My Case for Agile Development

Whitepaper based on original presentation by Cary Millsap, commissioned by Red Gate Software

DEPLOYMENT SUITE for Oracle

© Red Gate - www.red-gate.com

redgate
ingeniously simple tools

My Case for Agile Development

This article will cover the following agenda:

1. Agile and me
2. Five practices from XP
3. How Agile has helped me
4. What has *not* worked
5. Discussion

Agile and Me

This section begins with a joke and it's a true story about a joke. Six or seven friends and I were sitting around a dinner table one night in Denmark a few months ago–we'd known each other for a really long time–enjoying a drink before the food got there. We'd ordered already, it took 15 extra minutes or so, we expected we'd be eating by now but the food hadn't arrived. Finally the food started to arrive and none of it was right. The first guy said, "Hey, I ordered French fries not mashed potatoes," and the second person was complaining, "My meat is overcooked, I ordered it rare." The waiter, confronted with so many mistakes started apologising to us at the table and told us not to worry, he'll get it fixed as soon as he could. One of the database administrators at the table piped up with the comment "Ah, they must be doing Agile in the kitchen."

That's what Agile is to the database administrator community that I'm a part of and it's basically one more thing that developers and database administrators have to fight about. The relationship between database administrators and developers in the north of the universe is not always the smoothest in the world and I assume it's probably similar in other product disciplines. But my job is to take DBA communities and developer communities and mash them together so that they can cooperate more fully, and that's really the spirit in which I offer this today. Fundamentally, as mostly a developer, but a developer who has been immersed in the database administrator community for the last 20 years, I really truly believe Agile has a lot of good to it, but a lot of database administrators don't see it because the way it is presented to them makes it completely disgusting for them.

The first realisation that I'd like you to contemplate with me, and this is my own realisation in running a business as a software development company, is that change is for the most part unpredictable, it's absolutely inevitable, it's multidimensional, it's complex and it's a number of other things that make dealing with change really difficult.

However, responsiveness to change is an advantage.

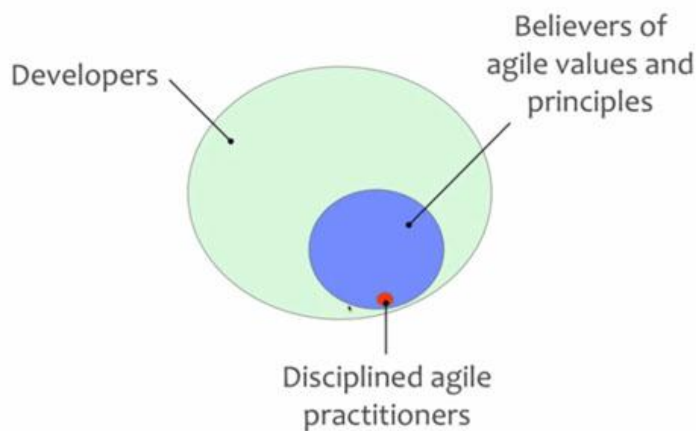
Now a lot of people don't like change, they think that their job is to prevent change from occurring, but as a business leader change is something that you need to embrace and it's something that you need to be able to handle well. There are all sorts of reasons that things change and I'll talk about a lot of them as we go through the paper, but being able to respond to that change is an advantage.

The third realisation and this is about a ten year old realisation for me, is that traditional design build methods were just not working for me, there were several problems which I'll explain in due course.

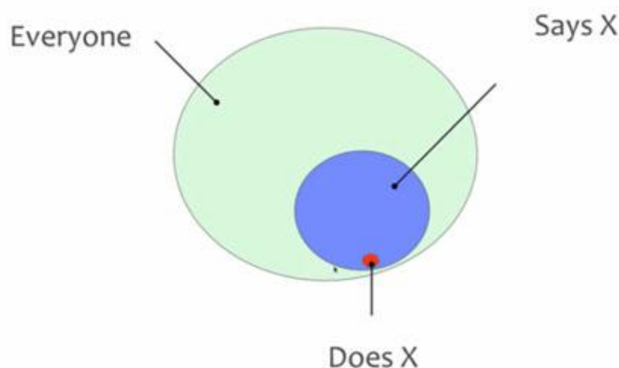
© Red Gate - www.red-gate.com

I happened to come across a book called *Extreme Programming Explained*, in other words XP explained. It has on the cover the phrase 'embrace change' and basically within that book I've found a discipline for responding to change. Those of you that have experienced Agile in a bad way are probably are thinking, "Wow, I thought that Agile and discipline were antonyms," but actually I want to convince you that Agile is not a synonym for undisciplined, and if Agile looks undisciplined to you then the people who are doing what they call Agile are just doing it wrong.

This Venn diagram is the set of all developers in the world, and I think that Agile is a fairly popular thing for people to say that they like or believe in.



But the disciplined practitioners of Agile are a very small subset of that set of developers who say they like Agile and this isn't really any big surprise, the whole world works this way. Everybody in the world is the big duck egg green circle and there are few people in the world that say something and there are far fewer that actually do the thing they say.



© Red Gate - www.red-gate.com

This article takes its inspiration from a book written by Kent Beck with Cynthia Andres called *Extreme Programming Explained* and the two fundamental core rudders in the book are basically that the purpose of software development is to satisfy the customer through early and continuous delivery of valuable software and that working software is the primary measure of progress. This rang a bell for me because back in the early 2000s I was attempting to create an optimisation method for Oracle database applications that would work uniformly across performance problem domains. Where I started was with this brilliant book written by a gentleman named Eli Goldratt who, I'm very sad to say, passed away in 2011, but *The Goal* is a book about optimising the manufacturing process. As *The Goal* is to the process of manufacturing optimism, the book that Jeff Holt and I wrote called *Optimising Oracle Performance* I hope, has the same relationship to database application optimisation. Goldratt's work really inspired our work and our method is very similar. It's, in fact, derived from Goldratt's method. What I found in Kent Beck's book is that his book actually has the same relationship to software development optimisation - basically the key of *Extreme Programming Explained* is to keep your eye on the overall goal of the process of software development and basically evaluate or optimise the process based on the impact of every decision you make upon that overall goal.

My 11 year old son is a professional baseball player and we travel with him and take him to games all around the US and there's another book called *Moneyball* written by Michael Lewis that actually is bizarrely similar to the other three books and it's about baseball club optimisation. So it's very interesting how many different disciplines have the same approach to optimising that Eli Goldratt started roughly 20 years ago.

As Goldratt explains, the global goal for an organisation, whether it's a profit or non-profit organisation, is to leverage its resources under its control as well as possible. He uses the three metrics: net profit, cash flow and return on investment as the things you want to maximise pretty much no matter what kind of a business you are in. Why Agile? Well it's because I want better net profit, cash flow and return on investment, I also want higher quality – this is one of the ways you get better net profit, cash flow and return on investment. I also want more fulfilment from my developers and myself; when I write a product I want to feel better about it, and I want to enjoy doing it more, and Agile gives me these things. So I

do Agile for these reasons, not because it's easy – as a matter of fact doing Agile correctly is extremely not easy.

Why **Agile?**

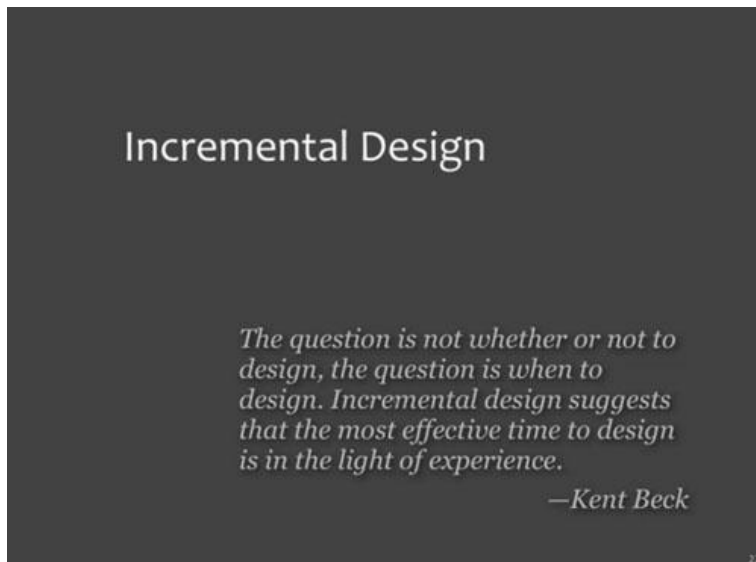
- ✓ better ❶ net profit ❷ cash flow ❸ return on investment
- ✓ higher **quality**
- ✓ more **fulfillment**
- ✓ more **enjoyment**

Not because it's **easy.**

It's **not.**

Five Practices from XP

The first practice is called Incremental Design. One of the reputations of Agile is that you just don't do design. I think a lot of people use Agile as a caricature or cartoon for bad project management.



The way that they would define Agile is to say "Oh Agile is a project where you don't design anything," and that's completely counter to how I see it and how I use Agile, and it's counter to how Kent Beck in his XP book describes it:

> *"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."*

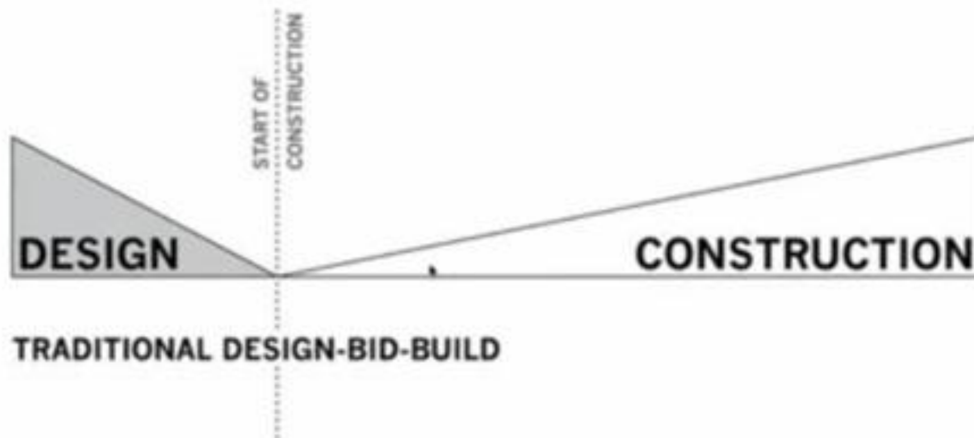
Fundamentally, the problem that incremental design is meant to address is that plans fail. No matter how carefully you plan for the year prior to the project, when you try to create a complete airtight plan that is going to define behaviour of a project team for the following five years, plans fail.

My wife and I built a house about 13 years ago and we built it from a pre-existing design where the architect allows you how to stretch a closet into a little bit of a different shape, but not materially change the layout of a house too much. So we picked a pretty decent plan and made a few minor customisations to it. One day when I walked out onto the site when the framers were putting up the wood to frame out the shape of the house, the leader of the framer company came over to me and said, "Hey, I want to ask you a question, this plan has a contradiction in it and I'd like you to tell me whether you want me to do it this way or that way." Basically the drawing was something that was impossible to implement in reality. If you want to think about an M C Escher print, that's essentially what my plan had been and the framer just wanted to know how to resolve the ambiguity. So there are ways to prevent a failed plan from failing your project and that's what this whole notion of incremental design is all about.

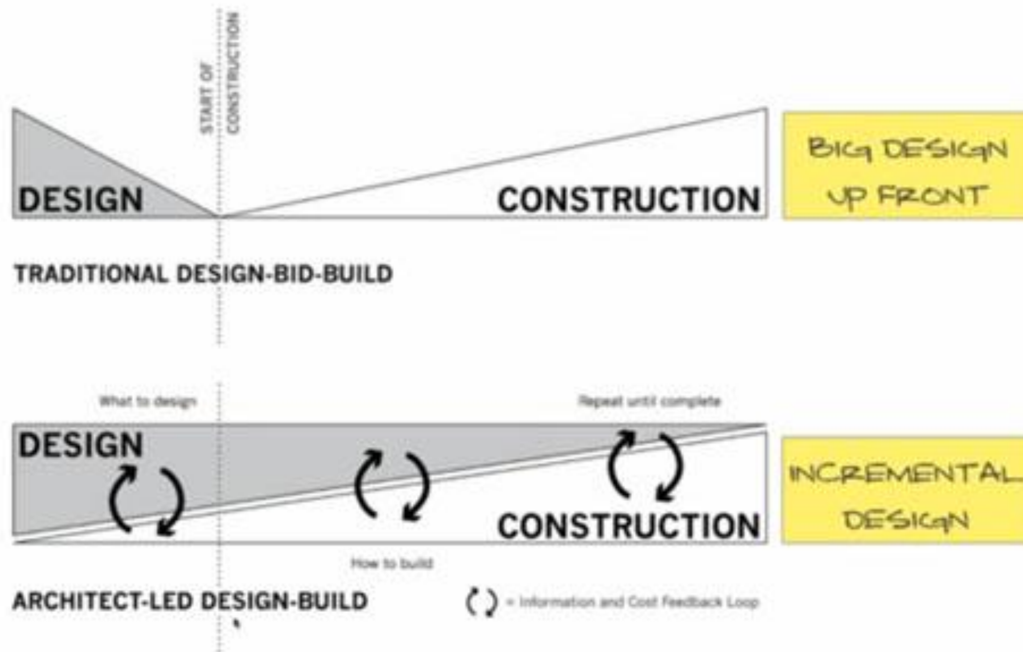
Now traditionally, and I say traditionally because it extends through the beginning of our lifetimes, but this traditional is only about 100 to 150 years old, back in the time of Frederick Taylor. His job of

trying to automate production and assembly lines meant that in order to optimise an assembly line he needed to separate the thinking from the doing.

So basically, traditional design build processes have thinkers front loaded in the project over in the following domain:

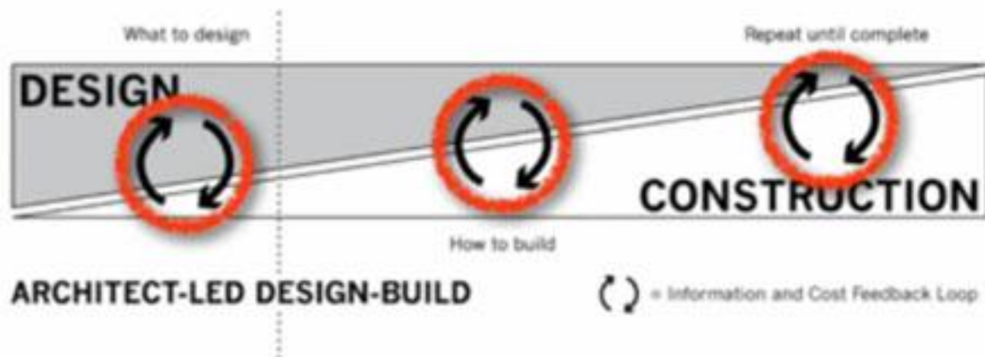


When construction starts the thinkers go off to another project and now the builders come in and build what the thinkers thought up. The plan is supposed to be airtight enough that there doesn't need to be a whole lot of thinking after construction begins. In the software industry, my experience is that, that just doesn't work out the way it's supposed to very often. Now these pictures I'm showing are actually from the construction industry, not the software industry. There's a new trend, or a newish trend, which is actually how people used to build things 200 years ago and 2,000 years ago, called architect-led design build. That's where the architect basically stays engaged throughout the duration of a project. Instead of the smart guys coming in, making a plan and then getting the heck out of the project, they stick around until the end of the project, but the decision-making dwindles because as you get deeper and deeper in a project more of the project gets constructed and so there's less need for design throughout the duration of a project. This is called architect-led design build. This was actually regarded immoral three decades ago because of the influence of unionisation and specialisation of labour in the western industrialised world, but nowadays it's more of a craftsman way to approach a project where the leader and the planner stays engaged throughout the entire project. So the top thing in the software world is what people refer to as big design upfront, the next picture represents what I'm calling incremental design.



The second practice of XP is called Rapid Iteration. This goes back to the notion that working software is the primary measure of progress. Do you know what the worst kind of software in the world is? It's that software that is 90% complete, but nobody can run it yet. I run across this kind of stuff in projects in my Oracle work all the time. I'll see a project that's been running for three years, but no users have been able to even see a prototype yet because the application is 90% finished and the vendor has been paid 90% of its price tag, but nobody can run the application until the final bits are put together. What typically happens in these types of engagement is when the final does get put together, then 20 people get to sit in a training course and they find out the application doesn't scale to 20 people in a training room. Well of course the project is late and over budget by now so they don't have time to fix the performance problems and it gets people in a real bind, especially when the project was supposed to be designed to support 20,000 users instead of just 20. Well the Agile answer to that is these loops inside the architect design build scenario, the method of doing design while you're building. These loops become very important and each one of these cycles needs to resolve in runnable software that people can actually get their hands on and experience how the stuff works and when you do that, you find that the design tends to refine itself rather quickly.

© Red Gate - www.red-gate.com



I've blogged a couple of cases, if you go to carymillsap.blogspot.com and search for the term 'messed up apps' you'll see a couple of applications that are painfully obviously not designed this way. They are painfully obviously not used by the people that designed them, otherwise you know the design would have never made it into production.

The third practice is called Test-First Progamming and as Kent Beck introduces it:

> *"The continuous testing reduces the time to fix errors by reducing the time it takes to discover them."*

That's exactly the experience that I've had. How many of you have ever been afraid, literally, to go and touch somebody's code. It might be yours that you wrote three years ago; it might be somebody else's code that left the company. I heard a story about a customer of ours who was afraid to go and touch somebody's code because the person that had wrote it had left three years ago. That's a big problem and it basically speaks to a lack of confidence in somebody's ability to understand what the code does, and also their lack of confidence that if they change the code they won't break something unintended. This is how Test-First Programming works: step one is you add a case to your case tracking system (we use a thing called FogBugz which is written by Joel Spolsky's company Fog Creek); that case then becomes a test, it's literally a piece of code that says I'm going to run the application and for this input it had better create that output. Well of course when a new test is integrated into the test suite, all the previous tests succeed, but the new test will fail and you want that because if the new test doesn't fail then the test doesn't really describe a new feature, it describes an old feature. Then you write code to accommodate passing that test and you run all the tests so that the tests will succeed. Basically there's a loop between potentially step five and step two that you may have to refine your test, you may have to refine your code, but ultimately all your tests will now succeed. At this point, it's safe to refactor, so this means you can honour Knuth's advice which says "write the thing to be maintainable and then go back later and if the code is slow or if the code is factored poorly where you've got the same few lines of code in 17 places throughout your application, now you can refactor safely and securely knowing that as long as your test passed you haven't broken the functional aspects of your code."

How **Test-First Programming** works:

1. Add a **case**
2. Add a **test**
3. Run all **tests** (✔✔✔✔✘ ...new test fails)
4. Write **code**
5. Run all **tests** (✔✔✔✔✔ ...all tests succeed)
6. **Refactor**

The fourth practice is called Pair Programming and this is one of the things when people talk about XP it's probably one of the first things people think of. Kent Beck has a nice way of turning a phrase:

> *"Silence is the sound of risk piling up."*

Pair Programming to me is a solution to problems like these:

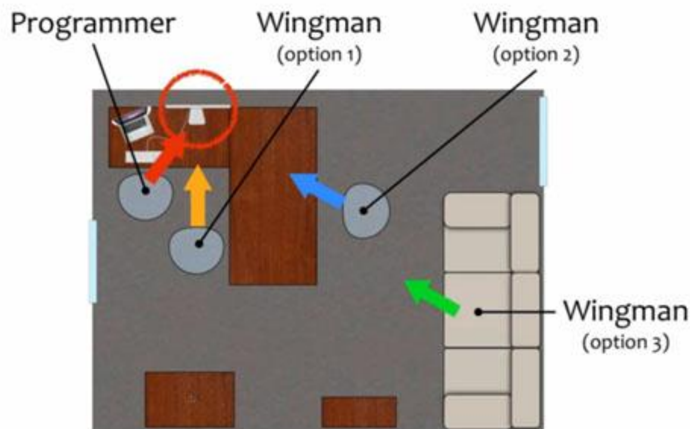
Stuck?

Not in the mood?

Skipping steps?

Occasionally I'll come into work and I'll just completely get stuck on something technical. Maybe I'm capable of doing it but I'm just not in the mood, I've got the thought that I just don't think I have the energy to spend four hours fixing this problem and I think it's going to take that long. Some days I come into the office and I really don't feel like making a test, I feel like writing code but I don't feel like doing step two which was create the test. Pair programming helps to solve those problems for the same reason that a workout buddy helps you not skip a day of going to the gym. When there's somebody there watching and you know that somebody is watching, you tend to be more careful and you tend to have more energy. This is a floor plan of my office and this is how we pair:

© Red Gate - www.red-gate.com



I sit in the programmer's chair and type, and Jeff Holt will be my wingman while I'm typing, so he can see the screen circled in red. Ron Crisco is currently sitting in the blue chair wingmanning this webcast. Sometimes I have somebody sit over on the couch if it's a PowerPoint show that I need to make sure the pixels are big enough for everybody to see from far away. But we do this fairly commonly and I'd say that we pair probably only about 10% of my time, maybe a little bit less.

Pairing is what we do to make sure that we're cross-cultivating the knowledge that is going into our code so that somebody else can maintain it later and it's also what we do to ensure that the person writing the code doesn't get lazy and starting skipping steps. It's amazing to me that as a child, I grew up an only child, I never did homework with friends, never had study groups in the lower grades at school so I just always considered work to be a solo activity. But once I started pair programming I realised that two people in a pair get far more work done together than if they work separately. The 10% that we spend paired up is some of the most fulfilling programming time that I've ever spent in my life, it has been world changing for me.

The fifth practice is called the Ten-Minute Build and Kent Beck talks about why ten minutes is about the right duration for all your automated test suites and your build scripts, Ant or Maven or whatever, should run. He says that practices that you execute every day should reduce your stress levels, not increase them and automated build becomes a stress reliever at crunch time. I did this just yesterday, I had a sticky situation where I wasn't sure whether the code I was writing was going to work and it's just my habit now, I go to my build screen and I build the app and I see whether it passes all the tests. If it does pass all the tests but I'm still concerned about my application working correctly, then maybe I'll add another couple of tests. Test coverage tends to grow over time and as your test coverage becomes more and more complete, when your test runs and passes, it just gives you that much more confidence.

This is what it looks like when the world is good:

```
[exec] Result: PASS
[echo]
[echo] mrtim
[exec] # ./t/4154.test
[exec] # ./t/4160.test
[exec] # ./t/4163.test
[exec] # ./t/4175.test
[exec] # ./t/4176.test
[exec] # ./t/core01.test
[exec] # ./t/opt01.test
[exec] # ./t/pod01.test
[exec] ./t/test.t .. ok
[exec] All tests successful.
[exec] Files=1, Tests=14, 29 wallclock secs ( 0.02 usr  0.00 sys + 26.73 cusr  3.22 csys = 29.97 CPU)
[exec] Result: PASS
[echo]
[echo] mrtimfix
[exec] # ./t/opt01.test
[exec] ./t/test.t .. ok
[exec] All tests successful.
[exec] Files=1, Tests=7,  6 wallclock secs ( 0.02 usr  0.00 sys +  4.52 cusr  0.56 csys =  5.10 CPU)
[exec] Result: PASS
[echo]
[echo] mrcallrm
[exec] # ./t/4119.test
[exec] # ./t/4137.test
[exec] # ./t/4138.test
[exec] # ./t/4139.test
[exec] # ./t/pod01.test
[exec] ./t/test.t .. ok
[exec] All tests successful.
[exec] Files=1, Tests=11,  5 wallclock secs ( 0.02 usr  0.00 sys +  4.28 cusr  0.54 csys =  4.84 CPU)
[exec] Result: PASS

BUILD SUCCESSFUL
Total time: 3 minutes 20 seconds
```

This is a little toolset that we sell call MR Tools and it's the test suite from MR Tools and you can see at the bottom that BUILD SUCCESSFUL took three minutes and 20 seconds to execute and that runs all the tests for our tools. For example, 4119 test might have 15 or 20 command line executions in it but it tests a particular feature of the tool called mrcallrm and I know that when all these tests pass we have a tool that is ready for release.

Big Spec == Big Mistake

the **testing-is-too-expensive** problem

the **antigravity** problem

the **gluttony** problem

the **I-know-it's-what-I-asked-for-but-it's-not-what-I-want** problem

The first mistake that I want to talk about that I used to make a lot is the big spec mistake. A lot of people believe that the way to do a software project is to write a great big spec that includes everything that you might need to put in the product, then that becomes the plan and you lock it down. In some cases the project planners actually leave the project and move onto another project and start planning project b, leaving the spec for the builders to make. In my experience that has always been a really big mistake for me. The testing-is-too-expensive problem is really about what do you do when you have a 100 page specification and you want to test to see if your software meets that specification. Well if that spec is written in English then it's going to require a human to read the spec, run the code, look at all the details that the spec describes and make sure that the code performs in a way that matches how those things are described in English in detail. It sounds good in theory, but that might take four or five weeks. It might require a tester to create lots of innovative ways to reproduce the cases that are described in English. So if your test suite takes four or five weeks for a human to execute and somebody needs to make a small change in the code, the tester really needs to start over with page one of the document again and not very many people can do that very many times without starting to skips steps. Testing to an English specification document is really expensive, because it involves labour, it involves somebody that is willing to pay close attention over and over doing the same repetitive task and a task like that I think is really better executed by a machine rather than a human.

The antigravity problem is the problem that I can actually specify that I want to levitate four tonnes of object 19 inches above the ground for a year and a half, and I can write that sentence in English quite easily, but it's very difficult to implement that sentence in reality. The fact is that there really are no physical laws or physical constraints on what you can write in English, that is why science fiction exists, but when it comes time for somebody to build it they get stuck in the same way that my framer got stuck, you can't build an Escher print out of wood, you can build one on paper but you can't build one out of wood. So it's very easy in an English specification, a big spec upfront to actually specify things that are contradictory to what is possible.

The gluttony problem is probably pretty easy to understand. The guy with the word processer and a big imagination can put a bunch of stuff into an application that really doesn't belong there.

Then finally the I-know-it's-what-I-asked-for-but-it's-not-what-I-want problem. It sounds undisciplined to say what you want, get it, and then later decide well, that's really not what I want.

There are all sorts of gender jokes about that kind of thing but it’s actually very common. That people are unable to describe what they want until after they feel something that is close to what they want, then it's much easier to refine than it is to imagine the perfect application from a blank sheet of paper. Kent Beck in the XP book advises that you should maintain only the code in the tests as permanent artifacts. That you should generate other documents from the code and tests. That's really what we do, when we generate our manual pages we generate those based on what the test suite does, instead of the other way around. If you think about this it's really one of the most elemental principles behind relational design in that you want to store data only once. You want to store a given piece of information once and only once. You want the spec in one place, and you want the code in one place. Otherwise you end up with the potential for update anomaly problems. The spec in my opinion, belongs in the test suite so that a machine can execute it over and over and the code belongs, of course, nobody debates whether the code belongs where the code goes.

Regression Testing == Awesome

far **less expensive** than I thought

makes **refactoring** so much **easier**

inspires **confidence**

makes **support** and **documentation** better

The next thing that blew me away is how awesome regression testing is. I used to think that only huge companies could do regression testing and I found out to the contrary that even our small company can afford to do it. In fact, I would argue now that we really can't afford not to. It makes refactoring so much easier. If you want to go and change the way you factor some subroutines, as long as you keep passing tests and as long as your tests are adequate in their coverage of your product, you’re safe. You're working with a huge, great, strong safety net when you use tests.

© Red Gate - www.red-gate.com

Incremental Design == Better Design

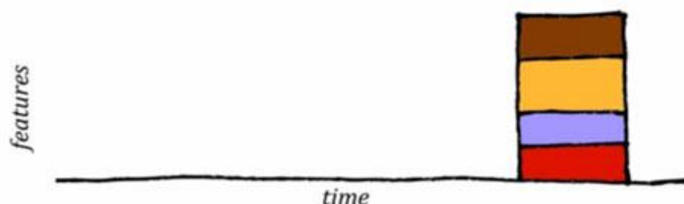
makes **decisions easier**, more **obvious**

thus **less expensive** and **just better**

creates **inspired innovation**

The incremental design to me means absolutely better design. It allows us to make decisions more easily and makes those decisions more obvious, it is less expensive and just better. Basically doing incremental design allows us to create much more inspired designs than if you have to sit down and imagine everything from front to back on a blank sheet of paper. In my experience most code that people are not satisfied with, is not because the code mismatches the specification, it's because the specification mismatches the need.

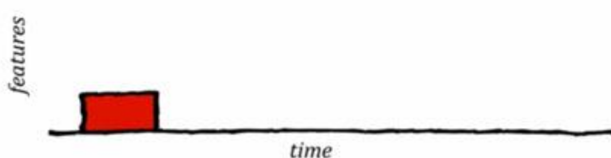
Below is the picture that appears inside my brain whenever I talk to somebody about incremental design and rapid iteration:



© Red Gate - www.red-gate.com

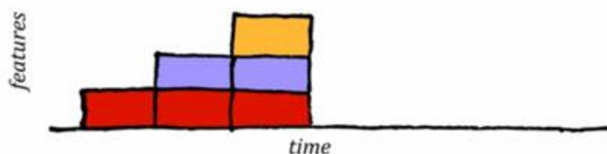
Here's a picture of a product that I thought I wanted when I designed it big upfront. I thought that I wanted the product to have a red feature, a purple feature, a gold feature and a brown feature and I knew it was going to take time to build this thing. So the distance from when I imagined this product and when I knew the product could be built is quite a long span of time. Ron Crisco has taught me a lot about how to write software projects for release and the way that Ron would do a project like this is start with something valuable that runs and then release that. So we decided in this particular product to do the red feature, because the red feature would be helpful and if our software tool only did the red feature and we were to perish after creating just the red feature, the red feature would in fact be useful and useable to a lot of people. So we built the red feature.

Build something **valuable** that **runs**,
and **release it.**



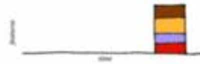
The next step, we built a little more and we released, so we built the purple feature and then over time we built the gold feature.

Build a **little more** and **release**,
a **little more** and **release**...

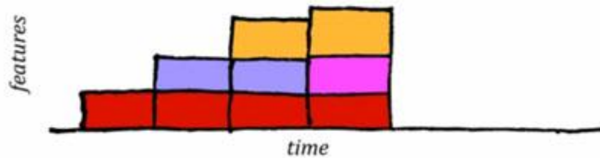


At that point, something magical happened. We discovered that we really didn't want the purple feature after all, after using it for some time and getting accustomed to what it could do, what we decided was instead of the purple feature, what we really wanted was the pink feature.

© Red Gate - www.red-gate.com

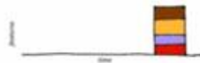


...and **discover.**

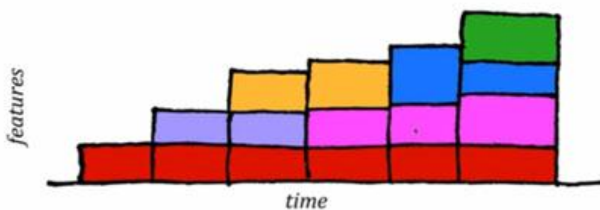


So instead of building the brown feature on top of the gold feature at this point, we decided to replace the purple feature with the pink feature, because we discovered that's what we really needed and we couldn't have known that, or at least we didn't know that at the beginning when we planned this project.

The next couple of iterations were now that we have this pink feature, we really don't like the gold feature as much as we would like a blue feature and in the final release we built a green feature on top of the blue feature.



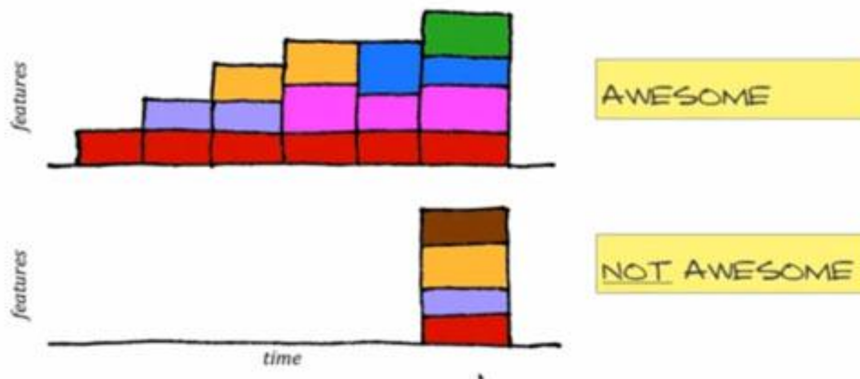
What I **want** is
not what I imagined.



The experience of using the product over time informed us that the design we initially imagined wasn't really optimal for the business anymore and what we ended up wanting is not very much at all like what I had initially imagined. Maybe there are people out there that are so awesome at designing things from scratch that they can design two, three, four years out into the future, but I'm just not that guy. I am pretty good at taking something that works and making it better, but I'm not nearly as good at taking nothing and turning it into something elegant.

DEPLOYMENT SUITE for Oracle
© Red Gate - www.red-gate.com
redgate ingeniously simple tools

So what we've got here is three things I want to focus your attention on. Firstly, we were able to use the red feature much earlier in the top project picture than we would have been able to use anything in the bottom project picture.



- ✓ **Usable** software **earlier**
- ✓ **Experience** informs the **design**
- ✓ **Better** design in the **end**

So at a given point in time in the bottom picture we still have nothing, but in the top project we've got the red and the purple feature, so we're at least able to use some software assistance from the project at this point.

I've already talked about how experience informs the design so the design actually changes for all the right reasons because as we use the software we decide what we really wish the software did. Instead of what I thought I wanted it to do, I wish it would do something slightly different. Then in the end you've got a better design because the upper software is informed by actual use and experience, whereas the lower software was informed only by someone's imagination. So it's better all round. You get use out of it earlier and you end up with a better design by the time you're finished.

What Has Not Worked

Probably the most damaging aspect of a project that is attempting to be an Agile project is the absence of a customer that has these five attributes, acronym: C.R.A.C.K.

If a customer representative isn't on the product team that is collaborative and willing to talk about things, representative meaning that this person has the best interests of the users that this application is going to be distributed to. Authorised meaning that the company has given this person power to decide what goes into the software and what doesn't go into the software. Committed, meaning that he cares and knowledgeable meaning that the person understands what the real needs are of the people are that he is representing. If nobody in the project will say no, so everything gets a yes. I've been on projects before where there is no real CRACK customer and it's suicidal. Basically, the team doesn't know what to do so it makes it up as it goes along. Most members of a team, if they're responsible human beings, will try to add more and more things hoping that this imaginary customer will be happy with having more and more things in the application and it's what causes applications to end up having 80% of its code path never being executed by anybody. It's what causes projects to go over time and over budget. The absence of a customer with these attributes is absolutely an Agile project killer.

The second one, ironically, not enough customers is bad, too many customers is just as bad as having no customer. You've got to remember a great design is just as much about saying no as it is about saying yes.

DEPLOYMENT SUITE for Oracle

© Red Gate - www.red-gate.com

redgate ingeniously simple tools

Too Many Customers

just as **bad** as no customer

Suicide!

great design is also about **No**

If that's vexing to you, think about the iPod, think about a device that does not have an on-off switch. Jonathan Ive and Steve Jobs decided that they were going to publish a device that didn't have the most elemental switch that every other device on the planet had, but they designed it simpler and more elegantly because they knew what features to say no to and they knew that an on-off switch was superfluous, the product ought to be able to know when you need it on and when you need it off by itself.

© Red Gate - www.red-gate.com

Cultural Mismatch

agile is about **decentralization** of responsibility, accountability, ...

centralization + agile == **hypocrisy**

agile requires **openness, honesty** about where **failures** are

The next one is cultural mismatch. Agile is all about decentralisation of responsibility and accountability into the hands of the people who were technical enough and knowledgeable enough to be able to do the job correctly. If you try to do an Agile project in a centralised organisation unfortunately all you end up with is hypocrisy. What you end up with is a centralised organisation that can't let go, trying to claim that it is letting go, but it's not. Agile requires openness and honesty about where the failures are. If you're in one of the types of projects that the management cannot stand to admit that anything is imperfect then Agile is not for you. Basically the whole key about Agile is that you want to try to find out where the failures are as early as possible so that you can redesign and surmount them.

DEPLOYMENT SUITE for Oracle

© Red Gate - www.red-gate.com

redgate ingeniously simple tools

Talent Mismatch

undisciplined + agile == chaos

participants must **actively design, optimize**

key skill: project **factorization** to produce **running, valuable software** every n weeks

The final point is about talent mismatch. Basically if you have a team that is not disciplined or not self-disciplined and you try to integrate Agile what you'll end up with is chaos. What you'll end up with is the joke about "They must be doing Agile in the kitchen". Basically on an Agile project participants have to be good at design, they have to be good at optimisation, sometimes they even have to be good at process optimisation. Sometimes they have to be able to look at what they're doing and decide that in order to do this we need to do it a different way. A key skill in any Agile project is the ability to factor a project so that it produces running, valuable software every few weeks. Now the value of your n might be two, it might be one, it might be 36, but basically if you decide what your iteration length is between product releases, the ability to continually chunk out pieces of software that work, that can be installed and be run, that's the key skill to Agile.

Q+A Session

Q. On the legacy product waterfall methodology is used, and we're going to move to Agile/Scrum, what are some of the things we should be careful of or look out for?

A. I think the cultural mismatch issue and the talent mismatch issue. There's a really good blog post that I saw two days ago, I Tweeted it, it was called 'Briefly about Agile' (seldo.com) and the blog said "When I hear Agile I hear cargo cult", and it's a very short blog post that says that basically a lot of people who implement Agile are really only implementing the things that are their favourite sounding parts. They might implement the idea that they don't need to write big, complicated documentation. The problem that not writing big, complicated documentation solves is that you don't have to have a big complicated team that had got big complicated integrations with your development team who are updating big complicated documentation. So that's one problem solved. But if you don't have anything to replace that big complicated documentation, for example, you don't have a comprehensive high coverage test suite, then you've got a huge gap in your project. So what I see happening in a lot of sites that call themselves Agile is they're not really following the disciplined practices of having an integrated test suite that runs within ten minutes of clicking the build button. What's happening is they're taking what they like but they're not taking the part of the process that requires hard work and actually fills in for the part of the waterfall process that they're taking out. So what I'd advise you to do, the biggest pitfall is that if the team are not capable of doing design or not empowered to do design by their organisation as they work, then Agile is not necessarily going to end well, if the culture of the company that is doing the project doesn't allow for failure. Agile is about failing fast so that you can fix it quickly. It's not that your goal is to fail, your goal is that if a design element is destined to fail that you fail quickly so that you can fix it early so that your product can have a better design suitor. If a company's leadership is culturally opposed to anything having to do with failure or admission of failure then Agile is really going to be difficult to pull off.

Q. What are the limitations and advantages of Agile application design compared with ITIL or waterfall in software application design for development and deployments?

A. One of the limitations of Agile is that you really have to have a different staffing mix than you can in a more upfront, planned type of project. There's a really good book called *Balancing Agility and Discipline* written by Barry Boehm and co-authored by Richard Turner and they talk in great detail about what your project staffing mix needs to be in order to pull Agile off properly. You basically need to have much more senior, much more mature, but much more out of the box thinking project participants on an Agile project than if you have a more traditional waterfall type of project. Remember, what waterfall is intended to do is take large groups of people who are not necessarily that inventive or highly trained, in other words people that don't cost

as much, and allow them to get something nice done. So the advantage of waterfall is that it's an attempt to separate thinking from doing, so you put the highly paid, very expensive thinkers at the front of the project and then you try to release them as early in the project as you can to save on cost, then you pass off the work to people that don't cost nearly as much. If you think about the construction industry, that's how you tend to think highways get done. You've got somebody in an office who is drawing up pictures and thinking about queuing theory and where to place the traffic lights, you don't want your builders out there just throwing up stuff and seeing if it works, you need an architect. Then as the build progresses, if it's a project that's been done six thousand times in the past 15 years, you can be reasonably certain that the plan is going to work because it's been thoroughly debugged. Agile has an advantage in applications that are executed using a plan that has not been debugged, projects that have never been done before, but it requires a certain type of staffing skills mix that you may not be able to afford. So the downside of waterfall is that it tends to diminish the individual creativity that the better people on the team may have. The upside of waterfall is that you're supposed to be able to use lots of expensive and experienced people and get a reasonably good job done. So part of how you should decide which process to use depends on what are you trying to accomplish. If you're trying to accomplish something fairly mundane that has been a lot of times before waterfall probably is a better way to do it. If you're trying to accomplish something that's never been done before then you probably need to have lots more rapid iteration, lots more incremental design, lots more integrated testing and the things that Agile really brings.

Q. My team have been having trouble in limiting our Scrum meetings to 15 minutes, often we take 25 minutes to an hour and 15 minutes. Do you have any tips to reduce it and keep the discipline?

A. I'm not a Scrum expert by any stretch of the imagination. I used to host a morning stand up meeting at a prior company that I was at and to be honest the main reason I did that is because the culture of the company required that there be a meeting so that somebody could take notes so that the management of the company could know what was going on in the development team at all times. But honestly it didn't help the development process very much for us and I've taken the advice of a gentleman named Jason Fried at 37signals and his book called *Rework*. One of the chapters in *Rework* is called 'Meetings are Toxic'. At Method R Corporation we don't have meetings anymore, I can't think of the last time I had a meeting that took longer than a few minutes. Now we do pair up and sometimes we have three people in a room talking about something very specific and I don't know if this is particularly helpful advice, but I have found that by eliminating almost all the meetings I used to have I haven't lost anything. When I need to know something technical or I need to share a design idea to make sure that it's valid there are one or two people I pull into the room with me, we discuss it and we move forward. Having said that, the software that we design here doesn't typically have a lot of integration points, so there's not six different interface groups that need to be aware of everything that's going on every time we make a decision.

My advice is to grab a copy of Jason Fried and David Heinemeier Hanssone's book called *Rework* and see if that might give you some good ideas.

Q. Design upfront versus incremental design – isn't there a huge possibility of needing to change the fundamentals of the design half way through when you start adding new requirements?

A. That's one of the places where the talents of your team is a huge determinant in whether you will succeed or fail. The whole thing I said at the end about factoring – factorisation. Ron Crisco is my product development director and his presentation next week at ODTUG is about how do you do data modelling in an environment where change is inevitable and it's unpredictable. Because one of the things that people talk about being a huge deficiency of Agile is when they misunderstand and think that incremental design means no design and then people think that Agile means you don't do data modelling. You absolutely have to do data modelling if you're going to use a database in your project because there are so many things that mess up if you don't have a sound data model. You can't have a high performance, highly scalable application if you don't have a sound data model. There is the possibility that you get a third of the way into the project and discover that your data model is just not good enough and I don't have a set of examples on the tip of my tongue that I can tell you about, but Ron has been putting together a sequence of presentations on this. Fundamentally the goal is to make sure that the things that you do design and you do lay down into concrete, so to speak, are well done and that they're extensible. An Agile project in which the data model ends up having 27 copies of the same data because basically the model was just accreted by different people and never really rationalised or centrally controlled or made elegant, that is not what you want. It is not a good data model if that's what you end up with. The bottom line is if the model needs to change a third of the way or half way through the project then it needs to change a third of the way or half way through the project. The level of talent of people that are designing the data model and their experience with doing rapid iteration is going to minimise the impact of needing to do that on the occasion when you do need to do it.

Q. What has been your experience with on site and off shore development centres? Does Agile work with that?

A. Again, it depends very much on the talent mix that's involved in the project. We're actually involved as a vendor except we do some consulting on the side, it's a little more than just on the side, we have one very large project that's actually in its third year of execution and we've been writing PLSQL code for a large company and we're basically an off shore development team for them, although they're also in the US and we are too. We live a two and a half hour flight away from them, so we exchange specifications through e-mail and we participate in quite a few phone calls with them that constitute meetings where we're having technical conversations about how the spec needs to be designed. The same company has off shore teams in

India that they have a completely different relationship with. I guess we are a much more experienced team and it doesn't have anything to do with whether we're in the US or India, but our team has experience and the team in India is much lower priced and consists of much less experienced developers. Those guys require a spec to be sent to them and they code to the letter of the specification. If something doesn't make sense in the spec they will ask a question to try to resolve the ambiguity but there's not a whole lot of attention paid to trying to improve the spec in that other relationship. The relationship our customer has with us is that we're expected to be a partner in creating the specification for the software that we're writing. We're expected to interface with the other teams and think ahead so that we're actually designing along with our client as we write our code. We've just returned from a site visit last week in which we talked about the Agile processes that we use, the automated test suite, we don't like documentation for documentation's sake, but there are some cases in which you absolutely have to have documentation to be able to lock down what the specification is between two teams that live in totally different time zones. For us we don't have the same amount of documentation as the company does with their Indian off shore team, but again, it's because of the role that we play. So I think that Agile works in so called off shore environments but it really comes back to what is the talent that's involved and that helps define what level of detail is required in the documentation that has to be transmitted back and forth between the teams.

Q. What is the relationship between XP and Agile?

A. The XP book was written 1999 and does not have the word Agile in it that I could find. I actually searched in Google Books and I could not find the word Agile. As far as I know it does not appear in Kent Beck's book. I subsequently found out that the word Agile was chosen by a group of 12 authors. Basically a group of likeminded people, Kent Beck included and a bunch of other people from Pragmatic Programming, from Scrum and from several other disciplines that had similar attitudes that Kent Beck had. They got together in Colorado and had a meeting for two or three days and tried to sit down and decide what they agreed upon. The principles they agreed upon they wrote down in a thing called the Agile Manifesto and the 12 authors put their names to this manifesto as the core values that they all thought represented what they did. The word Agile is really kind of a rollup word, therefore. I think as Agile as the parent node in a tree that beneath it contains XP and Scrum and Pragmatic Programming and several other methods that were created before the word Agile existed. But they all have the same sort of spirit of we've got to figure out a way to add discipline to responding to change. So the word Agile is the parent of a tree that has leaves that include things like XP and Scrum.

Q. How do we reconcile Agile practices against SOX and HIPAA and other legal mandates that tend to lead towards centralised committees and information governance?

A. That's a really good question and I don't know. I do believe that it's a similar issue to what an aerospace company would have to deal with. A place I would look for maybe some inspiration is the story of Kelly Johnson and the Skunk Works group at Lockheed, basically we're talking about an industry that is heavily regulated, it's a military industry. Kelly Johnson was an aerospace engineer that is responsible for the design of the Lockheed P-38 Lightning fighter that the allies used in World War II, the F-104, the U-2 spy plane, the SR-71, those all came out of Kelly Johnson's Skunk Works operation in Southern California near Long Beach. I know that that's a heavily regulated and heavily authoritarian type of an industry and if you read Kelly Johnson's rules for how they did some of the remarkable things they did, they're very similar to what the Agile guys talk about in their Agile Manifesto and the principles behind the Agile Manifesto. The way that I would try to think about it is to think in terms of how a group works inside versus its interface with the outside world. The story I told about us and our customer is similar to this, we are actually Agile inside our company doing a project as a subgroup of a company that is not particularly Agile, that we're feeding software back to. Now our project has been so successful that they're interested in learning more about Agile, but they themselves are not Agile but we are as a component. It's very similar to how you implement a module and how you publish when its interfaces are different. So perhaps your team can be Agile and operate within its boundaries as an Agile team, but if a document is required because of HIPAA, for example, or Sarbanes Oxley, you have to create that document as an output of your team and you might create that document in an Agile way. For example, you might try to find tools to automate as much of the document creation as possible, but if that's a requirement of your project because of the governance that surrounds your project, then obviously it's just as important as an output for getting paid as the code that you produce. It just comes down to understanding what your API with the outside world is as a project team and where you have liberty to choose how your project team can operate within its own membrane.

Q. The older spiral methodology as well as rapid prototyping methodologies seem similar to Agile, do you see major differences with Agile and these methodologies?

A. If you think about what incremental design and a rapid release cycle means it's very much like the old spiral or rapid prototyping methods. I think perhaps Agile is different in that it incorporates a broad variety of other practices as well: the pair programming and the ten-minute build. Those are almost like ornaments on the fundamental tree of the rapid prototyping idea. Basically rapid prototyping is about "Hey, I don't know if this thing is going to fly to the moon, I don't even know if it's going to fly off the backyard, so let's create a model first, see if that will fly. Then we'll see if we can scale it up, then we can see if we can put a guy in it, then we can see if we can put it into orbit, then we can see if we can get it out of orbit, then we can see

if we can get it to the moon, then we can see if we can land it on the moon." If you think back to a project as large as Apollo for example, that's basically how they did it. It may sound like a big design upfront but they did lots and lots of missions that taught them what they needed to know before they could ever design the next mission. We had to put people into orbit before we could kick people out of orbit. We had to orbit the moon before we could ever decide to land on the moon.

So I think in the days when that happened what they were doing was called rapid prototyping and in the software development world I think that's really what Agile is, is rapid prototyping. But added with it a lot of things that tools can enable us to do to basically...I don't know, it seems like Agile is a bigger kit to me that explains how to do some of the loose end details that you have to do day in and day out on a project in order to succeed.

Q. Do you in any way capture anything in a modelling tool afterwards? So for example, capturing tables and putting it in an entity model, or capturing business rules in codes to make it readable afterwards?

A. On the project that we're doing that I keep referring to one of the things that we're required to do is to produce an entity relationship diagram for the project team that we're working with so that they can understand the structure of the tables that are in the schema that we created. Of course that changes from time to time and we have to keep the remainder of the project team updated with what those changes are. What we try very diligently to do is to make sure we record that information in the database where it counts, and then we generate the documents based on that. So it's like our source code is the actual create table statements and the actual alter table statements and create index statements – that's our source code. The documents are created automatically by using tools to generate what the schema looks like, so any pictures that we draw are not drawn by hand, they're derived from the source code, so to speak, of the create table statements that we've generated. The remaining documentation, we write a lot of code with PLSQL and I believe we're actually pulling comments out of the PLSQL to create the user documentation. We have, for example, an Open Source tool called ILO at SourceForge, it stands for the Implementation Library for Oracle and that's how the documentation for that is built. It's basically extracted from comments within the code so our documentation and our source code are really one and the same. In our MR Tools product they're written in Perl and the documentation for the tools themselves are written inside the Perl source code. So it's one file edit, from writing the documentation about a paragraph, the code that is right next to the documentation that describes it. So back to that fundamental rule of relational design, you want to store data once and only once, well if you have different formats in which you need to publish given data, what you need to do is figure out what your source code is for that format and then try to find tools. That's one of the reasons Red Gate and Method R are friends because Red Gate is one of the key providers of tools that allow you to do things like schema reporting and schema differentiation and data differentiation. You don't want to have to do that stuff by hand, you want to be able to have tools that help you do it.

Q. What is the biggest challenge when implementing Agile methods? Can you elaborate more on the Agile method and database modeling in database architecture?

A. To the first question, the biggest challenge is to make sure that you don't view an Agile system as a bunch of buzzwords and a bunch of ceremonial things, that if you do those everything is going to be all right. Agile is a bunch of work. If you decide you're not going to write a big complicated specification in English, well then you need to write a big complicated specification in code, you've got to fill the gap one way or the other. I'm a huge proponent that a specification should be written to be machine executable as opposed to human executable, but that takes work. I think one of the reasons that Agile methods have such a bad name among my DBA friends is because the way that they see them implemented is that people just pick off the stuff that's easy and they don't do the compensatory things that are difficult and so they end up in a rut, they end up with no spec whatsoever. They don't have tests, they don't have an English document, they saying they're doing Agile and all that does is make Agile look bad.

In answer to the second question, one of my good friends that used to work with me at Oracle is named Dominic Delmolino, he's on Twitter and he talks a lot about database and Agile. If you just follow his Twitter stream you're going to get loads of pointers to different sources of information. Ron Crisco is an expert in the same area and he's speaking next week at ODTUG about data modeling and rapid iteration projects. There's a good book called *Database Refactoring.* There are places out there in the market to help learn more about how to do rapid iteration and incremental design using databases. I think one of the reasons it's so hard to redesign a database model is that the tools market is relatively immature. Red Gate is early to the table really in providing some of the tools that it creates for the Oracle database administrator and Oracle database developers. There's not a whole lot of good competition out there that helps people refactor databases and there are really two big problems: one is the schema differentiation, a developer changed six tables and needs to communicate to the DBA what they changed, so then the DBA can rationalise and publish that schema change for everybody else in the project. Well noticing what's different about the schema is only half the battle, maybe even less than half the battle. The rest of the battle is "Hey you added a column, what data did you put in that column? So what data in this database is different based on what you the developer did by adding a column to this thing." Of course it's probably not driveable data from anything that was previously in the database or you wouldn't have needed to add the column. So it's a big complicated job to refactor and if 20 years ago somebody had said, "Yeah, I want to take all instances of these five lines of code out of my project and turn them into a function call," the developer would have been aghast. They would have said, "Oh my God, that's going to take me three weeks to do that," whereas today in Eclipse you can do it in 20 minutes. The tools are just much more mature on the developer's desktop than they tend to be on the database administrator's desktop for doing refactoring. It's one of the reasons that I've been following Red Gate because they seem to be the leaders, even ahead of

Oracle and some of the more established players in the market, in understanding that these things are gaps that need to be filled with good tools.

Q. Do you have suggestions with respect to programming standards at the beginning of a project? One project that Chris was involved with was so focused on speed of development that management rejected his proposed requirement that all SQL used by the php web server be expressed within a packaged procedure. When the data model was later changed, we were not merely rewriting SQL but negotiating all the interfaces between the web server and the database server.

A. If I were to write a book about ten mistakes you need to watch out for, that's probably number one or number two. There's a concept called 'technical debt', that's basically a story about unpaid technical debt. Think about what debt is with money, debt with money is spending money that you don't have but you'll earn it later and pay it back later. That's essentially what you do when you compromise code quality for speed. Speed to market means "I've got to get this thing done in two weeks, I don't have time to make packaged procedures out of my SQL. So what I end up with now is I've got SQL in my php, my product works, I've released it. Now we're starting to make a little bit of money because some customers like it, but they need six new features. So instead of spending next week fixing my code so that it's easier to maintain later, I'm adding six new features to my code." Ultimately your debt catches up with you, there is no more that you can borrow, you hit your credit limit so to speak, and what has to happen at that point is you have to stop and start paying it back one way or the other. The story that Chris told in his question ended with the abrupt realisation that something that should have taken 20 minutes to do, will now take four weeks and that's the interest on that debt. So as you continue to leave yourself in technical debt and add more and more features to your product, you're basically compounding the interest that you're going to have to pay back on that technical debt.

Basically what I find Agile gives me is because of the regression testing suite that I have, it makes getting out of technical debt, putting the SQL into packaged procedures and making an API, it needs to be done. I'm a fan of releasing a product that works before having done all that, I think that's a fine idea, I think the place that the refactoring needs to take place is after the project has been demonstrated to be viable. I think that the gap in the story that Chris told is his management's appreciation of what the cost is of continuing to hold that technical debt as the product gets extended and extended. I think the right place to go fix that problem is not to have delayed the product's release in the first place by doing a complete architecture design to put all the stuff in store procedures. I think that proving the concept that the application works and is saleable was a good thing, but I think that somewhere between version one and version two or three there needed to have been a feature added that customers don't necessarily, that doesn't show up in the marketing material, that is "Look guys we've got to go and refactor this code so that it's easier to maintain later, because we don't want to get caught out." There's a number of reasons that you want your PLSQL interfaced instead of having SQL sitting in php pages.

There are a lot of performance reasons, you don't want to be passing long strings of SQL between a client and server for example. There are security implications, there are horrific performance implications of that implementation inside the database. As Knuth said 31 years ago, you don't optimise before you make sure that the thing works and in the software industry 'the things work' is not just does the code run, but is somebody willing to actually trade their hard earned cash for our code. I think the place that the mistake was made was not necessarily in releasing before the architectural work, but continuing to release without going back and revisiting the architectural work.

Q. Are there any publicised white papers that show some detailed examples of successful Agile development projects? Are there any public training sessions coming up this year that are hosted by Method R?

A. Off the top of my head I don't know, if I need to find them I would start with Google personally. There is one blog called agilewarrior.wordpress.com. A Google search is going to result in a rich answer to that question.

\- - - - -

Cary Millsap

Cary Millsap is an Oracle ACE Director, president and founder of Method R Corporation, former VP at Oracle Corporation, founding partner of the OakTable Network, Oracle Development Tools User Group "Editor's Choice" 2010 award, and Oracle Magazine's 2004 Author of the Year.