# An Agile Retrospective

Richard J. Dudley
http://rjdudley.com/
@rj_dudley
rich@rjdudley.com

A lot of agile talks tell you what you should do, and how to do it. This talk it a little different—this is a look back at what worked for us, and why. We began with a big idea to bring order out of chaos. Each sprint we tweaked our process, and this is the result of several dozen sprints over several years.

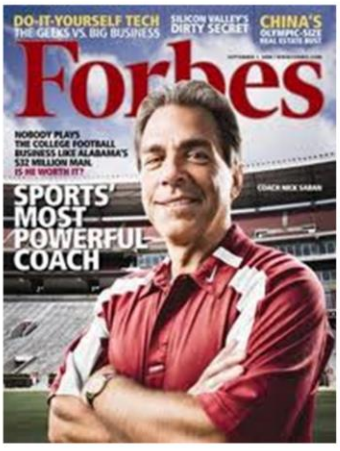
The process we ended up with is pretty similar to "scrumban", which is gaining in popularity recently. We started with just a kanban board, and as we learned more about Scrum, XP and other agile movements we'd try something out and kept what worked.

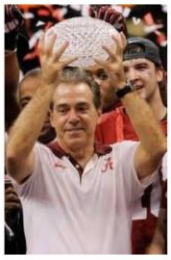
## Why agile?

"The great thing about agile is we don't have to figure it all out at the beginning.“

- David Juan (Quicken Loans) the BA who gets it

## “The Process”

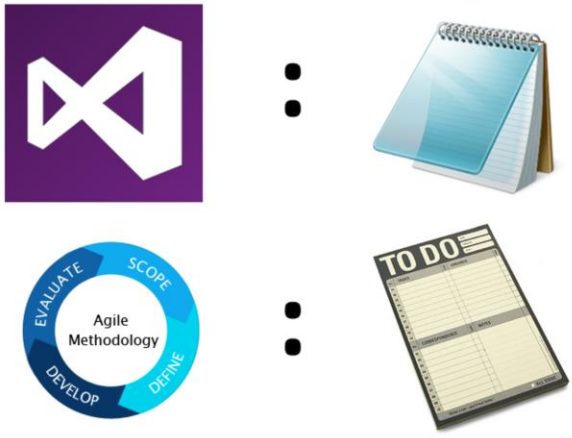


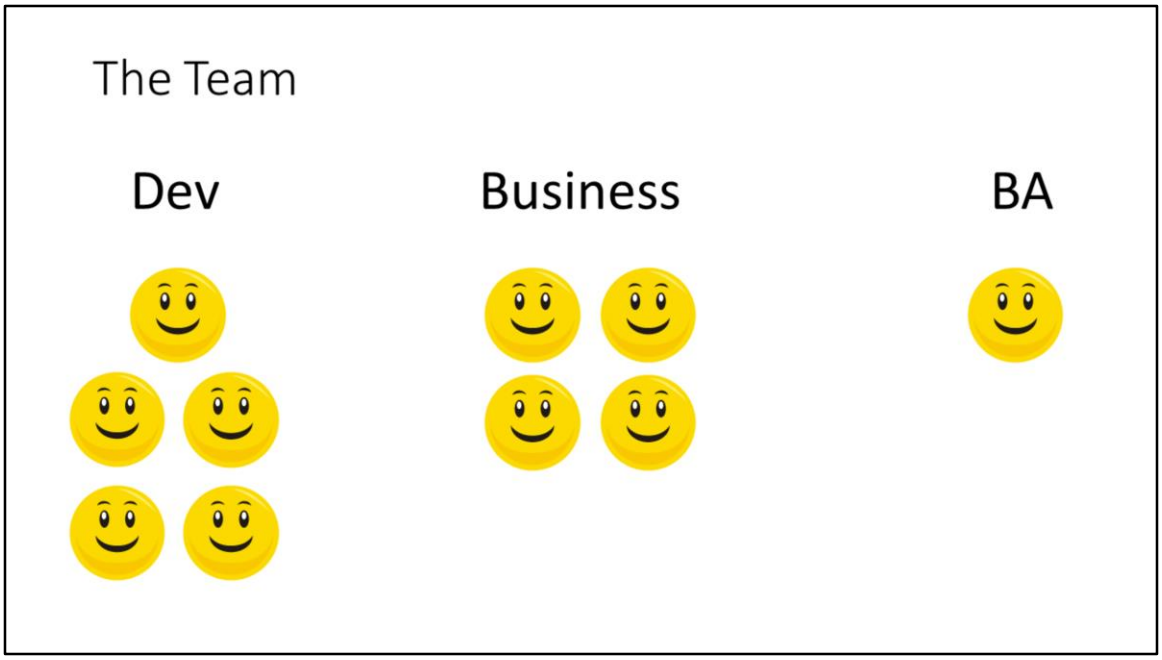
This guy has something called “The Process”, and it seems to work. It emphasizes planning, teamwork, trust, honesty, reliability, hard work and discipline. These are the same things good agile techniques seek to build, and the results can be very tangible in the corporate world, too.

Photo credits:
http://blog.al.com/bamabeat/2008/08/tide_coach_nick_saban_lands_a.html
http://www.al.com/sports/index.ssf/2011/06/will_auburn_have_a_50-year_dro.html
http://www.cleveland.com/browns/index.ssf/2012/12/nick_saban_happy_at_alabama_cl.html
http://www.ajc.com/weblogs/college-recruiting/2013/jan/20/nick-saban-presence-greatness/

## Level setting




Just as Visual Studio is s super pimped out text editor, agile methodologies are pimped out ways of managing a to-do list.




This guy has something called “The Process”. It emphasizes teamwork, trust, honesty, reliability, hard work and discipline. These are the same things good agile techniques seek to build, and the results can be very tangible.

At this time, the company I worked for was undergoing significant changes.

## Ch-ch-ch-changes

- 2005 - Established 3PL and warehousing, dedicated to one client ($250mln), 50 FTE
- 2009 - Managed freight and 4PL ($1bln), 200 FTE, several clients
  - First successful managed freight in food service

- McDonald's Supplier Innovation Award
- McDonald's Supplier of the Year
- William B. Darden Distinguished Supplier
- #32 on InformationWeek's "250 Most Innovative", 2010

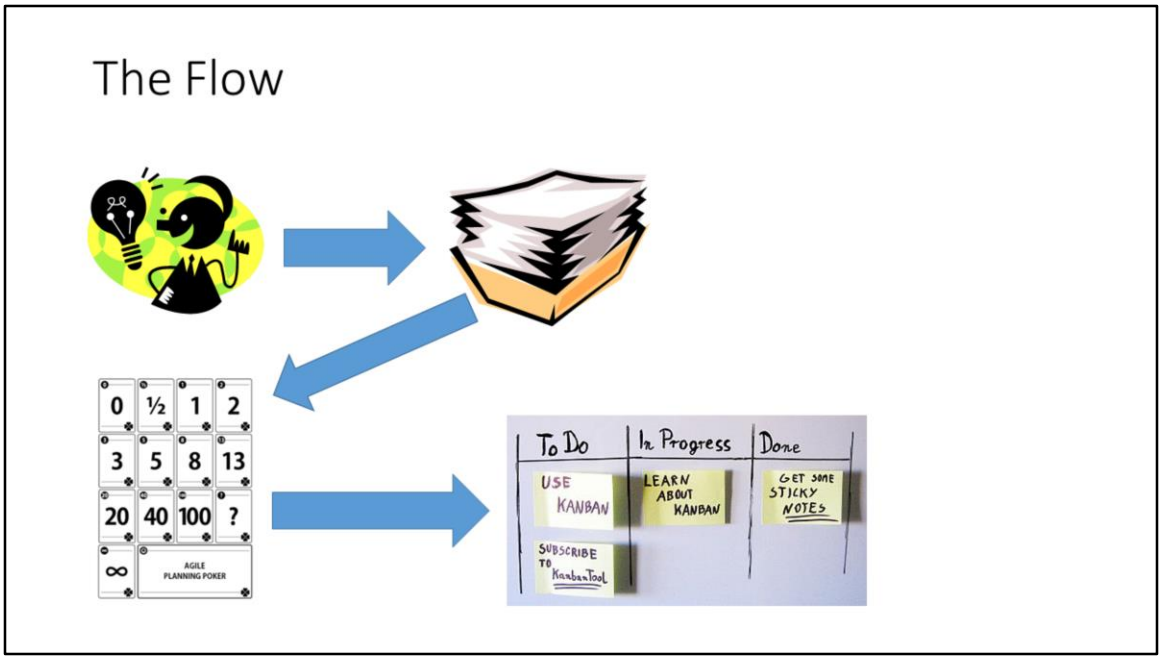
Our situation was an established 3PL warehouse operator, and began providing managed freight and forward (4PL) warehousing services. We were the first successful managed freight implementation in food service, and probably the largest scale yet. We grew from 50 to 200 people, and $250mln to $1bln in just a couple years. In order to make the 4PL and freight services a success, we had to have a lot of flexibility and responsiveness to client needs. The software was being written literally as the business was, so an agile methodology worked great for us. Many agile processes have a defined release schedule with iterations (containing several sprints), but our mode was responding quickly to changes, so we planned on a sprint-by-sprint bases for most of our work.

All the hard work paid off in the end, we were awarded a McDonald's Innovation Award, a McDonald's Supplier of the Year, a William B. Darden Distinguished Supplier of the Year and were listed #32 Information Week "250 Most Innovative" in 2010.

## What we did…

- Integration with JDE and LeanLogistics
- Custom onboarding tool for carriers, warehouses and routes
- Data warehouse (and ER/IQ) from JDE and LeanLogistics for reporting and integration
- Converted Access-based reports app to self-serve web based solution
- The “Cash Register”
- RedPrairie WMS implementation, customization and reporting

We needed better self-serve information with clients as well as internal LOB apps and systems integrations. In addition to the custom reporting, freight planning and management applications, we were also tasked with data cleanup and systems integration with JD Edwards and implementing a new WMS.




Ideas need to be captured as features and placed into a backlog. Features are selected from the backlog and estimated until capacity has been reached. Work then begins implementing these features.

Have a process for recording those thoughts and ideas that pop into your head at odd times. If you have an agile tool you can access at any time, update it. Otherwise, carry a small notebook, use your phone to email yourself, whatever. Don't lose the idea, you may not get it back until it's too late.

Since there were no agile products available at that time, we kept everything on index cards. The story went on top, and tasks and tests were stapled underneath.

## User Stories

As a customer service rep, I would like the order to be totaled automatically when I am done entering an item.

The most basic element of a to-do list is the thing you have to do, which in software development are usually called “user stories”. Stories are summaries of what needs to be done. There is some pretty heated discussion as to what form stories should take. There is little disagreement about user stories. User stories usually correlate to features, and are phrased “As a customer, when I save a transfer request, the designated amount of money should move between the selected accounts”. The nitty-gritty of input fields, validations, etc. aren’t essential at this point of the process.

Technical stories are what are generating a little controversy. Technical stories are things like “apply SP1 to all SQL Servers”. Some opponents of technical stories say these shouldn’t be presented to product owners, but we found it was helpful to let them know what we were up to, especially because service packs mean you need to schedule an outage. Also, if there are requests for features which would be easier or better implemented in a newer version of the software, you can enlist their help in getting approval for the money and time to make the upgrades happen. This was our case with the new geographic features in SQL Server 2008, and with the support of the business side, upgrading was deemed a no-brainer. Pro tip when discussing upgrades: include the phrase “sets us up for future success, rather than creating legacy systems”.

Everything you do during a sprint needs to be a fully carded story, task or test. If you need to, add them as necessary. Keep the stories simple. Sometimes, you have to add things like "move database to another server". There's no user story, it's purely a necessary technical task, but it takes up a measurable amount of time.

In very little time, you'll have a pile of stories. This is your backlog. You need a system to manage backlog. We used 3x5 index cards, and a small basket served as the inbox.

## Initial estimation


When you enter items into the inbox, add a t-shirt size estimate (S, M, L, XL). Best to do it at this time, when the idea is occupying the front of your mind. It can always be changed later, so don't worry about absolute accuracy. It was a gut feeling for the amount of work necessary, and was used when filling the sprint. Not tied to actual time, just a relative amount of work.

Image from our friends and local company, Café Press.

## Sprint Planning

| Monday | Tuesday | Wednesday | Thursday | Friday |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| Panning Lunch Preparation | Daily standup Work | Daily standup Work | Daily standup Work | Daily standup Work |
| 8 | 9 | 10 | 11 | 12 |
| Daily standup Work | Daily standup Work | Daily standup Work | Daily standup Work | Prod push Lunch Retrospective |

The goal of each sprint is to improve a certain product (although sometimes you have to throw in a "clean up a bunch of small stuff in a bunch of products" sprint). During our planning sessions, we broke all stories down into tasks and test cases. This is where the dev team communicating between each other was essential. You don't want to plan as you go during a sprint—you want to be on autopilot, cranking out good code.

When you have a choice, pick off low hanging fruit and greatest misery. Low hanging fruit is easy and gives you something to crow about. Greatest misery gets other people talking about you, which is great. Pro tip: time something big for a few weeks before annual review time.

We estimated time in each sprint as two business weeks, minus two days (Day 1 being planning and setup, day 10 morning being push to environments, PM being retrospective). We would get some work done on day 1 usually, but typically we'd usually install updates to VS, .NET, set up branches and daily builds, additional libraries, or do a little research. No push happened Friday afternoon, if we didn't hit Friday AM, we scuttled until Monday and shortened the next sprint. We figured 6 hours a day of programming time, and we protected this time pretty fiercely. Myself,

the BAs and prod owners ran interference to keep this time sacred.  We counted on 2 hours a day spent with daily standup, clarification of tasks, boot time, other distractions.

We did nightly builds, everyone updated all related libraries each day, which gave us the potential for testing and feedback quickly.  As features are completed, we'd deploy to the test environment and have the product owners test the feature.

We didn't count BA or product owner time in our projections, we focused solely on the dev effort, but we knew we had to deliver small builds during the sprint that they could test and provide feedback.

One of our favorite sprints was the holiday sprint - pile up a bunch of niggling little changes (renaming variables, reducing smells).  No real work, just cleaning off the annoyances, and let the skeleton crew have an easy couple weeks.

## Estimation

http://en.wikipedia.org/wiki/Dogs_Playing_Poker

At the beginning of every sprint, we'd spend the morning planning. We'd pick the features we were going to work on—greatest misery, hottest potato, low hanging fruit, it was more responsive to requests than scientific.

One user story or one technical objective = one feature. Each feature represents one or more tasks which need to be done. The more tasks in a feature, the bigger it is. Break the work down into specific actions--"add x field to the database", "change this stored procedure", "change the DAL", "add this field to the UI and code behind", "adjust this report". You're discussing the work, so record it while it's fresh.

I use the word "task" or "tech objective" rather than bug or feature. In the end, it's all work that needs to be done, some sort of change that needs to be implemented. And not everything is a feature or a bug fix, some things are just stuff we need to do.

Most tasks (or groups of tasks) should be a couple hours, UI might take a day or so because UI just takes that long. Don't estimate less than an hour. It takes 5 minutes to get VS started, plus the source control ceremony, documentation, test builds, testing/QA, time tracking, etc. You're not estimating the amount of time you're spending typing code, you're estimating the total time which needs to be spent on

the task and associated activities.

We estimated time using planning poker. It was both fun and effective. We'd discuss each task, then play our cards to estimate the time. The discussion was essential--doing this revealed the different understandings of what needed to be done, and usually simplified or clarified all the individual tasks. We grouped related tasks by stapling cards, one dev would work on everything related. This greatly reduced wait time or blocks. We used a regular deck of cards for straight time, ace=1 and going up from there.

If a certain resource in the business is going to turn a 2 hour task into an all-day event, put that in your estimate and be clear about it. You'll see those problems disappear over time as you build your team's cred.

Why time, instead of story points? It all comes down to time. There are a fixed number of hours you're planning for in your sprint. How do you know exactly what to promise the product owners? Planning poker is the best thing ever. I preferred straight up hours estimates, using a regular card deck. The most accurate estimates of time came from developers talking amongst one another. The least accurate are when the PM is standing in front of you asking "how long is this going to take?"

## Scrumboard in a Cube

http://rjdudley.com/blog/2008/10/06/adopting-scrum-scrumboard-in-a-cube/

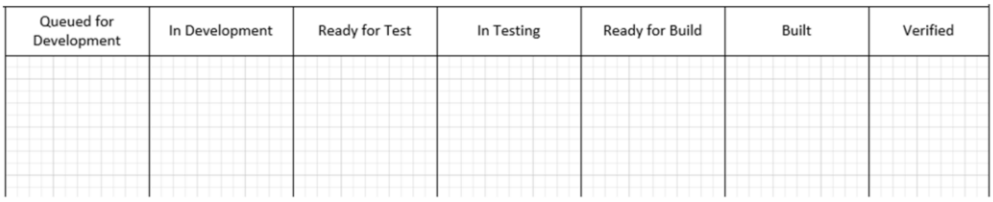
For organization and visibility, we used a kanban board, which was just index cards held to my hutch with magnets. Devs claimed a card, and put initials on the front and move it to the next status. The Task Order Up pages from David Seah's Printable CEO are also really good for this. I don't recommend starting with software, because you don't have a good idea of how to select the option that's right for you. Many tools lock you into their way of doing things, and you may be annoyed at the lack of flexibility, or possibly the lack of rigidity. Do a couple rounds on paper before looking at tools.

Back of task cards had the names of DB objects, classes, etc which were touched and needed merged in or migrated. As part of my morning routine, I kept a list of what had been touched so I could look for problems as I did the merges.

As we completed tasks, we marked "almost actual" time on the cards, and if we missed estimates, we made sure to understand why at the end of the sprint. We weren't anal about time tracking, if you needed a coke or a bio break, those were part of the time, too. Time was just kept manually ("I started this about 9:15, and finished about 11:30, so 2.25 hours"). This was more an exercise for us to match reality to guesses.

We didn’t watch individual burn down, since it really doesn't measure total work. If a dev causes other work or technical debt—writes code that needs to be refactored, or is buggy—you have to count the rework against that dev’s productivity. This is something that is a little squishy, and can be greatly alleviated by code reviews. Since we were a small team, it was easy for us to monitor when one was struggling a little and help out when necessary.

# Swimlanes


Queued for Development
In Development
Ready for Test
In Testing
Ready for Build
Built
Verified


Commit early, commit often! Having a secure backup of your code gives you the flexibility to make all kinds of changes, try new things, and the security of being able to completely revert it all back.

Options at this time were VSS, CVS and SVN, and we used Subversion. SVN wasn't as smooth as in Git is, but it was far better than the other two options.
The shared libraries and logical layers (like the DAL and CBO) were all in separate solutions, and had their own repositories.

Our branching strategy underwent several changes. Since Subversion is a centralized, we had a main trunk, and each dev had his own branch. Devs made sub-branches of their own as needed. When a feature was completed, commits were made to the feature branch. Daily builds were made from the main branch. It was a little dicey and took a lot of work. If there is one recommendation to be made for today, use Git.

We had nightly builds against the main trunk, and DLLs for the shared libraries were dropped into a network folder. Part of the daily task was to copy these DLLs to your local machine, and a pre-build step copied the DLLs into the output folder. If you need a build tool, talk to Alex Papadimolous about Build Master Pro, a sponsor for

codepaLOUsa.

With so many different, separate solutions, keeping the code quality high started to get tricky. TDD was gaining traction in .NET, and we started adding tests to our code. It wasn't easy, but we kept at it. Some of the best tests are written as you write your code. For instance, if a method falls apart when a null value is passed in, you need to handle that value as well and write the test as you think of it.

When testing code, the rule of thumb is to try for 100% coverage, but that's not really enough. You need like 500% coverage. You need to test every path through your code, and throw both positive and negative tests into the mix.

Acceptance tests were all manually performed by the person requesting the feature.

The techniques and tooling have improved for all manner of testing, and the importance of testing cannot be overstated.


Everything here actually happened to us, and then some. I wrecked a brand new Thinkpad with a large chai latter days after receiving it. The bacon factory caught fire and burned for days, disrupting orders and shipments. Storms, floods and major snowstorms took us out of commission at one time or another. These are macro level happenings—the business usually doesn't get upset for missing a deadline when the entire city is shut down for several days. Lightning struck our building, taking the entire network offline for a day. You need to be prepared to break the sprint and go all hands on desk to help out.

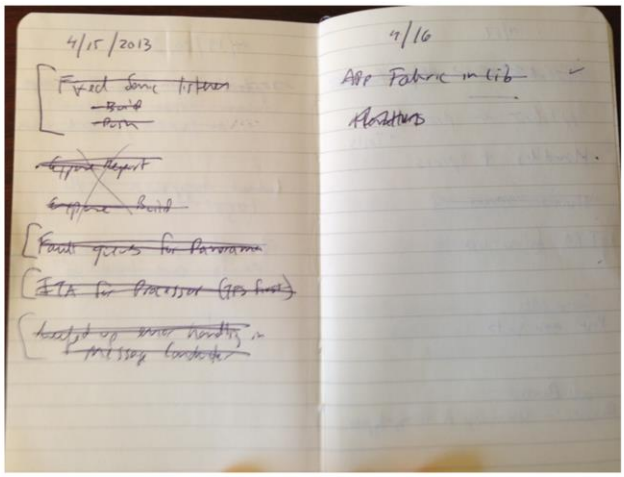
Micro level happenings are what affect your team, and are what usually damage your credibility.

The first couple sprints usually go pretty well, you're setting up projects, building base classes, things look easy. Then you'll come to a sprint where you try and tie everything together. You'll find dependencies you missed, inconsistent implementations, things that will require a lot of rework. You'll catch heat for this, and you want to try and prevent it. Being agile doesn't absolve you from planning. Look at cards in depth, look at contingencies and dependencies, and discuss interactions when planning. That is time well spent.

Don't beg forgiveness or try and explain anything away. If you f'd up, accept responsibility. You missed an estimate because _____. Being forthright and open goes a long way with the business.

If you don’t take your time planning, it’s very likely you’ll have a "we just proved waterfall is a good idea" sprint. It’s bound to happen at one time or another. Don’t blame, keep your team together and persevere. It’s only momentary, you can regain your momentum and reputation, and your team will emerge stronger if you support one another.

## Personal Standup




In addition to my cards, found it helpful to have a small checklist of all the things I needed to do that day, especially follow-up tasks ("is the server provisioned yet", etc.) This is something my dad got me doing. My dad still uses a Day Timer planner, and has for longer than I can remember. I prefer the little Moleskine notebook, but I've also tried using a portable wiki (Screwturn can be run from a thumb drive) and some more of the pages from the Printable CEO. There is also a huge selection of productivity apps. For me, the little notebook has worked the best. To each their own.

Don't use something where you can just roll incomplete tasks over, you want to recreate the list first thing every morning, reflect on yesterday and plan for the coming day. A tiny personal standup. Keep your notebook updated through the day with the unplanned work you do—updating libraries, etc.—as a reminder for what went on.

Don't mix personal and work in the same place, try and keep them separate. Use two notebooks if necessary.

## Continuous Improvement

- EF Code-first migrations or SQL Source Control
- Team Pulse, YouTrack, TFS, Axosoft, sprint.ly, Team City
- Build Master, TFS, Visual Build, Hudson, Jenkins, TeamCity
- NuGet, ProGet, MyGet
- Git
- Production diary

"If we had then what we have now, what would I have used?"

For databases, either code-first migrations or SQL Source Control. Some of the stored procedures we used for BI reports were very complex, and would be a pain to get right in a code-first migration (you don't use T-SQL, you use OSQL, which is a slightly different syntax at the command line). One thing we didn't get done was an automated DB update deploy, but there are better tools now which can make this happen. We did use SQL Delta for manual database deployments.

There were no good agile tools at this time, but today we're flush with choices. Use one when you get a handle on your process. Trello is not very good, since you can't have dependencies and subtasks. TFS, Axosoft, sprint.ly, Jira, Team City, YouTrack—look at a number of options and get the right one.

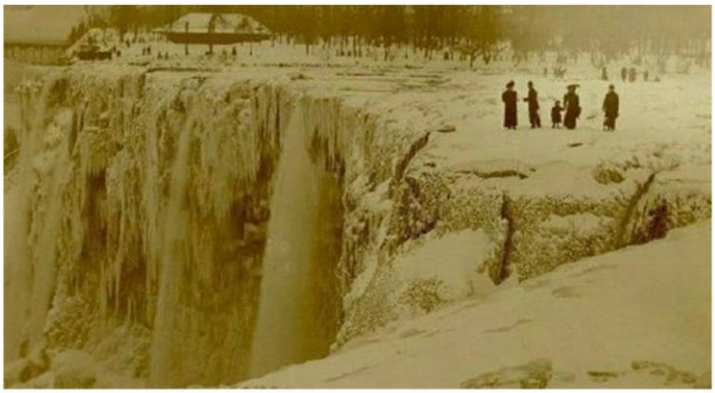
You also need a good CI tool. Again, TFS, or Build Master, Visual Build Pro, or the OSS tools Jenkins or Hudson. Automate as much as you can, the tools are so much better now and it makes your life so much easier.

Use Git. If you're a SAS-70 compliant space and you are stuck with TFS, you can still

use Git.  We use Git for everything until the final push.  Only the code for the final push is what gets checked into TFS.  Develop a good branching strategy, and don't be afraid to modify it as you gain more experience.  If you're not improving something after every sprint, you're not paying enough attention to how you work.

Keep a production diary—another small notebook of production changes.  What features were in what push, versions, branches, etc.  Maybe some of the reasoning behind places where you compromised.  Any technical debt or things to improve in the future should be put into the backlog right away.

## Waterfall



http://www.niagarafallslive.com/facts_about_niagara_falls.htm

Any time you have a defined delivery date and prescribed set of features, you're waterfall.

Being agile in a waterfall world is tough, but can be done. Usually you have to work with a defined due date, and plan everything up front. When this happens, this is usually a sign someone doesn't see development as a process of continuous improvement, but instead as a form of finite goods manufacturing. The conflict between waterfall and agile stems from a difference up front. Waterfall requires you to solve the problem up front, but the great thing about agile is that you don't have to solve everything up front.

Just because you're locked into waterfall doesn't mean you can't use many of these techniques here. Treat the cycle as a long sprint. Plan more up front, but use the backlog estimation and poker to get good estimates. Keep track of tasks with a kanban board, the visibility will be appreciated. So will the frequent updates of which feature is completed, so dependencies can be marked off in Project.

## BBQ for the Brain

- Won't be perfect from the start
- Agile != sloppy
- Be tolerant of iteration
- ~~If~~ When this changes…
- No dogma, unless dogma works for you

It won't be a perfect system at the start. Experience helps, when you gel as a team and start seeing commonalities in the code you write and the things you need to do. There will be bumps, there will be changes, but that's all good and all part of establishing a process.

Main takeaway (and a significant point to emphasize if you're introducing agile) is that agile is not sloppiness in action, it's not flying by the seat of your pants. It's fast moving and carefully orchestrated and flexible when necessary, designed for both communication and productivity.

The goal of an agile process is to quickly improve applications with few defects. This works by being well planned, with the details of every task clearly outlined, having every interested party on the same page, and everyone with something to do having the overall picture.

One of the most important concepts of agile is being tolerant of iteration. It's also one of the hardest to accept for those who practice a more traditional methods. Everyone involved needs to be aware that there is a difference between incremental enhancement and going back to do it over. Make sure everyone understands this.

Sometimes you partially implement some functionality, and this can seem incomplete to the product owners. There is a difference between technical debt ("we know this is bad, we'll clean it up later") vs. laying a foundation for future work. An example is the requirement for user logins. You can install the database and role providers, and put a management tool in place in a later sprint. As long as it's all completed by the end of the iteration.

Never, ever use the phrase "if this changes". Use "when this changes". Adjust tasks accordingly--build for now, or send back and make people think about it some more. The biz can agree to move things around and prioritize--this way, they won't be surprised.
You need to be a bit of a psychologist. When you use "if" that triggers a response, and includes an element of the unknown. When you say "when", that triggers a different thought process. People are prepared for change.

Eschew the idea that if you are not precisely dogmatic you're doing something wrong. If you're stumbling around for a bit, trying different things and incorporating what works, you're probably doing it right, and this talk is for you.

## Second helping of BBQ

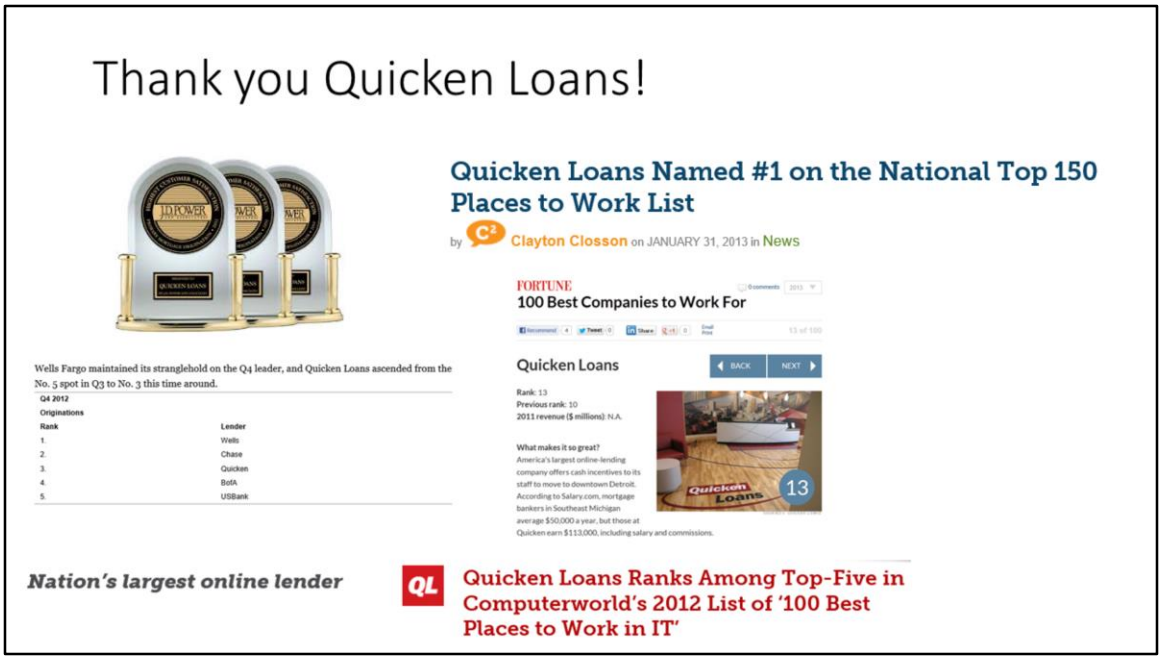
- Agile Manifesto: http://agilemanifesto.org/
- "Is Agile Dead" podcast? http://www.dotnetrocks.com/default.aspx?showNum=838
- Estimation by Joel Semeniuk: http://channel9.msdn.com/Events/TechEd/NorthAmerica/2009/DPR205
- http://agile.dzone.com/articles/10-scrum-methodology-best
- http://agile.dzone.com/articles/treat-technical-stories-user
- http://thecodist.com/article/what_the_hell_is_really_agile
- http://visualstudiomagazine.com/Articles/2012/11/07/Story-Points-in-Agile-Development.aspx?Page=1
- http://codebetter.com/johnvpetersen/?p=545
- http://dylanbeattie.blogspot.com/2012/08/planning-poker-with-lolcats.html

Here are some resources with some additional information or things to think about.

## Handy Things

- Printable planning poker cards http://www.tekool.net/blog/2009/07/21/printable-agile-planning-poker/
- PlanningPoker.com
- BitBucket.org = Git, free for 5 users, private repos
- YouTrack = free for 5 users, self hosted
- http://tfs.visualstudio.com/
- http://davidseah.com/productivity-tools/

Here are some resources I find helpful. David Seah has the Printable CEO series of productivity tools, and is on f my favorite sites for personal productivity matters.




I have to thank my employer Quicken Loans for sponsoring my trip to tell you about the work I did at a previous employer.

Quicken Loans is the nation's #1 online mortgage lender, the #3 lender overall, and no matter which survey you read, one of the top companies to work for.

If you need a mortgage or a great technology job, let me know!