The Rules and Practices of Extreme Programming

Planning

User stories are written.
Release planning creates the schedule.
Make frequent small releases.
The Project Velocity is measured.
The project is divided into iterations.
Iteration planning starts each iteration.
Move people around.
A stand-up meeting starts each day.
Fix XP when it breaks.

Coding

The customer is always available.
Code must be written to agreed standards.
Code the unit test first.
All production code is pair programmed.
Only one pair integrates code at a time.
Integrate often.
Use collective code ownership.
Leave optimization till last.
No overtime.

Designing

Simplicity.
Choose a system metaphor.
Use CRC cards for design sessions.
Create spike solutions to reduce risk.
No functionality is added early.
Refactor whenever and wherever possible.

Testing

All code must have unit tests.
All code must pass all unit tests before it can be released.
When a bug is found tests are created.
Acceptance tests are run often and the score is published.

User Stories

User stories serve the same purpose as use cases but are not the same. They are used to create time estimates for the release planning meeting. They are also used instead of a large requirements document. User Stories are written by the customers as things that the system needs to do for them. They are similar to usage scenarios, except that they are not limited to describing a user interface. They are in the format of about three sentences of text written by the customer in the customers terminology without techno-syntax.

User stories also drive the creation of the acceptance tests. One or more automated acceptance tests must be created to verify the user story has been correctly implemented. One of the biggest misunderstandings with user stories is how they differ from traditional requirements specifications. The biggest difference is in the level of detail. User stories should only provide enough detail to make a reasonably low risk estimate of how long the story will take to implement. When the time comes to implement the story developers will go to the customer and receive a detailed description of the requirements face to face.

Developers estimate how long the stories might take to implement. Each story will get a 1, 2 or 3 week estimate in "ideal development time". This ideal development time is how long it would take to implement the story in code if there were no distractions, no other assignments, and you knew exactly what to do. Longer than 3 weeks means you need to break the story down further. Less than 1 week and you are at too detailed a level, combine some stories. About 80 user stories plus or minus 20 is a perfect number to create a release plan during release planning.

Another difference between stories and a requirements document is a focus on user needs. You should try to avoid details of specific technology, data base layout, and algorithms. You should try to keep stories focused on user needs and benefits as opposed to specifying GUI layouts.

Release Planning

A release planning meeting is used to create a release plan, which lays out the overall project. The release plan is then used to create iteration plans for each individual iteration
.

It is important for technical people to make the technical decisions and business people to make the business decisions. Release planning has a set of rules that allows everyone involved with the project to make their own decisions. The rules define a method to negotiate a schedule everyone can commit to.

The essence of the release planning meeting is for the development team to estimate each user story in terms of ideal programming weeks. An ideal week is how long you imagine it would take to implement that story if you had absolutely nothing else to do. No dependencies, no extra work, but do include tests. The customer then decides what story is the most important or has the highest priority to be completed.

User stories are printed or written on cards. Together developers and customers move the cards around on a large table to create a set of stories to be implemented as the first (or next) release. A useable, testable system that makes good business sense delivered early is desired. You may plan by time or by scope. The project velocity is used to determine either how many stories can be implemented before a given date (time) or how long a set of stories will take to finish (scope). When planning by time multiply the number of iterations by the project velocity to determine how many user stories can be completed. When planning by scope divide the total weeks of estimated user stories by the project velocity to determine how many iterations till the release is ready.

Individual iterations are planned in detail just before each iteration begins and not in advance. The release planning meeting was called the planning game and the rules can be found at the Portland Pattern Repository.

When the final release plan is created and is displeasing to management it is tempting to just change the estimates for the user stories. You must not do this. The estimates are valid and will be required as-is during the iteration planning meetings. Underestimating now will cause problems later. Instead negotiate an acceptable release plan. Negotiate until the developers, customers, and managers can all agree to the release plan.

The base philosophy of release planning is that a project may be quantified by four variables; scope, resources, time, and quality. Scope is how much is to be done. Resources are how many people are available. Time is when the project or release will be done. And quality is how good the software will be and how well tested it will be.

Management can only choose 3 of the 4 project variables to dictate, development always gets the remaining variable. Note that lowering quality less than excellent has unforeseen impact on the other 3. In essence there are only 3 variables that you actually want to change. Also let the developers moderate the customers desire to have the project done immediately by hiring too many people at one time.

Make frequent small releases

The development team needs to release iterative versions of the system to the customers often. The release planning meeting is used to discover small units of functionality that make good business sense and can be released into the customer's environment early in the project. This is critical to getting valuable feedback in time to have an impact on the system's development. The longer you wait to introduce an important feature to the system's users the less time you will have to fix it.

Project Velocity

The project velocity (or just velocity) is a measure of how much work is getting done on your project. To measure the project velocity you simply add up the estimates of the user stories that were finished during the iteration. It's just that simple. You also total up the estimates for the tasks finished during the iteration. Both of these measurements are used for iteration planning.

During the iteration planning meeting customers are allowed to choose the same number of user stories equal to the project velocity measured in the previous iteration. Those stories are broken down into technical tasks and the team is allowed to sign up for the same number of tasks equal to the previous iteration's project velocity.

This simple mechanism allows developers to recover and clean up after a difficult iteration and averages out estimates. Your project velocity goes up by allowing developers to ask the customers for another story when their work is completed early and no clean up tasks remain.

A few ups and downs in project velocity are expected. You should use a release planning meeting to re-estimate and re-negotiate the release plan if your project velocity changes dramatically for more than one iteration. Expect the project velocity to change again when the system is put into production due to maintenance tasks.

Project velocity is about as detailed a measure as you can make that will be accurate. Don't bother dividing the project velocity by the length of the iteration or the number of people. This number isn't any good to compare two project's productivity. Each project team will have a different bias to estimating stories and tasks, some estimate high, some estimate low. It doesn't matter in the long run. Tracking the total amount of work done during each iteration is the key to keeping the project moving at a steady predictable pace.

The problem with any project is the initial estimate. Collecting lots of details does not make your initial estimate anything other than a guess. Worry about estimating the overall scope of the project and get that right instead of creating large documents. Consider spending the time you would have invested into creating a detailed specification on actually doing a couple iterations of development. Measure the project velocity during these initial explorations and make a much better guess at the project's total size.

Iterative Development

Iterative Development adds agility to the development process. Divide your development schedule into about a dozen iterations of 1 to 3 weeks in length. Keep the iteration length constant through out the project. This is the heart beat of your project. It is this constant that makes measuring progress and planning simple and reliable in XP.

Don't schedule your programming tasks in advance. Instead have an iteration planning meeting at the beginning of each iteration to plan out what will be done. Just-in-time planning is an easy way to stay on top of changing user requirements.

It is also against the rules to look ahead and try to implement anything that it is not scheduled for this iteration. There will be plenty of time to implement that functionality when it becomes the most important story in the release plan.

Take your iteration deadlines seriously! Track your progress during an iteration. If it looks like you will not finish all of your tasks then call another iteration planning meeting, re-estimate, and remove some of the tasks.

Concentrate your effort on completing the most important tasks as chosen by your customer, instead of having several unfinished tasks chosen by the developers.

It may seem silly if your iterations are only one week long to make a new plan, but it pays off in the end. By planning out each iteration as if it was your last you will be setting yourself up for an on-time delivery of your product. Keep your projects heart beating loud and clear.

Iteration Planning

An iteration planning meeting is called at the beginning of each iteration to produce that iteration's plan of programming tasks. Each iteration is 1 to 3 weeks long. User stories are chosen for this iteration by the customer from the release plan in order of the most valuable to the customer first. Failed acceptance tests to be fixed are also selected. The customer selects user stories with estimates that total up to the project velocity from the last iteration.

The user stories and failed tests are broken down into the programming tasks that will support them. Tasks are written down on index cards like user stories. While user stories are in the customer's language, tasks are in the developer's language. Duplicate tasks can be removed. These task cards will be the detailed plan for the iteration.

Developers sign up to do the tasks and then estimate how long their own tasks will take to complete. It is important for the developer who accepts a task to also be the one who estimates how long it will take to finish. People are not interchangeable and the person who is going to do the task must estimate how long it will take.

Each task should be estimated as 1, 2, or 3 ideal programming days in duration. Ideal programming days are how long it would take you to complete the task if there were no distractions. Tasks which are shorter than 1 day can be grouped together. Tasks which are longer than 3 days should be broken down farther.

Now the project velocity is used again to determine if the iteration is over booked or not. Total up the time estimates in ideal programming days of the tasks, this must not exceed the project velocity from the previous iteration. If the iteration has too much then the customer must choose user stories to be put off until a later iteration (snow plowing). If the iteration has too little then another story can be accepted. The velocity in task days (iteration planning) overrides the velocity in story weeks (release planning) as it is more accurate.

It is often alarming to see user stories being snow plowed. Don't panic. Remember the importance of unit testing and refactoring. A debt in either of these areas will slow you down. Avoid adding any functionality before it is scheduled. Just add what you need for today. Adding anything extra will slow you down.

Don't be tempted into changing your task and story estimates. The planning process relies on the cold reality of consistent estimates, fudging them to be a little lower creates more problems.

Keep an eye on your project velocity and snow plowing. You may need to re-estimate all the stories and re-negotiate the release plan every three to five iterations, this is normal. So long as you always implement the most valuable stories first you will always be doing as much as possible for your customers and management.

An iterative development style can add agility to your development process. Try just in time planning by not planning specific programming tasks farther ahead than the current iteration.

Move People Around

Move people around to avoid serious knowledge loss and coding bottle necks. If only one person on your team can work in a given area and that person leaves or you just have numerous things waiting to be done in that section you will find your project's progress reduced to a crawl.

Cross training is often an important consideration in companies trying to avoid islands of knowledge, which are so susceptible to loss. Moving people around the code base in combination with pair programming does your cross training for you. Instead of one person who knows everything about a given section of code, everyone on the team knows much of the code in each section.

A team is much more flexible if everyone knows enough about every part of the system to work on it. Instead of having a few people overloaded with work while other team members have little to do, the whole team can be productive. Any number of developers can be assigned to the hottest part of the system. Flexible load balancing of this type is a manager's dream come true.

Simply encourage everyone to try working on a new section of the system at least part of each iteration. Pair programming makes it possible without losing productivity and ensures continuity of thought. One person from a pair can be swapped out while the other continues with a new partner if desired.

Daily Stand Up Meeting

At a typical project meeting most attendees do not contribute, but attend just to hear the outcome. A large amount of developer time is wasted to gain a trivial amount of communication. Having many people attend every meeting drains resources from the project and also creates a scheduling nightmare.

Communication among the entire team is the purpose of the stand up meeting. A stand up meeting every morning is used to communicate problems, solutions, and promote team focus. Everyone stands up in a circle to avoid long discussions. It is more efficient to have one short meeting that every one is required to attend than many meetings with a few developers each.

When you have daily stand up meetings any other meeting's attendance can be based on who will actually be needed and will contribute. Now it is possible to avoid even scheduling most meetings. With limited attendance most meetings can take place spontaneously in front of a computer, where code can be browsed and ideas actually tried out.

The daily stand up meeting is not another meeting to waste people's time. It will replace many other meetings giving a net savings several times its own length.

Fix XP When It Breaks

Fix the process when it breaks. We don't say *if* because we already know you will need to make some changes for your specific project. Follow the XP Rules to start with, but do not hesitate to change what doesn't work. This doesn't mean the team can do whatever they want. The rules must be followed until the team has changed them. All of your developers must know exactly what to expect from each other, having a set of rules is the only way to set these expectations. Have meetings to talk about what is working and what is not and devise ways to improve XP.

The Customer is Always Available

One of the few requirements of extreme programming (XP) is to have the customer available. Not only to help the development team, but to be a part of it as well. All phases of an XP project require communication with the customer, preferably face to face, on site. It's best to simply assign one or more customers to the development team. Beware though, this seems like a long time to keep the customer hanging and the customer's department is liable to try passing off a trainee as an expert. You need the expert.

User Stories are written by the customer, with developers helping, to allow time estimates, and assign priority. The customers help make sure most of the system's desired functionality is covered by stories.

During the release planning meeting the customer will need to negotiate a selection of user stories to be included in each scheduled release. The timing of the release may need to be negotiated as well. The customers must make the decisions that affect their business goals. A release planning meeting is used to define small incremental releases to allow functionality to be released to the customer early. This allows the customers to try the system earlier and give the developers feedback sooner.

Because details are left off the user stories the developers will need to talk with customers to get enough detail to complete a programming task. Projects of any significant size will require a full time commitment from the customer.
The customer will also be needed to help with functional testing. The test data will need to be created and target results computed or verified. Functional tests verify that the system is ready to be released into production. It can happen that the system will not pass all functional tests just prior to release. The customer will be needed to review the test score and allow the system to continue into production or stop it.

This may seem like a lot of the customer's time at first but we should remember that the customer's time is spared initially by not requiring a detailed requirements specification and saved later by not delivering an uncooperative system. Some problems can occur when multiple customers are made available part time. Experts in any field have a tendency to argue. This is natural. Solve this problem by requiring all the customers to be available for occasional group meetings to hash out differences of opinion.

Coding Standards

When you create your tests first, before the code, you will find it much easier and faster to create your code. The combined time it takes to create a unit test and create some code to make it pass is about the same as just coding it up straight away. But, if you already have the unit tests you don't need to create them after the code saving you some time now and lots later.

Creating a unit test helps a developer to really consider what needs to be done. Requirements are nailed down firmly by tests. There can be no misunderstanding a specification written in the form of executable code.

You also have immediate feedback while you work. It is often not clear when a developer has finished all the necessary functionality. Scope creep can occur as extensions and error conditions are considered. If we create our unit tests first then we know when we are done; the unit tests all run.

There is also a benefit to system design. It is often very difficult to unit test some software systems. These systems are typically built code first and testing second, often by a different team entirely. By creating tests first your design will be influenced by a desire to test everything of value to your customer. Your design will reflect this by being easier to test.

There is a rhythm to developing software unit test first. You create one test to define some small aspect of the problem at hand. Then you create the simplest code that will make that test pass. Then you create a second test. Now you add to the code you just created to make this new test pass, but no more! Not until you have yet a third test. You continue until there is nothing left to test. The coffee maker problem shows an example written in Java.

The code you will create is simple and concise, implementing only the features you wanted. Other developers can see how to use this new code by browsing the tests. Input whose results are undefined will be conspicuously absent from the test suite.

Pair Programming

All code to be included in a production release is created by two people working together at a single computer. Pair programming increases software quality without impacting time to deliver. It is counter intuitive, but 2 people working at a single computer will add as much functionality as two working separately except that it will be much higher in quality. With increased quality comes big savings later in the project.

The best way to pair program is to just sit side by side in front of the monitor. Slide the key board and mouse back and forth. One person types and thinks tactically about the method being created, while the other thinks strategically about how that method fits into the class. It takes time to get used to pair programming so don't worry if it feels awkward at first.

Sequential Integration

Without controlling source code integration developers test their code and integrate believing all is well. But because of parallel integration of source code modules there is a combination of source code which has not been tested together before. Numerous integration problems arise without detection.

Further problems arise when there is no clear cut latest version. This applies not only to the source code but the unit test suite which must verify the source code correctness. If you can not lay your hands on a complete, correct, and consistent test suite you will be chasing bugs that do not exist and passing up bugs that do.

Some projects try to have developers own specific classes. The class owners then ensure that code for each class is integrated and released properly. This reduces the problem but interclass dependencies can still be wrong. It does not solve the whole problem.

Yet another way is to appoint an integrator or integration team. Integrating code from multiple developers is more than a single person can handle. And a team of people is too big a resource to integrate more than once a week. In this environment developers work with obsolete versions which are then erroneously re-integrated into the code base. These solutions do not address the root problem. You want developers to be able to proceed in parallel, courageously making changes to any part of the system required, but you also want an error free integration of those efforts. Like a dozen steaming locomotives headed for the switch house all at the same time, there is going to be trouble. Instead of restricting development to being sequential, or requiring complex integration procedures let's rethink the problem. Our locomotives can all get into the switching house without a crash if they just take turns. We need to do this with code integration as well.

Strictly sequential (or single threaded) integration by the developers themselves in combination with collective code ownership is a simple solution to this problem. All source code is released to the source code safe or repository by taking turns. That is, only one development pair integrates, tests and releases changes to the source code repository at any given moment. Single threaded integration allows a latest version to be consistently identified.

This is not to imply that you can not integrate your own changes with the latest version at your own workstation any time you want. You just can't release your changes to the team with out waiting for your turn.

Some sort of lock mechanism is required. The simplest thing is a physical token passed from developer to developer. A single computer dedicated to this purpose works well if the development team is co-located. Integrating and releasing code often shortens the time needed to hold the lock and thus reducing the wait time to acquire the lock.

Integrate Often

Developers should be integrating and releasing code into the code repository every few hours, when ever possible. In any case never hold onto changes for more than a day. Continuous integration often avoids diverging or fragmented development efforts, where developers are not communicating with each other about what can be re-used, or what could be shared. Everyone needs to work with the latest version. Changes should not be made to obsolete code causing integration head aches.

Each development pair is responsible for integrating their own code when ever a reasonable break presents itself. This may be when the unit tests all run at 100% or some smaller portion of the planned functionality is finished. Only one pair integrates at any given moment and after only a few hours of coding to reduce the potential problem set to almost nothing.

Almost continuous integration avoids or detects compatibility problems early. Integration is a "pay me now or pay me more later" kind of activity. That is, if you integrate through out the project in small amounts you will not find your self trying to integrate the system for weeks at the project's end while the deadline slips by. Always work in the context of the latest version of the system.

Collective Code Ownership

Collective Code Ownership encourages everyone to contribute new ideas to all segments of the project. Any developer can change any line of code to add functionality, fix bugs, or refactor. No one person becomes a bottle neck for changes.

This is hard to understand at first. It's almost inconceivable that an entire team can be responsible for the system's architecture. Not having a single chief architect that keeps some visionary flame alive seems like it couldn't possibly work.

But it is not uncommon to ask a chief architect a question and get an answer that is just plain wrong. It is not a failing of your lead programmers. Any non-trivial system can not be held in one person's mind. Other programmers are hard at work changing the system without benefit of the architect's vision. Whether you realize it or not your architecture is already distributed among your team. If the entire team already has some responsibility for architectural decisions, shouldn't they receive the authority as well?

The way this works is for each developer to create unit tests for their code as it is developed. All code that is released into the source code repository includes unit tests. Code that is added, bugs as they are fixed, and old functionality as it is changed will be covered by automated testing. Now you can rely on the test suite to watch dog your entire code repository. Before any code is released it must pass the entire test suite at 100%.

Once this is in place anyone can make a change to any method of any class and release it to the code repository as needed. When combined with frequent integration developers rarely even notice a class has been extended or repaired.

In practice collective code ownership is actually more reliable than putting a single person in charge of watching specific classes. Especially since a person may leave the project at any time.

Optimize Last

Do not optimize until the end. Never try to guess what the system's bottle neck will be. Measure it!
Make it work, make it right, then make it fast.

No Overtime

Working overtime sucks the spirit and motivation out of a team. Projects that require overtime to be finished on time will be late no matter what you do. Instead use a release planning meeting to change the project scope or timing. Increasing resources by adding more people is also a bad idea when a project is running late.

Simplicity is the Key

A simple design always takes less time to finish than a complex one. So always do the simplest thing that could possibly work. If you find something that is complex replace it with something simple. It's always faster and cheaper to replace complex code now, before you waste a lot more time on it. Keep things as simple as possible as long as possible by never adding functionality before it is scheduled. Beware though, keeping a design simple is hard work.

Choose a System Metaphor

Choose a system metaphor to keep the team on the same page by naming classes and methods consistently. What you name your objects is very important for understanding the overall design of the system and code reuse as well. Being able to guess at what something might be named if it already existed and being right is a real time saver. Choose a system of names for your objects that everyone can relate to without specific, hard to earn knowledge about the system.

For example the Chrysler payroll system was built as a production line. At another auto manufacturer car sales were structured as a bill of materials. There is also a metaphor known as the naive metaphor which is based on your domain itself. But don't choose the naive metaphor unless it is simple enough.

CRC Cards

Use Class, Responsibilities, and Collaboration (CRC) Cards to design the system as a team. The biggest value of CRC cards is to allow people to break away from the procedural mode of thought and more fully appreciate object technology. CRC Cards allow entire project teams to contribute to the design. The more people who can help design the system the greater the number of good ideas incorporated.

Individual CRC Cards are used to represent objects. The class of the object can be written at the top of the card, responsibilities listed down the left side, collaborating classes are listed to the right of each responsibility. We say "can be written" because once a CRC session is in full swing participants usually only need a few cards with the class name and virtually no cards written out in full. A short example is shown as part of the coffee maker problem.

A CRC session proceeds with someone simulating the system by talking about which objects send messages to other objects. By stepping through the process weaknesses and problems are easily uncovered. Design alternatives can be explored quickly by simulating the design being proposed.

If you find too many people speaking and moving cards at once then simply limit the number of people standing and moving cards to two. When one person sits down another may stand up. This works for sessions that get out of hand, which often happens as teams become rowdy when a tough problem is finally solved.

One of the biggest criticisms of CRC Cards is the lack of written design. This is usually not needed as CRC Cards make the design seem obvious. Should a more permanent record be required, one card for each class can be written out in full and retained as documentation. A design, once envisioned as if it were already built and running, stays with a person for some time.

Create a Spike Solution

Create spike solutions to figure out answers to tough technical or design problems. A spike solution is a very simple program to explore potential solutions. Build a system which only addresses the problem under examination and ignore all other concerns. Most spikes are not good enough to keep, so expect to throw it away. The goal is reducing the risk of a technical problem or increase the reliability of a user story's estimate.

When a technical difficulty threatens to hold up the system's development put a pair of developers on the problem for a week or two and reduce the potential risk.

Never Add Functionality Early

Keep the system uncluttered with extra stuff you guess will be used later. Only 10% of that extra stuff will ever get used, so you are wasting 90% of your time. We are all tempted to add functionality now rather than later because we see exactly how to add it or because it would make the system so much better. It seems like it would be faster to add it now. But we need to constantly remind our selves that we are not going to actually need it. Extra functionality will always slow us down and squander our resources. Turn a blind eye towards future requirements and extra flexibility. Concentrate on what is scheduled for today only.

Refactor Mercilessly

We computer programmers hold onto our software designs long after they have become unwieldy. We continue to use and reuse code that is no longer maintainable because it still works in some way and we are afraid to modify it. But is it really cost effective to do so? Extreme Programming (XP) takes the stance that it is not. When we remove redundancy, eliminate unused functionality, and rejuvenate obsolete designs we are refactoring. Refactoring throughout the entire project life cycle saves time and increases quality.

Refactor mercilessly to keep the design simple as you go and to avoid needless clutter and complexity. Keep your code clean and concise so it is easier to understand, modify, and extend. Make sure everything is expressed once and only once. In the end it takes less time to produce a system that is well groomed.

There is a certain amount of Zen to refactoring. It is hard at first because you must be able to let go of that perfect design you have envisioned and accept the design that was serendipitously discovered for you by refactoring. You must realize that the design you envisioned was a good guide post, but is now obsolete.

A caterpillar is perfectly designed to eat vast amounts of foliage but he can't find a mate, he must refactor himself into a butterfly before he is designed to search the sky for others of his own kind. Let go of your notions of what the system should or should not be and try to see the the new design as it emerges before you.

Unit Tests

Unit tests are one of the corner stones of Extreme Programming (XP). But unit tests XP style is a little different. First you should create or download a unit test framework to be able to create automated unit tests suites. Second you should test all classes in the system. Trivial getter and setter methods are usually omitted. And last you should try to create your tests first, before the code.

Unit tests are released into the code repository along with the code they test. Code without tests may not be released. If a unit test is discovered to be missing it must be created at that time.

The biggest resistance to dedicating this amount of time to unit tests is a fast approaching deadline. But during the life of a project an automated test can save you a hundred times the cost to create it by finding and guarding against bugs. The harder the test is to write the more you need it because the greater your savings will be. Automated unit tests offer a pay back far greater than the cost of creation.

Another common misconception is that unit tests can be written in the last three months of the project. Unfortunately, without the unit tests development drags on and eats up those last three months and then some. Even if the time is available good unit test suites take time to evolve. Discovering all the problems that can occur takes time. In order to have a complete unit test suite when you need it you must begin creating the tests today when you don't.

Unit tests enable collective code ownership. When you create unit tests you guard your functionality from being accidentally harmed. Requiring all code to pass all unit tests before it can be released ensures all functionality always works. Code ownership is not required if all classes are guarded by unit tests.

Unit tests enable refactoring as well. After each small change the unit tests can verify that a change in structure did not introduce a change in functionality.

Building a single universal unit test suite for validation and regression testing enables frequent integration. It is possible to integrate any recent changes quickly then run your own latest version of the test suite. When a test fails your latest versions are incompatible with the team's latest versions. Fixing small problems every few hours takes less time than fixing huge problems just before the deadline. With automated unit tests it is possible to merge a set of changes with the latest released version and release in a short time.

Often adding new functionality will require changing the unit tests to reflect the functionality. While it is possible to introduce a bug in both the code and test it rarely happens in actual practice. It does occasionally happen that the test is wrong, but the code is right. This is revealed when the problem is investigated and is fixed. Creating tests independent of code, hopefully before code, sets up checks and balances and greatly improves the chances of getting it right the first time.

Unit Tests provide a safety net of regression tests and validation tests so that you can refactor and integrate effectively. As they say at the circus; never work without a net! Creating the unit test before the code helps even further by solidifying the requirements, improving developer focus, and avoid creeping elegance.

When a Bug is Found

When a bug is found tests are created to guard against it coming back. A bug in production requires an acceptance test be written to guard against it. Creating an acceptance test first before debugging helps customers concisely define the problem and communicate that problem to the programmers. Programmers have a failed test to focus their efforts and know when the problem is fixed.

Given a failed acceptance test, developers can create unit tests to show the defect from a more source code specific point of view. Failing unit tests give immediate feedback to the development effort when the bug has been repaired. When the unit tests run at 100% then the failing acceptance test can be run again to validate the bug is fixed.

Acceptance Tests

Acceptance tests are created from user stories. During an iteration the user stories selected during the iteration planning meeting will be translated into acceptance tests. The customer specifies scenarios to test when a user story has been correctly implemented. A story can have one or many acceptance tests, what ever it takes to ensure the functionality works.

Acceptance tests are black box system tests. Each acceptance test represents some expected result from the system. Customers are responsible for verifying the correctness of the acceptance tests and reviewing test scores to decide which failed tests are of highest priority. Acceptance tests are also used as regression tests prior to a production release.

A user story is not considered complete until it has passed its acceptance tests. This means that new acceptance tests must be created each iteration or the development team will report zero progress.

Quality assurance (QA) is an essential part of the XP process. On some projects QA is done by a separate group, while on others QA will be an integrated into the development team itself. In either case XP requires development to have much closer relationship with QA.

Acceptance tests should be automated so they can be run often. The acceptance test score is published to the team. It is the team's responsibility to schedule time each iteration to fix any failed tests.

The name acceptance tests was changed from functional tests. This better reflects the intent, which is to guarantee that a customers requirements have been met and the system is acceptable.