


Extreme Programming: A gentle introduction.

*Lessons
Learned*

The goal of this site is to provide an introduction and overview of Extreme Programming (XP). For a guided tour of XP follow the trail of little  buttons, starting here. Returning visitors can jump to [recent changes](#) to see what's new.

Let's begin with a simple question: [What is XP?](#) As you will see, it is a deliberate and disciplined approach to software development.

Next we might wonder [when to use XP](#). Risky projects with dynamic requirements are perfect for XP. These projects will experience greater success and developer productivity.

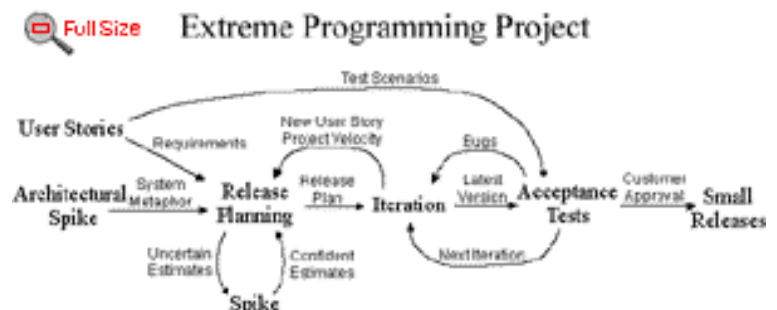
But do we need yet [another software methodology?](#) Actually we do. XP is a refreshing new approach. XP is successful because it emphasizes customer involvement and promotes team work.

So how could this possibly work? The most surprising aspect of XP is its [simple rules and practices](#). They seem awkward and perhaps even naive at first, but soon become a welcome change. Customers enjoy being partners in the software process and developers actively contribute regardless of experience level.

[What has changed here?](#) | [XP Practices and Rules](#) | [This site zipped](#) | [Email the webmaster](#)

Last modified January 26, 2003. See [recent changes](#).

Copyright (c) 1999, 2000, 2001 Don Wells. All Rights reserved.



The rules and practices must support each other. The [XP Map](#) shows how they work together to form a development methodology. Unproductive activities have been trimmed to reduce costs and frustration.

I want to try XP [how do I start?](#) Add a little to your current methodology or try it all at once. There is much here of benefit to any project. What have other projects already [learned about XP?](#) Some important lessons learned.

Where can I [get more information?](#) There are classes, conferences, books, and web sites. The [XP Agile Universe](#) conference will be held in New Orleans August 10-13, 2003.

A [Chinese Translation](#) is available.



What We Have Learned About Extreme Programming

*Lessons
Learned*

Release Planning

- The Team owns the schedule.

Simplicity

- Simplicity is easier to maintain.
- You aren't going to need it.

System Metaphor

- A metaphor can simplify the design.

Pair Programming

- The whole is greater than the parts.
- Some rules of thumb.
- Rein in the Cowboy Coders.
- Pairing reduces indecision.
- Make no mistake, pairing is hard work.
- Experimental evidence for pairing.
- Code reviews considered hurtful.

Integrate Often

- XP and Databases.
- Integration can be reduced to seconds.

Optimize Last

- It may not be as slow as you think.

Unit Tests

- Well worth the investment.
- Could have saved us some time.
- Testing first makes the code testable.

Acceptance Tests

- They give a feeling of stability.
- Create a tool to maintain them.

If you have been using Extreme Programming (XP) or a component practice tell us what you have learned! If you have a story to tell about something that saved you time please send it in. If you have a story about what doesn't work send that too. Please write a lesson learned about an XP practice and send it to the webmaster.

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules and Practices](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved



The Team Owns the Schedule

*Lessons
Learned*

The right team can accomplish goals beyond one's wildest dreams! These teams are also known as highly performing learning organizations; they elicit effective project management practices through team ownership of the plan, and they demand software development projects to be managed in two segments: a project deliverables schedule (**release plan**) and an **iteration plan** of programming (engineering) tasks.

These two are not the same; most project managers fail to let the engineering team work to the beat of the engineering plan, instead they

assign work and force teams to use a project plan focused only on customer deliverables.

In fact, we all know this erroneous practice is contradictory to good object thinking, let alone reusable frameworks, testing, refactoring, Extreme Programming (XP), and so on. Thus, we achieved our successes when we used extreme schedule negotiation [a.k.a **release planning**]. ☺ ☺

Jeanine De Guzman

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Lessons Learned](#) | [Release Planning](#) | [Email the webmaster](#)

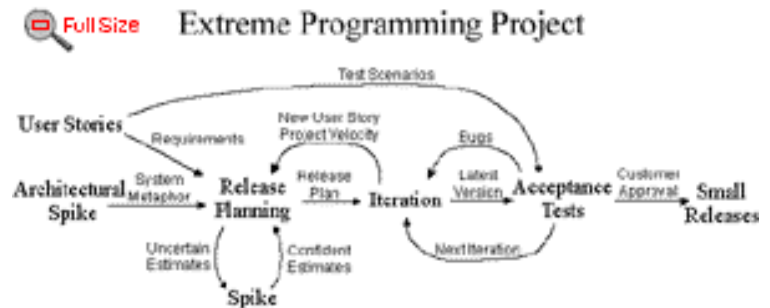
Copyright 1998 Jeanine De Guzman. Logos Copyright 1999 Don Wells all rights reserved.

XP Release Plan

After user stories have been written you can use a **release planning meeting** to create a release plan. The release plan specifies exactly which **user stories** are going to be implemented for each system release and dates for those releases. This gives a set of user stories for customers to choose from during the **iteration planning meeting** to be implemented during the next iteration. These selected stories are then translated into individual programming tasks to be implemented during the iteration to complete the stories.

Stories are also translated into **acceptance tests** during the iteration. These acceptance tests are run during this iteration, and subsequent iterations to verify when the stories are finished correctly and continue to work correctly.

When the **project velocity** changes dramatically for a couple iterations or in any case after several iterations go ahead and schedule a release planning meeting with your customers and create a new release plan.



The release plan used to be called the commitment schedule. The name was changed to more accurately describe its purpose and be more consistent with iteration plan.

 [Wiki Wiki](#)
The Portland
Pattern Repository

[ExtremeProgramming.org home](#) | [Release Planning](#) | [Small Releases](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.

XP Release Planning

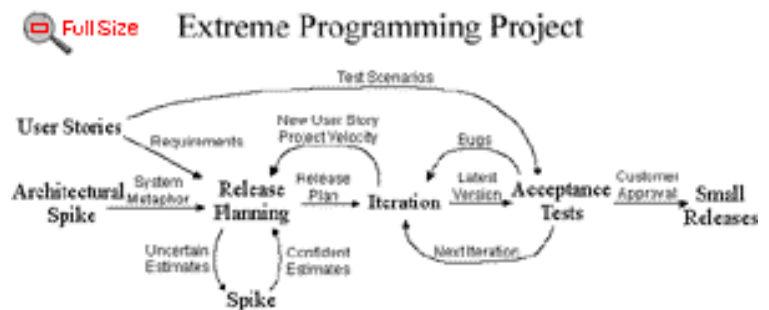
*Lessons
Learned*

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.

Continued on page 2 ↗ ↘

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules](#) | [Release Plan](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



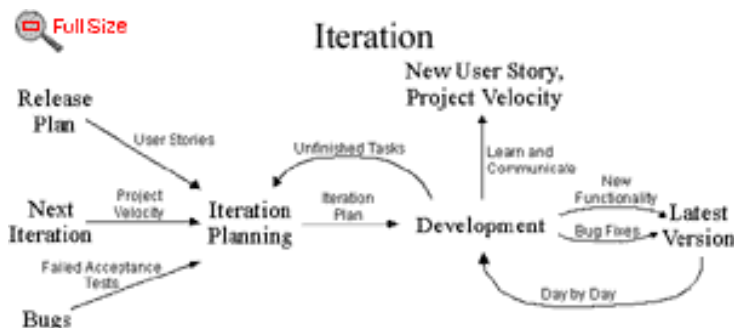
Iterative Development

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. ☺ ☑

Wiki Wiki
The Portland
Pattern Repository

XP
rogramming.com

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules](#) | [Iteration Planning](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



The load factor was how a project was tracked before **project velocity** became more popular. The load factor equals actual calendar days to complete a task divided by the developer's estimated "ideal" days to do it. That is, think of a task that would take you one day if you could focus completely on it. Now picture yourself trying to get it done in the real world. The number of days it actually takes is the load factor.

Load factors from 2 to 5 are normal. If you need to guess at a load factor to get started you should consider people's experience and the technology being used. A 2 is optimistic, a 3 is typical, while a 4 and 5 are for projects using unfamiliar technology. Ron Jeffries recommends just simply using a 3 as an initial guess for new projects.

After making an initial guess you must then measure and track either the load factor, or better yet, the project velocity throughout the project.

The load factor can not be used to compare two projects. Each project and team is unique and will have different load factors for different reasons.

Use a release planning meeting to re-estimate and re-negotiate the release plan if the load factor changes dramatically. Expect the load factor to change again when the system is put into production due to maintenance tasks. ☹ ☹

 [Wiki Wiki](#)
The Portland
Pattern Repository

 [XP](#)
programming.com

 [Wiki Wiki](#)
The Portland
Pattern Repository

[ExtremeProgramming.org home](#) | [XP Rules](#) | [Back to Project Velocity](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.

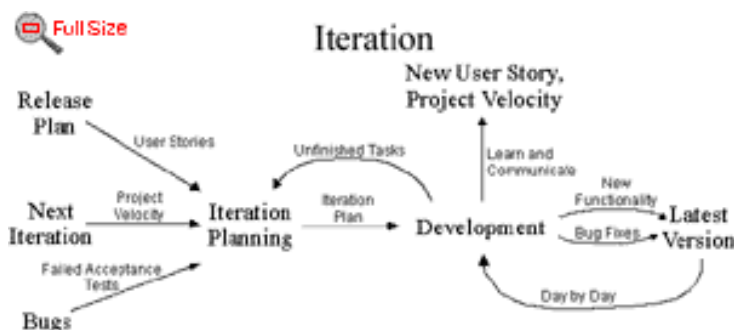
XP Project Velocity

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. ☺ ☺

 [Wiki Wiki](#)
The Portland
Pattern Repository

 [Wiki Wiki](#)
The Portland
Pattern Repository

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules](#) | [Iterative Development](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.

XP User Stories

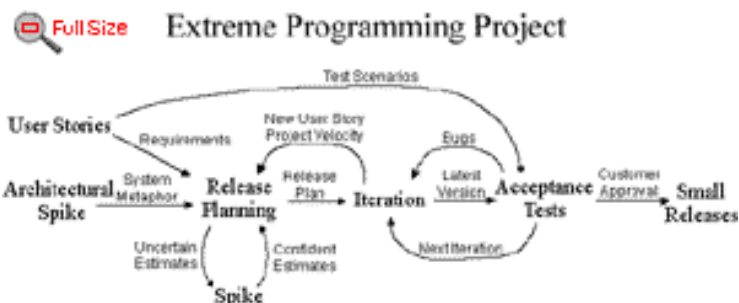
Extreme Programming

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. ☒ ☒



[ExtremeProgramming.org home](#) | [XP Rules](#) | [Release Planning](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved



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.

[Continued on page 2.](#)  

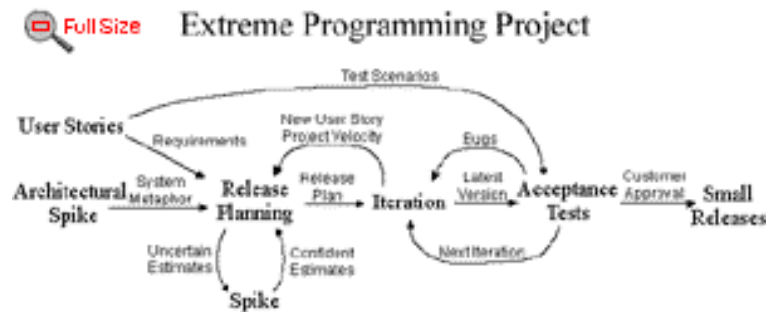
[ExtremeProgramming.org home](#) | [XP Rules](#) | [Coding Standards](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved. Any resemblance to actual customers is purely coincidental.



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. ☹ ☹



[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [XP Rules](#) | [Project Velocity](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved.



The Rules and Practices of Extreme Programming.

*Lessons
Learned*

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.

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.

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](#).

Testing

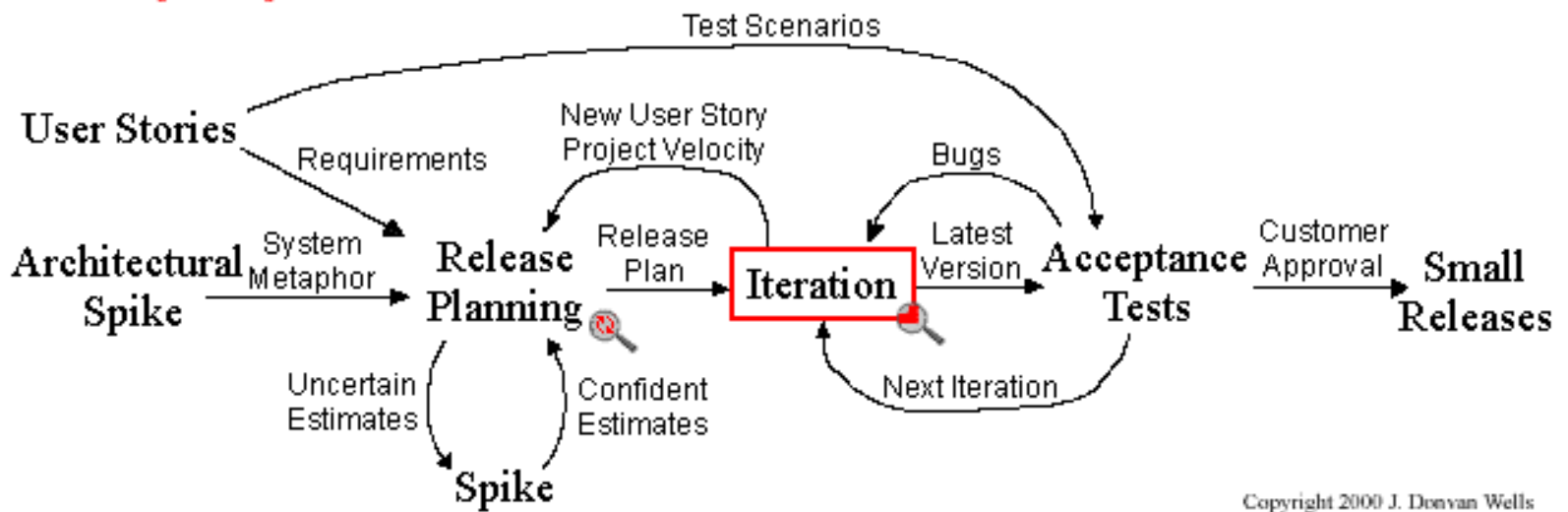
- ❏ ❏ All code must have [unit tests](#).
- ❏ ❏ All code must pass all [unit tests](#) before it can be released.
- ❏ ❏ When [a bug is found](#) tests are created.
- ❏ ❏ [Acceptance tests](#) are run often and the score is published.

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Map](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved



Extreme Programming Project



Copyright 2000 J. Donovan Wells

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [Zoom in on Iteration.](#) | [Starting with XP](#) | [Email the webmaster](#)



Copyright 2000 Don Wells all rights reserved

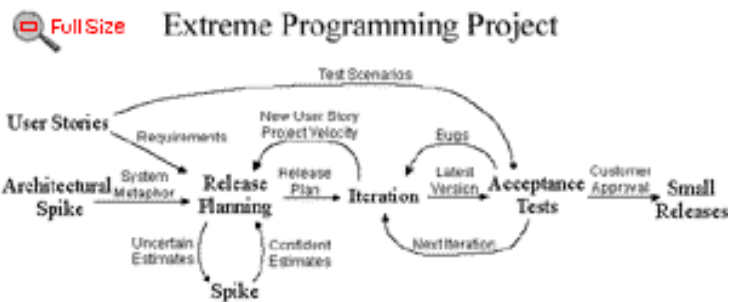


Choose a System Metaphor

*Lessons
Learned*

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. ☹ ☹



Wiki Wiki
The Portland
Pattern Repository

XPlorations

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules](#) | [CRC Cards](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



A System Metaphor can Simplify the Design

Lessons Learned

When we rewrote the VCAPS system we wanted to choose a system metaphor but found that it was difficult. We had data from a variety of sources that had no commonality it seemed.

A breakthrough was when we discovered that all the data could be uniformly represented as if it were a bill of material. So we chose bill of material as our system metaphor. In the old system it was often very difficult to find the right data. In the new system it was obvious where it would be.

From top to bottom our new model has the same metaphor. After we had gotten down

the road with this new metaphor we measured that this uniformity was going to save us 95% in database size. Also, we now had one and only one way of accessing data no matter where it came from or what it represented.

This uniformity also simplified our algorithms and reduced processing time. We went from processing times averaging about 15 minutes to coming back immediately. ☺ ☺

Don Wells

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Lessons Learned](#) | [System Metaphor](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.

XP CRC Cards

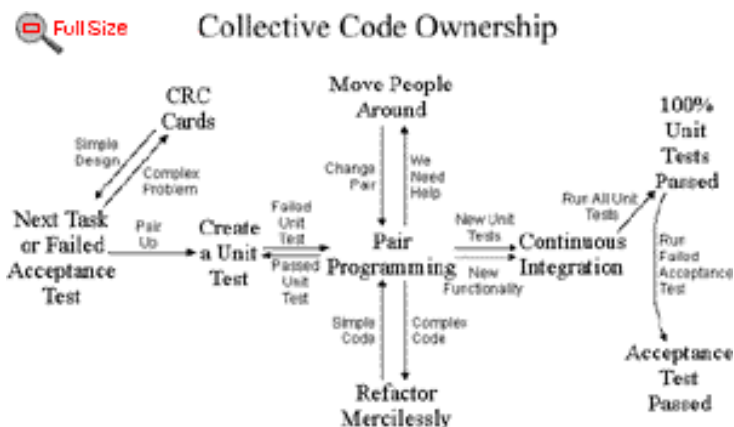
Extreme Programming

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.



Wiki Wiki
The Portland
Pattern Repository

XP
programming.com

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules](#) | [Spike Solution](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



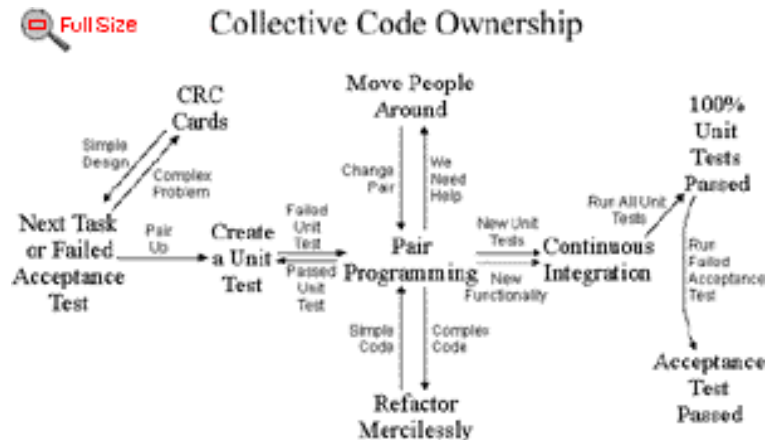
Collective Code Ownership

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. ☺ ☹

Wiki Wiki
The Portland
Pattern Repository

XP
programming.com

XPlorations

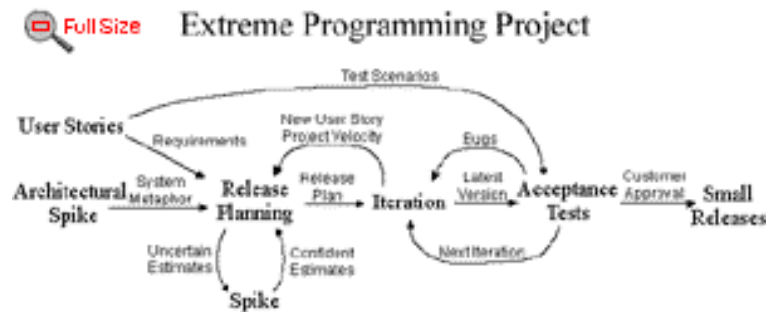
[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules](#) | [Optimize Last](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.

XP 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. 🗑️ 🗑️



[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [XP Rules](#) | [Acceptance Tests](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved.



Acceptance Tests

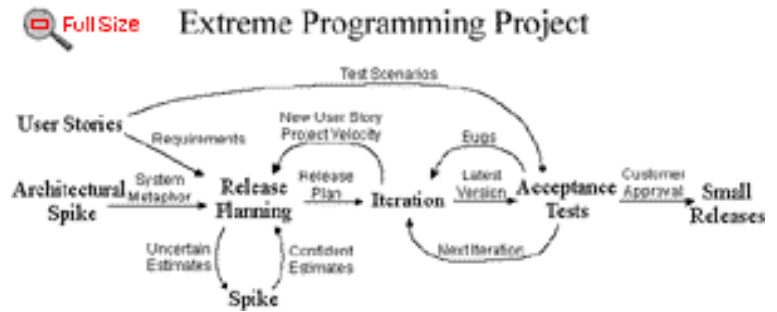
*Lessons
Learned*

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. ☺ ☺

Wiki Wiki
The Portland
Pattern Repository

XP
programming.com

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [Starting with XP](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved.

Acceptance Tests Don't Just

Eliminate

Bugs They Add a Feeling of

Stability

*Lessons
Learned*

When Extreme Programming (XP) was first introduced to the VCAPS project there were no automated acceptance tests. It took a while to add coverage. We added tests for all new functionality and old functionality as it required changes. After about a year we had an estimated 40% covered by tests and the trouble ticket count dropped by 40% as well. We don't consider this a coincidence.

But the customers had noted a different effect. By trapping bugs before they reached the production environment there were far fewer emergency production releases. Previously, it was not uncommon to release to production a couple times a day for a couple days due to bugs that

required an immediate fix. The acceptance tests improved the quality of the system to a point where a production release was rarely re-released because an emergency fix was required.

The customers experienced this as a system with a far greater feeling of stability. They had more confidence in the system and us. The customers also noticed that with fewer releases there was a large drop in spurious bugs often caused by quick and dirty fixes. ☺ ☺

Don Wells

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Lessons Learned](#) | [Next Lesson](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



Acceptance Tests Need to be Easy to Update

Lessons Learned

The VCAPS project used the unit test framework for both unit tests and acceptance tests. At the time it seemed like the simplest thing that could possibly work. But in order to use the more generic unit test framework we had to hand code large amounts of data. That is, our database was created with := (assignment) statements. The result was that it required someone familiar with the acceptance test data to update it or add new data and tests.

I had noticed that it was difficult. But continued on anyway. That was my mistake, we needed to create a tool to maintain our tests. A custom made, domain specific tool that would aid us and our customers in creating acceptance tests. My excuse was that there was never enough time to create it. We probably spent even more time creating test data the hard way.

Now a couple years later VCAPS has entered the maintenance part of it's life cycle. The acceptance tests are still being used to find integration bugs, but no new tests are being added. What the maintenance people are finding is that it is nearly impossible to add new acceptance tests and hard to maintain the ones that exist without some sort of tool designed to make all that data easy to handle.

The best thing we could have done for our project was to just go ahead and create the tool early on. In retrospect it would have been far simpler to have such a tool. Creating the tool would have helped us and saved us time over the entire life cycle of the project. ☹ ☹ ☹

Don Wells

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Lessons Learned](#) | [Acceptance Tests](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.

XP Unit Tests

Extreme Programming

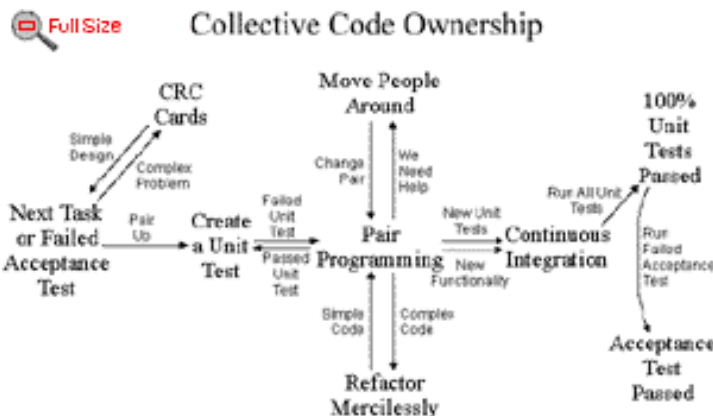
Lessons Learned

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.

[Continued on page 2](#)

[ExtremeProgramming.org home](#) | [XP Rules](#) | [Download a Unit Test Framework](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



Unit Tests Are Worth the Investment

*Lessons
Learned*

Over a year ago we stumbled upon the XP web page and applied two of the concepts espoused at that time: pair programming and automated unit testing. We were drawn initially to try the automated unit testing. I would like to report that the unit testing effort has returned huge dividends. We implemented Kent Beck's Simple Smalltalk Testing: With Patterns approach and implemented a "killer" user interface to organize, initiate tests and collect results. The initial effort

and learning curve was well worth the time investment. We have a dramatic increase in quality and an associated decrease in the overall test, find & fix intervals. ☺ ☺

Mark Bradac

[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [XP Lessons Learned](#) | [Next Lesson](#) | [Email the webmaster](#)

Copyright 1999 Mark Bradac. Logos Copyright 1999 Don Wells all rights reserved.



Unit Tests Could Have Saved Us Time

Lessons Learned

During the VCAPS project we introduced automated unit testing. It became the rule to include unit tests with any new functionality or modification, but in one instance a manager complained that a deadline would be missed if testing were required. In this case special dispensation was granted. After releasing the code about 32 developer hours were spent tracking down a problem that cost untold

customer delays. It was discovered that the errant development team had based their work on obsolete versions. For our own sake the unit tests were undertaken after the problem was found. Adapting existing unit tests to the new code took one hour. ☹ ☹

Don Wells

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Lessons Learned](#) | [Unit Tests](#) | [Email the webmaster](#)

Copyright 1998, 1999 Don Wells all rights reserved.



Unit Testing Framework

The most common misconception about unit testing frameworks is that they are only testing tools. They are development tools same as your editor and compiler. Don't keep this powerful development tool in reserve until the last month of the project, use it through out. Your unit testing framework can help you formalize requirements, clarify architecture, write code, debug code, integrate code, release, optimize, and of course test.

Unit testing frameworks are not hard to create from scratch. It is worth the effort to create your own because you will understand it better and be able to tailor it to your own needs. A simple change to the unit testing framework can often save you large amounts of development time. But to realize this savings you must feel comfortable and confident about extending your framework.

Most languages already have a unit testing framework available for download from XProgramming.com. Use this free version as a starting point. See how it works, then create your own. The team must claim ownership of the unit testing framework and be able to change any part of it. JUnit is quickly becoming the standard for unit testing in Java. At the very least refactor JUnit to make it your own and understand how to extend it. ☺ ☺ ☺



ExtremeProgramming.org home | [XP Rules](#) | [A Bug is Found](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



Code the Unit Test First

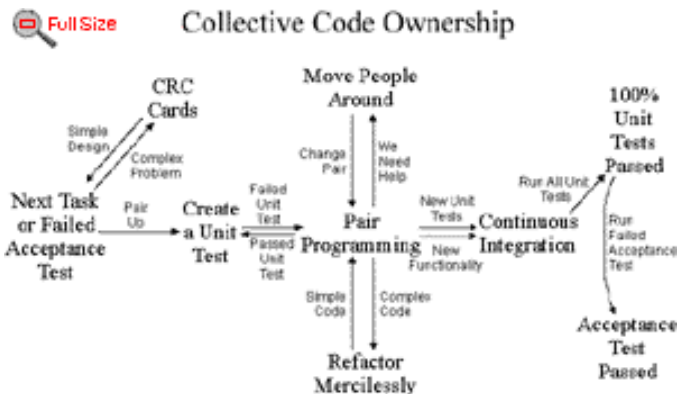
*Lessons
Learned*

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. ☺ ☹



[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules](#) | [Pair Programming](#) | [Email the webmaster](#)

Copyright 2000 Don Wells all rights reserved.



Testing First Makes the Code Testable

Lessons Learned

Massimo Arnoldi and I were working on booking payments for a life insurance policy. The requirements and the code were difficult to grasp because of all the special cases- what if they pay too much, what if they pay late, what if they pay for two months at once, etc.

We spent half a day trying to come up with the tests and code for this situation, without success. Every time we got one test working, we discovered the need for three others.

Cheese and oregano pizza and a couple of espressos later, we decided to solve a simpler problem. Could we just test and code the part that decided whether booking a payment was possible at all? Sure- if the amount matched, we could book. Otherwise we had to ask for human intervention. We made a couple of tests and an object that passed them. Elapsed time, 15 minutes.

Next, could we correctly book a payment? Given that the first test passed correctly, we could assume that the source account had the correct balance. Testing and coding was trivial- one test, one object, one method, done. Elapsed time, 5 minutes.



We wouldn't have created two objects if we hadn't been coding test first. The second object would have been god awful complicated if we couldn't rely on the first test passing. Net result of test-first (once we pulled our heads out)- cleaner design, correct behavior, simple tests, and simple code.

I now try to apply this strategy to all of the difficult problems I encounter. "What are the pieces, the combination of which I will have confidence in assuming that the pieces all work?" I don't always find the pieces right away. Sometimes it's months or years. In the meantime I have lots of tests for my less-than-optimal design. When insight rears its ugly head, I have all the resources I need to retrofit it. ☺ ☺

Kent Beck

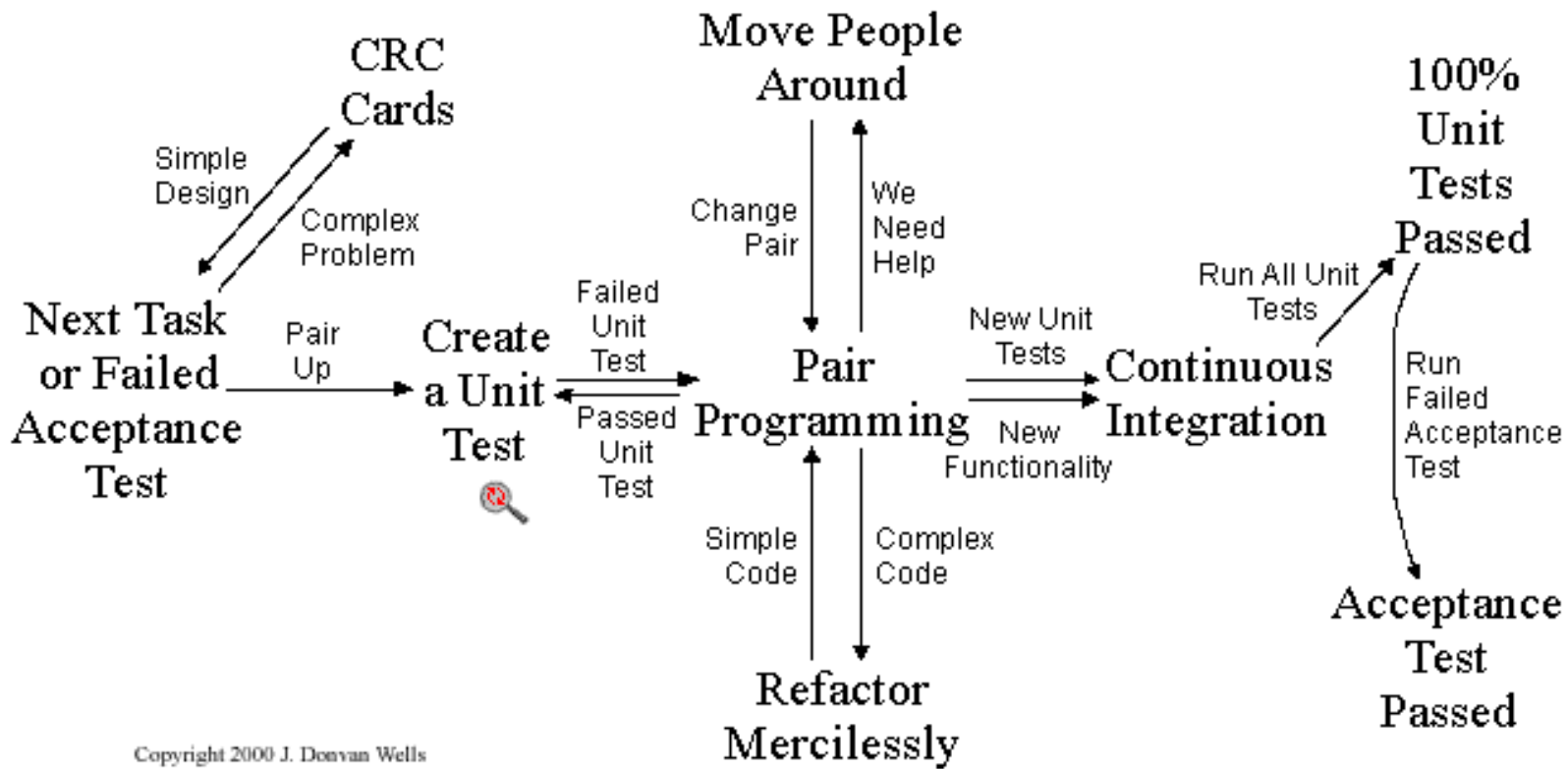
[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Lessons Learned](#) | [Test First](#) | [Email the webmaster](#)

Copyright (c) 2000, First Class Software, All rights reserved. Logos Copyright 1999 Don Wells all rights reserved.



Collective Code Ownership

Zoom Out



Copyright 2000 J. Donovan Wells

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [Zoom out to Development](#) | [Starting with XP](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved



Simplicity is the Key

*Lessons
Learned*

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. ☹ ☹



[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules](#) | [System Metaphor](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



A Simple Design is Easier to Maintain

Lessons Learned

We had to rebuild the VCAPS system. The original system was using an unsupported version of GemStone for a database. There was no way to upgrade to the new version directly because the system was too complex.

We created what we called the Replicator to move our data from the old system to the new. Months were invested in it's construction. But the Replicator had a great deal of flexibility and was built as a framework.

As the new system was being implemented we often had to change the Replicator. But it was so complex that making changes was hard and slowed us down. We had to have someone on the Replicator full time just to maintain it. When ever any team member needed a Replicator change they have to wait until the Replicator guy (a.k.a. Repli-Gator) could do it.

The only thing to do was get the team together to design a better solution. As we

worked through various ideas we came up with a couple good ones. We did spike solutions to see which would work the best.

We then wrote automated unit tests as we were creating the new Replicator so that we could refactor out complexity when ever possible. When we finished the new design in a few weeks and we had about 1/6th as much code.

This new system didn't have a framework like base. It just did what it needed to do and no more. We found it to be much easier to maintain and much easier to add just what we needed as we needed it. We found that we could collectively own it as well. Now anyone could quickly change it themselves as needed. ☺ ☺

Don Wells

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Lessons Learned](#) | [Next Lesson](#) | [Email the webmaster](#)

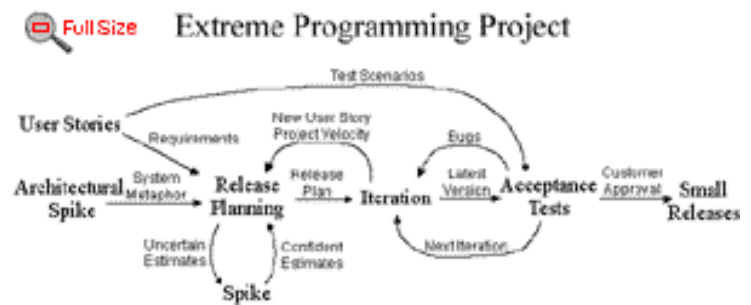
Copyright 1998, 1999 Don Wells all rights reserved.



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. 🧑‍💻 🧑‍💻



Wiki Wiki
The Portland
Pattern Repository

XPlorations

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules](#) | [No Early Functionality](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



Never Add Functionality Early.

*Lessons
Learned*

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. ☹ ☹



INCLUDE
THE [Wiki Wiki](#)
The Portland
Pattern Repository

XP
rogramming.com

[ExtremeProgramming.org home](#) | [XP Rules](#) | [Refactor Mercilessly](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.

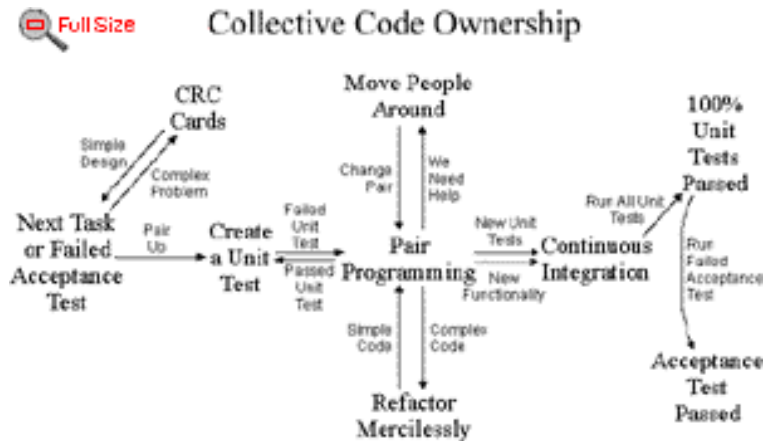


Refactor Mercilessly

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. 🦋



[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules](#) | [Customer Availability](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved. Refactoring graphic Copyright 1999 Addison Wesley Longman, Inc.

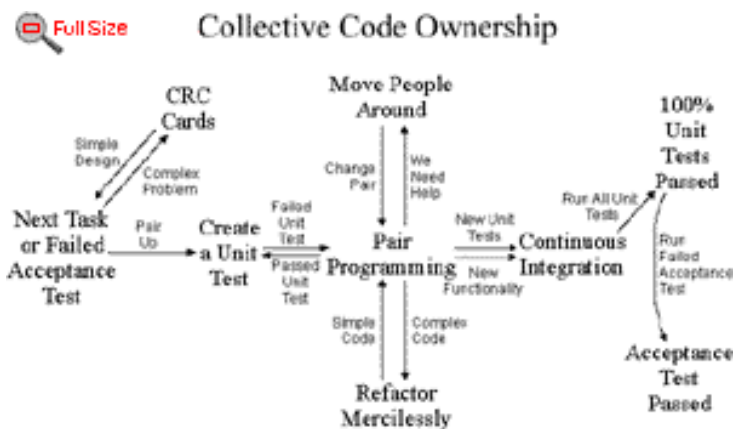


Move People Around

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. 🧑‍🤝‍🧑

Wiki Wiki
The Portland
Pattern Repository

XP
programming.com

[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [XP Rules](#) | [Stand Up Meeting](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



Iteration Planning

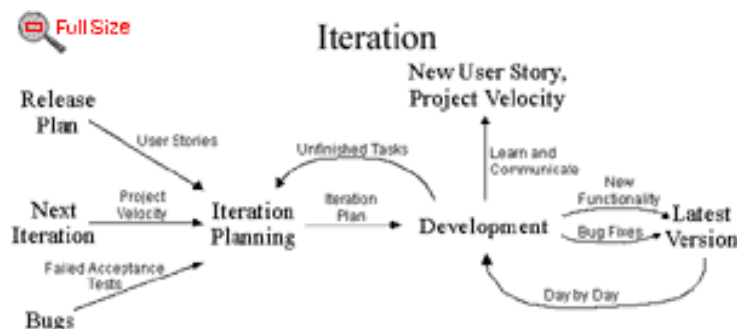
An iteration planning meeting is called at the beginning of each **iteration** to produce that iteration's plan of programming tasks. Each iteration is 1 to 3 weeks long. **User stories** are chosen for this iteration by the customer from the **release plan** in order of the most valuable to the customer 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. 🗓️ 📅



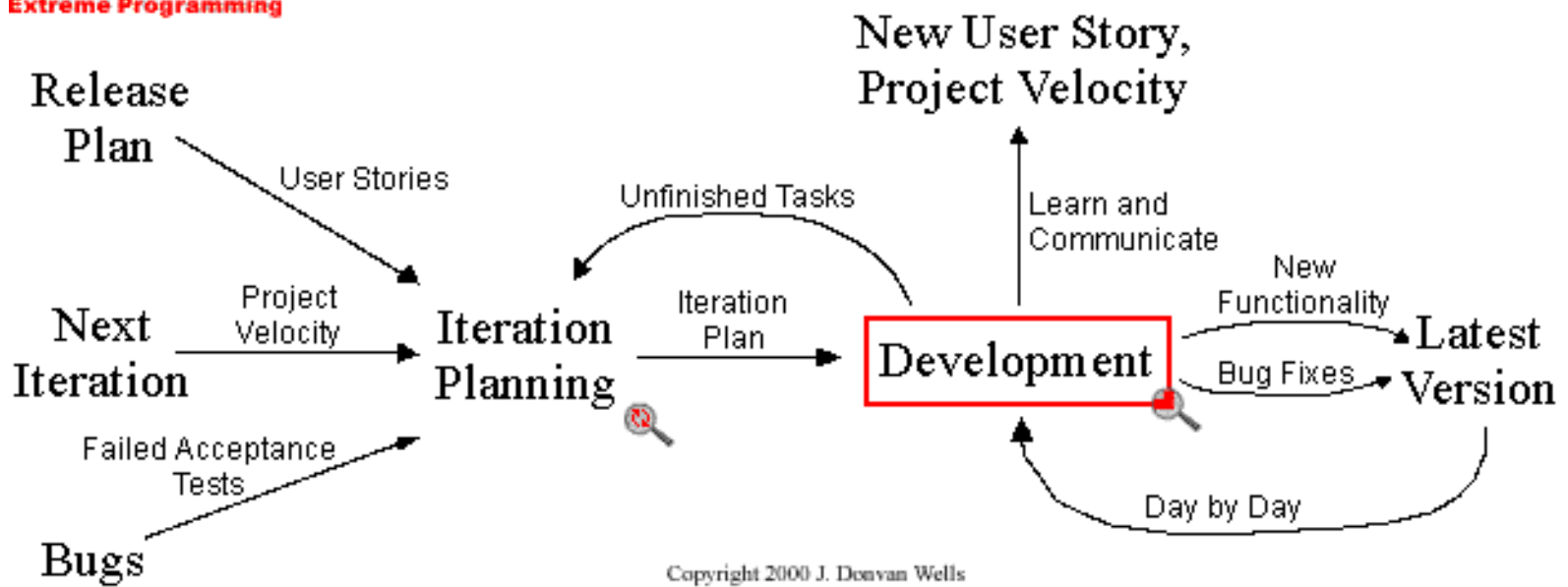
[ExtremeProgramming.org home](#) | [XP Rules](#) | [Move People Around](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



Iteration

Zoom Out



Copyright 2000 J. Doevan Wells

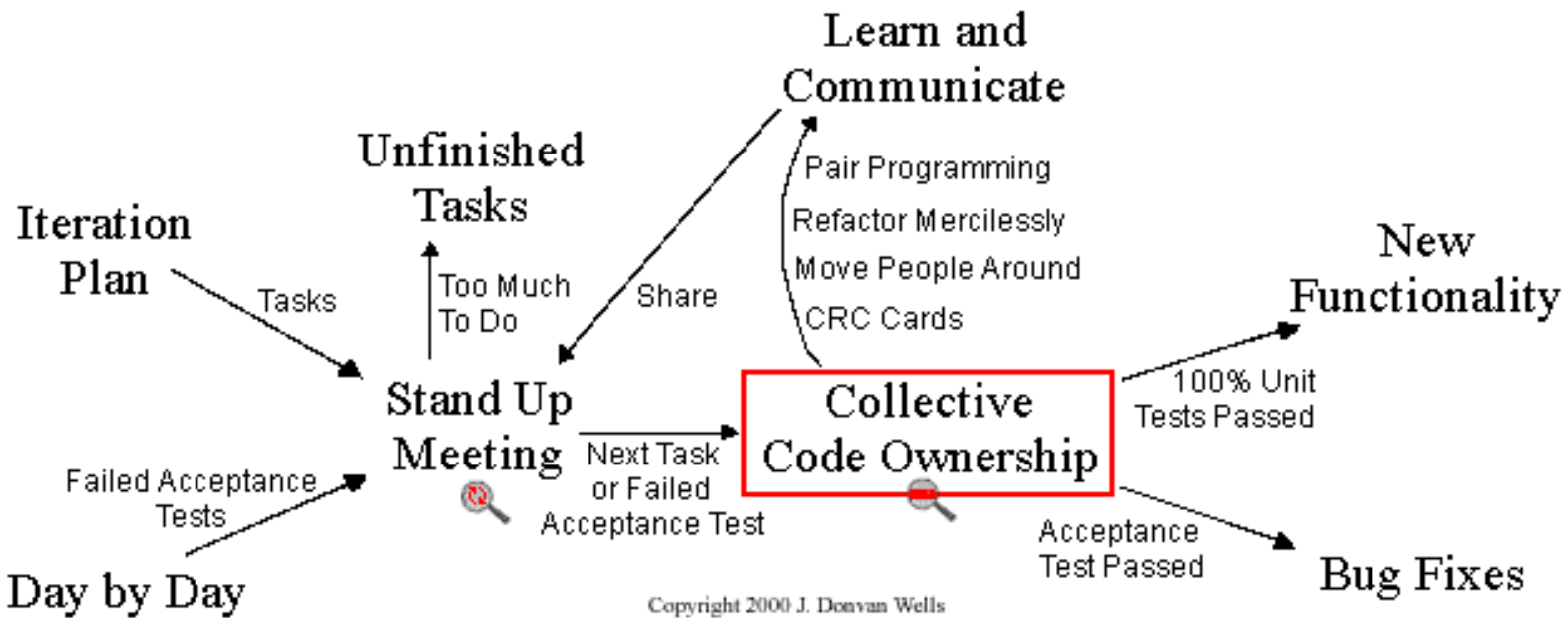
[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [Development](#) | [Project](#) | [Starting XP](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved



Development

Zoom Out



Copyright 2000 J. Donovan Wells

[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [Collective Coding](#) | [Iteration](#) | [Starting with XP](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved

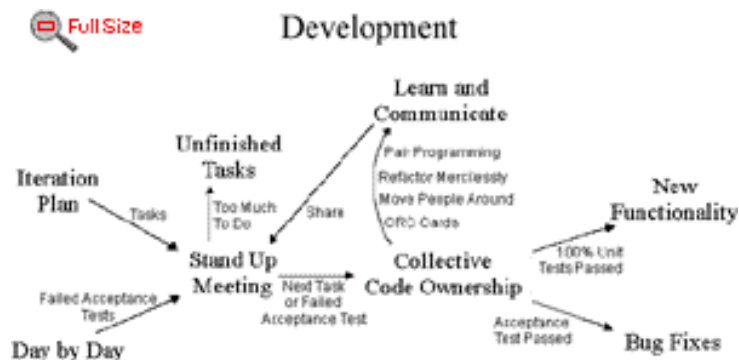


Daily Stand Up Meeting

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. 📄 📄

 [Wiki Wiki](#)
The Portland
Pattern Repository

[ExtremeProgramming.org home](#) | [XP Rules](#) | [Fix XP When It Breaks](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.

XP Pair Programming

Extreme Programming

Lessons Learned

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. ☹️ ☹️



[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [XP Rules](#) | [Sequential Release](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved. Photo of Pair Programming at DaimlerChrysler.



The Whole is Greater Than the Parts

*Lessons
Learned*

Over a year ago we applied [pair programming](#) and discovered that pairing improved the quality of design and implementation without a sacrifice in productivity. In fact, from our experience, the productivity of the whole proved to be greater than that of the individual parts. (Our only "fly in the ointment" was that

there are some people who don't work well together, and some individuals who are too individualistic to make pair programming work -- too bad!) ☹ ☹

Mark Bradac

[ExtremeProgramming.org home](#) | [XP Lessons Learned](#) | [Next Lesson](#) | [Email the webmaster](#)

Copyright 1999 Mark Bradac. Logos Copyright 1999 Don Wells all rights reserved.



Some Pair Programming Rules of Thumb

*Lessons
Learned*

We learned some fundamental rules for the pair programming construct itself during the VCAPS project:

- Never pair two people together who are brand new to programming in pairs (always one old-timer with a newcomer).
- When a pair takes the option of working separately (but with joint responsibility), they aren't really pair programming.
- If both people can't see what is happening on the monitor, they aren't really pair programming.

- Everyone works in a pair (no lone rangers allowed)
- People have to trust each other, and it may take time to build trust among everyone on the team

I think most people who have done pair programming unsuccessfully (and then successfully) have learned these sorts of rules of thumb. ☺ ☺

Jeanine De Guzman

[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [XP Lessons Learned](#) | [Next Lesson](#) | [Email the webmaster](#)

Copyright 1998 Jeanine De Guzman. Logos Copyright 1999 Don Wells all rights reserved.



Pair Programming Reins in the Cowboy Coders

*Lessons
Learned*

Extreme Programming (XP) is not just a codification of bad programming practices in bad environments. One key point is the constraining (read improving) effects that **pair programming** places on cowboy programmers with a JANGIT (Just Add New Garbage Ignoring Tests) mentality. Even if you tend to have cowboy coders in your group, pairing them up with an experienced extreme programmer virtually guarantees the code will be written properly and the necessary **testing** will also be put in place.

Even if two cowboys are paired together the resulting code will still be better than if either went off on their own. XP provides a certain set of checks and balances between a pair since no

two people think alike. When combined with the other **rules** of XP what results is the elimination of bad programming practices and bad environments.

Pairing also gets rid of any problems resulting from mediocre programmers writing poor code and documentation.

I'm speaking from the point of personal experience in that I didn't think pair programming was going to help me improve my code and make me faster. So I went in quite skeptical. As you can tell, my views have changed just a bit. ☺ ☺

Tom Kubit

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Lessons Learned](#) | [Next Lesson](#) | [Email the webmaster](#)

Copyright 1998 Tom Kubit. Logos Copyright 1999 Don Wells all rights reserved.



Pair Programming Reduces Indecision

Lessons Learned

One of the most powerful strengths of Extreme Programming (XP) is pair programming. We have found that the interaction that happens within a pair team is vastly more than the sum of the parts would produce. Even the most experienced coder finds that they are producing a design/code that is at a much higher level than they could do alone. We have found on the

VCAPS project that there is typically less indecision in a pair team, ideas are thrown back and forth, and the solution space is quickly narrowed as advantages and disadvantages are discussed within a pair team. ' : ']

Kevin Bradtke

[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [XP Lessons Learned](#) | [Next Lesson](#) | [Email the webmaster](#)

Copyright 1998 Kevin Bradtke. Logos Copyright 1999 Don Wells all rights reserved.



Pair Programming is Hard Work

*Lessons
Learned*

I've been experimenting with [pair programming](#) at my current position. I am fortunate to be working next to a programmer that I mesh with very well, and who is crazy enough to let me talk him into trying some XP practices. We've been spending about 50% of our time pair programming, and the results have been very encouraging. My observations are:

- It's somewhat harder work, because my partner requires me to justify everything that is unclear or disagreeable.
- It doesn't seem to me like we're being more productive at the time -- time does not go faster when you are always communicating, especially during vigorous arguments. But when we get done, we find that we've written a great deal of code that we both understand and like.
- Sometimes we both want to drive. We haven't had to flip quarters yet to decide, so I guess it hasn't been a problem.

- The shotgun partner can handle interruptions while the driver keeps coding. This is a big productivity gain.
- When I win an argument, it feels merely ok. When I lose one because my idea was not the best, it feels "great", because I know my partner just saved us from writing poor code.
- Sometimes we like to fork, with one going off to do a web search or write a quick test program, but we usually join before too long. Neither of us wants to let the other one get away with any unchallenged code!

Our team lead noticed us working as a pair, and officially paired us for the next project. I couldn't have hoped for better -- now we "have" to work as a pair (shucky darn). 'z' z

Wayne Conrad

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Lessons Learned](#) | [Next Lesson](#) | [Email the webmaster](#)

Copyright 1999 Wayne Conrad. Logos Copyright 1999 Don Wells all rights reserved.



Pair Programming Experiment

*Lessons
Learned*

We ran an experiment with 42 seniors at the University of Utah. Fourteen of them worked alone. The rest worked in pairs doing **pair-programming**. All students completed the same assignments.

Pair-programming definitely does not cost twice as much! In their first assignment (I call the "jelling-assignment") the pairs spent 60% more programmer hours than the individuals. In the second assignment, they had gotten used to this pair-programming thing. The pairs spent only 20% more total time than the individuals. By the third assignment, the pairs spent only 10% more time - so if an individual spent 10 hours on the assignment, the pair worked together for 5 hours and 15 minutes.

In all cases, the pairs passed about 15% more of the post-development test cases.



And, over 90% say they enjoy programming more and they feel more confident in their work when pairing.

As a posttest to the experiment, all students worked individually to complete one assignment. One student said of going back to solo programming, "Without my partner, I feel like I lost half my brain." ☹ ☹

Laurie Williams
North Carolina State University

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Lessons Learned](#) | [Next Lesson](#) | [Email the webmaster](#)

Copyright 1999 Laurie Williams. Logos Copyright 1999 Don Wells all rights reserved.



Code Reviews Considered Hurtful

Lessons Learned

It is all too easy to become emotionally invested in the code you've been creating. A formal group review process creates a stressful situation and fosters emotional reactions for both the developer and the reviewer.

There is just no good way for several people to examine someone's code and suggest changes without it seeming like personal criticism. There is a feeling of being attacked from all sides and it isn't always clear what changes need to be made to satisfy the reviewers once the review is complete.

I recently witnessed a developer feeling personally persecuted by the review process. I was part of the review team and the code was from an inexperienced developer. After only two reviews, this developer asked management to be excused from any more reviews.

We reassured her that it was not a personal attack, but her enthusiasm for the project was gone. She felt completely demoralized. We all wanted to help, but no one was available to sit with her and guide her. I dreaded the reviews almost as much as she did because I could feel stress levels rising, and knew



time was being wasted.

Pair programming changes the environment from criticism and competition to learning and cooperation. Programming partners must explain to each other what they are doing; one teaches and one learns, then the roles reverse. The learner is encouraged to participate with new ideas or new twists on old ideas, gaining confidence all the while.

There is a discussion between two developers instead of a short lecture from a superior. There are no criticisms thinly veiled as suggestions, but mutual discovery and agreement. The resulting code is always better because it has to pass two pairs of eyes. You end the day with a feeling of accomplishment instead of animosity.



Tom Kubit

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Lessons Learned](#) | [Pair Programming](#) | [Email the webmaster](#)

Copyright 1999 Tom Kubit. Logos Copyright 1999 Don Wells all rights reserved.

Photo of pair programming at Daimler Chrysler



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.

[Continued on page 2](#) 

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules](#) | [Integrate Often](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved.



Sequential Integration

[Continued from page 1](#)

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 without 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. ☺ ☺

[ExtremeProgramming.org home](#) | [XP Rules](#) | [Integrate Often](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved.



Dedicated Release Computer

A single computer dedicated to [sequential releases](#) works really well if the development team is co-located. This computer acts as a physical token to control releasing. There is also a place for developers to go to see the final word on what system configuration is current. Developers have a source for final arbitration on integration problems. The computer allows developers to see who is releasing and when. When the release computer is occupied no other changes can be released, stability is ensured.

The latest combined unit test suite can be run before releasing. Because a single computer is used the test suite is always up to date. If the unit

tests run at 100% the changes are released, if they fail the changes are debugged or backed out and debugged at the developers workstation. ↵ ↵



[ExtremeProgramming.org home](#) | [XP Rules](#) | [Integrate Often](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved. The featured insect is actually a moth.

XP Integrate Often

Extreme Programming

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.

Lessons Learned



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. ☺ ☹

Wiki Wiki
The Portland
Pattern Repository

XP
programming.com

Wiki Wiki
The Portland
Pattern Repository

ExtremeProgramming.org home | [XP Rules](#) | [Collective Code Ownership](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved.



Code Integration Can be Reduced to Seconds

Lessons Learned

The VCAPS Project has had considerable success with the practice of *Continuous Integration*. [Continuous Integration is an attitude towards integration where you integrate as often as possible.] A couple of key factors we found were required to be successful:

- **Unit tests** are an absolute must-have. No one releases unless they run at 100%!! If they don't, your changes are not going in.
- Everyone has **collective code ownership** for all of the code. That is, each time a **programming pair** touches the code, it should get improved, refactored, and simplified
- There must be a **release station**. No one releases from their own workstation.

Integrate Often!! The longer you wait to integrate, the more pain you will create.

Continuous integration (as a slogan) is done in our environment (Visual Works Smalltalk and GemStone Smalltalk) in a matter of minutes, usually seconds! Developers integrate multiple times per day in practice. If it takes too long to integrate your changes, then the unit of work being released was too big! The programming pair is responsible for creating their own integration hell, or heaven, whichever they prefer... ☹ ☹

Jeanine De Guzman

[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [XP Lessons Learned](#) | [Integrate Often](#) | [Email the webmaster](#)

Copyright 1998 Jeanine De Guzman. Logos Copyright 1999 Don Wells all rights reserved.



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. ☺ ☹



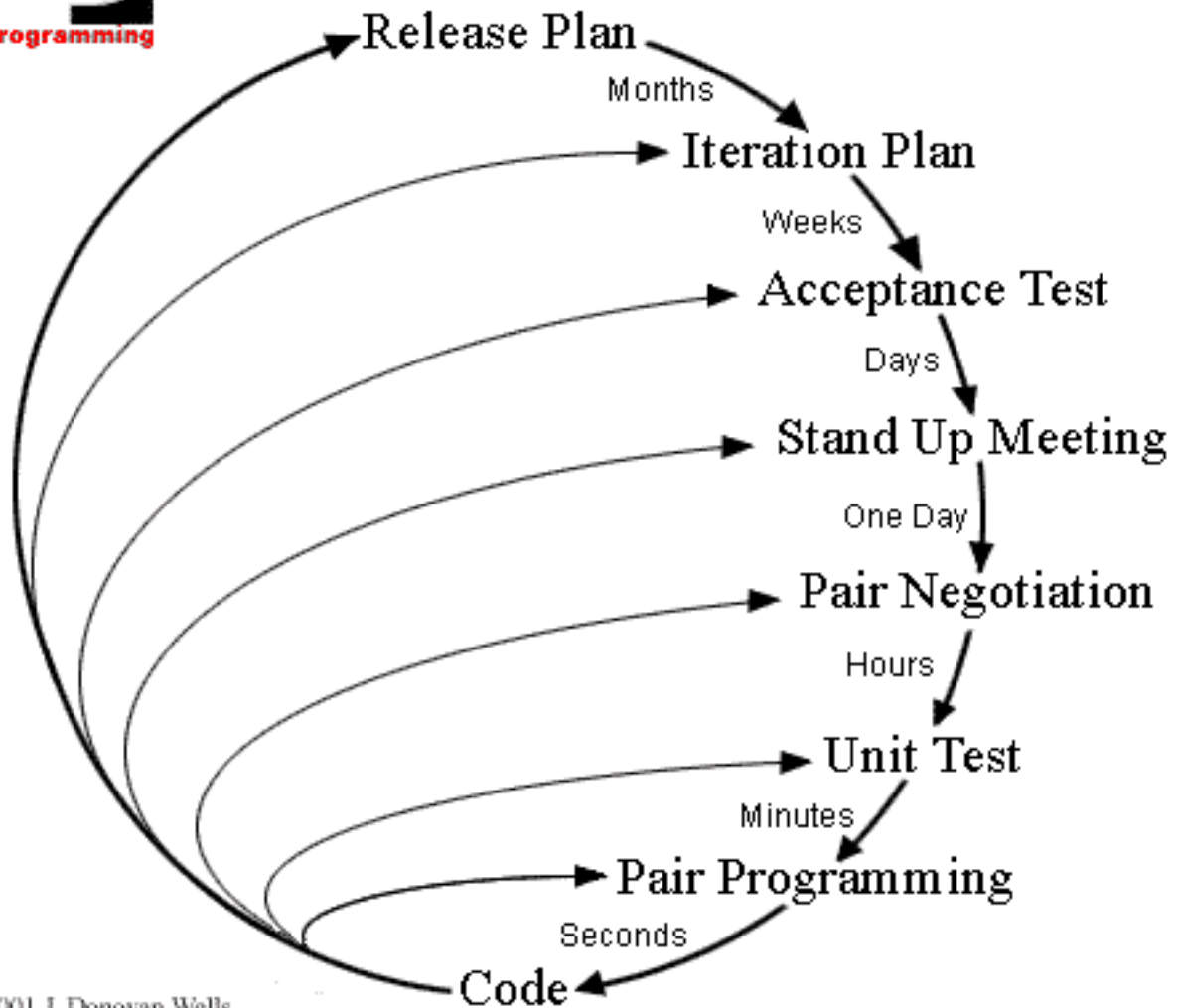
[ExtremeProgramming.org home](#) | [XP Rules](#) | [Simplicity](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



Planning/Feedback Loops

Zoom Out



Copyright 2001 J. Donovan Wells.

[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [Zoom out to Project](#) | [Starting with XP](#) | [Email the webmaster](#)

Copyright 2000 Don Wells all rights reserved



How do I start this XP thing?

The most obvious way to start extreme programming (XP) is with a new project. Start out collecting [user stories](#) and conducting [spike solutions](#) for things that seem risky. Spend only a few weeks doing this. Then schedule a release planning meeting. Invite customers, developers, and managers to create a schedule that everyone agrees on. Begin your iterative development with an [iteration planning](#) meeting. Now you're started.

Usually projects come looking for a new methodology like XP only after the project is in trouble. In this case the best way to start XP is to take a good long look at your current software methodology and figure out what is slowing you down. Add XP to this problem first.

For example, if you find that 25% of the way through your development process your requirements specification becomes completely

useless, then get together with your customers and write [user stories instead](#).

If you are having a chronic problem with changing requirements causing you to frequently recreate your schedule, then try a simpler and easier release planning meeting every few iterations. (You will need user stories first though.) Try an [iterative style of development](#) and the [just in time style of planning](#) of programming tasks.

If your biggest problem is the number of bugs in production, then try automated [acceptance tests](#). Use this test suite for regression and validation testing.

If your biggest problem is integration bugs then try automated [unit tests](#). Require all unit tests to pass (100%) before any new code is released into the code repository.

[Continued on page 2](#) 

[ExtremeProgramming.org home](#) | [Where can I get more information?](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



How do I start this XP thing?

Continued from page 1

If one or two developers have become bottlenecks because they own the core classes in the system and must make all the changes, then try **collective code ownership**. (You will also need unit tests.) Let everyone make changes to the core classes whenever they need to.

You could continue this way until no problems are left. Then just add the remaining practices as you can. The first practice you add will seem easy. You are solving a large problem with a little extra effort. The second might seem easy too. But at some point between having a few XP rules and all of the XP rules it will take some persistence to make it work. Your problems will have been solved and your project is under control. It might seem good to abandon the new methodology and go back to what is familiar and comfortable, but continuing does pay off in the end. Your development team will become much

more efficient than you thought possible. At some point you will find that the XP rules no longer seem like rules at all. There is a synergy between the rules that is hard to understand until you have been fully immersed.

This up hill climb is especially true with pair programming, but the pay off of this technique is very large. Also, unit tests will take time to collect, but unit tests are the foundation for many of the other XP practices so the pay off is very great.

XP projects are not quiet; there always seems to be someone talking about problems and solutions. People move about, asking each other questions and trading partners for programming. People spontaneously meet to solve tough problems, then disperse again. Encourage this interaction, provide a meeting area and set up workspaces such that two people can easily work together. The entire work area should be open space to encourage team communication. ↵ ↵

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [Where can I get more information?](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



More Information



Class Room Training



Object Mentor Inc. offers a **5 day class** featuring Kent Beck, Ron Jeffries, and Robert Martin in Vernon Hills, IL.



Industrial Logic teaches an **Extreme Programming Workshop** and a **Testing and Refactoring Workshop**.



Advanced Technologies Integration offers a three day class in Edina, MN.

Group Discussion



ObjectMentor Inc. has set up an **XP mailing list** for us at yahoo.com.

There is now a usenet news group **comp. software. extreme-programming**

Web Sites



The Portland Pattern Repository hosted by Ward Cunningham. A lively free wheeling discussion group. Ask questions and get answers. All points of view are well represented here. Information and experiences using XP from several projects.



XPprogramming.com hosted by Ron Jeffries. Articles by Ron and others. Lessons learned from the C3 project, Q&A and more.



XP Developer a wiki style discussion group. The discussions are about how to actually do XP. Find out about the Extreme Tuesday Club which meets in London.



Conferences

The Fourth International Conference on **eXtreme Programming** and Agile Processes in Software Engineering, will be held the week of May 26, 2003, in Genova, Italy.



The **XP Universe** and **Agile Universe** conferences will be co-located in New Orleans, August 10-13, 2003.

Books



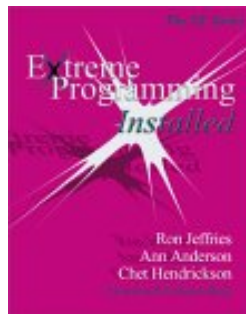
Extreme Programming Explained: Embrace Change.

Kent Beck explains the concepts and philosophy behind extreme programming. This book teaches what and why but not how.



Refactoring Improving the Design of Existing Code.

Martin Fowler writes the first authoritative volume on refactoring. Presented as patterns. There are plenty of examples in Java. This book teaches you how to refactor and why.

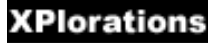


Extreme Programming Installed.

By Ron Jeffries, Chet Hendrickson, and Ann Anderson. This book covers specific XP practices in finer detail than Extreme Programming Explained. This book teaches how to program XP style.



Laurie Williams is researching pair programming at the University of Utah. She has conducted a survey of professional programmers who have experience working in pairs.



Willam Wake has written many good articles targeted at understanding specific XP topics.



Martin Fowler has created a web site for information about refactoring including updates to his book.



Jim Highsmith talks about XP in an article at e-business application delivery.



RoleModel Software has an eXtreme Programming Software Studio(tm) and an apprenticeship program.



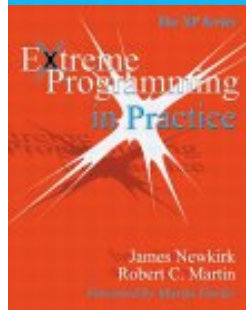
Yonat Sharon has a general OO site with some information about XP.



Planning Extreme Programming by Kent Beck, and Martin Fowler. This book presents the latest thoughts on how to plan software in a rapid delivery environment. This book teaches how to run an XP project.



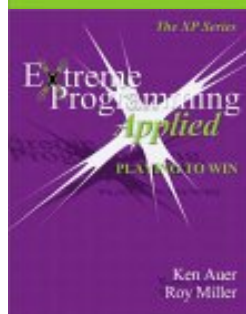
Extreme Programming Examined by Giancarlo Succi and Michele Marchesi. Papers presented at XP2000. A well rounded set of papers covers most topics.



Extreme Programming in Practice by Robert C. Martin, James W. Newkirk. A real project which used XP is described in gory detail.



Extreme Programming Explored by William C. Wake. Based on the popular XPlorations website. Specific subjects are explored in detail.



Extreme Programming Applied: Playing to Win by Ken Auer and Roy Miller. Experiences from pioneers in applying XP. To be published in September.

People



An excellent source of information are the people who have already been learning about XP. ☺ ☻

There is the **Michigan
eXtreme Programming
Enthusiasts.**

There is **XP Denver** for
the Colorado Front Range.

There is a **Hamburg XP
user's group.**

**XP user group Stuttgart
(XPUGS).**

The **eXtreme Tuesday
Club** meets in London (UK).

There is an **XP User's
Group** starting in Phoenix,
Arizona

[ExtremeProgramming.org home](#) | [Where can I find XP people?](#) | [Email the webmaster](#)

Exclusive of cover art and logos as noted, Copyright 1999 by Don Wells. Pattern logo Copyright 1997 Cunningham and Cunningham Inc. Red XP logo Copyright 1998 Ronald E. Jeffries. XPDeveloper logo Copyright 1999 XPDeveloper.com. Role Model logo copyright 1998 Rolemodel Software. eXtreme Programming Software Studio is a trade mark of Rolemodel Software. U of Utah logo copyright by the University of Utah. Refactoring graphic Copyright 1999 Addison Wesley Longman, Inc. Amazon.com pays a small fee for the sale of books through this website, all other companies do not pay for advertising and are listed here because I believe they have a good product. Please report any negative experiences to the webmaster.

People Are Your Most Valuable Resource.

Extreme Programming

There used to be a list of people and companies on this page. The spread of Extreme Programming (XP) has been so quick that there are now far too many projects using XP and people who know how to use XP for a listing. Please consider instead the Portland Pattern repository and the Groups.Yahoo.com list to find XP work or XP people. ☺ ☺



[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [Email the webmaster](mailto:webmaster@extremeprogramming.org)

Copyright 2000 Don Wells all rights reserved.



The October MXPE meeting has been scheduled. The meeting will be held on Thursday, October 9th from 5:30 - 7:30 pm. The meeting will be held at the DTE Energy building in downtown Ann Arbor. The address is 425 S. Main Street, Ann Arbor, 48104.

Unfortunately, due to scheduling conflicts, Henry Beitz, our scheduled presenter for the last meeting that was pre-empted by the blackout, will be unable to give his presentation this time around. Rest assured, we will find another time that work for a future meeting and invite Henry back.

This meeting will consist of two sessions. The first half will be a progress report from the Agile group at DTE. This should be a real treat to see what progress the team has made since our last meeting there one year ago. They will discuss the difficulties they have faced and how they have had to adapt in order to thrive in a large corporate setting. The session will also be open to some Q&A.

The second half of the meeting will be the free-for-all session where you get to bring up the issues and challenges you face in your work related to implementing XP or, for anyone just wanting to understand more about XP, to ask any pressing questions you might have about how any of this stuff can actually work. In the past, this meeting format has provided for lively discussions and a good learning experience for all involved. I'm sure the DTE guys will give us some interesting topics to cover in more depth, too.

If you are planning on attending, please send a confirmation e-mail to the group account at mxpe@extremeprogramming.org. Include the number of people planning on attending in the confirmation. If you have any questions, please contact any of the MXPE officers. Finally, if you have not yet submitted your membership dues, and since it's getting near the end of the year, talk to one of the officers at the meeting. Checks made out to MXPE are the preferred method of payment. Dues amounts and the mailing address can be found on the group web site (link below). Looking forward to seeing everyone there. ☺ ☺

MXPE yearly dues schedule is as follows; professional member \$20, corporate member \$100 (6 representatives), student member \$5. Single meeting tickets are \$5 or \$2 for students. Membership will be for the calendar year effective 1/1 - 12/31. New members pay for the current calendar year unless otherwise stipulated.

Membership dues can be paid by mail. Please make your check payable to "MXPE." If you need a receipt sent back to you, please be sure to include a return address with your payment. Payments can be sent to the following address:

MXPE Treasurer
2041 Miller Rd
Metamora, MI 48455-9222

Payments can also be made at the meeting. Please show up a few minutes early if paying on site. Payment by check is preferred.



MXPE Officers:

Tom Kubit, President, tom@extremeprogramming.org

Jason Rogers, Vice President, jacaetevha@fast-mail.org

Dave Bryant, Treasurer, dkbryant@urbanscience.com

Don Wells, VP Technology, don@extremeprogramming.org


Subscribe to the MXPE mailing list

[ExtremeProgramming.org home](#) | [Back to Top](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.

Coding Standards

Extreme Programming

Code must be formatted to agreed coding standards. Coding standards keep the code consistent and easy for the entire team to read and refactor. Smalltalk projects can use [Smalltalk Best Practice Patterns](#) as a coding standard. 



[ExtremeProgramming.org home](#) | [XP Rules](#) | [Code The Unit Test First](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



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. ☹ ☹ ☹

*Lessons
Learned*



[ExtremeProgramming.org home](#) | [XP Rules](#) | [No Overtime](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved.



Optimize Last Because it May Not be as Slow as You Think

*Lessons
Learned*

We needed to add some new functionality to the VCAPS project. Based on some other code in the system that ran 8 hours this new section would run in no less than 16 hours even after we squeezed ever little bit of performance out of it. We added the new functionality as simply as possible. We ignored concerns of speed in favor of concerns of code clarity and maintainability. Make it run, make it right, make it fast.

When we had finished the code we fired it up on a Friday afternoon in anticipation of several days of run time since we had not done any

optimizing. We were shocked when it came back in 1 hour. By keeping our design simple and understandable we had managed to avoid several "penny wise, pound foolish" types of optimization that other sections of code had fallen prey to.

At this point we could have considered optimization, but because it was so much faster than any other portion of the system we didn't need to do any optimization at all. ☺ ☺

Don Wells

[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [XP Lessons Learned](#) | [Optimize Last](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.

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. ☹️



[ExtremeProgramming.org home](#) | [XP Rules](#) | [Unit Tests](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved. Photo of the vampire from the movie Nosferatu.



You Aren't Going to Need It.

*Lessons
Learned*

During the initial requirements and design sessions of my current project, we continually had people wanting to push a lot of "future" requirements into the first phase. Our answer was always the same. "We'll write a card for it so we don't forget about it, but we won't put it into the design until the time comes when we need it."

This strategy saved us a tremendous amount of grief, especially with regard to one of the fundamental building blocks of the system. We were designing a model to be used for defining the possible combinations of product offerings.

The catch was that in about 6 months time a new corporate model was going to be released. We got numerous requests to try to predict and model the inevitable corporate direction. We resisted, and instead modeled the product definition piece of the system to reflect the current business practice, and only those portions of it that were relevant to our project.

We kept the design as simple as possible while still satisfying all project requirements. We practiced "you aren't gonna need it."

After six months passed the corporate model was postponed for another year. Meanwhile our project was considering a completely new and different model based on expanded requirements.

If we had tried to guess at a corporate model we would have needed months of rework to migrate to this new model. Because we kept our design very simple, we could easily evolve our model into the new model. We estimate it will only take a week or two of adjustments, which represents substantial savings. ☺ ☺

Tom Kubit

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Lessons Learned](#) | [Simplicity](#) | [Email the webmaster](#)

Copyright 1999 Tom Kubit. Logos Copyright 1999 Don Wells all rights reserved.



Design a Simulator for the Coffee Maker

Design is accomplished in three ways on an Extreme Programming (XP) project. There are CRC cards, refactoring, and pair programming. CRC cards can be considered a strategic level of design, pair programming is at the tactical level and refactoring serves both.

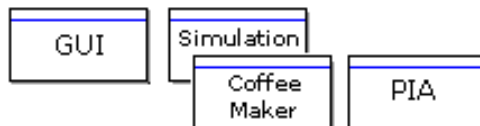
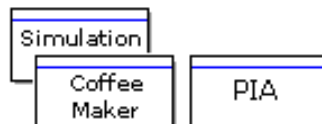
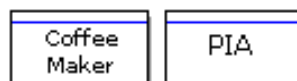
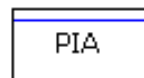
For this problem I would feel better with the team working together. So let's get out the cards!

Let's start out with our hardware's interface. We know we have a programmable interface adapter (PIA). We can set out a card to represent one object that will be our interface to the PIA.

Now let's add our coffee maker code as if it were a single object. It may or may not be, but right now we are designing the simulator. Let's put it right next to the PIA because it will interface with it.

Next we need to add the simulation object. This will be the object that loops simulating time and makes the PIA react as if it was being powered by real hardware. I am thinking that the simulation will hold onto the coffee maker code. I put the card slightly under the coffee maker code. The simulation will signal the coffee maker when ever the PIA has changed. The coffee maker can then react to the change.

The GUI will complete our design. It interacts with the simulation showing the state of our coffee maker and accepting user input like the brew button. But the team is not sure about this design. The simulation interacts with the PIA and needs to know the internals of the coffee maker code, which also interacts with the PIA. Wouldn't it be better if the PIA itself was the simulation? This cuts down on several interfaces.



Let's just start over. A good thing about designing with CRC cards is that we can sweep the desk clean as many times as we want and we have not wasted large amounts of time on creating diagrams for each alternative design.

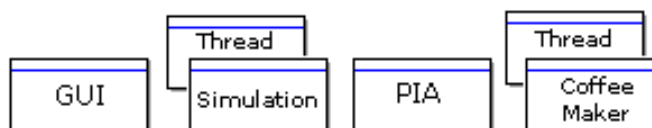
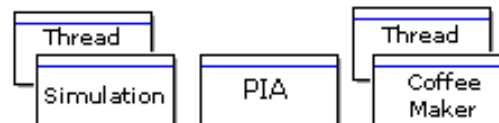
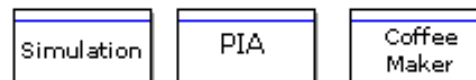
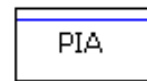
We are back to just the PIA. This is the one thing we must have.

To this we add the coffee maker object(s). This interfaces with the PIA. Now at this point we could just say that the PIA runs the simulation, but after some discussion the team doesn't like that. We want the PIA interface to be simple and generic, a closer representation of the hardware. Adding the simulation portion to it does not achieve that.

So let's put our simulation object back into play. But instead of the simulation owning the coffee maker, let's say that the simulation only interfaces with the PIA. The simulation has no internal knowledge of how the coffee maker works. We all like this better. It will even be a better simulation.

In order to make this work we will have to have separate threads for the coffee maker and the simulation. This adds a level of complexity, but removes about two levels of complexity in the exchange. We are reducing net complexity because we do not have to provide one coffee maker interface for the simulator and some other interface that will run the coffee maker on real hardware. On the other hand Java Threads are just not that complex. And as a bonus we can test the coffee maker exactly as it would run on real hardware. We agree this is better even though it is multithreaded.

Last, we add our GUI to interface to the simulation. This seems like a good design to start out with. Remember, we can change our minds when ever things become difficult to implement. We rely on this strength of XP so that we don't have to design out every detail of every class in advance. Now we need think of a system metaphor to fit this design. ☺ ☹



[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [A Spike Solution](#) | [System Metaphor](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Choosing a System Metaphor for the Simulator

The next step in creating our coffee maker simulator is the selection of a **system metaphor**. What we need is something that everyone can understand easily. From what we know about our simulator from our **CRC card session** we could consider a finite state automata, but this isn't exactly the case. We could think of it as an adapter sitting in between the PIA and the GUI. But this seems too abstract to me. We could think of it in terms of an alarm clock, but this isn't right either.

No, I think that the best metaphor for us is what **Kent Beck** calls the *naive* metaphor. This is the metaphor that uses the domain itself. We can use it here because our domain is actually very simple and generically understood already.

Let's start out writing code for the PIA. We need to see how well our design holds up under contact with actual code! 📄 📄

[ExtremeProgramming.org home](#) | [A Spike Solution](#) | [Code for the PIA](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



The XP Philosophy

In the early 1990s a man named Kent Beck was thinking about better ways to develop software. He had recently spent some time working with Ward Cunningham. Ward and Kent together had experienced an approach to software development that made every thing seem simple and more efficient. Kent contemplated on what made software simple to create and what made it difficult. In March of 1996 Kent started a project at DaimlerChrysler using new concepts in software development. The result was the Extreme Programming (XP) methodology.

What Kent came to realize is that there are four dimensions along which one can improve any software project. You need to improve communication. You need to seek simplicity.



You need to get feedback on how well you are doing. And you need to always proceed with courage. Communication, Simplicity, Feedback, and Courage are the four values sought out by XP programmers. ☺ ☹ ☺



[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [Back to What is XP?](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



What is Extreme Programming?

Extreme Programming (XP) is actually a deliberate and disciplined approach to software development. About six years old, it has already been proven at many companies of all different sizes and industries world wide.

XP is successful because it stresses customer satisfaction. The methodology is designed to deliver the software your customer needs when it is needed. XP empowers your developers to confidently respond to changing customer requirements, even late in the life cycle.

This methodology also emphasizes team work. Managers, customers, and developers are all part of a team dedicated to delivering quality software. XP implements a simple, yet effective way to enable groupware style development.

XP improves a software project in four essential ways; communication, simplicity,



feedback, and courage. XP programmers communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested. With this foundation XP programmers are able to courageously respond to changing requirements and technology.

XP is different. It is a lot like a jig saw puzzle. There are many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen. This is a significant departure from traditional software development methods and ushers in a change in the way we program. 🧩

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [A Change in the Way We Program](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved



A Change in the Way We Program.

Software which is engineered to be simple and elegant is no more valuable than software that is complex and hard to maintain. Can this really be true? Extreme Programming (XP) is based on the idea that this is not in fact true.

A typical project will spend about twenty times as much on people than on hardware. That means a project spending 2 million dollars on programmers per year will spend about 100 thousand dollars on computer equipment each year. Let's say that we are smart programmers and we find a way to save 20% of the hardware costs by some very clever programming tricks. It will make the source code harder to understand and maintain, but we are saving 20% or 20 thousand dollars per year, a big savings. Now what if instead we wrote our programs such that they were easy to understand and extend. We could expect to save no less than 10% of our people costs. That would come to 200 thousand dollars, a much bigger savings. This is certainly something your customers will notice.

Another important issue to customers are bugs. XP emphasizes not just testing, but testing well. Tests are automated and provide a safety net for programmers and customers alike. Tests are created before the code is written, while the code is written, and after the code is written. As



bugs are found new tests are added. A safety net of tight mesh is created. Bugs don't get through twice, and this is something the customers will notice.

Another thing your customers will notice is the attitude XP programmers have towards changing requirements. XP enables us to embrace change. Too often a customer will see a real opportunity for making a system useful after it has been delivered. XP short cuts this by getting customer feed back early while there is still time to change functionality or improve user acceptance. Your customers are definitely going to notice this.

Much of what went into XP was a re-evaluation of the way software was created. The quality of the source code is much more important than one might realize. Just because our customers can't see our source code doesn't mean we shouldn't put the effort into creating something we can be proud of. ☺ ☺

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [When Should We Use XP?](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



When should Extreme Programming be Used?

Extreme Programming (XP) was created in response to problem domains whose requirements change. Your customers may not have a firm idea of what the system should do. You may have a system whose functionality is expected to change every few months. In many software environments dynamically changing requirements is the only constant. This is when XP will succeed while other methodologies do not.

XP was also set up to address the problems of project risk. If your customers need a new system by a specific date the risk is high. If that system is a new challenge for your software group the risk is even greater. If that system is a new challenge to the entire software industry the risk is greater even still. The XP practices are set up to mitigate the risk and increase the likelihood of success.

XP is set up for small groups of programmers. Between 2 and 12, though larger projects of 30 have reported success. Your programmers can be ordinary, you don't need programmers with a Ph.D. to use XP. But you can not use XP on a project with a huge staff. We should note that on projects with dynamic requirements or high risk you may find that a small team of XP programmers will be more effective than a large team anyway.

XP requires an extended development team. The XP team includes not only the developers, but the managers and customers as well, all working together elbow to elbow. Asking questions, negotiating scope and schedules, and creating functional tests require more than just the developers be involved in producing the software.

Another requirement is testability. You must be able to create automated unit and functional tests. While some domains will be disqualified by this requirement, you may be surprised how many are not. You do need to apply a little testing ingenuity in some domains. You may need to change your system design to be easier to test. Just remember, where there is a will there is a way to test.

The last thing on the list is productivity. XP projects unanimously report greater programmer productivity when compared to other projects within the same corporate environment. But this was never a goal of the XP methodology. The real goal has always been to deliver the software that is needed when it is needed. If this is what is important to your project it may be time to try XP. ☺ ☺

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [Do We Need Another Methodology?](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved



Do We Need Yet Another Methodology?

When the Hubble Telescope was launched into space it had a mirror built to the wrong specifications. It wasn't until the telescope was fully deployed and in the hands of its customers that the problem was discovered. A satellite's customers can't try using it until after it's in outer space, but why do we do this with software?

What if you walked into an electronics lab and saw work benches covered in old appliances? Would you keep that huge old TV with the quaint little round picture tube because it still had a good power supply? Would you keep an old AM radio because it was useful as an RF generator? You wouldn't because your work bench would be so cluttered you couldn't get anything done. But this is exactly what we do with software.

The Zilwaukee Bridge over the Saginaw River needed to be built in a hurry. The bridge

was built up from both banks of the river to meet in the middle. But when they got to the middle one side was three feet higher than the other. When half of your project is on one side of a river and the other half is on the other you can't integrate your project until the very end, but why do we do this with software?

Extreme Programming (XP) was designed in response to these kinds of questions. XP was based on observations of what made computer programming faster and what made it slower. XP is an important new methodology for two reasons. First and foremost it is a re-examination of software development practices that have become standard operating procedures. And second, it is one of several new lightweight software methodologies created to reduce the cost of software. XP goes one step further and defines a process that is simple and enjoyable. ☺ ☺

[ExtremeProgramming.org home](http://ExtremeProgramming.org) | [What is a Lightweight Methodology?](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.

What is a Lightweight Methodology?

A software methodology is the set of rules and practices used to create computer programs. A heavyweight methodology has many rules, practices, and documents. It requires discipline and time to follow correctly. A lightweight methodology has only a few rules and practices or ones which are easy to follow.

In the late 1960s and early 1970s it was common practice for computer programmers to create software any way they could. Many programmers excelled at creating software too complex for anyone to understand. At that time it was a miracle if a program ran without any bugs. Making computers useful was considered a worthy quest and not unlike an adventure into the old west.

In 1968 Edsger Dijkstra wrote a letter to CACM entitled *GOTO Statement Considered Harmful*. The central ideas of software engineering were being born. At that time we believed that bigger, more disciplined methodologies would help us create software with consistent quality and predictable costs. The lawless cowboy coders were being reined in.



The 1980s were good times for computer programmers. We had a few rules and practices to create software that was far superior in quality to what we were creating only a few years earlier. It seemed like if we could just create enough rules to cover the problems we encounter we could create perfect software and be on time. We added more and more rules and practices to cover all the potential problems.

Now in the 21st century we find these rules are hard to follow, procedures are complex and not well understood and the amount of documentation written in some abstract notation is way out of control. Trying to come up with a bigger and better methodology was like a California gold rush; everyone headed west only to be disappointed.

[Continued on Page 2](#) 

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [The XP Rules](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.

What is a Lightweight Methodology?

[Continued from Page 1](#)

We created software to help us create software. But this quickly got out of control and dreadnought CASE tools were born. These tools, originally created to help us follow the rules, are too hard to use themselves. Computer programmers find it necessary to cut corners and skip important practices to stay on schedule. No one is actually following the heavy methodologies we have handcuffed ourselves with. The cowboys have returned and we find ourselves back at the OK Corral.

When programmers ignore the rules of their methodology they are instinctively moving away from heavyweight methodologies and back toward an earlier, simpler time of lightweight methodologies when a few rules were enough. But we don't want to forget what we have learned. We can choose to keep the rules that help us create quality software and throw away those that hinder our progress. We can simplify those rules that seem too complex to follow correctly.



We don't want to return to the early days of cowboy coding when there were no rules at all. But instead let's stop at just enough rules to keep our software reliable and reasonably priced. Instead of cowboy coders we have software sheriffs; working together as a team, quick on the draw, armed with a few rules and practices that are light, concise, and effective.

Extreme Programming (XP) is one of [several new lightweight methodologies](#). XP has a few rules and a modest number of practices, all of which are easy to follow. XP is a clean and concise environment developed by observing what makes software development go faster and what makes it move slower. It is an environment in which programmers feel free to be creative and productive but remain organized and focused. ☺ ☺

[ExtremeProgramming.org home](#) | [Introducing XP](#) | [Email the webmaster](#)

Copyright 1999 James D. Wells all rights reserved



Introducing Extreme Programming

*Lessons
Learned*

Let's turn now to Extreme Programming (XP) itself. It is a collection of rules and practices each of which supports several others, and are supported by several others in turn. When they are used together a methodology emerges.


We shall examine these rules and practices now. We will begin with the practices associated with planning. User stories are the heart of planning in XP. User stories can be printed or hand written on cards. The project scope and plan is simply and efficiently created by manipulating the cards by hand.

Next we will see how XP projects design the system architecture. Architectural spikes or prototypes are used to create a simple overall design also known as the system metaphor. CRC Cards, a simple groupware design technique, encourages all team members to understand and contribute to the system design. But unique to XP is a reliance on a programming technique called refactoring to help uncover the most effective system architecture.



Then we turn our attention to methods for code creation. As we noted earlier, code quality is very important on an XP project. Practices which enhance quality include pair programming, refactoring, and creating tests before the code.

Testing occupies the place of honor. Good unit test and acceptance test coverage is the hall mark of an XP project. An XP project takes the attitude that developers are responsible for proving to their customers that the code works correctly, not customers proving the code is broken.

Continue to follow the little  logos for a guided tour or jump to the [catalog of rules and practices](#) for easy reference. The *Lessons Learned* buttons link to experiences shared with us by people who have already tried XP.

[ExtremeProgramming.org home](#) | [User Stories](#) | [Email the webmaster](#)

Copyright 2000 Don Wells all rights reserved



Test the PIA Class Into Existence

The first bit of programming we can do is for the PIA class. We know that we need to simulate the hardware and we know what that needs to look like. There will be 3 methods, one for programming the input and output bits, one to read, and one to write to the PIA. So let's create the PIA first.

The way that we create code in Extreme Programming (XP) is to start with a test.

Fortunately we have already created a unit testing framework. So let's create some code for our first test. What we want here is to be able to read the PIA and have any bits programmed to output always be zero. Input bits are unaffected when being read.

```
package simulator.r1.unittest;

import unittest.framework.*;
import simulator.r1.*;

public class TestReadFromPIA extends Test
{
    public void runTest()
    {
        PIA.register = 0x0F0F;
        PIA.setInputs(0x00FF);
        should(PIA.read() == 0x000F, "Outputs should always be zero");
    }
}
```

To be able to compile and run this test we need to create a stub class for our PIA. We will just create the methods and not bother writing code.

```
package simulator.r1;

public class PIA
{
    public static int register;

    public static int read()
    {
        return 0x0000;
    }

    public static void setInputs(int aShort)
    {
    }
}
```

Next we can create a TestSuite for our first test.

```
package simulator.r1.unittest;

import unittest.framework.*;

public class SimulatorTests extends TestSuite
{
    public SimulatorTests()
    {
        tests = new Test[1];
        tests[0] = new TestReadFromPIA();
    }
}
```

Now we can run this test. It fails of course. But we like to run it to make sure. Occasionally the test will pass, which means that our code already does what we need or our test doesn't test what we need. Lets go back to the PIA class and write some code to make the test work. We add some code and get the following.

```
package simulator.r2;

class PIA
{public static int register = 0;
 public static int inputBits = 0;

 public static int read()
 {return register & inputBits;}

 public static void setInputs(int aBitMask)
 {inputBits = aBitMask;};}
```

Let's try the unit test now. It runs as expected. Our code is still nice and simple and clean so we don't need to refactor. So let's add a second test. We need to test writing to the PIA.



[ExtremeProgramming.org home](#) | [Back to Creating a Spike Solution](#) | [Test Writing to the PIA](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Unit Test Framework for a Coffee Maker

We can get a [unit testing framework](#) for Java easy enough. The one we will be using here is one which runs as an applet and is compiled with Sun's 1.0.2 JDK compiler. This code should run on most browsers.

We will create a test suite of three tests to just test the unit test framework. They are a test which passes, a test which fails due to a bad computation, and a test which aborts because some unexpected exception occurs.

The applet in the right column is our unit test framework, click run tests to see if it works. The source code is [available for download](#).

Things are going well, we have our unit

testing framework, [story cards](#), and we are ready to do our [spike solution](#). After that we can begin to estimate our project scope and create a schedule with a release planning meeting. ☺ ☺

[ExtremeProgramming.org home](#) | [XP and the Coffee Maker](#) | [A Spike Solution](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Coffee Maker Unit Test Framework

The source code for the very simple Java unit test framework can be downloaded as a [zip file](#). Or you can just cut and paste from here.

The first part is the TestGUI class. This

class is the applet that runs the tests and reports the results. You will probably notice at this point that the formatting is different from what you are used to seeing. This is an experiment.

```
package unittest.framework;

import java.awt.*;
import java.applet.*;

public class TestGUI extends Applet
{Label scoreLabel;
Button runTestsButton;
List listOfTests;
Test tests [];

public void init()
{initializeTests();
setLayout(new BorderLayout());
add("North", scoreLabel());
add("Center", listOfTests());
add("South", runButton());
scoreLabel.setBackground(Color.lightGray);}

public void initializeTests()
{String testSuiteName = getParameter("TestSuite");
tests = testSuiteNamed(testSuiteName).tests;}

private TestSuite testSuiteNamed(String aClassName)
{try
{return (TestSuite) Class.forName(aClassName).newInstance();}
catch (Exception exception)
{return new TestSuite();};}

void runTests()
{for (int each = 0; each < tests.length; each++)
{runTest(each);
showResults();};}

void runTest(int anIndex)
{tests[anIndex].setUp();
tests[anIndex].run();
tests[anIndex].tearDown();}

private void showResults()
{for (int each = 0; each < tests.length; each++)
{listOfTests.replaceItem(tests[each].result, each);}
showScore();}

private void showScore()
{int passed = numberPassed();
float total = (float) tests.length;
int score = (int)(passed / total * 100);
scoreLabel.setText(new Integer(score).toString() + "%");
showPassFail(score);}
```

```

private int numberPassed ()
    {int passed = 0;
    for (int each = 0; each < tests.length; each++)
        {if (tests[each].success) passed++;}
    return passed;}

private void showPassFail (int aScore)
    {scoreLabel.setBackground((aScore == 100) ? Color.green : Color.red);}

private Label scoreLabel()
    {return scoreLabel = new Label("Not Run", Label.CENTER);}

private List listOfTests ()
    {listOfTests = new List(tests.length, false);
    for (int each = 0; each < tests.length; each++)
        {listOfTests.addItem(tests[each].result);};
    return listOfTests;}

private Button runButton()
    {runTestsButton = new Button("Run Tests");
    return runTestsButton;}

public boolean action(Event anEvent, Object anObject)
    {if(wasRunTestsPressed(anEvent))
        {runTests();
        return true;}
    else
        {return false;};}

private boolean wasRunTestsPressed(Event anEvent)
    {return anEvent.target == runTestsButton;};}

```

The next portion is the Test class. This class will be the super class of any tests we will be creating.

```

package unittest.framework;

/*
 * This is the class to extend for each test you need to run.
 * One new class for each test. JUnit allows multiple tests
 * per test class and is becoming the standard. Use JUnit instead.
 * setUp() is called before the test is run and can be used to
 * initialize your test. runTest() is called to actually run
 * your test. Override it and send the message should(boolean, String)
 * to check if the test has passed or failed.
 * tearDown() is called after your test is run and can be used
 * to clean up.
 */

public class Test
    {public boolean success;
    public String result;

    public Test()
        {super();
        this.initialize();}

    private void initialize()

```

```

    {this.testFailed("not run");}

public void setUp()
    {}

protected void runTest()throws Exception
    {}

public void tearDown()
    {}

protected void should (boolean aTestPassed, String aMessage)
    {if (!aTestPassed)
        {throw new TestFailedException(aMessage);};}

public void run()
    {runAndCaptureAborts();}

private void runAndCaptureAborts()
    {try
        {runAndCaptureFailures();}
    catch (Exception exception)
        {testFailed("Aborted : " + exception.getMessage());};}

private void runAndCaptureFailures()throws Exception
    {try
        {runAndAllowExceptions();}
    catch (TestFailedException exception)
        {testFailed("Failed : " + exception.getMessage());};}

private void runAndAllowExceptions()throws TestFailedException, Exception
    {runTest();
    testPassed();}

private void testPassed()
    {success = true;
    result = message("Passed");}

private void testFailed(String aMessage)
    {success = false;
    result = message(aMessage);}

private String message(String aString)
    {return getClass().getName() + " : " + aString;};}

```

The third and last portion of the framework is the TestSuite class. This simple class holds a set of tests together.



```
package unittest.framework;

/*
 * Extend this class to create sets of tests to be run together.
 * Override the creation method and initialize the tests variable to
 * contain an array of Test subclasses. One instance for each
 * test that needs to be run. This superclass also provides a
 * default empty suite of tests.
 */

public class TestSuite
    {public Test tests [];

    public TestSuite()
        {tests = new Test[0];};}
```

It helps to see some examples of how to actually use this framework. Let's create a sample TestSuite with 3 tests. One each for pass, fail, and abort results.

The most instructional class is presented last. The AbortTest is more like a test we would create. An expression followed by what we are testing is passed to the should method.  

```
package unittest.framework;

public class FrameworkTests extends TestSuite
    {public FrameworkTests()
        {tests = new Test[3];
        tests[0] = new unittest.framework.GoodTest();
        tests[1] = new unittest.framework.FailTest();
        tests[2] = new unittest.framework.AbortTest();};}

public class GoodTest extends Test
    {protected void runTest()
        {should (true, "This test always succeeds");};}

public class FailTest extends Test
    {protected void runTest()
        {should(false, "this test always fails");};}

public class AbortTest extends Test
    {private int number[] = {0,1,2,3};

    protected void runTest()
        {should(number[1] / number[0] == 0, "test divide by zero");};}
```

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [Back to Unit Test Framework](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Story Cards for a Coffee Maker

Let's begin creating **user stories** for the Mark IV Coffee Maker. The Fictitious Advanced Product Design Department (FADD) is our customer in this case. They will determine what the coffee maker will do. They will make these decisions based on their experience with the coffee maker market and where they want this new coffee maker to be positioned strategically in the market. We create four stories making sure that the stories are about what the coffee maker will do and not about how it will do it. It is up to the hardware design group and us, the software group to decide if these things can work and how.

Brew some coffee. When the brew button is pressed boil the water until empty.

Keep the coffee warm. When the pot has coffee in it turn on the warmer. When the coffeepot is empty turn off the warmer. When the coffeepot is removed turn off the warmer.

Indicator light. Turn on the indicator light when the coffee is done brewing. Turn off the indicator light the first time the coffeepot is picked up.

Interrupt brewing if the coffeepot is removed. Opening the relief valve will stop the water flow. If the coffeepot is replaced continue.

When we met with the hardware designers they proposed we add a cancel brewing function. We ask the customers about it and they make it clear this is not a good idea. This feature will add nothing to marketability and will cost us to implement.

Meanwhile, our other team members are creating a **unit testing framework** and trying out a spike solution. The spike solution will be critical to our estimations at the release planning meeting. 🗺️ 📊

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP and the Coffee Maker](#) | [Testing Framework](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



XP and the Mark IV Coffee Maker

There has been some discussion on [comp.object.moderated](#) of how Extreme Programming (XP) would design a solution to Robert Martin's Mark IV Coffee Maker example. Use cases and a design based on them are provided by [Jim Weirich](#). The question is would XP generate a similar design? Let's begin XP style by scoping the project and seeing what kind of delivery date we can achieve.

The first phase of any XP project is to gather [user stories](#) and conduct some experiments. The stories will be used to estimate

the project and to schedule a release date. The experiments will allow us to make estimations with confidence.

Some of us will go [meet with the hardware people](#) now. Mean while, some of us will help the Fictitious Advanced Product Design Department (FAPDD) [create user stories](#), and some of us will work on a [unit testing framework](#) and then [a spike solution](#). ☒ ☑

[ExtremeProgramming.org home](#) | [The Coffee Maker's Hardware](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



XP and the Mark IV Coffee Maker

The meeting between the software people and the electrical and mechanical engineers on the project goes well. The hardware people have already been approached by the Fictitious Advanced Product Design Department (FAPDD) and have added several new features to this coffee maker.

A relief valve to interrupt coffee brewing, and a weight sensor in the coffeepot station are new. But most importantly to us is the microprocessor control system. A state of the art microcontroller with Java 1.0 burnt into ROM, half a Meg of FLASH, half a Meg of RAM and 32 GPIO pins.

Fabulous, we won't have to program this thing in assembler [and all of the people watching over our shoulders will be able to actually run the code from their Internet browser.] The electrical engineers will use the bottom most 8 GPIO pins for input from the A/D converter to get the pressure sensor reading. The next 2 pins are input for the brew button and the water sensor. The next 4 pins are output to the indicator light, warmer, boiler, and pressure relief valve in that order. The top most 18 pins are unused.

Things are going well, then one of the engineers asks a question: "What about the cancel brew function."

"We haven't heard anything about that."

"Well, we talked to the FAPDD guys and they don't want it but we could just add it easy enough."

"Any additional requirements will effect our time to delivery date."

"But it won't cost you anything to add it now, adding it later will."

"I don't think that is true. If we add a cancel brewing function we will have to read the brew button during the brew cycle right?"

"Well yes, I suppose but you were going to do that any way weren't you?"

"We don't know yet. Besides, there is another issue. We would be required to debounce the switch."

"Weren't you going to debounce the switch anyway?"

[Continued on Page 2.](#) 


[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP and the Coffee Maker](#) | [User Stories](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Debounce a Switch

The problem with hardware is it doesn't behave nicely. In the case of switches they don't just turn off and on. They bounce. When you push a button it will go on, then off, then on, then off, before it goes on and stays on. The common way to deal with this is to read a switch's input

and if it has changed state you must ignore it for a fraction of a second. Just long enough for it to stop bouncing. Then you may reliably read it as input again. 

[ExtremeProgramming.org home](#) | [Back to Hardware Design](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



XP and the Mark IV Coffee Maker

⌂ [Continued from Page 1.](#)

"No we were going to keep it simple. If we debounce the switch then we add a requirement to keep track of time. I would like to keep such things out of the software if possible."

"Sounds like a bad design decision to me."

"If we find out we need it we will just add it later, XP allows us to act as if future requirements don't exist, you'll see. Next we need to talk about functional testing."

"We don't test our products we send them off to QA. They test it and tell us what is broken. We fix it and send it back again until everything is working."

"I was thinking of a different scenario. Suppose we send a product to QA that has no bugs?"

"That has never happened."

"We can at least try. What we will need is a computer with a 32 bit PIA to connect to the microcontroller we are building for the coffee maker. An additional PIA to control the coffee maker's power would also be good. We can then program the functional test computer to run scenarios on the coffee maker's microcontroller and test it completely before it even goes to QA."

"We could make something available near the end of the project, but you will need one of our MDL's and a FLASH burner for it to do you any good. That kind of equipment is already in short supply. You guys have a lot to learn about building commercial software!"

"I suppose if you could help us verify a software simulation, which we could then supplement with an occasional run on real hardware that would be enough. We would also like to do a spike solution before we attempt to cost out this project. We will need a prototype coffee maker. We want to try a simple experiment of reading inputs and controlling outputs."

"We haven't finished our design yet. Circuit boards won't be available for awhile."

"How about if we wire wrap something together quickly that just has the processor, brew button and the indicator light?"

"We can help with a schematic and show you how to burn the FLASH memory, if you do all the wire wrapping."

"Agreed."

Next let's see how the rest of the team is doing with the FADD helping them create [user stories](#). ⌂

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP and the Coffee Maker](#) | [User Stories](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Coffee Maker Spike Solution

The spike solution we want to try here will answer the question: Can we interface to the hardware as easy as we think? These types of spikes are very common and very useful at the beginning of an XP project.

In order to do this spike we will need some hardware. [Since no hardware exists we will use a simulation instead.] Our simulated hardware will need to be designed first. We use CRC cards to design it. Our next job is to create a simulation of our PIA, the only hardware we will interface with. The next piece will be the

simulator itself followed by a GUI to control it. Then we will create the spike code and answer our question. If the answer is yes, it is simple to control the coffee maker hardware, then our user story time estimates will be lower and with less risk.

We have not finished our spike yet, but we should be done soon, so book mark this page and come back! What we need next is have a release planning meeting with the user story cards we created. I will add that example as time allows.

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP and the Coffee Maker](#) | [Designing the Simulator](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



A Second Test for the PIA Class

Next we can create a unit test for writing to the PIA register. In this case we are making sure that only output bits are changed.

```
package simulator.r3.unittest;

import unittest.framework.*;
import simulator.r3.*;

class TestWriteToPIA extends Test
{public void runTest()
  {PIA.register = 0x00FF;
  PIA.setInput(0x0F0F);
  PIA.write(0x3333);
  should(PIA.register == 0x303F, "Write never changes inputs");}}
```

We also need to put this test into our test suite.

```
package simulator.r3.unittest;

import unittest.framework.*;

public class SimulatorTests extends TestSuite
{public SimulatorTests()
  {tests = new Test[2];
  tests[0] = new TestReadFromPIA();
  tests[1] = new TestWriteToPIA();}}
```

If we stub out our write method we can compile and run our new test to be sure it fails. Now let's create the code for the write method.


```
package simulator.r3;

public class PIA
{public static int register = 0;
  public static int inputBits = 0;

  public static int read()
  {return register & inputBits;}

  public static void write(int theNewOutputs)
  {register = read() | (theNewOutputs & ~inputBits);}

  public static void setInput(int aBitMask)
  {inputBits = aBitMask;}}
```

Run the test again. This time it passes as expected. Things are going well. But we are not sure we like the way this code looks. Let's do some refactoring now. 

[ExtremeProgramming.org home](#) | [Back to Creating a Spike Solution](#) | [Refactor](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Refactor the PIA Class

Before we do any refactoring we need to run the unit tests just to be sure we are at 100%. We will rely on the judgment of the unit tests to tell us if we have made a mistake while refactoring.

What we want to do here is make the code more self explanatory. We will change the parameter to the `setInputBits()` method from `aBitMask` to `theInputBits`. We can create a method `inputs()`. We can use the `inputs()` method in both `read()` and `write()`. We can create a method `outputs()` to further simplify `write()`. We can also create a method `outputBits()` to explain what `~inputBits`

means to us. And last we should make the `write()` method synchronized for multithreading.

Between each change we run our unit tests. Each time we create a new method, each time we change a name. But for the sake of brevity we will just show you our completed code.

```
package simulator.r4;

public class PIA
{public static int register = 0;
  public static int inputBits = 0;

  public static int read()
  {return inputs();}

  public static void write(int theNewOutputs)
  {register = inputs() + outputs(theNewOutputs);}

  public static void setInputBits(int theInputBits)
  {inputBits = theInputBits;}

  private static int inputs()
  {return register & inputBits;}

  private static int outputs(int theOutputs)
  {return theOutputs & outputBits();}

  private static int outputBits()
  {return ~inputBits;};}
```

Now let's run those unit tests one last time now that we are done refactoring. We have correctly refactored our code making it easier to understand. Next let's work on the simulator.



[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [The Simulator](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Start Building the Simulator Class

As we have just seen with the PIA class we begin to build a class by creating a unit test. In this case we create a unit test for a message that the simulator will send to the GUI. Let's start with the boiler.

What we want to do here is start up the simulator in its own Thread then change the PIA register. We then wait a fraction of second and then make sure a message was sent to the GUI.

```
package simulator.r5.unittest;

import unittest.framework.*;
import simulator.r5.*;

class TestBoilerOn extends Test implements SimulationInterface
{private int messageSent;
 private Thread simulation;

 public void setUp()
 {messageSent = 0;
  PIA.register = 0x0000;
  PIA.setInput(0x003F);
  startSimulator();}

 public void runTest()
 {PIA.write(0x1000);
  pauseOneHalfSecond();
  should(messageSent == 1, "Got boilerOn " + messageSent + " instead of once");}

 public void tearDown()
 {stopSimulator();}

 public void boilerOn()
 {messageSent++;}

 private void startSimulator()
 {simulation = new Thread(new Simulator(this));
  simulation.run();}

 private void pauseOneHalfSecond()
 {try
  {Thread.sleep(500);}
 catch (Exception exception)
 {};}

 private void stopSimulator()
 {simulation.stop();
  simulation = null;};}
```

With the [test framework](#) we are using we will also need to add this test to the test suite we are building.

```
package simulator.r5.unittest;

import unittest.framework.*;

public class SimulatorTests extends TestSuite
{public SimulatorTests()
{tests = new Test[3];
tests[0] = new TestBoilerOn();
tests[1] = new TestReadFromPIA();
tests[2] = new TestWriteToPIA();};}
```

With Java we need to create stubs for all the messages we will send before we can compile.

```
package simulator.r5;

public class Simulator implements Runnable
{public Simulator (SimulationInterface aGUI)
{}

public void run()
{};}

public interface SimulationInterface
{public void boilerOn();}
```

Now let's compile and run this new unit test to make sure that it fails. It fails as expected so we can now create some code to replace the stubs we just created.

```
package simulator.r6;

public class Simulator extends Thread
{SimulationInterface gui;

public Simulator (SimulationInterface aGUI)
{super();
gui = aGUI;}

public void run()
{for (int each = 0; each < 50; each++)
{if ((PIA.register & 0x1000) > 0) gui.boilerOn();
try
{sleep(100);}
catch (InterruptedException exception)
{};};};}
```

Now we can run the unit tests again to see if we have the required functionality. The test still fails. Why? Let's read what the test says, we got more than one message. We forgot to make sure the message gets sent only the first time the register is changed. 'x' ↵

[ExtremeProgramming.org home](#) | [A Spike Solution](#) | [Fix the Bug](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Fix a Bug in the Simulator Class

We ran our unit tests and found that we have a problem. Our simulator sends the message too many times. So what we need to do is keep track of what state the switch was in and only send the message if it changes. Let's write some code.

```
package simulator.r7;

public class Simulator extends Thread
{SimulationInterface gui;
  boolean boilerIsOn;

  public Simulator (SimulationInterface aGUI)
  {super();
   gui = aGUI;}

  public void run()
  {for (int each = 0; each < 50; each++)
    {if ((PIA.register & 0x1000) > 0 && !boilerIsOn)
      {gui.boilerOn();
       boilerIsOn = true;}
     try
     {sleep(100);}
     catch (InterruptedException exception)
     {}};};};}
```

Run those unit tests again. Now they run just fine. Let's clean up a bit now. 🧹 🗑️

[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [A Spike Solution](#) | [Refactor the Simulator](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Refactor the Simulator Class

We have kind of a mess on our hands right now. We got the test to run, but the code isn't very pretty. So let's do some refactoring. First we always run the unit tests to make sure we have 100%, then we start. We can add lots of small methods named for what they do.

```
package simulator.r8;

public class Simulator extends Thread
{SimulationInterface gui;
boolean boilerIsOn = false;
static final int BoilerSwitch = 0x1000;

public Simulator (SimulationInterface aGUI)
{super();
gui = aGUI;}

public void run()
{while (true)
{checkBoilerSwitch();
sleepOneTenthSecond();}}

private void checkBoilerSwitch()
{if (wasBoilerJustSwitchedOn()) turnOnBoiler();}

private boolean wasBoilerJustSwitchedOn()
{return isBoilerSwitchedOn() && boilerIsOff();}

private boolean isBoilerSwitchedOn()
{return (PIA.register & BoilerSwitch) > 0;}

private boolean boilerIsOff()
{return !boilerIsOn;}

private void turnOnBoiler()
{gui.boilerOn();
boilerIsOn = true;}

private void sleepOneTenthSecond()
{try
{sleep(100);}
catch (InterruptedException exception)
{}};};}
```

You know what comes next. Run those unit tests again. They still run so we have not broken anything. Let's add another unit test.



[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [A Spike Solution](#) | [Next Unit Test](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Unit Test the Boiler Turning Off

The next thing we can test is turning the boiler off. In order to test the boiler turning off we must first turn on the boiler then shut it back off. So let's extend the `TestBoilerOff` test to also test for the boiler coming back on. We will need to rename it to be `TestBoiler`.

```
package simulator.r8b.unittest;

import unittest.framework.*;
import simulator.r8b.*;

class TestBoiler extends Test implements SimulationInterface
    {private int onMessageSent, offMessageSent;
    private Thread simulation;

    public void setUp()
        {onMessageSent = 0;
        offMessageSent = 0;
        PIA.register = 0x0000;
        PIA.setInputs(0x003F);
        startSimulator();}

    public void runTest()
        {testBoilerOn();
        testBoilerOff();}

    private void testBoilerOn()
        {PIA.write(0x1000);
        pauseOneQuarterSecond();
        should(onMessageSent == 1, "Got boilerOn " + onMessageSent + " instead of
once");
        should(offMessageSent == 0, "Got a different message");}

    private void testBoilerOff()
        {PIA.write(0x0000);
        pauseOneQuarterSecond();
        should(onMessageSent == 1, "Got a different message");
        should(offMessageSent == 1, "Got boilerOff " + offMessageSent + " instead of
once");}

    public void tearDown()
        {stopSimulator();}

    public void boilerOff()
        {offMessageSent++;}

    public void boilerOn()
        {onMessageSent++;}

    private void startSimulator()
        {simulation = new Simulator(this);
        simulation.start();}

    private void pauseOneQuarterSecond()
```

```

    {try
      {Thread.sleep(250);}
    catch (InterruptedException exception)
      {};}

private void stopSimulator()
  {simulation.stop();
   simulation = null;};}

```

We can change the test suite, create stubs for the methods we expect to be calling, compile them, and try to run it. As we can see we are not getting our boiler off message as we expected. So now let's create that code to do that. This time I am just going to show the changes we will make to the Simulator class. We will also update the SimulationInterface to include boilerOff().

```

private void checkBoilerSwitch()
  {if (wasBoilerJustSwitchedOn()) turnOnBoiler();
   if (wasBoilerJustSwitchedOff()) turnOffBoiler();}

private boolean wasBoilerJustSwitchedOff()
  {return isBoilerSwitchedOff() && boilerIsOn;}

private boolean isBoilerSwitchedOff()
  {return (PIA.register & BoilerSwitch) == 0;}

private void turnOffBoiler()
  {gui.boilerOff();
   boilerIsOn = false;}

```

**Now we run the unit test and it passes.
What now? That's right, another unit test! 📄**

[ExtremeProgramming.org home](#) | [A Spike Solution](#) | [Next Unit Test](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Add Yet Another Unit Test to the Simulator Class

Let's test the next switch. We choose the relief valve. First the test code. We notice (since we cut and pasted it) that this test is very similar to the boiler test. We have an opportunity for

refactoring, but let's wait until we have two complete examples before we try to merge them together.

```
package simulator.r10.unittest;

import unittest.framework.*;
import simulator.r10.*;

class TestReliefValve extends Test implements SimulationInterface
{private int onMessageSent, offMessageSent, otherMessageSent;
 private Thread simulation;

 public void setUp()
 {onMessageSent = 0;
  offMessageSent = 0;
  otherMessageSent = 0;
  PIA.register = 0x0000;
  PIA.setInput(0x003F);
  startSimulator();}

 public void runTest()
 {testReliefValveOn();
  testReliefValveOff();}

 private void testReliefValveOn()
 {PIA.write(0x2000);
  pauseOneQuarterSecond();
  should(onMessageSent == 1, "Got reliefValveOn " + onMessageSent + " instead of
once");
  should(offMessageSent == 0, "Got a different message");}

 private void testReliefValveOff()
 {PIA.write(0x0000);
  pauseOneQuarterSecond();
  should(onMessageSent == 1, "Got a different message");
  should(offMessageSent == 1, "Got reliefValveOff " + offMessageSent + " instead
of once");
  should(otherMessageSent == 0, "Got some other message");}

 public void tearDown()
 {stopSimulator();}

 public void boilerOff()
 {otherMessageSent++;}

 public void boilerOn()
 {otherMessageSent++;}

 public void reliefValveOff()
 {offMessageSent++;}

 public void reliefValveOn()
```

```
{onMessageSent++;}

private void startSimulator()
{simulation = new Simulator(this);
simulation.start();}

private void pauseOneQuarterSecond()
{try
{Thread.sleep(250);}
catch (InterruptedException exception)
{};}

private void stopSimulator()
{simulation.stop();
simulation = null;};}
```

We add this to the test suite, and create stubs for the methods we expect to be calling, now compile it all, and try to run it. It fails. Let's make some changes to the simulator class. And add our new methods reliefValveOn() and reliefValveOff() to our SimulationInterface.

```
package simulator.r11;

public class Simulator extends Thread
{SimulationInterface gui;
boolean boilerIsOn = false;
boolean reliefValveIsOn = false;
static final int BoilerSwitch = 0x1000;
static final int ReliefValveSwitch = 0x2000;

public Simulator (SimulationInterface aGUI)
{super();
gui = aGUI;}

public void run()
{while (true)
{checkBoilerSwitch();
checkReliefValveSwitch();
sleepOneTenthSecond();};}

private void checkBoilerSwitch()
{if (wasBoilerJustSwitchedOn()) turnOnBoiler();
if (wasBoilerJustSwitchedOff()) turnOffBoiler();}

private boolean wasBoilerJustSwitchedOn()
{return isBoilerSwitchedOn() && boilerIsOff();}

private boolean wasBoilerJustSwitchedOff()
{return isBoilerSwitchedOff() && boilerIsOn;}

private boolean isBoilerSwitchedOn()
{return !isBoilerSwitchedOff();}

private boolean isBoilerSwitchedOff()
{return (PIA.register & BoilerSwitch) == 0;}

private boolean boilerIsOff()
{return !boilerIsOn;}
```

```
private void turnOnBoiler()
    {gui.boilerOn();
    boilerIsOn = true;}

private void turnOffBoiler()
    {gui.boilerOff();
    boilerIsOn = false;}

private void checkReliefValveSwitch()
    {if (wasReliefValveJustSwitchedOn()) turnOnReliefValve();
    if (wasReliefValveJustSwitchedOff()) turnOffReliefValve();}

private boolean wasReliefValveJustSwitchedOn()
    {return isReliefValveSwitchedOn() && reliefValveIsOff();}

private boolean wasReliefValveJustSwitchedOff()
    {return isReliefValveSwitchedOff() && reliefValveIsOn;}

private boolean isReliefValveSwitchedOn()
    {return !isReliefValveSwitchedOff();}

private boolean isReliefValveSwitchedOff()
    {return (PIA.register & ReliefValveSwitch) == 0;}

private boolean reliefValveIsOff()
    {return !reliefValveIsOn;}

private void turnOnReliefValve()
    {gui.reliefValveOn();
    reliefValveIsOn = true;}

private void turnOffReliefValve()
    {gui.reliefValveOff();
    reliefValveIsOn = false;}

private void sleepOneTenthSecond()
    {try
        {sleep(100);}
    catch (InterruptedException exception)
        {}};
```

Now we run the unit test and it passes.
Before we can add our next unit test we need to
spend some time refactoring the two tests we
already have so that we can add the third simply.



[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [A Spike Solution](#) | [Next Unit Test](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Refactor the Simulator Unit Tests

Now let's refactor the unit tests we have written. The first thing we always do before refactoring is run the unit tests to be sure we start at 100%. In this case we are going to be working with the unit tests themselves. We are using our code now to verify the unit tests were refactored correctly instead of the other way around.

```
package simulator.r12.unittest;

import unittest.framework.*;
import simulator.r12.*;

class SwitchTest extends Test implements SimulationInterface
{protected int onMessageSent, offMessageSent, otherMessageSent;
private Thread simulation;
static final int BoilerSwitch = 0x1000;
static final int ReliefValveSwitch = 0x2000;

public void setUp()
{onMessageSent = 0;
offMessageSent = 0;
otherMessageSent = 0;
PIA.register = 0x0000;
PIA.setInputs(0x003F);
startSimulator();}

public void runTest(int theSwitch)
{testSwitchOn(theSwitch);
testSwitchOff();}

public void tearDown()
{stopSimulator();}

private testSwitchOn(int theSwitch)
{PIA.write(theSwitch);
pauseOneQuarterSecond();
should(onMessageSent == 1, "Got on message " + onMessageSent + " instead of
once");
should(offMessageSent == 0, "Got an off message");}

private testSwitchOff()
{PIA.write(0x0000);
pauseOneQuarterSecond();
should(onMessageSent == 1, "Got an on message instead");
should(offMessageSent == 1, "Got off message " + offMessageSent + " instead of
once");
should(otherMessageSent == 0, "Got some other message");}

public void boilerOff()
{otherMessageSent++;}

public void boilerOn()
{otherMessageSent++;}
```

```

public void reliefValveOff()
    {otherMessageSent++;}

public void reliefValveOn()
    {otherMessageSent++;}

private void startSimulator()
    {simulation = new Simulator(this);
    simulation.start();}

private void stopSimulator()
    {simulation.stop();
    simulation = null;}

private void pauseOneQuarterSecond()
    {try
{Thread.sleep(250);}
    catch (InterruptedException exception)
    {};};}

```

What we did to create this super class is to find everything that was in common between the two classes we already had and create a generic version. With this super class recreating our two tests is easy.

```

package simulator.r12.unittest;

class TestBoiler extends SwitchTest
    {public void runTest()
    {runTest(BoilerSwitch);}

    public void boilerOff()
    {offMessageSent++;}

    public void boilerOn()
    {onMessageSent++;};}

```

```

package simulator.r12.unittest;

class TestReliefValve extends SwitchTest
    {public void runTest()
    {runTest(ReliefValveSwitch);}

    public void reliefValveOff()
    {offMessageSent++;}

    public void reliefValveOn()
    {onMessageSent++;};}

```

Run the new unit tests and they pass. We can add the third simply now. But before we do that let's refactor our simulator class first. There is a lot of code that looks just alike and I think it could be simpler. ☺ ☺

[ExtremeProgramming.org home](#) | [A Spike Solution](#) | [Refactor the Simulator](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Refactor the Simulator Itself

Now it is the Simulator's turn for some refactoring. So let's run those unit tests and get started. The first thing I want to do is add a method called `isSwitchedOff()`. Now I can change both `isBoilerSwitchedOff()` and `isReliefValveSwitchedOff()` to use it. At this point I compile and run the unit tests to be sure my small change is correct.

Then I continue by changing `wasBoilerJustSwitchedOff()` and `wasReliefValveJustSwitchedOff()` to use `isSwitchedOff()` directly. Now I can delete both `isBoilerSwitchedOff()` and `isReliefValveSwitchedOff()`. Again I run the unit tests to verify this small change is correct.

I can now do the same with the "on" methods as well. After each small change I run the tests before continuing.

This refactoring eliminates several of the switch specific methods. Last I will create a `Switch` interface to keep track of all those bit masks.

```
package simulator.r13;

public class Simulator extends Thread implements Switches
{SimulationInterface gui;
 private boolean boilerIsOn = false, reliefValveIsOn = false;

 public Simulator (SimulationInterface aGUI)
 {super();
  gui = aGUI;}

 public void run()
 {while (true)
  {checkBoilerSwitch();
   checkReliefValveSwitch();
   sleepOneTenthSecond();}}

 private void checkBoilerSwitch()
 {if (wasJustSwitchedOn(BoilerSwitch, boilerIsOn)) turnOnBoiler();
  if (wasJustSwitchedOff(BoilerSwitch, boilerIsOn)) turnOffBoiler();}

 private void checkReliefValveSwitch()
 {if (wasJustSwitchedOn(ReliefValveSwitch, reliefValveIsOn))
 turnOnReliefValve();
  if (wasJustSwitchedOff(ReliefValveSwitch, reliefValveIsOn))
 turnOffReliefValve();}

 private void turnOnBoiler()
 {gui.boilerOn();
  boilerIsOn = true;}

 private void turnOffBoiler()
 {gui.boilerOff();
  boilerIsOn = false;}

 private void turnOnReliefValve()
 {gui.reliefValveOn();
```

```
    reliefValveIsOn = true;}

private void turnOffReliefValve()
    {gui.reliefValveOff();
    reliefValveIsOn = false;}

private boolean wasJustSwitchedOn(int aSwitch, boolean isOnNow)
    {return isSwitchedOn(aSwitch) && !isOnNow;}

private boolean wasJustSwitchedOff(int aSwitch, boolean isOnNow)
    {return isSwitchedOff(aSwitch) && isOnNow;}

private boolean isSwitchedOff(int aSwitch)
    {return (PIA.register & aSwitch) == 0;}

private boolean isSwitchedOn(int aSwitch)
    {return !isSwitchedOff(aSwitch);}

private void sleepOneTenthSecond()
    {try
        {sleep(100);}
    catch (InterruptedException exception)
        {};};
```

```
package simulator.r13;
```

```
public interface Switches
    {static final int BoilerSwitch = 0x1000;
    static final int ReliefValveSwitch = 0x2000;}
```

This looks like enough for now. If complexity creeps in again we will do more refactoring. Let's run those unit tests one last time to make sure we are ready to continue. How do we continue? Of course, another unit test. '↵' ↵

[ExtremeProgramming.org home](#) | [A Spike Solution](#) | [Another Unit Test](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Test the Warmer Switch Now

The next unit test will test the warmer switch. We refactored our unit tests so now this is trivial.

```
package simulator.r14.unittest;

class TestWarmer extends SwitchTest
{public void runTest()
{runTest(WarmerSwitch);}

public void warmerOff()
{offMessageSent++;}

public void warmerOn()
{onMessageSent++;};}
```

We always test our test by creating stubs, compiling and running. Of course it fails. Now let's add some code to make it pass. We can add three methods `checkWarmerSwitch()`, `turnOnWarmer()` and `turnOffWarmer()`. Then change our `run()` method to call them.

```
package simulator.r15;

public class Simulator extends Thread implements Switches
{SimulationInterface gui;
private boolean boilerIsOn = false, reliefValveIsOn = false, warmerIsOn = false;

public Simulator (SimulationInterface aGUI)
{super();
gui = aGUI;}

public void run()
{while (true)
{checkBoilerSwitch();
checkReliefValveSwitch();
checkWarmerSwitch();
sleepOneTenthSecond();};}

private void checkBoilerSwitch()
{if (wasJustSwitchedOn(BoilerSwitch, boilerIsOn)) turnOnBoiler();
if (wasJustSwitchedOff(BoilerSwitch, boilerIsOn)) turnOffBoiler();}

private void checkReliefValveSwitch()
{if (wasJustSwitchedOn(ReliefValveSwitch, reliefValveIsOn))
turnOnReliefValve();
if (wasJustSwitchedOff(ReliefValveSwitch, reliefValveIsOn))
turnOffReliefValve();}

private void checkWarmerSwitch()
{if (wasJustSwitchedOn(WarmerSwitch, warmerIsOn)) turnOnWarmer();}
```

```

    if (wasJustSwitchedOff(WarmerSwitch, warmerIsOn)) turnOffWarmer();}

private void turnOnBoiler()
    {gui.boilerOn();
    boilerIsOn = true;}

private void turnOffBoiler()
    {gui.boilerOff();
    boilerIsOn = false;}

private void turnOnReliefValve()
    {gui.reliefValveOn();
    reliefValveIsOn = true;}

private void turnOffReliefValve()
    {gui.reliefValveOff();
    reliefValveIsOn = false;}

private void turnOnWarmer()
    {gui.warmerOn();
    warmerIsOn = true;}

private void turnOffWarmer()
    {gui.warmerOff();
    warmerIsOn = false;}

private boolean wasJustSwitchedOn(int aSwitch, boolean isOnNow)
    {return isSwitchedOn(aSwitch) && !isOnNow;}

private boolean wasJustSwitchedOff(int aSwitch, boolean isOnNow)
    {return isSwitchedOff(aSwitch) && isOnNow;}

private boolean isSwitchedOff(int aSwitch)
    {return (PIA.register & aSwitch) == 0;}

private boolean isSwitchedOn(int aSwitch)
    {return !isSwitchedOff(aSwitch);}

private void sleepOneTenthSecond()
    {try
        {sleep(100);}
    catch (InterruptedException exception)
        {};};}

```

We now run the unit tests again and they pass. Something is wrong with this code. It is getting way to large and hard to understand and there are lots of methods that look alike except for a couple name changes. I think it is time for a big refactoring. 🤖

[ExtremeProgramming.org home](#) | [A Spike Solution](#) | [Refactor Again](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.



Refactor the Simulator Again

What we want to do here is remove the duplication of code. What we see is lots of code that does not use any local variable or methods which are exclusive to the owning class. We see the potential for a new class to eliminate the redundancy. We can try it and see if we like it. But first we run the unit tests to be sure we are at 100% to start. First I can extract out the boiler switch check into a separate class. We can then remove the boiler switch check from the simulator.

```
package simulator.r16;

public class BoilerSwitchCheck extends Object implements Switches
{private boolean boilerIsOn = false;

public void checkBoilerSwitch(SimulationInterface aGUI)
{if (wasJustSwitchedOn()) turnOnBoiler(aGUI);
if (wasJustSwitchedOff()) turnOffBoiler(aGUI);}

private void turnOnBoiler(SimulationInterface aGUI)
{aGUI.boilerOn();
boilerIsOn = true;}

private void turnOffBoiler(SimulationInterface aGUI)
{aGUI.boilerOff();
boilerIsOn = false;}

private boolean wasJustSwitchedOn()
{return isSwitchedOn() && boilerIsOff();}

private boolean wasJustSwitchedOff()
{return isSwitchedOff() && boilerIsOn;}

private boolean isSwitchedOff()
{return (PIA.register & BoilerSwitch) == 0;}

private boolean isSwitchedOn()
{return !isSwitchedOff();}

private boolean boilerIsOff()
{return !boilerIsOn;};}
```

Now let's run those same unit tests again. They run just fine so we refactored out the new class correctly. We can refactor out the relief valve check and the warmer switch check as well. Our simulator is now reasonably simple.

```
package simulator.r17;

public class Simulator extends Thread
{SimulationInterface gui;
private BoilerSwitchCheck boiler = new BoilerSwitchCheck();
private ReliefValveCheck reliefValve = new ReliefValveCheck();
private WarmerSwitchCheck warmer = new WarmerSwitchCheck();

public Simulator (SimulationInterface aGUI)
{super();
gui = aGUI;}

public void run()
{while (true)
{boiler.checkBoilerSwitch(gui);
reliefValve.checkReliefValve(gui);
warmer.checkWarmerSwitch(gui);
sleepOneTenthSecond();};}

private void sleepOneTenthSecond()
{try
{sleep(100);}
catch (InterruptedException exception)
{}};}
```

Let's run the unit tests for this simplified simulator class. We pass the test. Now we have a different problem to solve. We have three classes almost alike. Can we create a single super class to hold most of the common code? I haven't finished trying this yet, so come back soon and see what happens next! ☺ ☹

[ExtremeProgramming.org home](#) | [A Spike Solution](#) | [Back to Spike](#) | [Email the webmaster](#)

Copyright 1999 by Don Wells.

XP Unit Tests

Extreme Programming

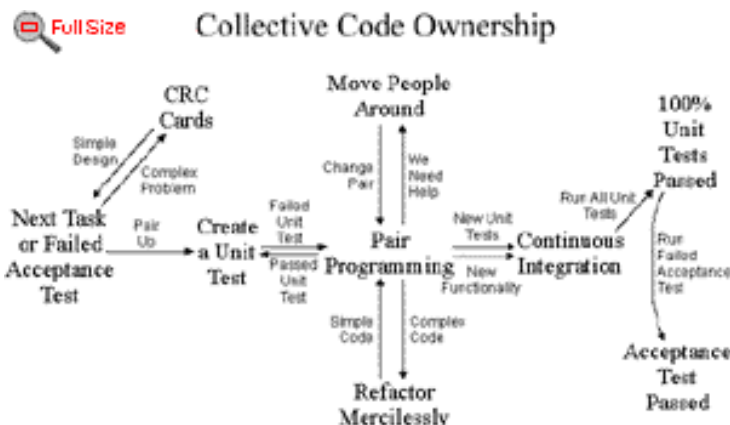
*Lessons
Learned*

[Continued from page 1](#)

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. 🐘 🐘



[ExtremeProgramming.org home](http://www.extremeprogramming.org) | [XP Rules](#) | [Download a Unit Test Framework](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



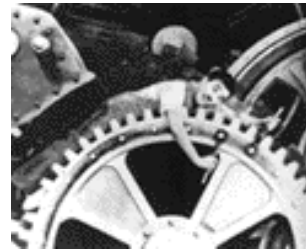
The Customer is Always Available

[Continued from page 1.](#)

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.

XPlorations

[ExtremeProgramming.org home](#) | [XP Rules](#) | [Coding Standards](#) | [Email the webmaster](#)

Copyright 1997, 1999 Don Wells all rights reserved. Any resemblance to actual customers is purely coincidental.



Release Planning

Lessons Learned

[Continued from page 1](#)

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. ☹ ☹



[ExtremeProgramming.org home](#) | [XP Rules](#) | [Release Plan](#) | [Email the webmaster](#)

Copyright 1999 Don Wells all rights reserved.



XP and Databases

*Lessons
Learned*

At the VCAPS project we found ourselves faced with the problem of XP and a large database. Our database was Object Oriented, but a relational database could be handled the same way. Remember that if you implement user stories customer value first your database tables and normalization will become stable faster.

The key point is taking the advice of Kent Beck, act as if the database is easy to change. Relational databases were created to be flexible, so flex them. Kent also advises that when something is very difficult try doing it more often not less. That way you get good at doing it and it won't be hard any longer. Get into the habit of migrating your database often, you will make less mistakes not more.

The VCAPS solution was to have a gold, a silver, and many bronze database versions. The gold is the one that resembles the production database. The silver is a migrated gold database. Each developer has a bronze database migrated from the silver.

A bronze database becomes silver when the developer's code is released. A silver database becomes gold when the production database is migrated.

It is important that everyone can easily get a copy of the gold or silver data base to use as a bronze quickly and that we keep track of migration paths. Setting up scripts to copy databases as files is very useful. You need one to make a copy of a bronze database promoting it to silver, and one to restore the current silver to the local host, which ever computer you are at.

Each database has the same set of test user ids and passwords. Developers and database connections can use the same user ids on any of the databases.

Each development pair will release newly developed code into the source code safe, promote their bronze database to silver, and add a migration script to a list. How you manage the list of migrations will depend on your data base software and how you access it. This list of scripts can then be executed in sequence to migrate the production database when the new

version is released to production. Independence from the application code is important since it can change causing the migration script fail. Use an **integration station** to control the sequence of changes.

At any moment in time the new silver database and the currently released code version are exactly in sync. This is important and requires discipline to maintain.

At any moment in time the gold database and the production release match up for production support. At any moment it time the data base migration scripts and the most current development release match up ready to be released.

Developers can integrate often because it is easy to copy the silver database and check out the current released code at the same time. The unit tests will run at 100%. Developers then add their own changes, perform any database migration, and integrate until the unit tests are running at 100% again. This method is very fast and only takes a couple minutes.

To support testing we had a script to create a new gold database with a predefined set of test data. Some data will be created by tests, but providing an example in the database helps developers create reliable migration scripts. We found it useful to create a new gold and migrate it to silver once a week to avoid the inevitable data corruption and to assure ourselves that our migration scripts were correct.

We did have an occasional problem with migration. We didn't do as good a job of keeping track of changes as we should have. What we ended up doing near the end was to formulate our migrations as executable methods on a DB maintenance object. This makes the production migration more reliable and a non-event. ☺ ☺

Don Wells



[ExtremeProgramming.org home](#) | [XP Lessons Learned](#) | [Next Lesson](#) | [Email the webmaster](#)

Copyright 1998, 1999, 2001 Don Wells all rights reserved.



What Has Changed Here?

August 17, updated [MXPE](#) next meeting.

March 29, added links to other XP summaries from the [project map](#) page.

March 27, small changes to the [front page](#), [iteration planning](#) and [project velocity](#).

February 17, Changed some icon links to new articles at www.xprogramming.com

January 30, [XP Universe and Agile Universe](#) will be in Chicago August 4-7, 2002.

January 23, Changes to the [feedback loop diagram](#).

January 5, Changes to the [integrate often page](#).

June 4, Changes to the [people page](#). New books on the [more page](#).

April 22, Updated the [XP and Database](#) page. Added a new diagram showing the [planning and feedback loops](#).

Thank You!

Changing this web site for the better is made possible by the people who have taken time to comment and make recommendations. Many people have contributed to this web site and the list has become too long to show. [Email the webmaster](#).

[ExtremeProgramming.org home](#) | [Email the webmaster](#)

Copyright 2000 Don Wells all rights reserved.