test, code, refactor, it will not happen.

Software Coding and Releasing:

- **Ten-Minute Build**: System should be built and all of the tests should be finished in ten minutes, in order to execute it often and obtain feedback .
- **Continuous Integration**: Developers should be integrating changes every two hours in order to

ease integration headaches.

Corollary Practices

Requirement Analysis and Planning:

- **Real Customer Involvement**: people whose lives are affected by your system must became a part of the team and they can contribute to quarterly and weekly planning.
- **Incremental Deployment**: when replacing a legacy system, start to replace some functionalities right away and gradually replace all system. Avoid the approach of "All or nothing."
- **Negotiated Scope Contract**: contracts for software development would have fixed time, costs, and quality, but the precise scope of the system would have to be negotiated during the same realization. Eventually it is better to have a sequence of short contracts in order to reduce risks.
- **Pay-Per-Use**: customer usually pays for each release of the software. This creates a conflict between the supplier and the customer, who would want fewer releases each containing a lot of functionalities. Connecting money flow directly to software development provides accurate, timely information with which to drive improvement.

Team and Human Factors:

- **Team Continuity**: the development teams must remain the same in several projects. The relationship they share in a project are precious and they do not have to be dispersed.
- **Shrinking Teams**: as the team becomes more capable and productive, keep its load constant but gradually reduce its size, send free members to form more teams.

Design:

- **Root-Cause Analysis**:every time you find a defect, eliminate it and its causes. In this away, not just you will eliminate the defect, but also you will prevent making the same mistake again.

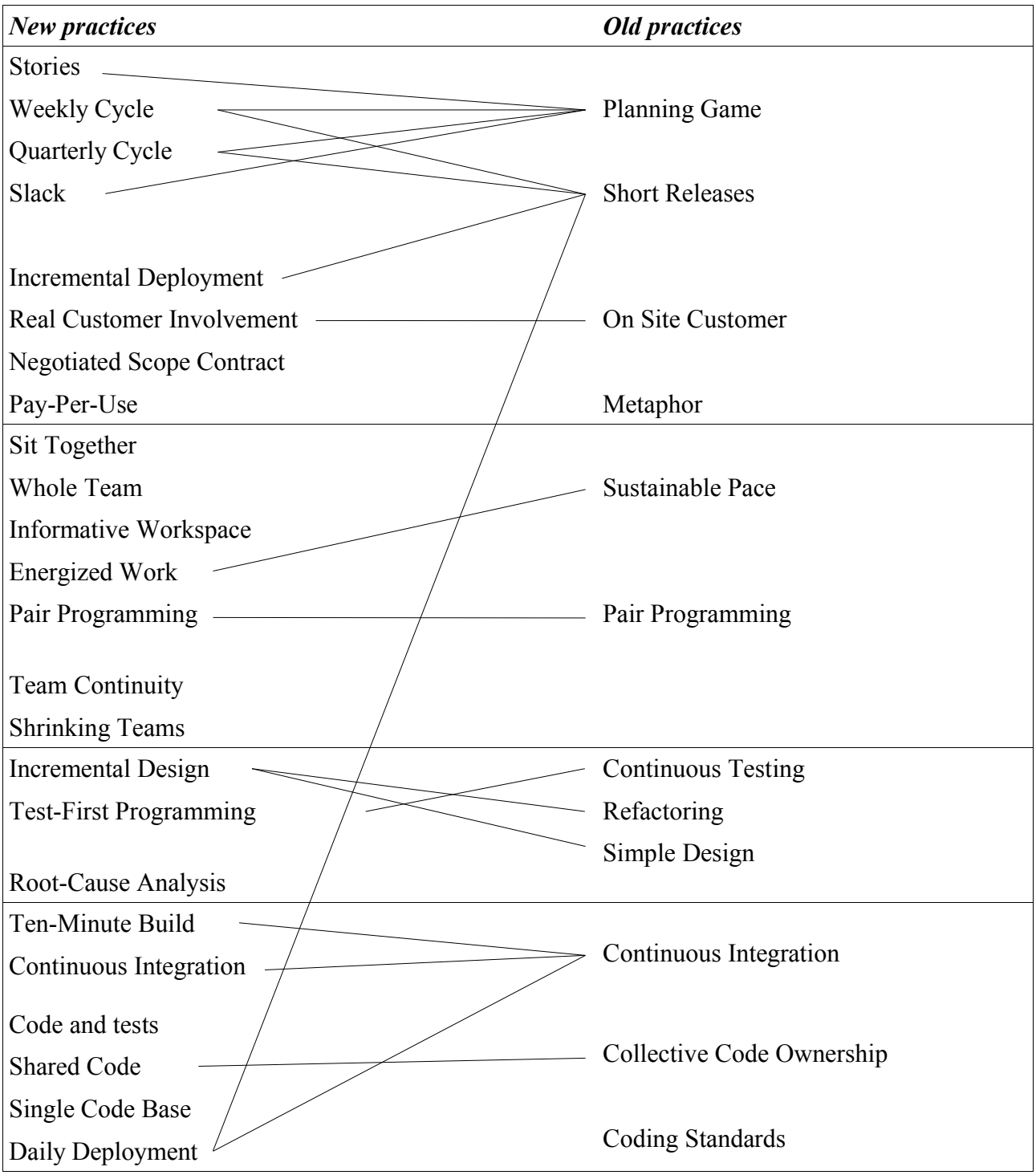
Software Coding and Releasing:

- **Code and Tests**: only code and tests are permanent artifacts and they have to be preserved. The other documents can be generated from code and tests.
- **Shared Code**: anyone in the development team must be able to change any part of system at any time. This practice was called "*collective code ownership*" in the original XP.
- **Single Code Base**: there is only one *official* version of system. You can develop a temporary branch, but it doesn't live longer than a few hours.
- **Daily Deployment**: every night you must put new software into production. It is risky and costly to have a gap between the version of software released into production and ones you have in your computer.

You must note that in the new XP we cannot find original practices of *coding standards*, that is considered obvious and not more essential thing, and *metaphor* which was the most complex to define and understand and the most difficult to implement among the twelve original practices.

After this shortly list of twenty-four practices, we can analyze the link between them and the twelve original practices. Indeed, only a few practices correspond with original ones. Some practices expand original ones and some are really new. For example, "planning game," which was the practice describing the XP process planning, is divided into four new practices ("stories", "weekly cycle", "quarterly cycle" and "slack".

The following table lists the new and old practices. They are divided into categories. The table shows the relationship between new practices and old ones. It shows easily the difference between old and new XP.



In conclusion, I have shown clearly, as Beck' s new book proposes, an agile way to produce software. The new XP is very different from the one in the original book. Surely, any original principle is negated and the second edition of the book is an evolution from the first one. It includes the lessons learned in these five years since the first book. However, many new things are introduced. We can find many new practices and principles we cannot find in the first edition. I don't know whether some supporters of original XP will accuse Beck of the heresy versus his same method or they will be faithful and consider to try and apply the new ideas. Anyway I think that Beck' s goal is not to give a solution “by the book,” to which you must adhere rigorously. He is opening a deep discussion about the way to develop software and suggest new ideas for each company to develop its own XP process with their own experience and context.