# Hidden Skills that Support Phased and Agile Requirements Engineering

Ben Kovitz
*bkovitz@acm.org*

## Abstract

*Phased development does requirements engineering in one or a small number of extended phases occurring early in a project. Agile development also does requirements engineering, but in thousands of small conversations spread throughout the development lifecycle. Each depends on subtly different skills and expertise to perform its practices—agile development depending heavily on ability to change working code, phased development depending heavily on foresight.*

*This paper surveys the special skills that each style of requirements engineering depends on in order to promise and deliver.*

## 1. There's no escape

Suppose a customer asks you to build an order-entry system for some new biological-research gizmo that requires extensive and unusual configuration for each unit ordered. Configuration will involve reading computerized lab results, downloading genomic data from the Internet, and combining all this information in complex ways. Neither you nor your team of programmers knows much about this field. How will you and the programmers learn the vast amount of knowledge and customer decisions required to implement the system?

One approach, which we will call *phased development*, is to acquire this information early, in a special phase that occurs prior to coding (aside from prototypes). Hopefully, by stabilizing and understanding this information early, we can make wise design decisions before coding, prevent costly recoding, and deliver a working system by a predictable date. We'll be able to make an agreement with the customer stating precisely what we will deliver and when, and we will know that we can fulfill our part of the agreement.

Another approach, which we will call *agile development*, is to acquire the same information concurrently with coding, in thousands of small conversations [1]. We'll write code with incomplete knowledge, fully expecting to rewrite it as we acquire new knowledge. Hopefully, we will reduce risk by delivering partial but highest-priority business value early. We will greet each coding "mistake" as an opportunity to learn and to allow the customer to change and refine requirements as his own understanding changes or as the business situation changes—leading over time to a system that is always finely tuned to both the current needs of the business and the available resources for software development.

Notice that in both approaches, the development team must acquire the same information and the customer must make the same decisions about the system. Of course the needed information is most commonly called *requirements*, and the job of specifying it and deciding it is most commonly called *requirements engineering*.

The principal difference between phased and agile development, then, is not *whether* to do requirements engineering, but *when* to do it. Phased development does requirements engineering in an early phase that precedes the majority of coding. Agile development does requirements engineering continuously throughout the project. Phased development usually embodies the requirements in a written document, and agile development does not [1, 2].

### 1.1. Promising and delivering

Why does anyone care? As long as you do a good job of requirements engineering, what does it matter whether you do it in a phase or continuously?

The decision matters because it affects what promises you make to the customer, and what skills and expertise you need in order to deliver. Typically (though not necessarily), developers in an agile project promise very small deliverables which they release to the customer once per "iteration"—one to three weeks, on most projects [3]. Beyond the current iteration, they don't offer the customer much certainty about precisely what functionality will be delivered on a given date.

Typically in a phased project, the developers promise a large set of specific features by a specific date. Only the phased approach can do this, because only in the phased approach are the features defined in enough detail for such a promise to be meaningful—combining both a precise delivery date and precise delimitation of scope.

In this paper, we'll examine the special skills that the development team must possess in order to deliver on its

promises predictably and economically. We'll see that success for each style of requirements engineering depends on the developers' possessing very different sets of skills. Some of these skills might be surprising, and most are not currently taught in universities.

While most of the skills are needed in some degree for both styles of development, we'll focus on skills that are pushed to extraordinary levels—levels that would be hard to anticipate without personally experiencing each style of development.

## 2. Skills in agile development

The principal skill in agile development is *modifying* working source code in response to concrete interaction and feedback. In agile development, you always have a live, working system. You make a small change, get the whole system working again, make another small change, and so on. The major difficulty to be overcome through special skills and practices is making continuous modification economical.

Programmers in a great many software shops live in terror of making even small changes to legacy code. "How can I be sure I'm not introducing a bug?" If software is anything, it's complicated. Seemingly minor code changes often have unexpected consequences.

In agile methods, it is not enough for code to be correct. The code must also be modifiable in such a way that we can easily tell if any given modification is also correct. Only in this way can we make continuous code modification economical.

Extreme Programming (XP), the best-known agile method, solves this problem through a variety of interconnected practices. Because the rules of these practices are described in depth elsewhere [4, 5], we'll describe only the most fundamental here, on our way toward understanding the complex skills that the practices depend on.

*Test First.* XP requires programmers to write a failing unit test for each bit of code before writing the code. Thus the software and a highly refined test suite grow together. Coding becomes a fast cycle of making a small change, seeing what tests fail, and making any needed changes to get 100% of the unit tests to pass again. The unit tests function as a project-specific debugger, pointing you to the subroutine that has gone wrong each time a new bug is introduced.

*Refactoring.* Refactoring is the act of changing working code so it still does the same thing but in a way that is easier to understand and modify. Programmers are thus free start with clumsy designs or inelegant code and continuously simplify.

*Pair Programming.* All production code is written by two programmers together: one at the keyboard, one looking over the other's shoulder for errors. Thus each line of code is reviewed as it is typed in. If the "observer" finds code difficult to understand, the "driver" is prompted to change it immediately. Pairing spreads knowledge of the code throughout the team, making everyone familiar enough with the system to modify any part of it.

*Small Releases* and *On-Site Customer.* The development team releases a usable new version of the software once per iteration. Any "user story" (customer-specified functionality) that is too large to implement in one iteration, complete with unit tests, must be broken into smaller chunks. During the iteration, users or domain experts work directly with the programmers, answering questions about details and resolving misunderstandings as they come to light. Decisions made by the on-site customer are embodied in acceptance tests (usually automated, not to be confused with the unit tests).

So in XP, code is made maintainable by the extensive unit-test suite and by continuously reworking the code to keep it simple and easy to understand. New ways to make the code simpler go into the code as people discover them. Gaps and inconsistencies in requirements are ironed out as they become apparent through writing tests.

Through these practices (and a number of others not covered here [6]), XP teams achieve an extraordinarily high degree of both code correctness and code modifiability. In contrast to expectations, the source code does not become more expensive or difficult to change over time. It actually becomes easier, because refactoring tends to break the code into modules tuned to the kinds of requests that the customer has really been making. After a year of delivering, implementing a new user story is often a matter of tying together or slightly modifying a few existing classes.

What does it take, though, to carry out the XP practices successfully? Just reading about the practices in a book is not enough to make a person *good* at them—good enough to make promises to a paying customer and be sure of delivering. A programmer with decades of experience can find starting on an XP project dizzying and upsetting, because his prior experience may not have developed the peculiar skills that XP depends on most heavily.

### 2.1. Breaking big things into tiny things

Typically a programmer or team is given a problem that is too large to solve in a few lines of code: draw molecules on the screen, look up information in a password-protected database on the Internet, etc. Rather than *think* and try to solve the whole problem at once, XP asks developers to find the smallest meaningful thing they could *implement* right now—preferably in a few minutes.

While breaking down large procedures is basic to all forms of software development (for example, "functional decomposition"), XP takes it to an extreme seldom

imagined outside the agile world. The extreme is not just the tininess of the tasks—many taking minutes or even seconds—it's the purpose that they serve in the process.

XP uses tiny tasks, not as elements of a detailed plan to reach a result known precisely in advance, but to explore the unknown in a productive and low-risk way. The programmers often do not know what they will do after completing the current task. They do the current task in order to find out. Each completed bit of functionality teaches them something about the problem, putting them in a better position to see what to do next. The next task may be to ask the customer a question, add a new test case that will "break" the code just written, start a new class, or even totally rewrite what was just written.

This skill of chipping away at large jobs, continuously relying on coding to bring you new insights that you will apply as you acquire them, cannot be taught by simply giving someone a set of step-by-step instructions. That's what makes it a "skill" for purposes of this paper. Experienced programmers who haven't tried XP often find it difficult to think of coding tasks that do not require full understanding of the problem. They find it hard to think of easy tasks, and easy to think of hard tasks.

The reason step-by-step instructions can't substitute for this skill is because each new big task is unique and must be broken down in a different way. What, for example, would make a good first microtask for drawing molecules on the screen? Draw a circle? Draw a graph? Make a text version? Make a data structure for one atom? It depends on the constellation of a thousand factors never to be seen again: your current understanding of chemistry, the graphics libraries available to you, your familiarity with those libraries, the current state of the code, your pair partner's current skills, etc.

Nearly every XP practice calls for breaking big things into things tinier than one might think possible. User stories must be broken into functionality small enough to deliver in one iteration and still deliver business value immediately. To be assigned to programmers, user stories must be broken into engineering tasks small enough to complete in no more than three days yet still perform some meaningful, testable function. Classes continually get broken into smaller classes. Subroutines get broken into smaller subroutines. Programmers need to break large refactoring jobs into tiny refactorings—often changing one line, or changing one variable name. Big refactorings are frowned upon because they tend to wander into long periods during which each partial change cascades into more partial changes that need to be made before the tests can pass again.

Notice that this skill, like all skills, admits of better and worse. This is different from knowledge of a certain fact or procedure: you either know it or you don't. Probably no one will ever hit on the *best* microtask to do next, but that's not necessary. A great many possible microtasks can work just fine as starters. Some are better and some are worse. The more highly skilled the developers, the better microtasks they'll think of, and the more quickly they'll deliver a usable product. Below a certain skill level, though, they'll simply be befuddled when presented with a large task.

## 2.2. Giving up control

In XP, any code that you write, someone else will probably rewrite, since your code will probably give them ideas for how to improve it. You must continually let code develop in ways that you yourself would never do. This lets designs emerge that are better than any individual could have thought of alone.

XP thus requires that programmers *give up control* to an extraordinary degree—a skill that many programmers not only lack, but resist acquiring.

Each member of a pair that works well tends to report that the other person had most of the ideas and did most of the work. The experience tends to be euphoric, and the resulting code astounding in its clarity and simplicity—even to the participants. The pair partners always learn from each other: seniors from juniors as well as juniors from seniors and peers from peers.

But when either partner tries to hold personal control over the code, the pairing experience can be agonizing. The partners argue over every tiny decision. Rather than let the code show them the way in tiny steps, they try to persuade their partner to do it their way. They start offering reasons to consider instead of offering code to improve or tiny tasks to perform.

If you feel personal ownership of the code, if you insist on holding true to an architectural vision invented prior to coding, if you insist on preventing your pair partner from doing anything "wrong", you will block one of the principal mechanisms by which XP creates reliable and maintainable code: the operation of many minds, each examining and improving the products of all the others.

Even the skill of doing tiny tasks is a form of giving up control. You have to code without knowing in advance how things will turn out. Many programmers like solving big problems in their heads. They hate imperfection, and they especially hate letting other people see their mistakes or improve their code. It takes *wisdom* to intentionally allow mistakes to happen because mistakes enlighten more quickly and effectively than argument. It takes wisdom, when you think your partner's idea is stupid, to tell yourself, "Either I am wrong or he will learn," and then say, "Okay, let's try it." Most likely, when you let him try it, you both learn, because a better design emerges than either of you can imagine now. The temptation to prevent mistakes and backtracking is powerful, but the developers must overcome it for XP to work.

Following the XP rules is another form of giving up control. Most arguments between pair partners can easily be resolved by doing a smaller task rather than a larger task, coming up with a test case instead of discussing architecture, separating refactoring from implementing new functionality, agreeing to try both ideas and then re-evaluate, etc. Indeed one of the functions of pairing is to "keep each other honest": to help the other person overcome temptations. Some programmers, though, resist when reminded. Some get angry, some resort to trickery to escape the rules, and some make the pairing experience so unpleasant for their partners that they regain control simply because no one will work with them.

## 2.3. Writing meaningful tests

Many experienced programmers, when called upon to write a test before writing code, find themselves in cognitive vertigo. "How am I supposed to write a test for a subroutine that isn't even defined yet?"

Yet test-driven design is one of the principal mechanisms through which XP keeps code simple and comprehensible. Writing a tiny test case first, and then writing code to pass just that test, leads you to define the simplest interfaces and always implement the bare minimum of functionality. [7] The design tends to become simple, and the unit tests serve as executable documentation of the intent of each subroutine.

Testing is a skill, and it's a skill that many programmers have not acquired. There is no step-by-step procedure to tell you how to make a test case. Rather, through experience, you learn a myriad of ways to make simple, meaningful, deterministic tests. To take one little example, to test code that generates random results (say, in a simulator), you pass the constructor a random-number generator object. The test seeds it with a known value, while production code lets it run indeterministi-cally. GUI code can be especially difficult to unit-test. Over time, you learn a hundred little ways to separate the GUI code from the code that performs the underlying computations. You learn the trick of passing a "mock object" which substitutes for the GUI objects, logging all of the subroutine calls it received. Any kind of interaction with external entities, whether over a network or though a user interface, tends to present problems. Tests for multithreaded code can be deceptive as well as difficult.

Programmers sometimes feel tempted to write tests by pasting the output of the code into the test. Sometimes there's value in that, but programmers can often find ways to make tests more meaningful by looking for other angles from which to examine the output. A subroutine that calculates a mathematical function can test for the intended properties of the function as well as specific known values. For example, if the function is intended to be periodic, the test code can verify the periodicity by comparing the results of two calls. It takes discipline and experience to dig for these other angles.

This skill is so highly varied, it's mostly learned through pairing, which brings us to the next skill.

## 2.4. Conversation

In many organizations, programmers spend most of their hours in solitude and silence. Not in XP.

Rather than relying on written documents, XP relies most heavily on source code and in-person conversation to communicate. It follows that XP depends heavily on social skills. It's not enough to be a good coder. You need the skill of both listening to and making yourself understood to another person. Since disagreements among programmers are common, you need the skill of presenting an opposing idea in a way that doesn't offend the other person's ego. A simple but often neglected skill is knowing when to interrupt and when to let your pair partner finish the current task while you quietly jot down your idea.

While conversational skill is pretty basic to nearly all software development (especially the requirements phase of phased development), it plays an especially important role in XP because the programmers talk directly with the customer. It is through these conversations that XP performs requirements engineering. It is through these conversations that trust between customer and developer is built or destroyed.

The customer, too, must be willing and able to converse. For an XP project to succeed, the users or domain experts must be available to the programmers as well as politically in a position to make authoritative decisions on the spot. A customer who takes the attitude of, "I want to throw the specification over a wall and not hear from you again until you've implemented it exactly as I've written it" undermines the foundation of XP: continuous feedback and small course-corrections.

All of the XP practices aim at supplying people *real* information: the concrete, here-and-now truth of what the code is doing, unexpected difficulties as soon as they arise, and so on. If the customer won't listen well enough to answer questions, or can't be bothered with details, the developers are cut off from business priorities and domain knowledge. Requirements engineering halts, and the viability of the project comes into doubt.

## 2.5. Object-oriented design

Object orientation's primary advantage is that it reduces the amount of context you need in order to understand a given snippet of code. XP draws upon this aspect of object-orientation to give programmers license to change any code at any time—because they can

understand it at a glance, or with very little research elsewhere in the code.

Refined XP code typically consists of many small, simple classes and tiny subroutines. Many subroutines are just one line long, their purpose being to have a simple name that can be used elsewhere in code to express intent. Most or all of the code is uncommented, because the code itself, together with unit tests, communicates the human intent.

Programmers who've never seen code at this level of refinement may not know what XP demands. They may put in large "design patterns" from books where simple solutions would do, undermining the comprehensibility needed to keep the code modifiable. An especially strong temptation is to use object-oriented techniques to write clever code that a reader cannot understand without first understanding a great deal of other code.

Again, the skill of object-oriented design grows through experience refactoring many classes and cannot be taught in the form of simple rules (except for "once and only once" [4, 8]). Most of the skill consists of seeing tiny ideas for improvement rather than overarching architectural ideas—contrary to the intuitions of many programmers. For example, you might pass a "configuration object" to a constructor rather than a long list of parameters. With that in place, you can add new parameters to the constructor without modifying existing code that calls it with fewer parameters. There is no limit to the number of such small, helpful techniques a programmer can learn.

### 2.6. Tools for fast cycle times

Finally, agile development depends crucially on tools that enable fast cycle times. "Ten minutes to green bar" is the XP rule of thumb that says you usually should go from introducing a failing test to getting all the unit tests running again in no more than ten minutes. If compilation takes more than ten minutes, then this goal cannot be achieved. Any slowness in the compiler or development tools will be "leveraged" to slow down every aspect of the project. It would have been difficult to do agile development in the days of punch cards.

Fast cycle times can be undermined in a variety of ways. A geographically dispersed team causes more problems for agile development than phased development, due to the former's greater need for quick, informal conversation. It's difficult to pair effectively when you're not physically in the same room. If the software consists of multiple versions on different branches in the source-control system, it may not be feasible to refactor code in small bites. Each refactoring would occur on only one branch, resulting in confusion for programmers moving from one branch to another for different jobs.

## 3. Skills in phased development

The principal skill in phased development is *foresight*—accurately predicting the future without concrete interaction and feedback (except prototypes). The only feasible way to make plans without feedback from actual construction of the product is by interacting with a *representation* of the product or the jobs involved in constructing it. Thus the key to making phased development work is to have a trustworthy representation.

The representation, whether in paper or in software, must allow you to perform operations on it that are analogous to actually doing the construction. You must be able to ask questions of it and it must give you answers that accurately match the reality to come when real construction begins.

Planning and estimating with representations is commonplace in non-software forms of engineering. Electrical engineers draw schematic diagrams, architects and civil engineers draw blueprints, mechanical engineers draw orthographic projections. Even figuring the capacitance of a circuit on a hand calculator is manipulation of a representation.

All of these work because the elements of the representation map in a simple and direct way to components or measurable properties of the constructed artifact. A transistor symbol on a schematic diagram maps to a real transistor in the product. Looking just at the diagram, you can tell whether the circuit will do a job that you can precisely define in advance. You can even tell how much the transistor adds to the cost of the product.

Software development presents a special difficulty for the representation approach. The product—a computer program—is itself a description, and indeed the vast majority of software development consists of inventing appropriate terms and structure for the description. Electrical engineers don't routinely conceive of new kinds of components or invent new ways of describing them. For computer programmers, the crafting of *ways of describing* is daily work.

What makes a computer program likely to be bug-free is not physical materials, but the faithfulness of its representations (data structures and procedures) to both the reality they model and the cognitive needs of the programmers who create and verify them.

A requirements document or functional specification is thus a representation of representations yet to be created. It describes properties of descriptions yet to be written. This brings us to the first and perhaps most basic skill of phased development.

### 3.1. Technical writing

Writing a natural-language requirements document calls for technical writing at the highest skill level. To

serve as a basis for phased development, the descriptions it contains must have numerous attributes, such as completeness, correctness, non-ambiguity, and so forth, as set forth in standards such as [9]. We'll examine some of the standard attributes, plus a few more.

First, the document must be lucid enough that readers can easily spot gaps or internal contradictions, if there are any. To trust the document, the text must be so lucid that readers can be confident that if they see no gaps or contradictions, the reason is that there really are none.

The document alone must do this job, without the aid of trying to implement its descriptions in executable code. If gaps or contradictions emerge in later phases, during coding, then the document has failed.

Second, the document must be interpreted exactly the same way by both programmers and customer. If there is any dispute about what belongs in the software or what doesn't, the developers must be able to resolve it by looking in the document and not asking the customer. In fixed-bid contracts, the document exists partly to protect the programmers against the customer interpreting their agreement to include more development, partly to protect the customer against the developers interpreting the agreement to include less. In these projects, the document must settle every possible question about scope in advance of coding. (In in-house projects, non-ambiguous writing is not quite as important, as long as the ambiguities don't significantly affect the schedule.)

Third, the document must be readable enough that the developers and customer actually read it. In many projects, programmers skim through requirements documents or ignore them completely. Often customers don't read requirements documents. Many people find reading requirements documents boring and difficult. If they have a question or want to say something, they prefer to speak in person. This is quick and easy. In face-to-face conversation, people can ask, get an answer, and put the answer in their own words to confirm that they've understood correctly. The document must communicate so clearly that two-way conversation is not required—or preferred.

To put this third property another way, the document must *communicate* roughly as well as in-person conversation [10]. When you ask a person a question, you don't need to look through a table of contents or index, or piece together little clues from all throughout a document. You can ask your question in speech that comes to you naturally, and the other person can compose an answer from the totality of his knowledge, tailored to just your question and your needs—in a few seconds.

All of the above properties are achievable with a high level of technical-writing skill. Like writing tests first or breaking big classes into small classes, the skill of technical writing consists of a wealth of knowledge of different approaches, good judgement in choosing an approach for the job at hand, and ability to invent new approaches for each new situation.

Many requirements engineers do not possess this skill. Some look down upon technical-writing skills, such as framing clear sentences, saying one thing at a time, giving examples, structuring the document to fit the audience, and so on, as "mere secretarial work" or irrelevant to their jobs. Informing a system analyst that people are ignoring his work can cause professional humiliation. Consequently an analyst's poor technical-writing skills may go unaddressed for years or permanently.

Fourth and finally, the requirements document must be completed quickly enough to enable development to start—and deliver—in time for the product to provide maximum business value.

Writing a useful requirements document is possible, but it takes time. It takes a great deal of time and thought to understand a large software system. There is much more to the job than merely collecting statements from the customer. The requirements engineer must find a way to cognitively structure the information so that people can easily understand it and easily see gaps or internal contradictions. Only when something is well understood can a person express it lucidly, unambiguously, readably, etc. The time taken for so much forethought may itself endanger the schedule even if the resulting requirements document renders coding a breeze.

Given the above difficulties, many practitioners of phased development have evolved a compromise approach. They make the requirements document a summary of conversations between the customer and the developers. Rather than making the document so refined that it can introduce a new programmer to the project on its own, or covers every detail, the document assumes that its readers participated in the elicitation. This enables people to write the document with enough accuracy to help the programmers do their work, without demanding a huge investment of time or exceptional technical-writing skills. It adds some risk to the schedule, and makes development contingent on the programmers' remaining with the team until delivery, but these risks may be acceptable in many projects.

### 3.2. A battery of proven methods

To serve phased development, a requirements document must be backed up by programmers who can translate the requirements into a working product in known time. This means that the programmers must be able to determine which components and subcomponents they'll need to build, and how long each will take—with enough accuracy to calculate development time and resources for a project lasting several months or perhaps more than a year before deployment.

Because "construction" of software is really much more like drawing a blueprint than constructing a building, such precise foreknowledge of development tasks and time is rare. Often programmers work on tasks they've never encountered before. The majority of their time is spent thinking and understanding, not typing in source code. Once they've found a description that fits the task well, they can code it quickly. Before they've found such a description, predicting the time required for the job is difficult.

In software, once a form of description is understood especially well, like a grammar or a GUI widget, someone automates it with a standard component or language. This changes the task into using the new tool, and because this can be done faster than coding from scratch, it raises customer expectations of software developers. This brings us to our next skill.

### 3.3. Negotiation

In practice, the requirements phase is really a negotiation phase. The customers and developers try to find a set of features that they can implement within a time that delivers acceptable return on investment for the customer. Especially in phased development, the developers can feel a great deal of pressure to make promises that they can't keep. The intangibility of software tempts them into thinking they can do more than they really can [11], and many customers leverage this optimism into one-sided contracts.

So, during the requirements phase, the requirements engineer often finds himself in the role of negotiator, trying to find a realistic schedule and present it in a way that is acceptable to the customer. Many requirements engineers are familiar only with the technical side of software development and are not prepared for the kind of gamesmanship that occurs in negotiation with a savvy customer. Yet this negotiation may be the single most important factor in the success of the project. In larger organizations, politics often overrides technical factors, and phased development puts the requirements engineer into the thick of politics—whether he knows it or not.

Developers who know when to say "no"—who can state bluntly that they lack the kind of proven know-how needed to implement a certain component in a known time—provide a crucial aid to the negotiation process. However, many developers feel a strong temptation to always say "yes" lest they appear less competent than their co-workers or competitors.

### 3.4. Advance prioritization

Finally, for successful phased development, the customer must be able to prioritize details far in advance of delivery. The requirements engineer needs to guide the customer early to see all problem-domain details that may become relevant later so the customer can prioritize them before major design decisions are made. The customer must take responsibility for these decisions, and for the costs, in both time and money, for changing them later.

For example, during elicitation the customer might state that one of the computations he needs the software to perform requires data from a database on the Internet. It's up to the requirements engineer to ask how often the data changes, how often the computations are performed, etc., since these have great impact on what kind of design architecture will best fit the customer's needs. Changing it late may undermine hundreds of design decisions and significantly delay release of the software, possibly throwing the customer out of business. The ability to anticipate and prevent such problems grows only from years of experience with real software development.

## Acknowledgements

Thanks to Asim Jalis for endless brainstorming and war stories, and to Daniel Berry for some much-appreciated encouragement.

## References

[1] Ron Jeffries. "Natural Documentation," *XP Magazine*, October 7, 2001.

[2] Ron Jeffries. "Essential XP: Documentation," *XP Magazine*, November 21, 2001.

[3] Ron Jeffries. "What's in it for the business?" *XP Magazine*, September 30, 2001.

[4] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.

[5] Martin Fowler and Jim Highsmith. "The Agile Manifesto." *Software Development*, August, 2001.

[6] Anonymous. Wiki page: http://www.c2.com/cgi/wiki?ExtremeProgrammingCorePractices, as of June, 2002.

[7] Kent Beck. *Test-Driven Development by Example*. Forthcoming.

[8] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[9] "Recommended Practice for Software Requirements Specifications," IEEE Standard 830-1998, June 25, 1998.

[10] Martin Fowler. "The Almighty Thud," *Distributed Computing*, November/December 1997.

[11] Tom DeMarco. *Why Does Software Cost So Much?* Dorset House, 1995.