Extreme Programming
A Lightweight OO Development Process

**Introduction**

Extreme Programming (XP) is the name that Kent Beck (kentbeck@csi.com) has given to a lightweight development process that has been evolving over the years. Many experienced developers have contributed to the process as defined by Kent.

This article contains many excerpts from many of his posts to otug@rational.com (the object technology email discussion group). It was originally compiled by Yonat Sharon and Mark Collins-Cope based on information sent by Kent Beck to the otug@rational.com discussion group. This article was not structured by Kent himself. It has since been edited to reach a broader audience than those that shared the context of his original postings, and additional insights from Ken Auer (kauer@rolemodelsoft.com) of RoleModel Software, Inc. have been added.

Although we believe we have not changed the spirit of Kent's postings or misrepresented the XP process, this overview is based on the original compilation, and reflects our own experiences in addition to Kent's words.

This is a work in progress as of June 21, 1999. The vast majority of the credit to the compilation and format of the contents go to Yonat Sharon and Mark Collins-Cope. The credit for the vast majority of the content goes to Kent Beck.

**Motivation for Extreme Programming**

People don't enjoy, and don't actually use the feedback mechanisms that they read about—synchronized documentation, big testing processes administered by a separate group, extensive and fixed requirements. So XP attempts to employ feedback mechanisms that
- people enjoy, so they will be likely to adopt them,
- have short-term and long-term benefits, so people will tend to stick to them even under pressure,
- will be executable by programmers with ordinary skills, so the potential audience for XP is as large as possible and,
- has good synergistic effects, so we can pay the cost of the fewest possible loops

**The Planning Game**

**Collecting User Stories**

Before you code, you play the planning game. The requirements are in the form of User Stories, which you can think of as just enough of a use case to estimate from and set priorities from. Our experience of customers using stories is that they love them. They can clearly see the tradeoffs they have available, and they understand what choices they can and can't make.

Each story typically translates into one or more functional test cases, which you review with the customer at the end of the iteration that delivers the story [An iteration is 1-4 weeks worth of stories]. The test cases can be written in any of a number of forms that are easily readable (and if you're smart, easily writable) by the customer-directly reading spreadsheets, using a parser generator to create a special purpose language, writing an even simpler language that translates directly into test-related objects.

[...]

Our experience with the Planning Game is that it works wonderfully at conceptualization. You get 50-100 cards on the table and the customers can see the entire system at a glance. They can see it from many different perspectives just by moving the cards around. As soon as you have story estimates and a project speed, the customers can make tradeoffs about what to do early and what to do late and how various proposed releases relate to concrete dates. And the time and money required to get stories and estimates is miniscule compared to what will eventually be spent on the system.

The stories are written by the customers with feedback from the programmers, so they are automatically in "business language".
The strategy of estimation is:

- **Be concrete.** If you don't know anything about a story or task, go write enough code so you know something about the story or task.
- **If you can, compare a task or story to something that has gone before,** that is concrete, also. But don't commit to anything on speculation. [...] 
- **No imposed estimates.** Whoever is responsible for a story or task gets to estimate. If the customer doesn't like the estimate, they can change the story. The team is responsible for delivering stories, so the team does collective estimates (everybody estimates some stories, but they switch around pairs as they explore so everybody knows a little of everything about what is being estimated). Estimates for the tasks in the iteration plan are only done after folks have signed up for the tasks.
- **Feedback.** Always compare actuals and estimates. Otherwise you won't get any better. This is tricky, because you can't punish someone if they really miss an estimate. If they ask for help as soon as they know they are in trouble, and they show they are learning, as a coach you have to pat them on the back.
- **Re-estimation.** You periodically re-estimate all the stories left in the current release, which gives you quick feedback on your original estimates and gives Business better data on which to base their decisions.

**Scheduling**

There are two levels of scheduling in XP:

- **The commitment schedule** is the smallest, most valuable bundle of stories that makes business sense. These are chosen from the pile of all the stories the customer has written, after the stories have been estimated by the programmers and the team has measured their overall productivity. So, we might have stories for a word processor:
  - Basic word processing - 4
  - Paragraph styles - 2
  - Printing - 4
  - Spell checking - 2
  - Outliner - 3
  - Inline drawing - 6

  (the real stories would be accompanied by a couple of sentences). The estimates are assigned by the programmers, either through prototyping or by analogy with previous stories.

Before you begin production development, you might spend 10-20% of the expected time to first release coming up with the stories, estimates, and measurement of team speed. (While you prototype, you measure the ratio of your estimates to make each prototype to the calendar- that gives you the speed). So, let's say the team measured its ratio of ideal time to calendar time at 3, and there are 4 programmers on the team. That means that each week they can produce 4/3 ideal weeks per calendar week. With three week iterations, they can produce 4 units of stories per iteration.

If the customer has to have all the features above, you just hold your nose and do the math- 21 ideal weeks @ 4 ideal weeks/iteration = 5 1/4 iterations or 16 calendar weeks. "It can't be four months, we have to be done with engineering in two months."

Okay, we can do it that way, too. Two months, call it three iterations, gives the customer a budget of 12 ideal weeks. Which 12 weeks worth of stories do they want?

[...]

XP quickly puts the most valuable stories into production, then follows up with releases as frequent as deployment economics allow. So, you can give an answer to the question "How long will all of this take," but if the answer is more than a few months out, you know the requirements will change.

[...]

To estimate, the customers have to be confident that they have more than enough stories for the first release, and that they have covered the most valuable stories, and the programmers have to have concrete experience with the stories so they can estimate with confidence.

**Planning is continuous**

XP discards the notion of complete plans you can stick with. The best you can hope for is that everybody communicates everything they know at the moment, and when the situation changes, everybody reacts in the best possible way. Everybody believes in the original commitment schedule- the customers believe that it contains the most valuable stories, the programmers believe that working at their best they can make the estimates stick. But the plan is bound to change. Expect change, deal with change openly, embrace change.

For more information on the planning game:
Requirements

Dealing with changing requirements

Where we live the customers don’t know what they want, they specify mutually exclusive requirements, they change their minds as soon as they see the first system, they argue among themselves about what they mean and what is most important. Where we live technology is constantly changing so what is the best design for a system is never obvious a priori, and it is certain to change over time.

One solution to this situation is to use a detailed requirements process to create a document that doesn’t have these nasty properties, and to get the customer to sign the document so they can’t complain when the system comes out. Then produce a detailed design describing how the system will be built.

Another solution is to accept that the requirements will change weekly (in some places daily), and to build a process that can accommodate that rate of change and still be predictable, low risk, and fun to execute. The process also has to be able to rapidly evolve the design of the system in any direction it wants to go. You can’t be surprised by the direction of the design, because you didn’t expect any particular direction in the first place.

The latter solution works much better for us than the former.

[XP says] to thoroughly, completely discard the notion of complete up-front requirements gathering. Pete McBreen <mcbreenp@cadvision.com> made the insightful observation that the amount of requirements definition you need in order to estimate and set priorities is far, far less than what you need to code. XP requirements gathering is complete in the sense that you look at everything the customer knows the system will need to do, but each requirement is only examined deeply enough to make confident guesses about level of effort. Sometimes this goes right through to implementing it, but as the team’s skill at estimating grows, they can make excellent estimates on sketchy information.

The advantage of this approach is that it dramatically reduces the business risk. If you can reduce the interval where you are guessing about what you can make the system do and what will be valuable about it, you are exposed to fewer outside events invalidating the whole premise of the system.

So, what if I as a business person had to choose between two development styles?

- We will come back in 18 months with a complete, concise, consistent description of the software needed for the brokerage. Then we can tell you exactly what we think it will take to implement these requirements.
- We will implement the most pressing business needs in the first four months. During that time you will be able to change the direction of development radically every three weeks. Then we will split the team into two and tackle the next two most pressing problems, still with steering every three weeks.

The second style provides many more options, more places to add business value, and simultaneously reduces the risk that no software will get done at all.

Documenting Requirements

You could never trust the developers to correctly remember the requirements. That’s why you insist on writing them on little scraps of paper (index cards). At the beginning of an iteration, each of the stories for that iteration has to turn into something a programmer can implement from.

This is all that is really necessary, although most people express their fears by elaborating the cards into a database (Notes or Access), Excel, or some annoying project management program. (We use a custom Wiki [http://c2.com/cgi/wiki?WikiWikiWeb]. Using cards as the archival form of the requirements falls under the heading of "this is so simply you won’t believe it could possibly work". But it is so wonderful to make a review with upper management and just show them the cards. "Remember two months ago when you were here? Here is the stack we had done then. Here is the stack we had to do before the release. In the last two months here is what we have done."

The managers just can’t resist. They pick up the cards, leaf through them, maybe ask a few questions (sometimes disturbingly insightful questions), and nod sagely. And it’s no more misleading (and probably a lot less misleading) than showing them a Pert chart.
**Architecture (System Metaphors)**

The degree to which XP does big, overall design is in choosing an overall metaphor or set of metaphors for the operation of the system. For example, the C3 project works on the metaphor of manufacturing: time and money parts come into the system, are placed in bins, are read by stations, transformed, and placed in other bins. Another example is the LifeTech system, an insurance contract management system. It is interesting because it overlays complementary metaphors—double entry bookkeeping for recording events, versioned business objects, and an overall task/tool metaphor for traceability of changes to business objects.

The system is built around one or a small set of cooperating metaphors, from which class, method, variables, and basic responsibilities are derived. You can’t just go off inventing names on your own. The short term benefit is that everyone is confident that they understand the first things to be done. The long term benefit is that there is a force that tends to unify the design, and to make the system easier for new team members to understand. The metaphor keeps the team confident in Simple Design, because how the design should be extended next is usually clear.


Communicating the overall design of the system comes from:
- listening to a CRC overview of how the metaphor translates into objects
- pair programming new member with experienced member
- reading test cases
- reading code

Kent arrived at "system metaphor" as the necessary and sufficient amount of overall design after having seen too many systems with beautiful architectural diagrams but no real unifying concept. This led him to conclude that what people interpret as architecture, sub-system decomposition, and overall design were not sufficient to keep developers aligned. Choosing a metaphor can be done fairly quickly (a few weeks of active exploration should suffice), and admits to evolution as the programmers and the customers learn.

**Design**

**Simple Design (or Precisely Enough Design)**

The right design for the system at any moment is the design that
1) runs all the tests,
2) says everything worth saying once,
3) says everything only once,
4) within these constraints contains the fewest possible classes and methods.

You can’t leave the code until it is in this state. That is, you take away everything you can until you can’t take away anything more without violating one of the first three constraints. In the short term, simple design helps by making sure that the programmers grapple the most pressing problem. In the long run, simple design ensures that there is less to communicate, less to test, less to refactor.


**Countering the Hacking Argument**

During iteration planning, all the work for the iteration is broken down into tasks. Programmers sign up for tasks, then estimate them in ideal programming days. Tasks more than 3-4 days are further subdivided, because otherwise the estimates are too risky.

So, in implementing a task, the programmer just starts coding without aforethought. Well, not quite. First, the programmer finds a partner. They may have a particular partner in mind, because of specialized knowledge or the need for training, or they might just shout out "who has a couple of hours?"

Now they can jump right in without aforethought. Not quite. First they have to discuss the task with the customer. They might pull out CRC cards, or ask for functional test cases, or supporting documentation.

Okay, discussion is over, now the "no aforethought" can begin. But first, they have to write the first test case. Of course, in writing the first test case, they have to precisely explore both the expected behavior of the test case, and how it will be reflected in the code. Is it a new method? A new object? How does it fit with the other objects?
and messages in the system? Should the existing interfaces be refactored so the new interface fits in more smoothly, symmetrically, communicatively? If so, they refactor first. Then they write the test case. And run it, just in case.

**Refactoring**

Now it's hacking time. But first a little review of the existing implementation. The partners imagine the implementation of the new test case. If it is ugly or involves duplicate code, the try to imagine how to refactor the existing implementation so the new implementation fits in nicely. If they can imagine such a refactoring, they do it.

You can't just leave duplicate or uncommunicative code around. The short term value is that you program faster, and you feel better because your understanding and the system is seldom far out of sync. The long term value is that reusable components emerge from this process, further speeding development. Refactoring makes good the bet of Simple Design.

More information:  

**Coding**

**The Hacking Phase**

Okay, now it really is hacking time. They make the test case run. No more of this aforethought business. Hack hack hack. This phase usually takes 1-5 minutes. While they are slinging code around with wild abandon, they may imagine new test cases, or possible refactorings. If so, they note them on a to do list.

Now they reflect on what they did. If they discovered a simplification while they were "no aforeshothing", they refactor.

This process continues until all the test cases they can imagine all run. Then they integrate their changes with the rest of the system and run the global unit test suite. When it runs at 100% (usually the first time), they release their changes. One never thinks more, both before and after coding, than when they are in the flow of this process. So we're not worried about accusations that XP involves not thinking. It certainly isn't true from our perspective. And people who have actually tried it agree.

---

**Collective Code Ownership**

If you run across some code that could be improved, you have to stop and improve it, no matter what. The short term benefit is that your code is cleaner. The long term benefit is that the whole system gets better all the time, and everyone tends to be familiar with most of the system. Collective Code Ownership makes refactoring work better by exposing the team to more opportunities for big refactorings.

More information:  

**Coding Standards**

You can write code any way you want, just not on the team. Everybody chooses class names and variable names in the same style. They format code in exactly the same way. There isn't a universal short term benefit to coding standards that we can think of. Longer term, coding standards that are chosen for communication help new people learn the system, and as the standards become habit, they improve productivity. Pair programming works much more smoothly with coding standards, as do collective code ownership and refactoring.

More information:  

**Pair Programming**

This is the master feedback loop that ensures that all the other feedback loops stay in place. Any myopic manager can tell you with certainty that pair programming must be slower. Over the course of days and weeks, however, the effects of pairing dramatically reduce the overall project risk. And it is just plain fun to have someone to talk to. The pairs shift around a lot (two, three, four times a day), so any important information is soon known by everyone on the team.


**Integration & Testing**

**Unit testing**

You can't release until the unit tests are 100%. The short term value is that you program faster over one hour and greater time scales, the code is higher quality, and there is much less stress. Over the
longer term, the unit tests catch integration errors and regressions, and they communicate the intent of the design independent of implementation details. The unit tests enable refactoring, they drive the simple design, they make collective code ownership safe, and act as a conversation piece to enhance pair programming.


You can’t have certainty. Is there a reasonable amount of work that you can do that lets you act like you have certainty? For the code we write, the answer is yes. We write a unit test before writing or revising any method that is at all complicated. We run all of the unit tests all the time to be sure nothing is broken. Then we can act like we have certainty that our software functions to the best of our understanding, and that we have pushed our understanding as far as we can.

The cool thing about this strategy is it makes great sense short term and long term. Short term we program faster overall because we force ourselves to think about interfaces before thinking about implementation. We are also more confident, less stressed, and can more easily explain what we are doing to our partners.

Long term the tests dramatically reduce the chance that someone will harm the code. The tests communicate much of the information that would otherwise be recorded in documentation that would have to be separately updated (or not). The writing of the tests tends to simplify the design, since it’s easier to test a simple design than a complex one. And the presence of the tests tends to reduce over-engineering, since you only implement what you need for tests.

Oh, and just in case the programmers missed something in their unit tests, the customers are writing functional tests at the same time. When a defect slips through unit testing and is caught in functional testing (or production), the programmers learn how to write better unit tests. Over time, less and less slips through.

A real testing guru would sneer at this level of testing. That doesn’t bother us so much. What we know for certain is that we write better code faster and have more fun when we test. So we test.

Continuous Integration

Code additions and changes are integrated with the baseline after a few hours, a day at most. You can’t just leap from task to task. When a task is done, you wait your turn integrating, then you load your changes on top of the current baseline (resolving any conflicts), and running the tests. If you have broken any tests, you must fix them before releasing. If you can’t fix them, you discard your code and start over. In the short term, when the code base is small, the system stays in very tight sync. In the long term, you never encounter integration problems, because you have dealt with them daily, even hourly, over the life of the project. Continuous integration makes collective code ownership and refactoring possible without overwhelming numbers of conflicting changes, and the end of an integration makes a natural point to switch partners.


Functional testing [black box testing]

You can’t continue development until the functional test scores are acceptable to the customer. The short term value is that the programmers know when they are done with the functionality, and the customers are confident that the system works. Longer term, the functional tests prevent regressions and communicate important historical information from a customer perspective. The functional tests back up the unit tests to ensure quality and improve the unit testing process.


On Customer Site

You can’t just take your understanding of requirements and design and implement with them for a month. Instead, you are in hourly contact with a customer who can resolve ambiguities, set priorities, set scope, and provide test scenarios. In the short term, you learn much more about the system by being in such close contact. In the longer term, the customer can steer the team, subtly and radically, with much more confidence and understanding because such a close working relationship is built. The Planning Game requires an on-site customer to complete requirements as they are about to be built, and functional testing works...
much better if the author of the tests is available for frequent consultation.

**Culture and Environment**

**Open Workspace**

The best XP workspace is a large bull-pen with small individual cubbies around the walls, and tables with fast machines in the center, set up for pair programming. No one can go off and hack for hours in this environment. That solo flow that many developers get addicted to simply isn't possible. However, in the short term it is much easier to get help if you need help just by calling across the room. In the long term the team benefits from the intense communication. The open workspace helps pair programming work, and the communication aids all the practices.

**Forty Hour Week**

Go home at 5. Have a nice weekend. Once or twice a year, you can work serious overtime for a week, but the need for a second week of overtime in a row is a clear signal that something else is wrong with the project. Over the very short term, this will definitely feel slower. But over a few weeks, and certainly over the course of months, the team's productivity will be higher and the risk will be lower. Over the long term, XP's reliance on oral history demands a certain level of stability in the staff, and 40 hour weeks go a long way towards keeping people happy and balanced. Rested programmers are more likely to find valuable refactorings, to think of that one more test that breaks the system, to be able to handle the intense inter-personal interaction on the team.

**Less Stress!**

These are all the practices we regularly teach. Contexts with less stress will require fewer of the loops to maintain control.

**Is XP Always Appropriate**

There are certainly contexts that will require additional practices- ISO certification, FDA auditing requirements, complicated concurrency problems requiring careful review, complicated logic that no one can figure out how to segment. In these exceptional cases, you can figure out additional practices, or just forget about XP. But, when you throw out XP, you may be throwing out your shot at the benefits XP offers.

Looking at the above practices, it is still hard to imagine how XP can fly, not because it so out of control, but because it is so completely controlled. I think the reason it works is because none of the practices require a PhD to execute, and they are all fun, or at least stress relieving, and they all contribute, directly and obviously, to the system.

The result of this process is clean, tight, communicative code that is flexible where it needs to be, but without an ounce of fat. And it has a test suite that allows dramatic changes to be put in with confidence, years after the system was originally built.

**Summary**

XP values analysis and design so much that on a 12 month project, the team will spend no less than 12 months on analysis and design (and testing and integration and coding). It's not "are these kinds of questions addressed", but rather when and how. XP does it throughout and in the presence of the production code.

The percentages that have been promoted for successful development- 40-65% A&D, 5% coding, 30-55% test and support - XP seems to correspond with them 100%. It's just that we can measure those proportions over pretty much any given day for every programmer on the team.

[Ed note: Ralph Johnson <johnson@cs.uiuc.edu> said that in XP the software life-cycle is: Analysis, Test, Code, Design.]

**More Information**

