Development Methodology

Introduction

There have been numerous requests regarding the Atomicat development methodology. In a nutshell, we use the X-Treme Methodology (XP) to develop solutions and this brief will provide an overview thereof.

What Is The X-Treme Programming Methodology (XP)?

Extreme Programming is also know as X-Treme Programming or XP. In this paper we will refer to it as “Extreme Programming”.

In case you’re wondering, “XP” has absolutely nothing to do with a Microsoft operating system, the acronym was in use for many years before to refer to the Extreme Programming methodology. It’s a fluke that Microsoft have a similarly named product series but if they used the methodology of the same name it can only bode well for them.

Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.

In Extreme Programming, every contributor to the project is an integral part of the "Whole Team". The team forms around a business representative called "the Customer", who sits with the team and works with them daily.

The core practises, in brief, are as follows;

- **Whole Team** – Extreme Programming teams use a simple form of planning and tracking to decide what should be done next and to predict when the project will be done. Focused on business value, the team produces the software in a series of small fully-integrated releases that pass all the tests the Customer has defined.

- **Planning Game, Small Releases, Customer Tests** – Extreme Programmers work together in pairs and as a group, with simple design and obsessively tested code, improving the design continually to keep it always just right for the current needs.

- **Simple Design, Pair Programming, Test-Driven** – Development, Design Improvement The Extreme Programming team keeps the system integrated and running all the time. The programmers write all production code in pairs, and all work together all the time. They code in a consistent style so that everyone can understand and improve all the code as needed.

- **Continuous Integration, Collective Code Ownership, Coding Standard** – The Extreme Programming team shares a common and simple picture of what the system looks like. Everyone works at a pace that can be sustained indefinitely.

- **Metaphor, Sustainable Pace** – A single understanding and vision of what is being developed tempered by realistic goals

We will now review the core practises in greater detail.

Planet Limited (Company Nr 5952219)
Level 1, 120 Eleventh Avenue, Tauranga, New Zealand
www.*planetlimited*.io

Whole Team

All the contributors to an XP project sit together, members of one team. This team must include a business representative - the "Customer" - who provides the requirements, sets the priorities, and steers the project. It's best if the Customer or one of her aides is a real end user who knows the domain and what is needed.

The team will of course have programmers. The team may include testers, who help the Customer define the customer acceptance tests. Analysts may serve as helpers to the Customer, helping to define the requirements. There is commonly a coach, who helps the team keep on track, and facilitates the process.

There may be a manager, providing resources, handling external communication, coordinating activities. None of these roles is necessarily the exclusive property of just one individual: Everyone on an XP team contributes in any way that they can. The best teams have no specialists, only general contributors with special skills.

Planning Game

XP planning addresses two key questions in software development: predicting what will be accomplished by the due date, and determining what to do next.

The emphasis is on steering the project - which is quite straightforward - rather than on exact prediction of what will be needed and how long it will take - which is quite difficult. There are two key planning steps in XP, addressing these two questions:

- **Release Planning** is a practice where the Customer presents the desired features to the programmers, and the programmers estimate their difficulty. With the costs estimates in hand, and with knowledge of the importance of the features, the Customer lays out a plan for the project. Initial release plans are necessarily imprecise: neither the priorities nor the estimates are truly solid, and until the team begins to work, we won't know just how fast they will go. Even the first release plan is accurate enough for decision making, however, and XP teams revise the release plan regularly.

- **Iteration Planning** is the practice whereby the team is given direction every couple of weeks. XP teams build software in two-week "iterations", delivering running useful software at the end of each iteration. During Iteration Planning, the Customer presents the features desired for the next two weeks. The programmers break them down into tasks, and estimate their cost (at a finer level of detail than in Release Planning). Based on the amount of work accomplished in the previous iteration, the team signs up for what will be undertaken in the current iteration.

These planning steps are very simple, yet they provide very good information and excellent steering control in the hands of the Customer. Every couple of weeks, the amount of progress is entirely visible. There is no "ninety percent done" in XP: a feature story was completed, or it was not.

This focus on visibility results in a nice little paradox: on the one hand, with so much visibility, the Customer is in a position to cancel the project if progress is not sufficient. On the other hand, progress is so visible, and the ability to decide what will be done next is so complete, that XP projects tend to deliver more of what is needed, with less pressure and stress.

Planet Limited (Company Nr 5952219)
Level 1, 120 Eleventh Avenue, Tauranga, New Zealand
www.planetlimited.io

Customer Tests

As part of presenting each desired feature, the XP Customer defines one or more automated acceptance tests to show that the feature is working. The team builds these tests and uses them to prove to themselves, and to the customer, that the feature is implemented correctly. Automation is important because in the press of time, manual tests are skipped. That's like turning off your lights when the night gets darkest.

The best XP teams treat their customer tests the same way they do programmer tests: once the test runs, the team keeps it running correctly thereafter. This means that the system only improves, always notching forward, never backsliding.

Small Releases

XP teams practice small releases in two important ways:

First, the team releases running, tested software, delivering business value chosen by the Customer, every iteration. The Customer can use this software for any purpose, whether evaluation or even release to end users (highly recommended). The most important aspect is that the software is visible, and given to the customer, at the end of every iteration. This keeps everything open and tangible.

Second, XP teams release to their end users frequently as well. XP Web projects release as often as daily, in house projects monthly or more frequently. Even shrink-wrapped products are shipped as often as quarterly.

It may seem impossible to create good versions this often, but XP teams all over are doing it all the time. See the Continuous Integration section below for more on this, and note that these frequent releases are kept reliable by XP's obsession with testing (described here in the sections about Customer Tests and Test-Driven Development).

Simple Design

XP teams build software to a simple design. They start simple, and through programmer testing and design improvement, they keep it that way. An XP team keeps the design exactly suited for the current functionality of the system. There is no wasted motion, and the software is always ready for what's next.

Design in XP is not a one-time thing, or an up-front thing, it is an all-the-time thing. There are design steps in release planning and iteration planning, plus teams engage in quick design sessions and design revisions through refactoring, through the course of the entire project. In an incremental, iterative process like Extreme Programming, good design is essential. That's why there is so much focus on design throughout the course of the entire development.

Pair Programming

All production software in XP is built by two programmers, sitting side by side, at the same machine. This practice ensures that all production code is reviewed by at least one other programmer, and results in better design, better testing, and better code.

It may seem inefficient to have two programmers doing "one programmer's job", but the reverse is true. Research into pair programming (www.pairprogramming.com) shows that pairing produces better code in about the same time as programmers working singly. That's right: two heads really are better than one!

Some programmers object to pair programming without ever trying it. It does take some practice to do well, and you need to do it well for a few weeks to see the results. Ninety percent of programmers who learn pair programming prefer it, so we highly recommend it to all teams.

Pairing, in addition to providing better code and tests, also serves to communicate knowledge throughout the team. As pairs switch, everyone gets the benefits of everyone's specialized knowledge. Programmers learn, their skills improve, they become move valuable to the team and to the company. Pairing, even on its own outside of XP, is a big win for everyone.

Test-Driven Development

Extreme Programming is obsessed with feedback, and in software development, good feedback requires good testing. Top XP teams practice "test-driven development", working in very short cycles of adding a test, then making it work. Almost effortlessly, teams produce code with nearly 100 percent test coverage.

It isn't enough to write tests: you have to run them. Here, too, Extreme Programming is extreme. These "programmer tests", or "unit tests" are all collected together, and every time any programmer releases any code to the repository (and pairs typically release twice a day or more), every single one of the programmer tests must run correctly. One hundred percent, all the time! This means that programmers get immediate feedback on how they're doing. Additionally, these tests provide invaluable support as the software design is improved.

Design Improvement

Extreme Programming focuses on delivering business value in every iteration. To accomplish this over the course of the whole project, the software must be well-designed. The alternative would be to slow down and ultimately get stuck. So XP uses a process of continuous design improvement called Refactoring, from the title of Martin Fowler's book, "Refactoring: Improving the Design of Existing Code".

The refactoring process focuses on removal of duplication (a sure sign of poor design), and on increasing the "cohesion" of the code, while lowering the "coupling". High cohesion and low coupling have been recognized as the hallmarks of well-designed code for at least thirty years.

The result is that XP teams start with a good, simple design, and always have a good, simple design for the software. This lets them sustain their development speed, and in fact generally increase speed as the project goes forward.

Refactoring is, of course, strongly supported by comprehensive testing to be sure that as the design evolves, nothing is broken. Thus the customer tests and programmer tests are a critical enabling factor. The XP practices support each other, they are stronger together than separately.

Continuous Integration

Extreme Programming teams keep the system fully integrated at all times. We say that daily builds are for wimps: XP teams build multiple times per day. (One XP team of forty people builds at least eight or ten times per day!)

The benefit of this practice can be seen by thinking back on projects you may have heard about (or even been a part of) where the build process was weekly or less frequently, and usually led to "integration hell", where everything broke and no one knew why.

Infrequent integration leads to serious problems on a software project.

First of all, although integration is critical to shipping good working code, the team is not practiced at it, and often it is delegated to people who are not familiar with the whole system.

Second, infrequently integrated code is often - I would say usually - buggy code. Problems creep in at integration time that are not detected by any of the testing that takes place on an un-integrated system.

Third, weak integration process leads to long code freezes. Code freezes mean that you have long time periods when the programmers could be working on important shippable features, but that those features must be held back. This weakens your position in the market, or with your end users.

Collective Code Ownership

On an Extreme Programming project, any pair of programmers can improve any code at any time. This means that all code gets the benefit of many people's attention, which increases code quality and reduces defects.

There is another important benefit as well: when code is owned by individuals, required features are often put in the wrong place, as one programmer discovers that he needs a feature somewhere in code that he does not own. The owner is too busy to do it, so the programmer puts the feature in his own code, where it does not belong. This leads to ugly, hard-to-maintain code, full of duplication and with low (bad) cohesion.

Collective ownership could be a problem if people worked blindly on code they did not understand. XP avoids these problems through two key techniques: the programmer tests catch mistakes, and pair programming means that the best way to work on unfamiliar code is to pair with the expert. In addition to ensuring good modifications when needed, this practice spreads knowledge throughout the team.

Coding Standard

XP teams follow a common coding standard, so that all the code in the system looks as if it was written by a single - very competent - individual. The specifics of the standard are not important: what is important is that all the code looks familiar, in support of collective ownership.

Metaphor

Extreme Programming teams develop a common vision of how the program works, which we call the "metaphor". At its best, the metaphor is a simple evocative description of how the program works, such as "this program works like a hive of bees, going out for pollen and bringing it back to the hive" as a description for an agent-based information retrieval system.

Sometimes a sufficiently poetic metaphor does not arise. In any case, with or without vivid imagery, XP teams use a common system of names to be sure that everyone understands how the system works and where to look to find the functionality you're looking for, or to find the right place to put the functionality you're about to add.

Sustainable Pace

Extreme Programming teams are in it for the long term. They work hard, and at a pace that can be sustained indefinitely. This means that they work overtime when it is effective, and that they normally work in such a way as to maximize productivity week in and week out. It's pretty well understood these days that death march projects are neither productive nor produce quality software. XP teams are in it to win, not to die.

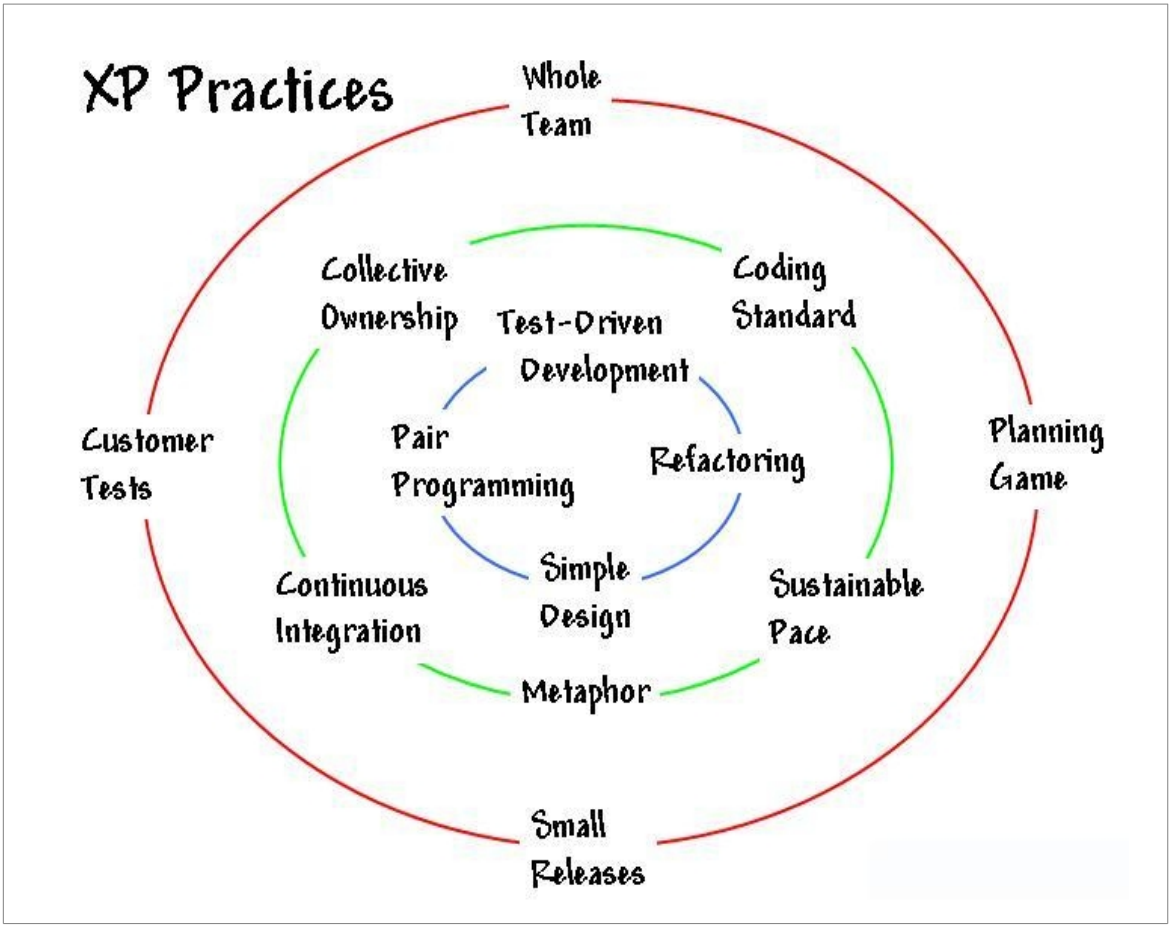
Last Words

Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.

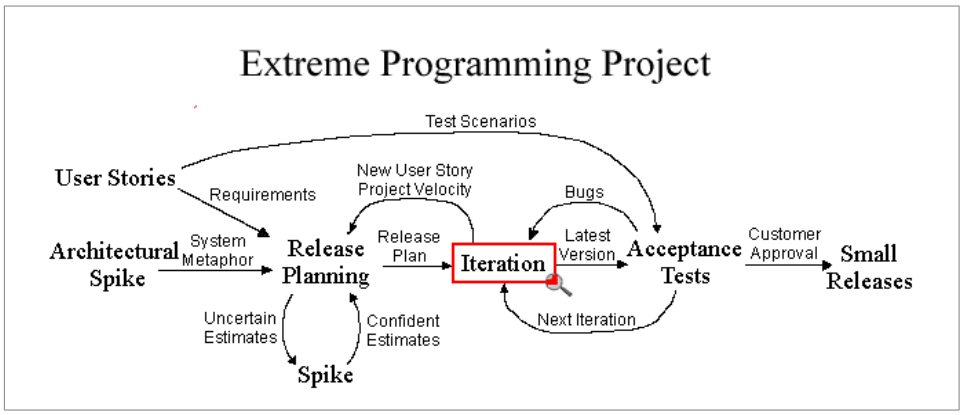
Graphic Overview Of Extreme Programming

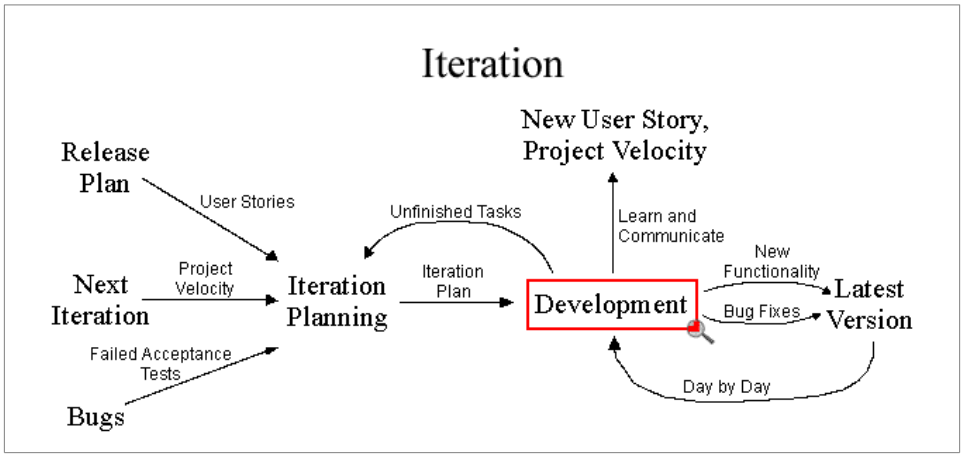
The following illustration shows the practices and the main "cycles" of XP.

Planet Limited (Company Nr 5952219)
Level 1, 120 Eleventh Avenue, Tauranga, New Zealand
www.planetlimited.io

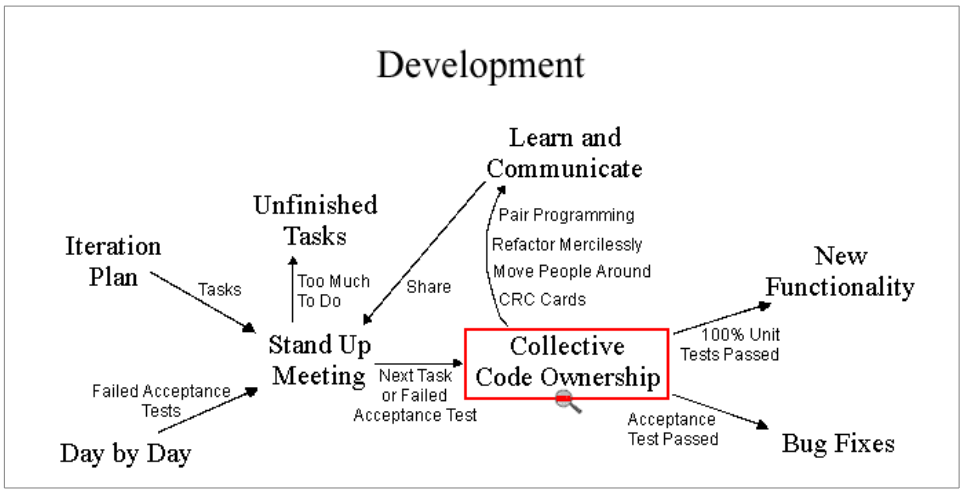


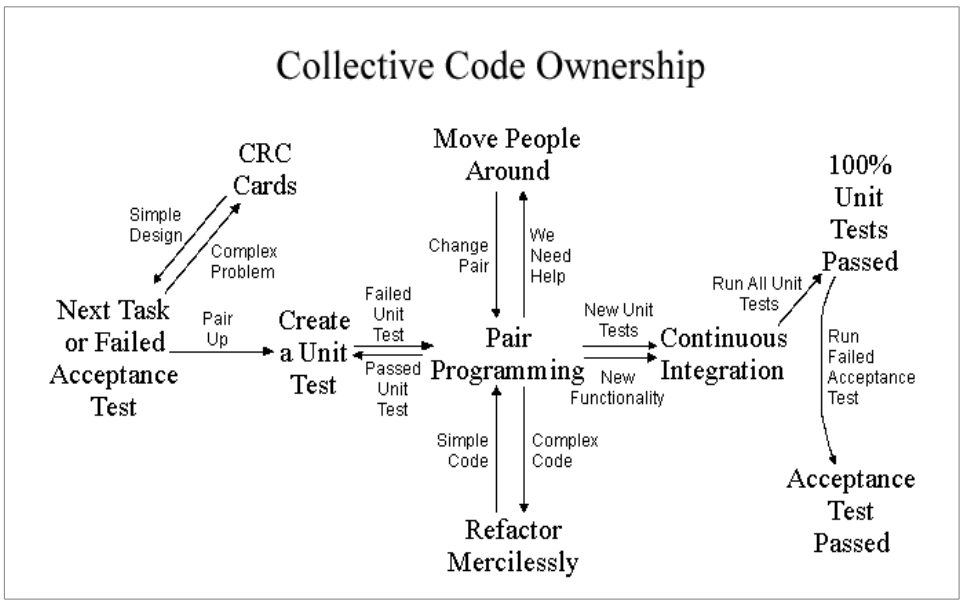
Graphical Overview Of Extreme Programming Models

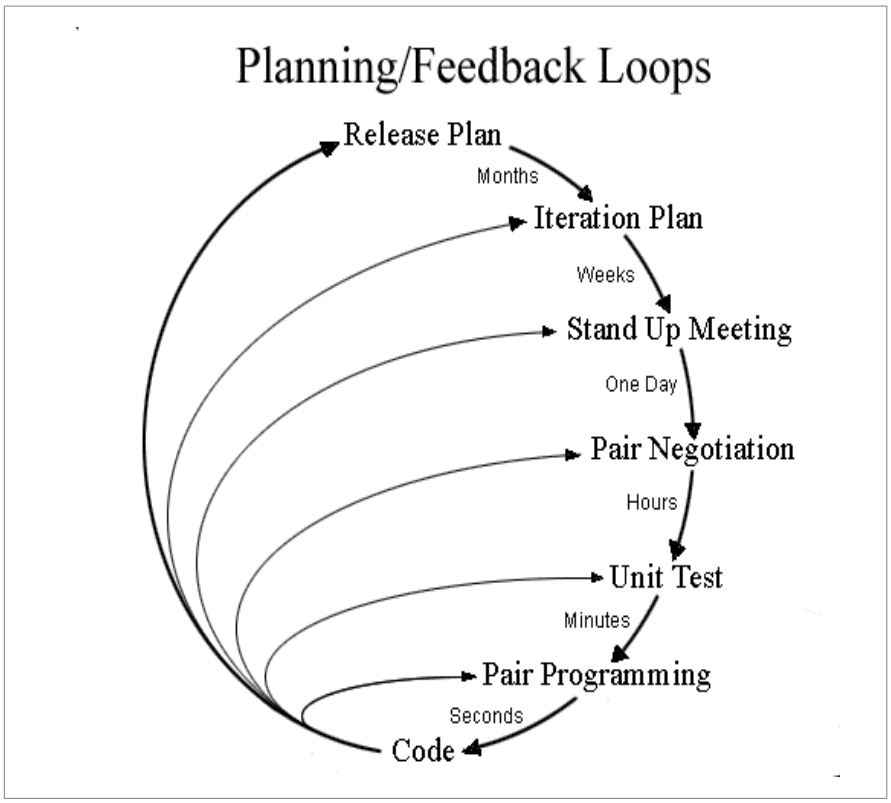




Planet Limited (Company Nr 5952219)
Level 1, 120 Eleventh Avenue, Tauranga, New Zealand
www.planetlimited.io







Planet Limited (Company Nr 5952219)
Level 1, 120 Eleventh Avenue, Tauranga, New Zealand
www.planetlimited.io

Some Questions & Answers About Extreme Programming

(The section by Ron Jeffries, editor XP Magazine)

Absence Of Errors

From an ongoing discussion on comp.software-eng:

Testing can only show the presence of errors, not their absence.

You XP guys advocate testing, not inspections or program proofs. Everyone knows that testing can only show that errors exist, not that they don't.

This famous saying, originally made up by a programmer who hated to write tests[1], is one of those logic games we all like to play when we're in school. It's formally correct, and completely misleading. The fact is that NOTHING, not inspection, not formal proof, not testing, can give 100% certainty of no errors. Yet all these techniques, at some cost, can in fact reduce the errors to whatever level you wish.

We teach our customers that they don't have to test anything unless they want it to work. What happens in the real world is that a large network of customer-specified functional tests in fact generate confidence. Not 100% confidence, but very great confidence. When we run all our tests, the programmers and the customers are very confident that we haven't screwed up anything important. Why? Because we have tested the hell out of everything important!

One final thought experiment. You're going to have to work with one of two word processors for the next six months. All I'll tell you is that the one in box A has been extensively tested and all the errors found were removed. The one in box B has not been tested. Which one do you choose, and why?

[1] A customer points out that E. W. Dijkstra actually first observed that testing can only show the presence of errors, not their absence. We knew that, but thought our fake attribution was more amusing. No disrespect was intended either to Professor Dijkstra or to the humour-challenged.

Tradeoffs

Worst things first vs User choice of stories

We have Worst Things First, which tells us what to worry about. On the other hand, our division of responsibilities tells us that the customer decides which stories to work on. Does WTF only apply within the particular story we have in hand now, or do we somehow exercise programmer forethought in deciding what we need to worry about?

Remember that when we do the Commitment Schedule, at that time we assign *two* priorities to stories: business value, and risk. Then we move stories toward the front which are high on either one, favouring risk reduction as much as possible.
Because one of the Extreme Values is Communication, it's quite important that the programmers educate the customers on the risks that they foresee. Much of this education happens during the commitment schedule, but effective teams do it all the time.

Perhaps a couple of developers sit down with customers and talk about risky areas: "You know, some of the approaches we're taking could slow down worse than one-for-one. When we get to 100 users, parts of the system could be 1,000 or 10,000 times slower, not just 100 times. You might want to be sure to write a story about performance and put it in the hopper." There are two key things to remember: first, XP programmers don't care at all if a story they've never heard of shows up in the next iteration. So if a story they *suggested* shows up, like "System must support 100 users with response time under one second", it's no problem at all.

Second, remember that the XP response to risk is to *reduce* it, not to solve the entire problem. So favor doing an experiment, or a *spike solution*, to find out how to deal with the performance issue. Once you know how you'll solve it, you can wait to put it into the system until the customers give it business value.

XP vs The real world

At some level, XP principles don't apply in the non-programming world. We can't build a bridge and have a user story when we're 75% done which says that we'd really like it to be about 12 feet to the left over here. XP seems to be built on the malleability of software, and on practices that help maintain that malleability. Do you have any feeling for where the boundary lines in these fall. One obvious grey area is the non-software parts of software projects. e.g. we need to train N-thousand users on this system, providing inertia once we start training them.

You're on the right track. There are no magic answers in this area. Certainly you try to defer decisions that will lock you in. The rule *Model First*, which says to defer working on the GUI as long as possible, is an example. As soon as you start working on the GUI, you're in an area where you can't change things without confusing your users.

The C3 team put off GUIs for a long time. But now even they find that when they make changes to the GUI that wouldn't bother a technical person, the users often get confused. We have to inform them, make changes really obvious, and so on. Even so, mostly the users come and ask.

C3 has the advantage that all the users are right there. With a delivered product, changing things that humans know about is much harder. There are lots of tricks out there: compatibility modes, configurable human interfaces, and so on. None of them are very satisfying. All I can offer is to try not to lock in, make the system as self-documenting as possible, and keep the locked parts of the system isolated from parts that may need to change.

Incrementalism vs Rewrite

XP seems built on incremental modification of software, fixing things that exist by gradual transformation, even if those things are really bad. Is there a point at which you say forget it, start again from scratch. Presumably there is, since I think that's what happened with the original C3 system. How does one tell?

XP is about ways to keep a system from needing a rewrite. By *refactoring mercilessly*, you can keep the code clean and malleable. When you walk into an existing system that wasn't built that way, or a system that somehow got out of hand, the temptation to start over is strong.

You're right that C3 started over. We have even reported that we wish we had trashed the few

things from the old system that we did keep! But it's a rare manager who will let this happen, since tons of money have probably already been invested in the old system.

If you do get the chance to start over, using XP is very important. Remember the famous "second system syndrome", where the developers throw in all the stuff they didn't get to use in the first system, resulting in a Fat Albert program that never gets done, and would consume all the computers in the universe if it ever did run. Now, more than ever, you need to *do the simplest thing that could possibly work*.

Much as I hate to say it, as a guy who hates working with horrible code, often your best chance is to clean it up as you go. It's possible to address one area at a time, write tests for it, refactor it, and move on. But the pain is sometimes just too much to bear.

Remember the rule, *Doctor, it hurts when I do this*. If the most mature and strongest developers on the team think that the program is doomed, it very likely is. Even if management is pushing to maintain the existing program back to health, measure your progress and feed it back to them. If it's good enough, fine, keep going. If it's not, help them to see that fact.

And don't forget *spike solution*! Sometimes you just have to take a week off and rewrite the system, or build a prototype. I've used code written in my free time to sell new ideas, and I'm sure you have too.

Final Comments

XP is a process allowing development to proceed smoothly, if it's applied throughout the project. If you're playing catchup, it'll be harder to do.

And never underestimate the value of your past experience. A team of experienced developers doing XP will blow away an inexperienced team. Let your educated intuition work for you. If you think the program should be destroyed, quite possibly it should. If you think an area is risky, it almost certainly is. And if you think some part of the system will be hard to evolve, you're probably right.

Follow your intuition and apply the XP rules as you develop. No one can do any better!

Quality Assurance

How is Software Quality Assurance and Software Configuration Management integrated into Extreme Programming?

XP defines two levels of testing. The first is *unit testing*, which must be performed by the programmers as they work. Each class implemented must have programmer-developed unit tests, for everything that "could possibly break". These tests are to be written during coding of the class, preferably right before implementing a given feature. Tests are run as frequently as possible during development, and all unit tests in the entire system must be running at 100% before any developer releases his code. (By release, we mean transferring from his own code space to the code integration area. This is handled differently, of course, depending on the code management tools in place.)

The second level of testing is called *functional testing*. Each feature of the system (which is defined by something we call a User Story, rather like a Use Case) must have one or more functional tests that test it. The functional tests are the responsibility of what we call the

Planet Limited (Company Nr 5952219)
Level 1, 120 Eleventh Avenue, Tauranga, New Zealand
www.planetlimited.io

"customer", the body responsible for defining the requirements.

The implementation and running of functional tests can be done by the Software QA group, and in fact this is an ideal way to do it.

Within XP, are there any specification baselines, test baselines, QA Acceptance testing, and CM Release Management/Change Control?

Let me take a shot at how XP works in these areas.

XP is an inherently incremental process, with software being released to "production" as frequently as possible. This generally means that programmers release their work to the common development pool approximately daily, and that means that if a full system were built on that day, their code would be included in that build. The time period between full system builds varies depending on the environment: since you have chosen a particularly difficult integration language (C++), I could imagine that you would build less frequently. We would recommend, however, that the full system be integrated as often as possible, at least daily. (This may seem aggressive to you. We'd have to talk about what is possible in your environment.)

Since XP is incremental, developers are working in short time increments we call *iterations*: we recommend about three weeks. Features (*user stories*) are broken down to the point of detail that allows a developer and his partner to implement the stories they're working on in that time period. We like the functional tests for that iteration to be complete and available no more than half-way through the iteration. (This usually means that QA is writing tests for the next iteration while this one is going on.)

All through the iteration, programmers can use QA's functional tests to determine whether they have met the requirements. (They are also using their own unit tests to determine whether their individual classes are doing what they should. This is usually at a much finer level of detail.)

Baselines work this way: when the code for a story is released, all the functional tests for it should be in place, and will ideally be working. Inevitably some will not, especially with teams just beginning with XP. One of the quality measures in the process is the daily graph of performance on functional tests. The general shape of this graph, over the course of the full system release period, is that of two s-curves: the upper curve is the total number of tests written, the lower curve is the number running at 100%. A healthy project of course shows these curves coming together at 100% by the end of the schedule.

The code management software needs of course to reflect the requirements scheduled for release. This is determined by the "customers", as part of the planning components we call the *commitment schedule* (overall plan for a major release) and the *iteration plan* (plan for a (three week) iteration). The baseline of what is in the system tracks what is actually requested by the customers.

Development doesn't care whether this is new functionality or a change to old. They don't care whether a given user story addresses something that was planned for or not. XP is completely flexible with regard to change management: development merely estimates how long any desired feature will take, and works on it when "customer" schedules it into an iteration. (Dependencies of course exist, but we find that far fewer exist than most developers believe. Drilling into that subject is beyond the scope of this appendix.)

Planet Limited (Company Nr 5952219)
Level 1, 120 Eleventh Avenue, Tauranga, New Zealand
www*.planetlimited.*io

When do the all the customer sign-offs occur?

Customer sign-off is continuous. Each iteration has its functional tests. Everyone is fully up to date on which tests are working and which are not. If tests scores are trailing implementation by too much, the customer will inevitably schedule more work against older features that are incorrect (or whose requirements have changed). When test scores are tracking implementation, the customer knows it and is comfortable requesting new functionality.

Because the test scores are public and visible, everyone has the same level of understanding of where quality is. Generally scores are showing a good curve toward release, and everyone gets increasing comfort as the release date shows up. And, of course, if tests are not tracking, everyone knows that and the priority of getting things right naturally increases.

The overall idea of this part of the process is to provide the most rapid feedback possible to everyone, customers and developers alike. That's why we like all the functional test run every night. Next morning, if anything has been broken the day before, everyone knows it and can deal with it effectively (since it was only yesterday's work that could be the problem). The faster the feedback, the faster development of quality software can proceed.

What are the Quality Assurance and Software Configuration Management roles and responsibilities with Extreme Programming?

We prefer for there to be a separate organization for functional testing (probably exactly like your QA function, with testing results made public very quickly). XP, however, only says that there must be functional tests: it does not specify organizationally how they must be done. Experience is that testing is best done by a separate function - but one that is very tightly integrated with development rather than at the end of a long pipeline.

Configuration management is also up to the team. It is usually necessary to have one or more individuals responsible for CM. We have no special rules or practices addressing how a group would manage the requirement to build multiple systems from one code base. Our main approach would be: for each release configuration, there must be corresponding functional tests, and these must be run before that configuration is released to the (real) customer. We would think that development would proceed by running kind of a "union" of all the functional tests of all the configurations.

We'd probably have to talk more specifically about how your specific organization needs to build configurations to say much more about that.

Do you use IEEE, SEI, ISO9000 standards as references to acquire the fundamentals of defining accurate requirements for customers and software engineering users? How can a person write storyboards without having the basics of pinpointing and developing sound requirements?

We would agree that those who play the customer role have to know what they want. We do not, however, recommend any particularly formal requirements writing or recording mechanism. Instead, what we are working toward (XP is for small teams, after all) is to have a clear understanding in the heads of customers, developers, and testers as to what is wanted.

Rather than have, say, an "analyst" sit down with the customer and laboriously translate his mumblings into something representing what is wanted, and then having a "designer" take the analysis and build a design, and so on, small teams function best if the customers and designer/developers talk to one another until they develop a common vocabulary of what is

Planet Limited (Company Nr 5952219)
Level 1, 120 Eleventh Avenue, Tauranga, New Zealand
www.planetlimited.io

needed and how it will be done. In XP, we would like to have a common level of understanding in all heads, each focused on its own particular interests:

Customers: what's needed, what's the business value, when do we need it?
Developers: what's needed, how can I build this, how can I test my code, how long will it take?
Testers: what's needed by the customers, how can I test whether developers have done it?

As you can see, the testers' functional tests are what close the loop, assuring everyone that what was asked for was what we got. The best way to do XP is with a separate functional testing organization that is closely integrated into the process. It would be delightful to have that organization run by an experienced QA manager trained in XP.

Is Extreme Programming not for Software Quality Engineering and Software Configuration Management practitioners?

XP is a development discipline that is for customers (in their role as specifiers and their role as investors and their role as testers and acceptors) and for developers. As such, the Quality Engineering and Configuration Management roles are critical to the effort. They have to be assigned and played in a way that is consistent with the mission of the group, the level of criticality of quality, and so on. We'd need to talk in detail about your situation to see just where the XP terminology connects with yours, but your QA functions need to be done in any effective software effort, whether in a separate organization or not. So XP certainly is for software quality engineering and software configuration management, as part of a healthy overall process.