eXtreme Programming

[Home] [Armaties Software] [Ron Jeffries] [eXtreme Programming]
[Recommended Reading]

EXtreme Programming (XP) is the name that Kent Beck has given to the lightweight team process which he has been evolving over the years.

In XP, we use a very lightweight combination of practices to create a team that can rapidly produce extremely reliable, efficient, well-factored software. Many of the XP practices were created and tested as part of the Chrysler C3 project, which is a very successful payroll system implemented in Smalltalk.

You may find some of these practices quite useful - a few may be controversial. I'd like to hear from you in either case! Enjoy them!

[C3 Practices] [Portland Pattern Repository] [SQUEAK]
[Testing Framework] [VA Testing Framework]
[NEW! Squeak Testing Framework]

C3 Programming Practices	Here are the XP rules and practices that the C3 project team follows, together with some discussion of why we do what we do. Some of the XP practices are controversial. In these pages I describe some of the objections and answer them. Finally, like any set of patterns or ideas for a complex effort, there are places where judgment is needed. Sometimes C3 doesn't have it right yet; I'll describe the learning we have done and are doing.
Portland Pattern Repository	Ward Cunningham is Kent Beck's long-time associate and co-inventor of many of the ideas that underly XP. He is active in object consulting and runs this exciting site on the patterns movement. "We're writing about computer programs in a new stylistic form called pattern languages. The form has many internal references which map well to hypertext links. We've added links to published (or soon to be published) documents. "

Introduction

The C3 project team has voluntarily adopted a large number of practices. Every team member follows these practices closely. We have found that designing, coding, and testing in a consistent way means that the team becomes quite well integrated and that team members know just what they can expect from each other. Software quality is kept more uniform, and at a high level.

All team members, new and experienced, are expected to follow the practices. We do not waste a lot of time debating them, although we do modify them from time to time in light of actual team experience.

We do not implement a lot of checking or security, to ensure that people follow the standards. We expect and require full voluntary compliance. .

In this document, I will describe our key practices. In some sections, I will present some possible concerns that the reader might have, prefixed with "**On the contrary**". Then I will present a few bullet items responding to the concern.

A general comment: we wouldn't be using these practices if they didn't work. C3 has been a very successful project both in relation to earlier attempts to write the same program, and by comparison with team members' other projects. The reader may find some of these practices surprising or even questionable: in fact they do work, by actual experience. We do remain open to improvements, and we do expect that developers will use good judgment in specific situations.

When in doubt, get a few people together and decide what to do.

As a member of the C3 team, you will be expected to follow these practices, not blindly but thoughtfully. As you become expert in following them, you may see improvements. Discuss them with the team and help us refine them. But whatever the practices are, we all agree to follow them, not to strike off on our own.

The Four Project Values
Simplicity + Communication + Testing = Aggressiveness

Simplicity

We strive always to "do the simplest thing that could possibly work". We're quite serious about this. We have found that with the right architecture and correctly-factored code, we can rapidly extend our software when the need arises. If we work on something "we're gonna need", we're not working on something we DO need.

Communication

We emphasize communication in every way. In our code, we all use the same formatting conventions, naming conventions, and coding standards. This means that all methods, no matter who wrote them, look familiar and are easy to understand when we come upon them.

We code so as to express intention, not algorithm. Our method names say what they accomplish, not how they accomplish it. At every level, we try to make our methods read like a human description of what is going on.

So that we can communicate design, we do it the same way, using CRC cards. Whenever we begin to work on some new objects, a few of us will sit down and do some cards. Whenever we make big changes, we'll get a larger group together and work out what is to be done. But what about the formal documentation?

Testing

We are fanatics for testing. We would like to have more lines of test than we do of actual code. Every class must have a corresponding unit test class. Each public method should have at least one unit test method. We presently have over 1400 unit tests in the system.

When we release code, all the unit tests must run at 100%. You can't release unless they do: if they don't, you fix the problem immediately.

We have functional tests as well, that test the system from end to end by paying one or more people and checking the results. There are hundreds of functional tests.

The result of all this testing is that we know the system works, and when we make errors we find them immediately.

Aggressiveness (Fearlessness)

The result of the other three values is that we can be aggressive, or fearless. We can change any part of the system to be better because we know we have a solid system of tests. We can try things and if we don't like how they work, we throw them away and try again. We know we won't break the system, which gives us the confidence to move forward rapidly.

Code Formatting

Here are a few of the key code formatting patterns we use. The names and some of the examples are from Kent's book.

Inline Message Pattern

The message pattern at the beginning of the method is always formatted on one line, never indented.

Indented Control Flow

Warning, tabs below are larger than in actual code. I haven't coerced HTML to do what I want yet.

Zero or one argument messages go on the same line as their receiver:

foo isNil.

2 + 3. a < b ifTrue: [...]

Messages with two or more keywords have each keyword-argument pair on its own line, indented one tab under the receiver.

```
size > 0
   ifTrue: [ ... ]
   ifFalse: [ ... ]

array
   at: 5
   put: #abc

aCollection
   copyFrom: 1
   to: aString size
   with: aString
   startingAt: 1
```

Blocks are made rectangular, with the starting bracket as the upper left corner, and the ending bracket as the lower right. Use one-line blocks if other rules permit. All other rules apply inside blocks.

```
ifTrue: [self recomputeAngle]

ifTrue: [^angle*90 + 270 degreesToRadians]

ifTrue:
   [self clearCaches.
   self recomputeAngle]

ifTrue:
   [self
```

```
        at: each
        put: 0]
```

Use cascades if a bunch of zero- or one-argument messages are being sent to the same object.

```
self listPane parent
   color: Color black;
   height: 17;
   width: 11
```

Do not use cascades with multiple-argument messages. They are hard to read. For example, if height:width: is a single message, do NOT write:

```
self listPane parent
   color: Color black;
   height: 17
   width: 11
```

Instead write:

```
self listPane parent color: Color black.
self listPane parent
   height: 17
   width: 11
```

or, better:

```
| parent |
parent := self listPane parent.
parent color: Color black.
parent
   height: 17
   width: 11
```

If these rules result in code that looks ugly, chances are that what is really needed is to refactor the method. Don't question your formatting rules: question the code. Here's an example:

```
removeStep
   | stepToRemove |
   stepToRemove := self list selection.
   stepToRemove isNil ifFalse: [stepToRemove isExecutable ifTrue:
      [self list remove: stepToRemove.
      steps remove: stepToRemove]]
```

The above code is correctly formatted. However, it is ugly. We could try adding extra returns and tabs, in violation of our formatting rules:

```
removeStep
   | stepToRemove |
   stepToRemove := self list selection.
   stepToRemove isNil ifFalse:
```

Coding Standards

We all code to the exact same standards. We name our classes and methods the same way, we format code the same way. The particular standards aren't as important as the principle: we all code the same way, which means that all the code is easily understandable and maintainable by all of us.

If anyone stumbles across a mistake where code doesn't meet the standards, they just fix it.

Fundamentally, we follow the guidelines in Kent Beck's book: Smalltalk Best Practice Patterns. Kent provides the rationale for the patterns, and our experience is that they work just fine.

On the contrary, I have a style of coding that I like and that I'm used to. I should be allowed to use it.

- You are welcome to use your style. Just not on our project.

    (Alistair Cockburn accused Kent and me of playing "Do it My Way or Get Out". Follow the above link for some discussion of that.)

[Home] [XP] [XP Practices Frame]
© 1997, 1998, Ronald E Jeffries
ronjeffries@acm.org
http://www.armaties.com

Method comments

In general, we do not comment our methods. Beck teaches that a comment is used to indicate that a method is not yet finished. That's how we use them.

When we have a method that is unclear on the face of it, we try to rename or refactor the method so that it becomes clear. If we fail to make it clear, we'll put a comment in it.

If a method uses an unusual algorithm, or one that is published somewhere, to which we wish to refer, we might leave a comment in the method.

If a method has been shown by a performance measurement to be a bottleneck, and we have written it in a peculiar way for efficiency, we will leave a comment in the method.

Probably less than one percent of the methods in the system have comments. If they needed a comment, they would have them. Most methods written using our style do not need them.

On the contrary, all methods should have comments to make the code more clear.

- Properly written Smalltalk code usually expresses intention directly, so that comments are not needed. If code needs commenting, it needs refactoring even more.

- If you have an example of commented Smalltalk code that you think needs its comments, please send it to me and I will see whether I can refactor it not to need them. I'll post the results, either way, on my page. Be sure to include the unit tests!

[Home] [XP] [XP Practices Frame]
© 1997, 1998, Ronald E Jeffries
ronjeffries@acm.org
http://www.armaties.com

Class comments

We do use class comments to describe what a class is for, and sometimes how it works. We probably don't do this often enough, and we probably don't keep them as up to date as we should.

In my opinion, this argues for stronger focus on writing clear code, not just cracking down on class comments. A developer's nature is to develop code, not to develop documentation. The best process is one that works with nature, not against it. (Tree-hugger theory of software development.)

Still ... for most classes a class comment can be quite useful. Please create them when you create new classes.

Continuous Integration, Relentless Testing

We call the combination of frequent release with tests at 100%: **continuous integration, relentless testing.** The result is rapid progress and a system that always works better than it did the day before.

... to be continued ...

CRC Card Design

We use CRC card design almost exclusively. Developers, managers, and users are all comfortable with the cards, since they are non-threatening. Moving the cards around makes it easy to describe how classes work, and it is easy to throw a couple of cards away and create new ones if the design isn't working out.

We do not save the cards, nor document what they were. The classes are the documentation of the design: they represent what the system really is.

On the contrary, without a "rigorous design process", you can never create a quality system.

- Our design process is actually quite rigorous. Many developers contribute to our carding sessions, and then a team of developers realizes the design in classes. It just happens that we do not produce ancillary documentation as a by-product of our design process. We also do not waste time updating documentation, nor live with out-of-date documentation. It's a win-win approach.

- The system's quality is ensured by over 1300 unit tests. Every class has extensive unit testing that ensures it is doing what it is supposed to do.

- The system's quality is ensured by hundreds of functional tests. Each test exercises the whole system, end to end, and checks that it computes correct outputs from its inputs.

On the contrary, without a "rigorous design process", you never know what you are building or have built.

- Design documentation, in the real world, rarely reflects the reality of the software written: it is not kept up to date. Our focus is on the quality and maintainability of the actual software.

- Smalltalk's development environment lets the software stand for itself. With proper naming conventions, well-designed classes, and the powerful Smalltalk environment, it is easy to find out how anything works.

[Home] [XP] [XP Practices Frame]

Don't Go Dark

A few times during the project, a team took on a task and did not complete it within the iteration. They wanted to complete the task and kept it for the next iteration. This generally turned out to be a mistake.

If the estimates for a task are so wrong that it doesn't complete within an iteration, it is likely that something isn't understood about the task. The manager should detect this before the iteration ends in most cases.

The problem is as likely outside the task than inside. Quite possibly the team is trying to solve a problem that is too hard because of something wrong elsewhere in the system. Consider getting a larger group together to do CRC cards on the problem.

~~In the next iteration, change out at least one of the team members.~~

On the contrary, the existing team has more experience with the problem, and they are probably close to cracking it. You should let them go ahead.

- Experience on this project, and on many others, clearly indicates otherwise: if a task isn't going well, it's probably on the wrong track and needs to be reset. Even if the code finally works, it will continue to be a problem.
- When a solution isn't fitting into the system, either the system is wrong or the solution is wrong. In Smalltalk we have the ability to do substantial refactoring, even late in the project. Find out what is wrong and fix it.

On the contrary, it's demoralizing to be declared to be a failure by having someone else take over your task. You should let them go ahead.

- No one is declaring anyone to be a failure. Just as two sets of eyes make development go faster, bringing in a fresh set of eyes will help a bogged-down task come back on track.

In addition, frequently changing tasks is a way to keep the team fresh. Even though we do not practice code ownership, sometimes developers become particularly enamored of certain objects that they have created. This usually indicates a problem. Switch Teams Around.

Software Documentation

There is some written documentation for the software, more commonly interpreting requirements than documenting design. Sometimes we document an approach to some particularly difficult area. We "never" document code. (We rarely even comment it.)

Here are some good reasons why we do **<u>not</u>** do more documentation:

- We do not believe that it would help us go faster;
- We do not believe that it would make the system more reliable;
- We do not believe that it would make the system more maintainable.

Here are some good reasons why we **<u>should</u>** do more documentation:

- It might help other groups benefit from our process;
- It would make interesting reading to our customers;
- It would serve as the basis for useful papers in journals and conferences.

One area that may concern you is our use of CRC cards. We generally do not write anything on the cards, or if we do, we might write the class name. We move them around, touch them, hand them back and forth, let them act out the process we're designing. Then we throw the cards away and write the code!

You may ask how we later know what the design is, since the cards are gone. Our answer is that the design is represented in the code. If we did have a document, it would either be out of date, or we would have to spend time updating it. We make the code readable, and perhaps use a class comment to describe what a class does. And our high level of communication, and the fact that we do not practice code ownership, ensure that we all know what we need to about the entire system design.

We have written some literate programs for parts of the system, early on in the process. Like most system documentation world-wide, these were not kept up to date. Developers did not enjoy the process, nor find it useful enough to continue.

Should we do more? We don't know. Will we suffer at some time in the future? We don't know. We stay alert to signs that we need to change our process, and if we see signs that we need to do more documentation, we will do it.

Do it in an Inspector!

Never underestimate the power of inspecting and changing things with Inspectors. Perhaps your GemStone database is messed up: a bunch of People have excess Parts in certain Bins.

Grab an Inspector on some root object you can get a hold of. Send it a message, #people, and Inspect It. In the Inspector on the people, type an expression and Inspect It:

```
self select:
    [ :each | | found |
    found := each bins
        detect: [ :eachBin | eachBin size > 1]
        ifNone: [nil].
    found notNil]
```

In the Inspector on the collection of affected people, write a loop to fix each one:

```
self do:
    [ :each |
    (each bins select: [ :eachBin | eachBin size > 1]) do:
        [ :eachBad |
        (eachBad parts copyFrom: 2 to: eachBad parts size) do:
            [ :eachPart | eachBad removePart: eachPart]]]
```

The problem is resolved. Close your inspectors and carry on with your day.

Note: the code example above is not correctly formatted by our standards. In a couple of inspectors it is hard to get to well-factored code. Furthermore, what you have done to the system will be documented nowhere. These are rather troubling issues: what do they suggest to you?

No, Wait! Do it in a Workspace!

No, wait! The problem with what we just did (in an inspector) is that it isn't reproducible. We have no way to save a record of what we have done. If this task, or one like it, ever has to be done again, we'll have to type it in all over again. If we are called upon to do it, we might be able to remember a little about it. Others will have to invent the approach, and theirs might not be as effective as ours was.

When you're working on some task that you haven't done before, and it is going to require some sequential processing (grab an object, do something to it to get another object, etc), do it in a workspace. Not step by step, inspecting, sending, modifying, inspecting deeper.

Rather, set up the workspace to do the entire task, showing all steps. Let the sequence become obvious, and retain it in the workspace, rather than in your head. When you've got the workspace code working, the shape of the objects and methods you need will be much more obvious. You'll make faster progress because Smalltalk is remembering things for you.

When you're finished, save the workspace for future reference. One good way to do that is to make the workspace a method on the object you started with.

Note: the code we get in the workspace still isn't very well factored. In some ways it is even worse than the code in the inspectors: at least that code referred basically to only one object. This is troubling: what does it suggest to you?

No, Wait! Do it in an Object to Begin With!
EXAMPLE

Just this weekend, I went through the inspector / workspace / object cycle with a partner. We had damaged the database and needed to remove the damaged objects. We inspected until we understood what was going on. They we put the code into a workspace, and ran it until we were sure it was finding all the bad objects. Then we quit for the day – it was already quite late.

The next day we built an object (DatabaseFixer), moved our workspace method into it, and began refactoring. We tested frequently along the way. At the end, the method was refactored into about 8 or 10 methods, and we were much more certain that the Fixer was working correctly. Then we ran it without incident. The resulting Fixer class is quite likely to be useful in similar circumstances, and we can see how to enhance it if the situation isn't quite the same.

All in all, it took a couple of hours to develop the thing in a workspace, and a couple of hours to refactor it. Not bad at all.

In retrospect, however, it would have been much faster if we had started with the DatabaseFixer object.

When we refactored, we looked at our huge method, and removed chunks that might make a separate method. The original method was a huge loop that processed: all people; selected people; their bins; their parts; their bins with bad parts.

The final object actually looks like that is what it's doing:

    self peopleToProcess do: [:each | self processPerson: each]

and so on. Each individual method just does one loop, and states clearly exactly what it is about. There was one odd blob of code that built a dictionary of dictionaries of dictionaries of collections, and it's still in there, all in one method. Because it isn't in the middle of a loop and processing eachThis and eachThat, even that method reads more clearly than the original.

If we had just begun that way, we could have expressed our intention as we went along. Instead of concentrating on our weird dictionary structure (even if we had wound up with it), we would have been concentrating on the overall operation of the fixer. Methods would practically have written themselves, I'm sure of it.

Let's look at an example!

[Home] [XP] [XP Practices Frame]
© 1997, 1998, Ronald E Jeffries
ronjeffries@acm.org
http://www.armaties.com

ENVY Discipline

Here's a quick list of ENVY guidelines:

- Always work under the name of one of the two people presently pairing. If you are logged in as someone else, everyone will have trouble sorting out where your code is.
- Version frequently. Always version at end of day, to make it easier to bring the code up tomorrow. Don't count on saving the image: you might be working at another machine tomorrow, or someone else might take over because you don't come in.
- Begin each version name with your initials: rej.231, rch.232, etc. Optionally use the initials of both team members.
- Use specialized version names sparingly, when major changes have been put into a broad-ranging number of classes: Part>>posPart.103, Bin>>posPart.106, Station>>posPart.254.
- Use frequent releases.
- Load the config frequently. Load it at least daily. Ideally, load it before beginning any updating task that will lead to versioning or release. Your work will go more easily if you are running and testing on current code.

Frequent Release

Our development teams typically release their software at least once per day. Yes, release. Yes, at least once per day.

We have found that when developers sit on code for more than a day, it is usually because they are in trouble. When we understand what we are setting out to do, it is almost always quite possible to break down the work so that our changes can be released daily. (With all the tests working.)

On the contrary, there are some changes that are just so complex that they can't be done in a day.

- Yes, there are. We encountered one during the course of the project, when we were making a simple, but very fundamental change in how the values in the system were interpreted. It was incredibly difficult to get the system to work in the new mode. And it wasn't just a matter of a couple of fixes.
- We had several other tasks where the developers didn't release for several days. In every one of these other cases, the result was difficult integration, and unreliable code. In almost every case, we wound up removing the software and doing it over.

As a general guideline, when developers are holding back changes, treat it as a serious trouble sign and deal with it promptly. There can be exceptions, but fewer than you think.

[Home] [XP] [XP Practices Frame]

Functional Tests

On C3, there are also literally hundreds of functional tests, which test the system from end to end, input files to output files. The test cases were defined by the users, and developed by a separate team whose job was to create tests.

The functional tests have their own special GUI, which displays each step of the test, its execution time, and whether or not it succeeded. There is a score displayed.

Functional tests did not have to score 100% during the project: adding features makes their scores go up. They do have to score 100% before release, and any substantial change in the tests is always worthy of note.

We have a short suite of functional tests that we run before moving the code to GemStone, q.v., and during development all functional tests were run every day and the scores reported to the team. The testing team was proactive in recognizing when test failed that used to run, and would inform the developers as soon as something appeared to have gone wrong.

One mistake that we made with the functional tests was that they were not sufficiently end to end. Even after we made them run from input files to output files, we subsequently discovered differences in the downstream programs that used the files. This typically happened when we had misinterpreted what the output should be; the file had on it what we intended, and the test checked for that value, but the legacy program expected something else. Let your functional tests span as far as possible from input to output: the defects will appear wherever you stop.

[Home] [XP] [XP Practices Frame]

Short Iterations

We use a three-week "iteration", into which the project is time-boxed. User stories were broken down into engineering tasks. If an engineering task appears to require more than one engineering week, it must be broken down further.

Engineers estimate their own tasks, in "perfect engineering days". That is, they estimate the time it would take to do the task if everything went perfectly, they weren't interrupted, and so on. We assume (and it has held up well under measurement) that there are two real days per perfect day. Therefore each engineer signs up for enough tasks to fill 7 or 8 days in each iteration.

On the contrary, since you do pair programming, doesn't that cancel out the factor of two?

- You might think that this would happen, but it seems not to. We track how we do against the factor, and for some iterations we used a factor of 0.3 rather than 0.5. (We were doing a lot of necessary refactoring during those periods.)
- **Update**: recent events make us suspect that pairing does impact the factor of two, but does not fully cancel out. We are presently using an overall performance ratio of 1 / 2.5, or 0.4 instead of 0.5. We can't fully explain why the ratio has changed, but it clearly has. We are working on reducing interruptions, and on producing more tests at the beginning of tasks. The main thing, however, is that we know the ratio has changed and we use the new ratio in our planning. That lets us predict progress most accurately (if not always popularly).

On the contrary, the ratio of 1 to 2 accounts for engineer's errors in estimating how effective they are. Don't different engineers have different ratios?

- A refinement of our technique might be to maintain a separate factor for each engineer, but this would make estimating and managing the iterations quite difficult. So far, it hasn't been necessary.

Don't Think, Just Set the Halt

When partnering with Kent Beck, most everyone has this experience:

Something goes wrong. The code doesn't work. You start to think: "What could cause that to happen?" Kent doesn't think about what the problem is. He just sets a halt in the system and lets Smalltalk tell him what the problem is.

Sometimes you're right about what the problem is. If you're really quick you'll be able to tell Kent what to edit in the window he's already looking at. If you're really quick.

Sometimes you're not right about what the problem is. Forget it, he has already fixed it.

Train yourself to think about where to put the halt, not to think about what the problem is. Of course it's a great feeling when you can reason to the problem. But we're not here to make our brains feel good, we're here to get the code working as quickly as possible. Setting the halt and letting Smalltalk tell you will help you build working code faster.

[Home] [XP] [XP Practices Frame]

Let Smalltalk Tell You

Here's another guideline that is very easy to forget, but very important. The general notion is that Smalltalk is very good at incrementally changing things, breakpointing, and checking values. Instead of wasting time thinking about what to do, or theorizing about what went wrong, just ask Smalltalk. Here are some examples:

- To work out some new capability, inspect an object, browse the class if you need to, then send the object messages, building the code you need right in the Inspector workspace. Then copy the code to an appropriate method.
- When working on a new feature, don't implement all the necessary methods on all the objects you use. Instead, wait for the walkback for a method you haven't defined. Then go to the browser, implement the method, and proceed from the walkback. (Kent calls this "just-in-time programming".) Smalltalk is keeping track of the order in which you need to do things.
- When the system breaks, don't theorize: put in a breakpoint and step through the code until you see what is wrong. You'll find the answer much faster.

This next one is a little harder. Sometimes someone else will see this before you do.

- Be sensitive to how things feel as you work. If you are slowing down, or something seems to be hard to do, or the methods are getting longer and uglier, Smalltalk is trying to tell you that you are doing the wrong thing. Remember the old joke about "Doctor, it hurts when I do this." "Then don't do that."

  Smalltalk is telling you to take a break. Get up and walk around. Go get some coffee, with someone else. Let them tell you what they're doing. Take some kind of a mental break.

  You know that thing where you're trying to think of something and you can't remember it, and as soon as you start doing something else, you remember? Let that happen. Smalltalk is telling you to take a break. Take a break.

[Home] [XP] [XP Practices Frame]

You're NOT gonna need it!

Often you will be building some class and you'll hear yourself saying "We're going to need...".

Resist that impulse, every time. Always implement things when you **actually** need them, never when you just **foresee** that you need them. Here's why:

- Your thoughts have gone off track. You're thinking about what the class might be, rather than what it must be. You were on a mission when you started building that class. Keep on that mission rather than let yourself be distracted for even a moment.
- Your time is precious. Hone your sense of progress to focus on the real task, not just on banging out code.
- You might not need it after all. If that happens, the time you spend implementing the method will be wasted; the time everyone else spends reading it will be wasted; the space it takes up will be wasted.

You find that you need a getter for some instance variable. Fine, write it. Don't write the setter because "we're going to need it". Don't write getters for other instance variables because "we're going to need them".

The best way to implement code quickly is to implement less of it. The best way to have fewer bugs is to implement less code.

You're not gonna need it!

Code Ownership

We do not practice code ownership. When the classes for some feature are first developed, only one team will typically work on them, during the one iteration it takes to develop them initially.

Thereafter, whenever any team has occasion to work with a class, whether to extend it or repair it, they do so freely. Because we release frequently, there are few conflicts. We have extended ENVY to warn us when we version a class that is not based on the released, and when it does, we integrate our changes into the released version if that isn't the one we started with. (The team who released ahead of us will help us if we need it.)

On the contrary, lack of ownership might cause people to feel less responsible for their work.

- Since the entire group will soon see our work, we become more aware of quality concerns, not less.

On the contrary, the non-owner might not understand the class and break it.

- Since the class must run its unit tests at 100%, changes made by anyone are automatically subject to the quality requirements.
- Pair programming ensures that even if the team changing the class has no experience with it, two pairs of eyes look at the changes.
- Commonly, if the class is complex, one of the original authors is invited to help with changes.

Pair Programming

All significant development is done in pairs. We have found that progress is faster, we can work longer without losing headway, and quality is higher. Typically the person types who has the best feel for where the code is going. The observer catches detail errors (spelling, formatting, method names) and at the same time tends to have a wider overall view of how the development is going.

It is as if the typist is crafting the individual methods, while the observer is monitoring strategy and tiny detail at the same time.

We have found that with very little practice, most people adapt quickly to pair development. They get to like it, because progress is discernibly faster.

From a project viewpoint, the practice ensures that at least two people are very familiar with all the code. This pays off very quickly as teams go on to other tasks.

On the contrary, some complex tasks require concentration and developers need privacy to concentrate.

- We do allow people to prototype on their own. The recommended practice is then to throw the prototype away and develop it anew with a partner. If we develop something alone (such as over the weekend), we will review it with a partner. This is OK if kept in check. Do not think of it as a clever way to avoid pairing.
- If a task is so complex that it requires that much concentration, it is almost certain that we're doing the wrong thing. We have found no valid exceptions to this guideline in our entire project: simplicity has always been possible, and always been better.

On the contrary, I just like to work alone.

- That's fine. You just can't work with us.

    (Alistair Cockburn accused Kent and me of playing "Do it My Way or Get Out". Follow the above link for some discussion of that.)

Pair Programming Screenplay

An exchange recently on comp.lang.smalltalk, though it actually involved three people, showed a good example of how pair programming works. Check it out.

[Home] [XP] [XP Practices Frame]

Do the simplest thing that could possibly work

The most important rule in our development is always to do the simplest thing that could possibly work. Not the most stupid thing, not something that clearly can't work. But simplicity is the most important contributor to the ability to make rapid progress.

We find that we have to remind ourselves of this rule continually. Developers like to develop, and most of us have years of experience in creating a "general solution" to whatever we're asked for. Real progress against the real problem is maximized if we just work on what the problem really is.

On the contrary, if we know we're going to need something we should build it in while we're building the object the first time. It'll save time.

- How can it possibly save time to do more rather than less? The best you can hope for is to break even. A little bad luck, and you'll come out behind.
- When you're thinking about "we're going to need this someday", you're not thinking about "we need this today". You just distracted yourself from your goal. Don't compound the error by chasing tomorrow.
- Software follows an 80/20 rule: 80 percent of the benefit comes from 20 percent of the work. Find that simplest 20 and do it.
- Studies show that developers are not really that good at predicting what will be needed. It's better to wait for a real need, and provide for it then.

Smalltalk code is extremely easy to modify. We do not have to design or build for the future. Progress is fastest if we just do what we need to do now: leave the future to the future.

On the contrary, when I'm immersed in a new object, I may see how to do something that will later be more difficult because I won't be up to speed.

- If the object is so complex that it will be hard to modify later, it is just too complex. Simplify it so that adding new capabilities will be easy, but don't make it even more complex by adding them now.
- Add commentary (a class comment or method comment) describing the key idea. Your mission is to make the fastest progress against what the problem really is, not against what the problem might be or might become.

Switch Teams Around

In general , we have had our best results by switching teams around. There are a few areas where one developer has so much experience or knowledge that they tend to come back to that area whenever there's more to do.

Sometimes this has worked well, usually when it is specialized domain knowledge that's in question. We've had developers who have tried to stick with internal parts of the system, and by and large that works out less well. I suspect that it's because it's hard to build good systems tools if you don't have to use them. In any case, if you have someone who is stuck on some area, it's to your advantage and theirs to move them to something else: you'll get cross-training in their specialty area, and you'll keep everyone fresh.

As a general rule, in every iteration, move the person who has been on the team longer to a new area.

Don't Go Dark.

© 1997, 1998, Ronald E Jeffries
ronjeffries@acm.org
http://www.armaties.com

Unit Tests

Each class must have unit tests. Every class's unit tests must score 100%. On C3, we use Kent Beck's public domain testing framework, augmented with a GUI that runs all the tests and shows the percent correct. At this writing, there are over 1300 unit tests, and they all run at 100 percent.

We recommend that unit tests be written before the class is written. This is a good way to focus attention on what the class is really about. In any case, a class isn't done until its unit tests are in the test suite.

When classes are released, all unit tests must be running at 100 percent. All. That is, if the changes you make break the unit tests for some other class, the problem must be resolved before you release.

On the contrary, what if I'm not the one who broke the test, or what if someone is using my class incorrectly?

- Since the unit tests ran at 100 percent before you released, you did break the tests.
- If someone is using your class incorrectly, why did the tests run at 100 percent when **they** released? Something you changed has caused the problem, because you know the tests ran at 100% before you started to release your code.
- Even if the class is used incorrectly, you need to resolve the problem before you release. Partner with the user of your class if need be.

To emphasize the point: all the unit tests must run at 100 percent, all the time.

The Four Variables
Scope, Quality, Resources, Time

Projects are often given to developers in terms like these: "Take these four people, and get back here in three months with a perfect program". The developers ask "What does it have to do," and are given a huge list.

Often (it may seem like always), the developers kill themselves to produce something, but fall short. Either they don't get done on time, or the program they produce is of low quality, i.e. it doesn't make the customer happy.

Rarely, you get to add or remove people, but this doesn't help much either.

Enlightened managers do it differently: they tell you the task and the time, and let you estimate resources. (Some even give you the resources you ask for.) Or they tell you the resources and the task, and let you estimate the time. (Some even give you the time you ask for.)

This turns out not to work either. You still kill yourself, and you're still not done on time or else the customer isn't happy.

What's going wrong?

There are four variables you have to deal with: resources, time, quality, and scope. Here are examples of how they interact:

- Try to make more time using overtime. This never works. Quality suffers, and you wind up late anyway.
- Stick to your scope and quality goals. You wind up late.
- Try to make good decisions about what to leave out, but deliver on time. Customer isn't happy with what you left out.
- Ask for more resources. You can't have them. You wind up late.
- Ask for more resources. You get them. Overhead buries you. You wind up late.

The end result is always the same: you deliver junk on time, or you deliver the expected program so late that no one remembers you. Either way, the customer isn't happy.

The reality is this: time and resources are probably fixed. Quality goals may be a little flexible, but only within narrow limits. The only thing that can be varied is scope. How can we vary scope to result in a perception of success rather than failure?

Review: the variables

There are four inter-related aspects to a software project:

- Scope - what is to be done; the features to be implemented;
- Quality - the requirements for correctness and other "good" things;
- Resources - the investment of personnel, equipment, and so on;

They're just rules!

Alistair Cockburn and Kent and I have been chatting via email concerning the eXtreme Programming methodology, and a family of lightweight methods that Alistair is formulating. As part of getting to know each other, we have been accusing the other of this or that and then defending ourselves. Alistair said:

> Extreme programming says, Do it My Way or Get Out. My philosophical stance is, Permit the Maximum Personal Variation. (It is not lost on me that the CCC project benefited from the Do it My Way approach.)

This really is not the case. Here's my reply to Alistair, slightly edited:

> . . . I didn't want to let this one lie. XP, to the extent that there is such a thing, doesn't say "Do it My Way or Get Out". Quite the contrary ... the XP-like rules we use on C3 are rules the **Team** chose to live by, and the **Team** says "Do it OUR way, or get out". And within the team's rules, there is individual variation. In some areas (you will write unit tests, you will run the unit tests before releasing), there is very little variation. In others, there's more. But there's never much because the team has an internal social contract to do it the way we do it.

> Kent and I like to talk tough. The XP rules, version C3, evolved as part of the process of getting the C3 team from dead stop to full speed. They learned a lot. I learned a whole hell of a lot. Kent can speak for himself, but my observation is that he learned a lot too.

This is the way of it: the rules expressed so strongly here aren't my rules or Kent's rules. They are the rules that the team embraces. We do require everyone to follow them, because we believe that they make us more effective. When the situation changes, or we get a better idea, we discuss possible changes and often change the rules. After all, "They're just rules" is one of our rules.
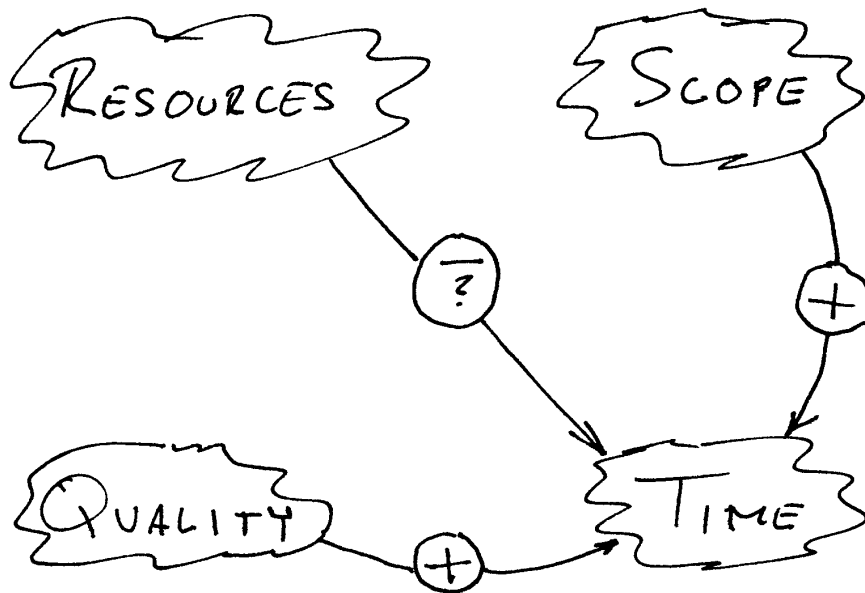

[Home] [XP] [XP Practices Frame]
© 1997, 1998, Ronald E Jeffries
ronjeffries@acm.org
http://www.armaties.com

- Time - the duration of the project.

How are the variables related?

The primary relationships are shown in the picture above:

- Adding Scope (features) makes Time (duration) increase. Do more, take longer.
- Increasing Quality requirements makes Time increase. Do better, take longer.
- Increasing available Resources can <u>sometimes</u> make Time decrease. Rarely.

Reverse relationships hold. If you increase Quality requirements and hold Time constant, then you must add Resources (sometimes) or reduce Scope.

Similarly, if you want to decrease Time, you must do one or more of:

- Decrease Scope;
- Decrease Quality;
- Increase Resources.

Who owns the variables?

Finally, let's examine who owns which variable:

- Scope is owned by the Customer, with some input from Management and Developer;
- Quality is owned by the Customer, perhaps again with input;
- Resources are owned by Management.

If Management and the Customer exercise control over Scope, Quality, and Resources, then Time is completely determined by the dynamics of the development process. It is a

matter of measuring progress and predicting the final result. You hold your nose and do the math.

Conclusions

Learn how to measure and predict your progress. In a context of short iterations with defined goals, this is actually fairly easy. When you predict, measure, and report progress, Customer and Management can make good business decisions based on reliable predictions. And your credibility will be high, because everyone can see what is really happening.

Analyze results, determine what slows you down, and improve the process to help speed up. Don't make assumptions about the effect: measure results again and when there is a real effect, feed back into the predictions. This gives Customer and Management increasingly reliable information as time goes on.

The bottom line is that Customer and Management have direct control over all aspects of the project: Scope, Quality, Resources, and Time. You can adjust Time, for example, by tuning the other variables: and you can do it with high confidence in the results.

The customer can make the project go better by controlling Scope. Management can make the project go better by providing enough Resources and by protecting the team from Time-wasting activities. The developers can make the project go better by working as effectively as possible (the point of the rest of these pages), and by reporting accurate information on the Quality and Scope produced over Time.

You control your project. Make it be what you want, by controlling the variables.

[Home] [XP] [XP Practices Frame]
© 1997, 1998, Ronald E Jeffries
ronjeffries@acm.org
http://www.armaties.com

```
        [stepToRemove isExecutable ifTrue:
           [self list remove: stepToRemove.
            steps remove: stepToRemove]]
```

Let's face it, it's still ugly. A better solution is to refactor, for example:

```
removeStep
   self removeStep: self list selection

removeStep: aStep
   aStep isNil ifTrue: [^self].
   aStep isExecutable ifFalse: [^self].
   self list remove: aStep.
   steps remove: aStep
```

When we feel like breaking the formatting rules to make a method look better, we get better code by reformatting, in almost all cases. Give it a try.

[Home] [XP] [XP Practices Frame]

Wiki-Wiki SQUEAK Site	Ward also hosts the Wiki-Wiki SQUEAK Site: "Squeak is a dialect of Smalltalk with a portable implementation and liberal license terms. A community of Squeak developers and users meet here to share facts and pointers to other Squeak related sites on the web. "
Kent Beck testing framework	Testing Framework This is the core of the testing framework we use on C3. It is in the public domain. Try the framework: you'll be glad you did! On my FTP Site.
VA Testing Framework Port	Jeff Odell has provided a port of Kent's testing framework to Visual Age, together with an excellent writeup of the port. Highly recommended! On my FTP Site.
Squeak Testing Framework Port ? ?	Bill Trost has given us a port of the testing framework to Squeak. Check it out! On my FTP Site.