Among many of my Oracle database administrator (DBA) friends, “agile” is widely regarded as a dirty word, a synonym for “sloppy.” However, adopting the principles of the Agile Manifesto (specifically, the implementation of the Agile Manifesto called Extreme Programming, or XP) has radically improved the commercial and technical success of projects that I've worked on. Agile principles have enriched my entire life—not just professionally, but personally. The contradiction between the typical DBA's perception of “agile” and my own is profound. This paper describes my experiences with Agile values and my implementation of them. I describe the circumstances that have led me to believe passionately that it's XP that will best assure the success of my projects. I describe what has worked for me and why, and I describe what hasn't worked and why.

**AGILE AS A JOKE**

True story:

True story: That’s what “agile” is in the Oracle DBA community: a snarky code word for sloppy, undisciplined behavior, sanctioned by some goofy manager who doesn’t know what he’s doing. When one of my DBA friends mentions an “agile” project, here comes a fun story

**TABLE OF CONTENTS**

1. Agile as a Joke .................................................1
2. Agile Changed My Life ........................................2
3. My Introduction to Agile .....................................2
4. What is Agile? ................................................3
5. What is Agile Not? ............................................3
6. Incremental Design ...........................................5
7. Rapid Iteration ................................................8
8. Pair Programming ...........................................11
9. Test-First Programming .................................12
10. The Parable of the Tool Chest .........................15
11. What Hasn't Worked ........................................17
12. Conclusion and Summary ...............................19
13. Bibliography and Further Reading .................19
14. Acknowledgments .........................................20
15. About the Author .........................................20
16. Revision History ...........................................20
about a project ruined by shortcuts and stupid decisions. And probably a Dilbert cartoon.

AGILE CHANGED MY LIFE

In spite of the “A”-word’s bad reputation among my colleagues, Agile software development principles have enriched my life tremendously. I’m practically as enthusiastic as a person could be about it. So, there’s the conflict. On the one hand, so-called Agilists systematically ruin projects attended to by some of the most talented DBAs and consultants in the world. On the other hand, Agile principles have profoundly enriched my life—not just my work life, my whole life. What is the basis for this contradiction? Is it reconcilable? That’s what I aim to answer in this paper.

I’m not trying to convince you that you should “do Agile.” There are plenty of circumstances in which I would implore you not to; I’ve listed a few of those at the end of the paper. However, I believe that the bad reputation of Agile principles and practices among my friends is a mostly unfair result of misunderstanding. I believe that executing my favorite Agile practices would actually improve discipline on even some of the most disciplined software development projects that I hear about. My goal here is to tell you what I’ve found Agile to mean and to describe a few of the benefits that adopting Agile practices have brought into my life. From there, you can decide how you feel about Agile for yourself.

MY INTRODUCTION TO AGILE

I didn’t realize it at the time, but my introduction to Agile principles came from reading Eli Goldratt’s The Goal. The Goal is an instructional book about the science of system optimization. The style is odd for a technical book: it is written as a novel, complete with a plot, a hero, and chapter-ending cliffhangers. This story style makes The Goal a fun book to read, which is good because its message is so important. The text of The Goal does not, to best of my knowledge, contain the string “agile”, but its lessons are synchronized with the principles of Agile software development.

The Goal and Theory of Constraints

The Goal illustrates Goldratt’s theory of constraints (TOC), which he describes in technical detail in another book called Theory of Constraints. Goldratt’s method for optimization begins like this:

The first step is to recognize that every system was built for a purpose. We didn’t create our organizations just for the sake of their existence. Thus, every action taken by any organ—any part of the organization—should be judged by its impact on the overall purpose. This immediately implies that, before we can deal with the improvement of any section of a system, we must first define the system’s global goal; and the measurements that will enable us to judge the impact of any subsystem and any local decision, on this global goal.

—Eli Goldratt
Theory of Constraints, page 4

This single purpose—the system’s global goal—runs core throughout The Goal as Goldratt teaches how to optimize a system. The clarity and focus of The Goal helped to inspire the work that Jeff Holt and I did to create the book Optimizing Oracle Performance, which also teaches the importance of keeping focus upon your overall purpose—again, the system’s global goal—when you optimize Oracle-based software.

Extreme Programming

The pursuit of clarity and purpose was my mindset when I read Extreme Programming Explained, by Kent Beck. Before XP Explained, I had never really learned how to optimize a project. Beck begins by defining the global goal of software development in a manner that I personally honor and respect:

[The goal] is outstanding software development. Software can be developed at lower cost, with fewer defects, with higher productivity, and with much higher return on investment.

—Kent Beck
Extreme Programming Explained, page xxi

Throughout the remainder of the book, he describes how to optimize the process of software development to meet that global goal. Every software development practice that Beck examines—and then either includes in XP or omits from XP—is judged by that practice’s contribution to the global goal. Extreme Programming Explained is a Goldratt-style optimization of the software development process. Beck’s analysis of how
to develop software better contained exactly what I was looking for: clarity and purpose.

*Extreme Programming Explained* slotted everything I love about developing software into place in my mind—it fortified my understanding of practices that had worked well for me—and it taught me how to fix what I had hated about developing software by clarifying the sources of the problems I had experienced. Beck’s work kindled and then helped fulfill the hope that my software development project teams could become more focused, more effective, more efficient, more fulfilling, and more fun.

I think that ultimately, *Extreme Programming* has mushroomed in use and interest, not because of pair-programming or refactoring, but because, taken as a whole, the practices define a developer community freed from the baggage of Dilbertesque corporations.

—Jim Highsmith

*History: Agile Manifesto*

It's funny how people who love Agile and people who hate Agile can unite behind at least one common idea: we are all secure in our knowledge that Dilbert proves our point.

**Agile Manifesto**

Studying XP led me to the *Manifesto for Agile Software Development* (the *Agile Manifesto*) and the twelve *Principles* upon which it was formed (shown next page). The Agile Manifesto was written in 2001 by seventeen “representatives from Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming, and others sympathetic to the need for an alternative to documentation driven, heavyweight software development processes.”¹ Kent Beck, representing XP, was one of the authors. The Agile Manifesto frames the overall purpose—the global goal—of software development against which agile practitioners judge the effectiveness and efficiency of the software development practices we execute.

I have reproduced both the Agile Manifesto and the twelve principles behind it, here in this document for your convenience (page 4). These declarations should give you the information you need to figure out whether agile software development is something you want to consider learning more about.

---

¹ See [http://agilemanifesto.org/history.html](http://agilemanifesto.org/history.html).

**WHAT IS AGILE?**

So then, *Agile* is a label applied to a set of values and principles. Values and principles are ideas that guide you. *Practices*, on the other hand, are how you implement those values and principles. They’re what you *do* every day.

It’s easier to agree on values and principles than to agree on practices. For example, my friends might agree that the Agile Manifesto sounds good in theory, but the practices they see people implement in the name of Agile are horrific. To have a constructive discussion, about “Agile,” then, let’s define some practices and see if we agree on those.

![It's easier to agree on values and principles than to agree on practices.](http://agilemanifesto.org/images/conversation2.jpg)

**WHAT IS AGILE NOT?**

A fun way, I think, to get the conversation moving is to give some examples of practices that people may call “Agile,” but that certainly are *not* sound software development practices. These are some of the reasons that “Agile” is a dirty word among my friends:

- Some people believe that being Agile means that you don’t write a project specification.
- Some people believe that being Agile means that you don’t do design (for example, that you don’t do data modeling when you’re developing a database application).
- Some people believe that being Agile means that you can skip whatever software development project steps you just don’t like doing.

If these practices were all I knew about Agile, I would choose sides against agility, too.

The points are great conversation starters about Agile methods, because they force into the daylight some of the key friction points that people must resolve when they consider “going Agile.” Each point raises legitimate questions about software development processes. However, if you consider the Manifesto and
## Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others to do it. Through this work we have come to value:

- *Individuals and interactions over processes and tools*
- *Working software over comprehensive documentation*
- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan*

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck, Mike Beedle, Arie vanBennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

© 2001, the above authors. This declaration may be freely copied in any form, but only in its entirety through this notice.

## Principles behind the Agile Manifesto

We follow these principles:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
Principles shown on page 4, you can see that the values and principles of Agile software development do not inevitably lead to the abhorrent practices that some people attribute to “Agile” methods. Therefore, I reject the idea that these poor practices should be allowed to define what “Agile” is:

Agile is not the opposite of disciplined.

In the remainder of this paper, I will focus on four specific practices from XP that convey what I consider to be the most important distinctions between agile software development methods and the traditional methods that my DBA friends would consider to be safer and more disciplined. The four XP practices that I shall discuss are:

• Incremental design
• Rapid iteration
• Pair programming
• Test-first programming

INCREMENTAL DESIGN

The worst design mistake I’ve ever made went something like this:

1. I write an English-language specification, replete with words and pictures. I add more and more details, more and more features, because—you know what, I’m king!—and I can spec anything I want. Configuration options: they’re in there. Antigravity: it’s in there. If I can dream it and then describe it in English or draw it with Visio, it’s in there.

2. People build it.

3. The software matches the spec exactly. ...Except of course wherever the spec was self-contradictory, but even then, the people who built the software did an exceptional job of reconciling the contradictions, because the people who built it are great at building software.²

4. ...But the application ended up being not really what I wanted, so I don’t want to use it.

I’m ashamed to admit that I’ve made the same mistake more than once.

Big design up front (BDUF)

The project I’ve just described is a big design up front (BDUF) project. The BDUF approach has burned me many times, but it hasn’t always burned me. Sometimes BDUF has been exactly what I needed. Here’s an example...

A few years ago, I built a walnut sewing center for my wife. Most of my shop projects are design-it-myself projects, but for this one, I bought a ready-made plan. The plan consisted of drawings (representing the design), a bill of materials, and a sequence of work steps. This plan was my BDUF. The project went well. The design was fantastic; my wife loves using it. The prescribed bill of materials and work steps allowed me to complete the project within a reasonable amount of time, using only intermediate-level woodworking skills. I completed the project far more quickly, to a higher quality level, and in far less time than if I had created the design myself. It was a thoroughly successful BDUF project.

What about this project made BDUF the right approach? The key aspect of this project that called for BDUF was the existence of a ready-made plan that other people had written, tested, and debugged. (You can still see the debugging process in action if you visit the plan’s reviews.) Instead of spending dozens of hours designing something I didn’t know enough about, I exchanged $13.95 for the pre-tested plan: a tremendous bargain!

In contrast, many of the software projects that I’ve designed, and that I’ve seen other people design in the BDUF manner have been like the story about my worst

² The same thing happened with the crew who framed my home, by the way. The blueprint—a stock blueprint that had been tested hundreds of times, but that had been customized for my wife and me—contained a contradiction. Since my framer (who was great at building houses) is not capable of framing out M. C. Escher’s Relativity, he had to use his own good sense and a little input from me about how to resolve the contradiction.
design mistakes at the beginning of this section. Too many features that made sense on paper before the build didn’t make enough sense once we used them. I cringe to imagine the features that I might have designed into a sewing desk.

What is incremental design?

Agile principles reminded me that the goal of computer software is not to match a specification; it is to solve a need. Big design up front operates under the assumption that you’re able—early in a project—to accurately determine a customer’s needs,5 but in the vast majority of products I see that aren’t delightful, the problem isn’t the code; it’s the spec!

In an XP project, design is a process that continues throughout the project. It’s not an isolated phase that ends before building commences, and it’s not a process executed by specialists who “design and dash”—who create a design and then move on to another project. In XP, you design what you know how to design when you know how to design it. This type of design is called incremental design.

The advice to XP teams is not to minimize design investment over the short run, but to keep the design investment in proportion to the needs of the system so far. The question is not whether or not to design, the question is when to design. Incremental design suggests that the most effective time to design is in the light of experience (Beck 2005, p. 52).

What’s the difference between incremental design and no design at all?

Incremental design can look just like no design. That’s a problem. How can you tell whether someone is doing incremental design or no design at all? Or is it all just a word game?

...it’s not just a word game. The most important practice in any design process—no matter how you do it or what you call it—is that the designer should test.

3 In Agile parlance, customer is a role. It refers to the project team member who is authorized to determine the priority of all the feature and bug-fix requests competing for your project team’s resources.

4 It is possible, of course, that the project team did try their design, saw the horrible flaws, and recommend a change to their leadership, and that their leadership made the business decision that changing the design would not deliver sufficient value to justify the expense. After all, it’s a business decision, which is optimization with a more global scope.
needs to be a part of your project plan. Ideally, this experience comes in the first person. Excellent designs can come from second-person experience, too, but only if (1) the designer genuinely understands how the design will work in reality, and (2) the designer feels a personal and social responsibility to do right by the ultimate users of the design.

I suspect few housing projects in the U. S. were designed by architects who expected to live in them. You see the same thing in programming languages. C, Lisp, and Smalltalk were created for their designers to use. Cobol, Ada, and Java were created for other people to use. If you think you are designing something for idiots, odds are you are not designing something good, even for idiots.

—Paul Graham
Hackers & Painters

Think of the big design mistakes you see around you. Is the design broken because one of the following statements about the team who built it is untrue?

• The team is required to test its own creation, and
• The team is enabled to make changes to the design before going live with it.

Incremental design admits that the best design emerges from actual use of the design, and it engages the power of the designer’s own social and personal responsibility to improve the experience for the people who will use it.

Incremental does not imply undisciplined

Conventional wisdom says that if you don’t know exactly what your code needs to do, how it needs to do it, how its interface needs to look on the front end, and how its data need to be structured on the back end, then you have no business trying to write it yet.

What I have learned by practicing incremental design is that I can make better designs, less expensively by having a process in place to allow design evolution to occur. Incremental design is admission that it is at least possible (if not downright inevitable) that some aspects of my design will be found deficient at some point in my project. When that happens, you’ll have two choices: (1) fix the design deficiency; or (2) press forward with design deficiency in place. If your development practices prohibit you from fixing the deficiency, or—worse yet—if it is politically or culturally impossible for your organization even to admit that a design can have a deficiency, then the best you can hope for is a system that works in spite of its design flaws.

Responsiveness to change is a competitive advantage. One of the changes you have to plan for is the evolution of your customers’ understanding about what they really need. Incremental design means that you should not pretend that you know how to design something before you have sufficient information to design it well.

Is incremental design too expensive?

Design changes are expensive. Everybody knows it...But so is moving forward with a flawed design. The constraints can be difficult, even overwhelming:

Example: You’ve never done this project before. Maybe you’ve done projects like this, which certainly helps, but building general ledger systems doesn’t completely prepare you to build a payroll system. So you don’t have a big design up front. ...Well, you may have what you’re calling a design, but it has never been implemented before, so it has never been tested and debugged. Any complex design that has never been tested and debugged is approximately 100% likely to contain flaws. The expensive team that created the design (and the expensive approval committee that approved it) have long since “left the building.” They’re no longer part of the project team, so making a formal design change and getting it approved is highly unlikely for the remainder of the project.

You need a process that helps you out whenever you find a defect in the design. Lots of programming teams work around design deficiencies and never communicate to their leadership when they’re doing it. They have deadlines to meet, after all. The more defects there are in the design, the more likely the project will be late and over budget. It’s not just a software development problem. Google for the words “late over budget,” and you’ll see lots of “real” engineering projects (like roads and bridges and weapons and such) that had big upfront designs and that have gone outrageously late and over budget.

How you’ll react to a design flaw comes down to economics. How much does it cost to refine or redesign? How much does it cost to push onward with a flawed design? How much does it cost to spend extra
time up front trying to ensure that the design doesn’t have any flaws? The economically responsible decision is to choose the option with the greatest cost-benefit advantage. Don’t just assume that the cost of redesign is “infinite.” It’s not. Sometimes the cost of redesign is well worth the benefits of doing it.

Ways to reduce the redesign cost

By far, the most important way to reduce the cost of redesign is to keep the iterations between design decisions and design validations short. I’ll cover that idea in the section on rapid iteration.

Good software tools also make it easier to reduce the design debt of your project. With the right IDE, you can refactor code quickly in ways that would have been unthinkable in the 1980s, and with the right testing tools, you can make those changes safely (I’ll say more about testing shortly, too.)

Database redesign is often more expensive than other types of code redesign. For example, fixing a bad data model can become a start-the-whole-project-all-over-again kind of experience. I think this is why so many Oracle DBAs so aggressively resist agile practices, and it’s why so many experienced leaders insist on BDUF for their data models. It’s why a project team with insufficient understanding of the relational model puts a database application project at grave risk.

However, the expense of redesign doesn’t change the fact that you’re going to learn about deficiencies in your database design throughout your project, no matter how carefully you plan upfront. You need to dedicate some of your upfront planning to making your design flexible enough that you can respond to unplanned circumstances.

For example, if your design distributes your SQL code throughout your PHP code, then renaming a column will be outrageously expensive: you’ll have to change the name of the column in potentially hundreds of files. But if you design your database dependencies into a PL/SQL data access layer, you’ll prevent many code refactoring problems, in addition to preventing many system performance problems (and even many automated testing problems, which I’ll discuss later). If you don’t have the staff who know how to do a fantastic job designing and writing that PL/SQL data access layer, then it’s a significant risk to your project.

One problem is that the software market for database refactoring tools is immature. There aren’t many tools that can help. The situation is improving, though. Oracle is building new refactoring tools to help with data modeling, versioning, and unit testing in its Oracle SQL Developer product. Red Gate has recently released new software tools to help with schema and data comparison mechanics. These are steps in the right direction. The market will inevitably continue to improve the ease with which we can both design and improve those designs.

Reconciling BDUF and incremental design

There are smart people on both sides of the big-design vs. incremental-design argument. Each approach has its place. I view the situation as a continuum:

1. If you have access to a BDUF for your specific project that has been tested and debugged, then use it. Examples: a sewing machine cabinet, a tract home, the rewrite of a product that has been refined already in production.

2. Otherwise, if you have tested and debugged BDUFs for components of your system, then use them, but since you haven’t tested and debugged the integration of those components before, you’ll probably need to allow for incremental design of the integrated whole. Examples: a customized or fully custom home, a payroll system written by a team experienced in large non-payroll systems.

3. Otherwise, you’ll need to design incrementally. You may create a BDUF as part of the project (especially if you want your project to be reproducible), but you won’t have trustworthy design until after much testing and refinement takes place, which by definition, is incremental design. Examples: the iPod, the Wright Flyer, the Apollo space program.

I believe that virtually all successful design is incremental; it’s just a matter of how you define your observation interval. During my sewing machine cabinet construction project, I did very little incremental design myself, because I bought the big design up front. However, the plan I used was really the result of an incremental design process. It’s just that those increments mostly concluded before my project began.

Rapid Iteration

Do you know the very worst kind of software there is? It’s software that is 90% complete, but that still doesn’t run yet. By “doesn’t run yet,” I mean literally that nobody can experience—see and feel—a runnable version of the software. Software like this is what causes those projects that are two years late and 350% over budget. They’re what result in big software deployments where users never see anything until their first day of training, where people discover (1) that the system looks nothing like they’d hoped, and (2) that the system is supposed to accommodate a department of 400 people isn’t scalable enough to accommodate the 20 people in the training course this morning.
You know the second-worst kind of software? Software that doesn’t act the way you want it to, and that’s not going to change until late next year.

These two problems are what make rapid iteration important. Rapid iteration means that you create runnable new releases frequently. It means to release a few new features and bug fixes earlier than many people are used to, rather than waiting until all the new features and bug fixes are complete. Rapid iterations for a conventional build-and-ship type of software company might mean monthly or biweekly releases. Rapid iterations for a software-as-a-service company might mean weekly releases or even continuous deployment, which means that software is put into production immediately as source code is checked into the source code versioning system.\(^5\)

The basic idea [of XP] is to get something valuable into production as soon as possible.

—Pete McBreen
*Questioning Extreme Programming*

You can’t do incremental design without a working product. Every minute you don’t have working software is a minute that you cannot do incremental design. Agile practices thus emphasize making software that runs from the very first phase.

Rapid iteration is the reflection of two essential facts:

- Design is the hardest part of a project.
- Long feedback loops kill.

Design can be horrendously difficult; yet unfortunately, it’s vital. Design literally defines what your product will become. I’ve seen far more software projects fail because of inferior design than because the programmers writing the code couldn’t write code to match a specification.\(^6\)

The easiest way for an architect to cover his lack of confidence in a design is to delay its release. “What we need is more time to think it through.” It’s intellectually comfortable to delay releases, but delaying releases actually compounds your troubles, because it lengthens your feedback loops. It worsens the delay between decisions and the events that inform you whether those decisions were good or bad.

Long feedback loops kill. Literally:

- Imagine what your hands would look like by now if your central nervous system required 30 seconds to register to your brain that your hand was being burned by a hot stove.
- Read the story of the [Radium Girls](http://www.wolframalpha.com). Imagine how their story would have changed if touching their paint brushes to their lips had caused bleeding and an immediate burning sensation.
- Imagine how much more difficult it is to remotely control the [Mars Rover](http://www.wolframalpha.com) with a controller round-trip latency of 38.70 minutes\(^7\) than it would be to remotely control a similar vehicle in your living room.
- Imagine how long it would take you to learn how to putt a golf ball if feedback about whether you’ve hit it left or right, short or long, required 7 days to reach your brain.
- Imagine how long it would take you to fix a fundamental software design flaw if a project team of 100 programmers had worked for more than a year writing code using the flawed design as a specification.

Think of the projects you’ve seen that were “90% complete.” What was missing? For me, the following reasons have accounted for nearly every almost-but-never-quite-finished project I’ve seen:

- “We’re done with all the features, but our [insert infrastructure element here, like ‘build script’ or ‘test harness’ or ‘installation wizard’ or ‘final integration’ or ‘porting team’] isn’t finished yet.”
- “Our most important feature isn’t done yet. We can’t release without it.”
- “We’re done with the application. Now we have to figure out how to make it scale to the size of our user base.”

If you’re several months into your software project, and you’ve never once been able to run that software, then you’re probably on the road to building an application that everybody hates. The chances of your project producing something delightful are vanishingly small.

---

5 As you’ll see shortly, any such check-in must not take place until all of the code’s tests pass.

6 Conversely, some of the most delight-inspiring products of our generation—the iPod, the iPhone, and the iPad—are delightful, famously, because of their design.

7 On 2008-04-26T16:44:00-0500, the distance between Mars and Earth was 19.35 light minutes. Information courtesy of [http://www.wolframalpha.com](http://www.wolframalpha.com).
It takes practice to structure a project so that rapid iterations make sense. Here are some guidelines that have worked well for me:

- The build infrastructure comes first. I must be able to compile and test my software using a script from the very first day of the project. No real code until push-button compiling and testing is in place.
- The first code that gets written is a stub that does very little, but it does compile, it does create executable software, and it does pass the regression test suite (even though the very first regression test suite says that the code really doesn’t do much yet).
- Any feature that is estimated to require more than 40 hours of labor gets decomposed into smaller, testable features.

Reducing design cost while building something unexpectedly delightful

Rapid iteration enables incremental design. The essential argument for incremental design is that doing design any other way is just too expensive. Agile critics say, “If your BDUF is defective, it just means that you didn’t do your design job properly up front!” I urge you to consider a different angle: How long can you afford to spend thinking about how a complex system should work, when you could be testing it and earning experience with it instead of merely theorizing about it?

Getting good at rapid iteration stacks the deck in your favor. Factor your project so that you can verify your design in small, economically affordable steps. Don’t invest months’ worth of risk into a design between opportunities to test the design in real life. The goal is not to avoid design changes; it’s to avoid big design changes. You avoid big design changes by never biting off a chunk of your design that you can’t afford to change. So then, segment your project into steps that allow you to put working software into production frequently. Consider the idea that a product that doesn’t do everything you envision can still provide value to the customer.

Example: The picture here represents a BDUF project with a single deliverable that will occur some time in the reasonably distant future. The design includes four features, represented by the red, purple, gold, and brown boxes.

However, we were able to factor our project so that it delivers the red feature right away, in just a few weeks.

<table>
<thead>
<tr>
<th>features</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td></td>
</tr>
</tbody>
</table>

Incremental design project, iteration 1

While the business used our red feature, we added the purple feature, which we released when it was ready. The business used the red and purple features while we worked on the gold feature, which we released when it was ready.

<table>
<thead>
<tr>
<th>features</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>purple</td>
</tr>
</tbody>
</table>

Incremental design project, iterations 2 and 3

That’s when the magic happened. As the business used the red-purple-gold product, it becomes abundantly clear that a new pink feature should replace the purple feature. So we build it.

<table>
<thead>
<tr>
<th>features</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>purple</td>
</tr>
</tbody>
</table>

Incremental design project, iteration 4

The use of the red-purple-gold product informed the design in an unexpected way. Here, the actual experience of using the application gave birth to a design element that nobody expected when we were thinking about the BDUF. The unexpected idea created an unplanned—and I would argue unplannable—benefit that the BDUF would have prohibited.

The project has now diverged from the alternative-universe BDUF plan, but only because the divergence created superior business value for the products users. Using the red-pink-gold product informed the designers that a red-pink-blue product would be better. Using red-pink-blue in the real world created the awareness that the green feature was what the business really needed next.

<table>
<thead>
<tr>
<th>features</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>purple</td>
</tr>
</tbody>
</table>

Incremental design project

A business that does BDUF would have thought that the red-purple-gold-brown feature set was the way to go. However, by the same date in the project plan as the BDUF would have delivered its first version of the application, the business that did incremental design knew that red-pink-
blue-green is a better way to maximize profit, cash flow, and return on investment.\(^8\)

Notice the following:

- In an incremental design project, users can derive value from the using product features very early in the project.
- In an incremental design project, the earliest implemented features (like the red feature) have a chance to undergo considerable debugging and refinement by the time version 1.0 of a fully implemented BDUF product can be released.
- An incrementally designed product (red-pink-blue-green) can be substantially different from what the BDUF product would have been (red-purple-gold-brown). The differences are motivated by real-life business requirements that are difficult, if not impossible, to envision before accumulating real-life experience using the product.

It takes practice to build software incrementally, and for many project leaders, it takes a whole new mindset. Many people cannot envision anything short of a software project’s total completion as having any value to a customer. But it’s often possible to build something simple that works, that creates value (possibly even tremendous value), and that informs the next design step in ways that you could never have envisioned merely by thinking more carefully about the product in the beginning.

A complex system that works is invariably found to have evolved from a simple system that worked…. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system.

—John Gall
The Systems Bible

PAIR PROGRAMMING

Through my first nineteen years of school, I only worked alone. I never perceived that working together with my peers offered me a significant advantage. This was my culture growing up. I read, then, with great interest and fascination, Kent Beck’s prescription to write code in a configuration that he called pair programming. The idea that programming could benefit from working with a partner seemed absurd to me, but Beck’s description of pair programming was so compelling that I tried it. Here is what I learned:

Two people working together deliver higher quality results and get more things done than two people working separately for the same duration.

It sounds crazy. Two people, working on one thing, together, accomplish more than if they worked separately, on two things. I’ve proven it over and over to myself. It happens like this: loner tendencies build up; I work dozens of hours by myself; I get stuck; I pair; dam breaks; ideas flow; code and tests result. Through pairing, I’ve accomplished tasks that seemed impossible. Pairing has delivered into my life some of the best times and best memories that I’ve created at work.

...Talking and working go together on an XP team. The hum of conversation is a sign of health. Silence is the sound of risk piling up.

—Kent Beck
Extreme Programming Explained

Here are some of the specific benefits that pair programming has provided me:

- Two people are smarter than one. One types, the other one—the wingman—watches. Usually the person typing isn’t thinking as well as the wingman. Sometimes, what the team really needs is the question, “Why are we doing this at all?” The person at the keyboard, continually executing specialized tasks, is much less likely to ask that. Sometimes, there’s an easier, smarter way. It’s easier for a wingman to see that.
- Two people are more tenacious than one. When you’ve burned tremendous cognitive energy trying to “maintain stack” in your mind as you debug, and you feel like even after hours of work, you’re only going to be 2% done, your spirit can fade. Sometimes you need a wingman to say, “C’mon, we

\(^8\) These are Goldratt’s optimization constraints, from The Goal, for a business.
just have to push through this,” before you can take another bite of the elephant.

- Two people are more careful than one. When your mental and physiological energy reserves are depleted, it’s easy to just add the feature and move on without creating the test. It’s like workout discipline: when your workout buddy’s out of town, it’s easy to skip a day. But when your pair partner is looking over your shoulder, you write the test, because it’s the right thing to do. And because somebody’s watching.

- Two people are safer than one. Again, especially when the person typing is tired, the wingman is essential for pulling the emergency brake and saying things like, “Stop: don’t commit that!”

- Two people design better than one, and two people write code better than one (or—in my experience—than three, or four, or five, ...).

- Having at least two people be intimate with all the code your company writes is also a valuable form of redundancy.

There ain’t no rules around here. We’re trying to accomplish something.

—Thomas Edison

My staff and I pair on about 20% of the programming time we invest into our commercial products. We have done it enough that we have a good feel for when it’s appropriate to pair and when it’s appropriate to focus in solitude on a problem. Whenever we’re paired, if a distracting little voice inside my mind says that what we’re doing is so straightforward or so tedious that I shouldn’t be keeping my colleague from doing the other work on his schedule, then we split up and work separately.

We pair predominantly in the following situations:

- When we’re inventing a new feature, especially one that changes the user interface.
- When we’re blessing new tests (that means “changing our design”) to implement changed product specifications.
- When we’re developing code that we know will affect other people on the team.
- When we know that someone else on the team is particularly expert in the type of code we’re going to be writing.
- When we’ve written a lot of new code that nobody else has seen yet (the standard code review context).
- When we’re stuck.

TEST-FIRST PROGRAMMING

One of the most appealing principles behind agile software development for engineers and managers alike is principle #10:

Simplicity—the art of maximizing the amount of work not done—is essential.

A popular interpretation of that principle is that you don’t have to write a specification:

Extreme Programming expert Bob Martin said at the 2001 XP Universe conference that he ran into someone who said his organization was using XP. Martin asked him how the pair programming was viewed, and the reply was, “We don’t do that.” Martin asked how refactoring was working out, and the reply was, “We don’t do that.” Martin asked how well the planning game was working, and the reply was, “We don’t do that.” “Well,” Martin asked, “then what are you doing?” “We don’t document anything!” was the answer [Paulk 2002, pp. 15–18].

Not writing a specification is a reasonably certain way to inflict death upon just about any project. However, there are many ways to write a specification. Microsoft Word, Powerpoint, and Visio aren’t the only ways.

For all the obvious benefits that a detailed specification document affords us, there are actually several disadvantages:

- Someone has to write it.
- Someone has to maintain it. Every time the design evolves a bit (which I hope your project is flexible enough to allow), your specification document has to be changed to keep up.
- Someone has to read it. How do you know if your code meets the specification? Someone has to (1) read the specification, (2) run the code, and (3) compare to find out. This has to happen every time something in the code, the design, or the specification changes.
- Specification documents don’t have to make sense. Since a specification document is written in a natural language (like English or Russian), it can contain self-contradictory statements. It can violate the laws of physics. A problem that is far less dramatic but just as debilitating: a specification can...

---

9 I’ll show you why this is important in a moment.
simply be vague and open to ambiguous interpretation.

I have known about regression testing for a long time. But until I began studying XP, I had always considered it a luxury item, something that a company like Oracle Corporation could afford but that I really couldn’t afford to do myself on small projects. That study of XP turned my mind around. Now I consider the regression test suite essential to any project I do, and the natural language specification is the luxury item. For most of the products I work on, all we need is a test suite and a user guide. My ultimate goal is that our application design is so good that we don’t even need the user guide (like the user guide I don’t need for my iPhone).

XP prescribes the practice called test-first programming. The software development method literature these days more often uses the term test-driven development (TDD). Same thing. Creating a regression test suite is not the easiest thing you’ll ever do, but it’s far simpler than I used to think. It’s one of those skills that just takes practice. The benefits are tremendous:

• Before you commit any change to your source code management system, you regression test everything first. That way, every time you check in a code change, you know that your software still works to specification. ...Try that using a group of human testers interpreting the behavior of your code using a 300-page English specification.

• You don’t need as many writers dedicated to keeping a specification document up-to-date with every evolution in the design.

• You don’t need as many testers dedicated to making sure that every little change in the code still matches the specification document.

• Your programmers don’t have to spend so much time talking to not-so-technical writers and testers to keep them updated with the latest changes in the code.

• The specification is black and white, recorded in a formal programming language (not an ambiguous natural language). There are no gray areas subject to interpretation.\(^\text{10}\)

• It puts more pressure upon the product design team to create only sensible specifications.

• When a code change flunks a test, of course it’s not what you had hoped for, but the failed test is a remarkable clarity-inspiring tool for the mind of a programmer.

• The feeling you get by pushing that button to run your test suite and seeing your tests all pass is priceless, whether you’re a programmer, a designer, an architect, an owner, or a customer. If your test coverage is complete, then the “Result: PASS” that you get is a guarantee that your product performs to specification.

There are two possible outcomes: if the result confirms the hypothesis, then you’ve made a measurement. If the result is contrary to the hypothesis, then you’ve made a discovery.

—Enrico Fermi

As quoted in Nuclear Principles in Engineering (2005)
by Tatjana Jevremovic

Test-first programming works in my company like this:\(^\text{11}\)

1. Add a case. Each new feature or bug fix begins with a case that we add into our bug tracking system.\(^\text{12}\) This case is written in English. Cases are how we communicate our intent within our team.

2. Add a test. Each case we choose to implement begins with a test. This test must inevitably fail because it is written before the corresponding feature has been implemented.

3. Run all tests and see if the new one fails. This validates that the test harness is working correctly and that the new test does not accidentally pass without requiring any new code.

4. Write some code. This is the code that will cause the test to pass.

5. Run the automated tests and see them succeed.

6. Refactor the code. Now the code can be cleaned up as necessary. The tests, to the extent of the existing

---

\(^{10}\) Of course, you can program shades of gray into your test suite, if you want. For example, it’s just as easy to test whether a result matches /this\s*string[!?\.]i as it is to test a match for “that\s*string.”

\(^{11}\) Adapted from Wikipedia: “Test-driven development.”

\(^{12}\) My company uses FogBugz.
Test-first development delivers massive technical, cultural, and psychological benefits. It’s the biggest payoff, in my opinion, in all of Agile. It’s also one of the most difficult agile practices to do well. Whenever you hear someone refer to “Agile” as a “lightweight methodology [sic],” you can bet that the person doing the talking isn’t doing test-first programming.

If you want to do test-first programming well, here are some of the first things you’ll need to do:

• Decide how your build script will evaluate whether your code meets your specification. Some of the best help for me has come from understanding the Test Anything Protocol (TAP) and exploiting the power of the Unix `diff` command. You’ll have to write code into your build script that will do this evaluation automatically. Much of what you need is available free-of-charge, in open-source form, but it might still take considerable investment to implement your practices.

• Decide how extreme your code coverage requirements are and how granular your tests need to be. We like 100% code coverage for our lowest level utility functions, at least 75% code coverage for the text and numerical data that lie immediately beneath our graphical interfaces, and we use people to test our graphical interfaces, working from a natural language user guide where necessary. The need to use humans to test human interfaces is one of the prime motivations for the extremely agile software development practice of continuous deployment famously advocated by Eric Ries.

• Create a culture in which programmers create tests for features they add and for bugs they fix. Testing is simply a new habit that programmers need to establish. Like any new habit, it takes time to feel comfortable with it and practice to get good at it. The habit has clicked only when programmers design and debug their tests with the same high level of care and discipline with which they design and debug their code. (Tests are, in fact, just code.)

• Create software tools that allow you to identify and explain the differences between your program’s output and its spec. The tools have to make doing this so quick and easy that programmers will actually use them.

• Pair your programmers to make tests. Wingmen design better tests and provide far better coverage than the programmer who is the principal author of the code being tested.

In almost all the work I do now (except for GUI work), my test suite is my spec. That test suite is the Rule of Law that determines whether our code meets its specification. Thus, our tests are just as valuable of an asset as our code, and I cannot imagine doing a commercial project without them. The cases in our bug tracking system that describe those tests help us remember the details of our specification, and they help us communicate those details to others, but the tests are the law. I will write an English specification document only if it’s in our economic best interest to do so. For internal projects, it’s generally not.

So that’s my priority system:

1. Tests are the Law that define how our product must work. We maintain our tests just as diligently as we maintain our code.

2. Cases are necessary to track the bugs and features we haven’t implemented yet, so that we can prioritize our work. When we’ve implemented a case, the text of the case helps us document our product changes for our customers to see.

3. We decide on a project-by-project basis whether a user guide (man page, etc.) is necessary.

4. We decide on a project-by-project basis whether an English specification document is necessary. When it is, we generally use a wiki.

Not writing the specification document is a privilege that I have to pay for by honoring my obligation to write cases and write tests. To not write the cases and the tests would be dangerous and reckless. It’s the kind of behavior that gives Agile the bad name that my friends joke about. I go further:

I believe that not writing cases and tests is dangerous and reckless even if you do have a formal BDUF-style specification made of pictures and natural language.

Here are the two primary obligations I must honor:

• I must write tests either before I write my code (as the name “test-first programming” implies), or at least while I write my code. As onerous as it sounds, this is not too much of a habit change. As I develop code, I always run the code I’m writing on as many interesting input values as I feel demonstrates sufficiently that my code in fact works. To “write a test” simply means that I have to persist these executions in a file that will execute in the future every time someone tests the product.

---

13 *Method* and *methodology* are not synonyms. For illustration, see Common Errors in English Usage or this essay by Peter Klein.

14 We use FogBugz for this, too.
• I must complete the product build and test cycle before every code commit. I cannot commit code to the repository unless the code passes all the tests. Committing code to the repository that doesn’t pass all the tests (that breaks the build) is a sin against the project.

It takes time to learn how to do these things, and doing them can be difficult. What do you do when your code doesn’t pass all its tests yet, so you’re not supposed to commit, but your colleagues desperately need you to commit your utility module so she can get on with her work? What do you do when it takes three minutes to fix an urgent bug for someone, but creating the corresponding test will take another hour, and you have other deadlines you’re desperate to meet? How do you create a test for a tool that emits 100 megabytes of data per execution?

..I didn’t say Agile is easy.

Your team might decide that you need to commit right now, even if some tests do fail, but you commit to a branch, not the trunk. You might compromise test coverage for your schedule, but you make yourself a case in your bug tracking system that you can prioritize and manage so that you’ll remember that you owe the project a test. For code that performs very complex transformations upon its input data or that creates very large output, you might write software tools that allow you to “bless” the output of a program as a new specification.

Over the course of nearly a decade of doing test-first programming, my teams have written, refined, and rewritten—that is, we’ve incrementally designed—software tools that help us remove friction from our test-first practices.

THE PARABLE OF THE TOOL CHEST

A few years ago, I wrote a paper called “Measure Once, Cut Twice” that illustrated the problem of too much upfront design and the elegance of allowing your construction experience to inform and improve your design. I used an actual experience from my wood shop as a metaphor. (I build furniture as a hobby.) Many aspects of that story are appropriate here. What I learn in my shop is often metaphorically applicable to my work. I’ll tell the story; you should decide for yourself.

The design, BDUF style

I wanted a chest to store the many small tools that I keep in my shop. I was aware of, for example, H. Gerstner & Sons’ fine tool chests for machinists, but I needed a different design for a variety of reasons. So I decided to create my own custom design. I knew better than to start cutting wood without a plan, so I spent dozens of hours on my computer, making drawings over the period of a couple of months, designing my new tool chest.

When I thought my design was perfect, I started cutting. I built the chest. As soon as I finished building it, I began using it. When I began using it, I immediately began discovering flaws in my design. The flaws were so critical that I decided to build version 2.

I had built exactly what I had designed, what I thought I wanted. I had executed my design perfectly enough, but I wasn’t happy with the result. What I really wanted was what I thought I wanted. I learned far more in ten minutes of having my hands on the physical chest than I had learned in two months of studying a paper design.

In my experience, this happens all the time. When I build something that nobody has ever built before (even if it’s a variant of something quite common, like a tool chest), I don’t know whether the design is going to work until I actually try it with my hands. With software, the problem is even worse. The gap between what I think I want and what I learn that I really want, once I get my hands on it and begin using it, is routinely chasmic.

I find that, while designing things from scratch is quite difficult, I’m quite good at redesigning things that I’ve used. Judging from all the fighter jets whose designations end in B, C, D, E, F, and G, and all the people I know who refuse to run x.1 versions of software in production (and the software companies who refuse even to name their versions x.0), I am not alone.
What distinguishes a really nice tool chest from just an okay one? If you don’t spend much time using tool chests (or designing them for others to use), then I’ll bet that I can guess your number one answer: the first thing you’ll say that distinguishes a really nice tool chest from an okay one is the material from which it’s made. Many people will probably tell you that they’d much prefer to own a chest made of solid oak than a chest made of plywood. That’s the first thing I thought of, too....Until I actually used one. The characteristics that I learned distinguish a really nice tool chest from an okay one are completely different. I’ll come back to that later.

When you build your first tool chest in oak, you’re making a bet. You’re betting that the first tool chest you make is going to be so good that you’re going to want to keep it. I’ve learned that this is almost never a good bet for me. As I’ve already mentioned, my first tool chest had design flaws; some were bad enough that I didn’t even want to keep the first chest I made.

Oak is a moderately expensive material to buy. The material cost is a problem for some people, but even if the material cost isn’t a problem for you, there’s a much bigger problem in store for you if you choose oak—or any hardwood for that matter.

The most important reason not to build your first tool chest out of hardwood is that it’s going to cost you a lot more time and labor than working with a man-made material. You can’t buy hardwoods in thin, flat panels that are the sizes you need to make the carcass of a tool chest. To make a 12”×18”×½” oak panel, you (or your specialty supplier) have to resaw and thickness plane oak planks that are an inch or thicker down to ½”-thick stock. Then you have to joint and plane the edges. If your planks were less than 12” wide, then you have to edge join the planks to form a 12” wide panel.

If you had chosen ½” fiberboard, like MDF, which is strong and flat, all you have to do is cut your 12”×18” panel out of a 24”×48” handy panel that you can buy at The Home Depot for less than $10. A big problem of working with oak on version 1 of a tool chest is that you spend too much time dealing with the difficulties of working with oak instead of focusing on the real problem: the question of whether you are building something you’ll actually want to use!

Of course, the big problem you perceive beforehand of not working with oak on version 1 is time pressure. You don’t feel like you have the time to do it twice, so you work with oak on version 1. Because what if you build an MDF tool chest in version 1, and someone decides to put that into production? Embarrassing, right?

The fact is, it takes a lot longer to make an oak tool chest than it takes to build an MDF one. Working nights and weekends, making time for my family responsibilities in the midst of it all, it takes me about three months to build an oak tool chest. It takes me about one month to build an MDF one.

Here’s the reason that knowing this is so important. Imagine that I’m destined to be disappointed in my design. My own experience shows that there’s about a 90% chance that this will be the case. Then if I build in oak, it’s going to take me three months before I learn where my design flaws are. It’s going to take me six months (two builds) before I have a tool chest that I really like. (That’s if I get it all right on version 2, and don’t have to build a version 3.)

If I build version 1 out of MDF, then I learn after just one month that I don’t like my design. And I now have a functional tool chest that I can use for the next three months while I build my version 2 chest out of oak....And do you know what I found out? MDF painted in a complementary color to my countertop is an even better solution for me than oak! Priorities change, as you become better informed by using your own design.

The problem with my plan

What are the real characteristics that distinguish a really nice tool chest from just an okay one? The material—whether it’s MDF or oak or mahogany—seems like a big deal up front, but once you actually use your tool chest, you learn that you stop noticing what material it’s made of very soon after you begin using it. The things that become more noticeable instead are:

- Are its outer dimensions suitable for your workspace? Or is it unexpectedly awkwardly large and constantly in your way now that you’ve put it where it’s going to sit for the rest of your life?

- Are its inner dimensions well matched to the things you need to store in it? If the drawer depths are too shallow to hold some of your tools, you leave the “I have a new tool chest” experience feeling like you wish you didn’t have this one. This is a difficult one to pick up on paper, unless you have accurate three-dimensional scale drawings of all the tools you’re planning to store in there.

- Do the drawers slide in and out smoothly? Sticky drawers prohibit the tool chest from giving you a pleasurable experience when you use it. How the actions—anything describable with verbs instead of nouns—are going to feel is very difficult to envision when you’re looking at a plan for a tool chest, but the quality of an action is incontrovertible when you feel it with your hands.

- Does your divider strategy work nicely? The whole point of a tool chest is its utility in helping you keep
the things inside it organized. Organized means separated, and separated means dividers. The best divider strategy is flexible; it allows you to choose where the dividers go as you buy new tools or decide to reorganize the ones you have, and it allows you to change those decisions later.

- Does it have sufficient capacity? Is the usable inner volume big enough to store all your stuff?

The big problem for me was the overall capacity. I loaded my tool chest full of an incredible volume of tools, which provided an immediate benefit. But I had more tools that I wanted to keep in there, and I could see that because of the way I had designed my drawer glides, I was wasting 21% of my interior case volume. It hadn’t looked like a problem on paper, but seeing it now on my workbench, it was a big problem.

At this point, I couldn’t have cared less what material the case was made of. If it had been made of oak instead of MDF, it would have just made matters worse. I wouldn’t have learned about the problem for another two months, and it would have been more difficult to discard. The very fact of having chosen nicer materials for the version 1 chest would have made me more likely to settle for something that was thoroughly un-delightful to use.

Putting something into production earlier allows you to evolve your design sooner.

- Making decisions that quicken the release cycle (like choosing easier-to-use materials) helps you put versions into production earlier.

The concept of prototyping is a bridge between agile software development principles and traditional big-design-up-front project management principles. If you’re building something that you’ve never built before, then don’t count on your first version being one you’ll want to keep.

WHAT HASN’T WORKED

Agile software development projects fail, too. Here are some of the problems I’ve seen.

No customer

One of the worst problems you can face is an antagonistic relationship between the customer and the team designing and building the product. In agile parlance, the customer is a role of a person on the product development team. It may not be the actual paying customer (just like the designer of a spacesuit may not be an astronaut), but it is someone who is CRACK: collaborative, representative, authorized, committed, and knowledgable (Boehm and Turner). The absence of a full-time on-team customer with all of these attributes severs the necessary connection between the project team and the business purpose of the software they’re trying to create.

Designs made in a vacuum usually suck.

—Mogens Nørgaard & James Morle

Too many customers

On the other hand, too many customers can be just as bad of a problem as not enough. Remember, the word customer in the agile parlance doesn’t refer to the headcount of people sending revenue to your company; it refers to the project team member who is authorized to determine the priority of all the feature and bug-fix requests competing for your project team’s resources. Systems designed by committees seldom have the elegance or usefulness of systems designed by just one or two people.

Lessons learned

Clearly, this is a story about the importance of prototyping a product that has a new and unfamiliar design. Here are some points that I think survive the metaphorical relationship between this woodworking story and computer software development:

- Your design goals before you use something are different from the design goals you’ll wish you had specified after you get your hands on the product. In other words, experience changes your understanding of your needs.
I don’t know the key to success, but the key to failure is trying to please everybody.  

—Bill Cosby

Cultural mismatch

When you use agile practices—if you do it right—you learn about project failure areas sooner rather than later. It takes courage to learn that your ideas were destined not to work, but if they were, it’s far better to learn you were doomed after $20,000 of investment than to learn it after $20,000,000. If your culture is not willing to accept the truth when the truth is inconvenient, then agile practices are not going to work for you.\(^1\)

You should be glad that bridge fell down
— I was planning to build thirteen more to the same design.

— I. K. Brunel
Addressing the directors of the Great Western Railway

Bureaucratic practices evolve in organizations committed to bureaucratic principles. Agile principles directly contradict bureaucratic principles. At the root of the contradiction, bureaucratic principles dictate the centralization of command and control, whereas agile principles dictate that command and control be distributed. Bureaucratic practices dictate the processes that people may use to create a work result; agile practices dictate that the work result is more important than controlling the means to create it.

Agile principles and practices cannot thrive in a bureaucratic culture, for two key reasons:

• Bureaucratic culture tends to attract and retain a distinctive type of staff skills mixture that does not work well with agile practices.

• Bureaucratic culture tends to attract and retain a distinctive type of leadership style that conflicts with agile practices.

Also, if your centralized leadership can’t bear to loosen its grip on every technical implementation detail in your product, then pretending to use agile practices will create an environment of hypocrisy that nobody good will want to be a part of.

Staff skill mismatch

Alistair Cockburn has written about three levels of listening, which is a useful categorization of staff skill for determining how plan-driven your project needs to be:

• Level 1 people can reliably perform a task only if all the steps are described in detail.

• Level 2 people have grown beyond Level 1 and can experiment with variations in the rules.

• Level 3 people are able to revise a method (break its rules) to fit an unprecedented situation.

Barry Boehm and Richard Turner recommend that if fewer than about a third of your project team are Level 3 experts, then your project probably needs more centralized control than agile practices can provide. Bureaucratic practices in general and plan-driven methods in particular are intended to allow project teams to get by with larger numbers of lower-cost staff, like those that Boehm and Turner designate as “Level 1B”.\(^1\)

Plan-driven methods can succeed with a smaller percentage of talented people [Boehm & Turner, p52].

Projects that are stacked with larger numbers of Level 3 people can get by with less safety netting.\(^2\) I believe that agile methods are less likely to impede outstanding people from doing outstanding work.

\(^1\) Neither is any other method, but that’s not the subject of this paper.

\(^2\) Boehm and Turner’s Levels 1A and 1B further refine Cockburn’s Level 1-2-3 hierarchy: “Level 1B people are average-and-below, less-experienced, hard-working developers. They can function well in performing straightforward software development in a stable situation. But they are likely to slow down an agile team trying to cope with rapid change. ... Level 1A people can function well on agile or plan-driven teams if there are enough Level 2 people to guide them. When agilists refer to being able to succeed on agile teams with ratios of five Level 1 people per Level 2 person, they are generally referring to Level 1A people.”

\(^2\) However, please see Dietrich Dörner’s The Logic of Failure for examples of how the abundance of self-confidence in groups of Level 3 people creates unexpected jeopardy in projects like the Chernobyl disaster.
What distinguishes people who are ten times more effective than the norm is not that they work ten times as hard; it’s that they use their creativity to come up with solutions that require one-tenth of the effort.

—Jason Fried & David Heinemeier Hansson

Rework

Disrespect

Finally, I’ll mention another problem that will sink a project; not just an agile project, but any project: lack of mutual respect among project team members, their customers, and their leadership. If you are a member of a culture that tolerates references to users as users (losers) or developers as developers or managers as managers, then I would encourage you to escape. If you have been infected with the disrespectful thinking that results in using terms like these about the people for whom and with whom you make software, then I would submit that you are not maximizing your own potential.

CONCLUSION AND SUMMARY

Your fondness or hostility toward agile software development methods will depend upon your core values about “political” subjects like freedom, respect, motivation, and talent. I have no intention of changing your attitude toward those subjects. However, for people whose values are similar to those expressed in the Agile Manifesto and the twelve principles behind it, I hope that I’ve provided some evidence that, if you’ve had really bad experiences with “Agile” projects, then it’s because of the specific practices you’ve seen implemented, not with the whole movement itself.

I believe...

• that excellent process is the result of excellent talent, not the other way around.
• and that agile values, principles, and practices help make the best use of excellent talent.

Here are the main ideas I’ve tried to convey in this paper:

• Agile is not a synonym for sloppy. The practices that people call “Agile” often aren’t.
• Some agile practices require extraordinary discipline.
• Plans fail. There are ways to prevent a failed plan from failing your project.
• Using something yields far more information about design than thinking about it. Using it—even after having thought profoundly about it—can change your understanding of your own needs.
• Incremental design reduces waste by permitting some design questions to go unanswered until a later phase when the answers become considerably less expensive (and less risky) to obtain.
• Rapid iteration maximizes the value of incremental design.
• Creating frequent releases of runnable, valuable software requires practice, dedication, and—for experienced BDUF project managers—mindset change.
• Pair programming can deliver remarkable result-to-time performance and reduce project risk.
• Automatic tests that implement your specification in machine executable form reduce risk and the need for disposable documentation.
• Excellent tools are critical to doing agile projects well. Without good tools, redesigning is too expensive, releasing is too expensive, and testing is too expensive. Some of the tools you can buy; many you’ll have to build or assemble yourself.
• Agile is not for everybody. However, I am convinced after a decade of experience with agile practices that “Agile” indeed is for me.

BIBLIOGRAPHY AND FURTHER READING

“The Agile Warrior” at https://agilewarrior.wordpress.com/tag/sp/ A blog about creating valuable software. It is an active discussion forum about Agile values, principles, and practices.


A remarkable business book that has the same kind of focus, honesty, and uncompromising purpose as Goldratt’s and Beck’s work.
large projects that lead to the need for expensive redesign.

ACKNOWLEDGMENTS
Thank you Ron Crisco, Ken Ferlita, and Robyn Sands for your assistance with this paper. Additionally, thank you Ron for your project management mentoring, which has inspired me. Thank you Jeff Holt and Harold Palacio for your unending help with everything I do. Thank you also to Eli Goldratt, Kent Beck, Jason Fried, David Heinemeier Hansson, Eric Ries, and Steve Johnson for your inspiration.

ABOUT THE AUTHOR
Cary Millsap is well known in the global Oracle community as a speaker, educator, consultant, and writer. He is the founder and president of Method R Corporation, a small company devoted to optimizing software performance. Method R offers software tools, education courses, and consulting services that help people make their Oracle-based systems faster.

Cary wrote Optimizing Oracle Performance (O’Reilly), for which he and Jeff Holt were named Oracle Magazine’s 2004 Authors of the Year. He received the Editor’s Choice award for best paper of the ODTUG 2010 international conference, and he is published in CACM and ACM Queue. He is an Oracle ACE Director and a founding member of the OakTable Network. He blogs at http://carymillsap.blogspot.com, and he tweets as @CaryMillsap.

REVISION HISTORY
2011-06-10 Initial publication.
2014-01-29 Major copyediting, cleanup, and revision.
Kelly’s 14 Rules and Practices

Kelly’s rules got their start on the XP-80 project in 1943, but it wasn’t until the early 1950’s that they were formalized and set in place as the Skunk Works’ rules of operation (from http://www.lockheedmartin.com/aeronautics/skunkworks/14rules.html).

1. The Skunk Works manager must be delegated practically complete control of his program in all aspects. He should report to a division president or higher.

2. Strong but small project offices must be provided both by the military and industry.

3. The number of people having any connection with the project must be restricted in an almost vicious manner. Use a small number of good people (10% to 25% compared to the so-called normal systems).

4. A very simple drawing and drawing release system with great flexibility for making changes must be provided.

5. There must be a minimum number of reports required, but important work must be recorded thoroughly.

6. There must be a monthly cost review covering not only what has been spent and committed but also projected costs to the conclusion of the program. Don’t have the books ninety days late and don’t surprise the customer with sudden overruns.

7. The contractor must be delegated and must assume more than normal responsibility to get good vendor bids for subcontract on the project. Commercial bid procedures are very often better than military ones.

8. The inspection system as currently used by the Skunk Works, which has been approved by both the Air Force and Navy, meets the intent of existing military requirements and should be used on new projects. Push more basic inspection responsibility back to subcontractors and vendors. Don’t duplicate so much inspection.

9. The contractor must be delegated the authority to test his final product in flight. He can and must test it in the initial stages. If he doesn’t, he rapidly loses his competency to design other vehicles.

10. The specifications applying to the hardware must be agreed to well in advance of contracting. The Skunk Works practice of having a specification section stating clearly which important military specification items will not knowingly be complied with and reasons therefore is highly recommended.

11. Funding a program must be timely so that the contractor doesn’t have to keep running to the bank to support government projects.

12. There must be mutual trust between the military project organization and the contractor with very close cooperation and liaison on a day-to-day basis. This cuts down misunderstanding and correspondence to an absolute minimum.

13. Access by outsiders to the project and its personnel must be strictly controlled by appropriate security measures.

14. Because only a few people will be used in engineering and most other areas, ways must be provided to reward good performance by pay not based on the number of personnel supervised.