Hi, My name is Troy Hurteau and I am a programmer for the NCSU Libraries. Today I would like to talk about using Agile for project management. It is something we have been doing in my development group for a couple of years, and hopefully this presentation will give you some ideas on how you might improve your own processes.

The presentation is titled, "Agile in Higher Education IT: Myths and Methods". Agile is not native to the higher education environment, but rather comes from the corporate world. Adapting Agile to our own organizational culture takes some consideration.

I use the word "myths" because Agile isn't very well understood, and has become a bit of a buzzword. The methods do have merit when applied judiciously, but Agile is no silver bullet.

To begin, Agile is an umbrella term, not a brand. Before the 1990's there were a number of software development methodologies that were designed to solve some pretty significant problems that are typical in software development. As it turns out, many of these issues are not uncommon in other types of projects.

I would love to explore Agile for project management in general, but since we don't have much time today I'm going to keep the topic focused to IT projects.

Most of my experience with Agile comes from software development specifically, but our department has been pretty happy with what Agile has done for my team and is interested in applying Agile techniques to projects in our desktop support group as well.
Agile developed as a reaction to the software development methods popular prior to the 1990’s.

The methods of that time mostly focused on meticulous planning and requirements gathering up front with the intention of building and delivering complex software systems in a single push.

If you have heard of the Waterfall method, it is a paragon for the methodologies popular in the corporate world at the time.

That is not to say that Waterfall does not work. There are many things that work.

A lot of things I will present today may seem like common sense, and that is because they are. You may already be doing some of these things even if you are not familiar with Agile.

Agile represents a way of thinking about process that reminds us about some common sense best practices. We all need reminding sometimes.

So my question is, “What are the people in these pictures doing?” (someone’s version of Agile, but it doesn’t look anything like what my team does)

Now to give you an overview of how the presentation is structured, I will begin by building a shared definition of Agile.

Next I will illustrate the basics of Agile by going a little more in depth with how we have been applying Agile to software development in my team.

Agile isn’t just about process, there are tools that help teams work together and individuals get their work done. I’ll talk about some things we use.

After that I will talk a little about some of the assumptions that Agile makes coming from the corporate world, and how we might adjust it to higher education institutions.

Finally I will explore how you might go about extending the benefits of Agile to the broader IT enterprise.
The worse possible way I can think to define Agile is to try and describe the myriad of existing Agile methodologies. There are a lot of them, and there is a small set of things that they all truly have in common.

Instead I'll begin by talking about some of the core concepts of Agile.

The keystone of Agile is "teamwork". Everything in Agile is geared towards helping people excel at work together.

Teamwork isn't necessarily the most efficient way to develop software. There are some advantages to the lone-coder.

The problem is that many projects are too complex for that approach to be practical. Projects often require a diverse skill set, and they also will often have deadlines that make working alone impossible.

Because we are working as a team, communication plays an important role.

Projects involve a lot of unknowns. The team has to learn about the problem they are tasked with solving. They will have to communicate to coordinate their efforts and to work effectively they will need to develop a shared understanding of the goal.
Defining Agile

Projects have a goal: to produce something that meets needs. Meeting these needs creates value.

Communication plays a major role in discovering and understanding needs. A good solution can't be crafted without this understanding.

Not only do projects have to meet needs, often they must identify them.

The team gathers requirements to help them understand what is needed. Some Agile methodologies call these requirements "User Stories".

A core concept that is distinctly Agile is working iteratively, specifically working in short cycles.

It is important to distinguish a "cycle" from a "phase".

The point of a cycle is to deliver a usable product. It might not be a "finished" product, meaning it won't meet all needs within the scope of the project.

Instead what is available at the end of each cycle is a product that delivers value because it meets more needs than what was available before.

In this way we say Agile teams deliver value iteratively.

Defining Agile

Needs, value, requirements

- Projects meet needs to deliver value
- Needs are initially unknown
- Gathering requirements, user stories, helps the team discover needs

A project, a team, communication, meeting needs... so far this sounds a lot like any project...

Nothing special yet
Defining Agile

Phases and the Waterfall method:
- Phases don’t release value
- Imagine a six month project...
  - Month 1, gather requirements
  - Month 2, build prototype

In a traditional methodology each phase builds on another, but no phase other than the final phase is intended to deliver a "usable" product.

Imagine a project that takes six months to complete. In the first month the team carefully analyzes the needs. In the next month they come up with a prototype.

You wouldn't want to use a prototype to do real work. Prototypes lack features, userfriendyness, reliability.

Defining Agile

Still working on that six month project...
- Month 3, work on architecture while waiting for prototype feedback
- Month 4, finally start real work
- Month 5, finishing touches
- Month 6, usable product...if it passes quality assurance

Another month after the prototype the team has all the feedback they need to round out the features. Assuming a best case scenario they were doing some low-level architectural work in the meanwhile.

By the end of the fourth month all of the features are done. We're over half way in now, no time for major changes.

In the fifth month they have to iron out some of the workflow issues or come up with documentation on the convoluted ways the software has to be used.

If all goes well by the end of the sixth month the product has been tested... and passed.

It is ready to be used.

Defining Agile

What if...
- Team learned as they went
- No throwaway prototype
- Feedback all the way through

That's Agile!

What if we compartmentalized the work differently so that:

The team could discover the unknowns while working on the knowns.

We didn't waste time on a throw away prototype.

Feedback was more evenly distributed through the life of the project.

That would be Agile. If cycles are scoped to be a month each, the team could have released a gradually improved product 6 times.
Defining Agile

Adapting iteratively

- Waterfall places all risk hunting up-front
- Hands-off approach while risk is assessed
- Iterative teams build understanding of needs and risks each cycle

Aside from the potential to deliver value iteratively, working in cycles makes the project adaptable.

In a traditional approach the team would try and hunt down as many risks as possible before work begins. This is accomplished by trying to understand the needs of a project in a very hands-off manner. Look, but don't touch, if you will.

The Agile approach is to spend a little time learning about the needs at the start of each cycle and identify the most obvious risks. In later cycles, as the team's understanding grows from hands-on experience, less obvious needs and risks will emerge.

Defining Agile

Adapting to change

- Rapid pace of change
- Products have expiration dates
- Working iteratively to delivery early, often
- Prioritized work

Agile’s cyclical approach also makes more sense when it comes to addressing change.

Needs, like technology, change at a rapid pace these days. It doesn’t do much good to start a six month project when the nature of the problem could be fundamentally different in a year or less.

Working in cycles allows the team to get a solution in the hands of people that can benefit from it sooner. Instead of exhaustively trying to understand and analyze needs, the team prioritizes work based on what can be done now, that will have the best return on investment.

Defining Agile

Important assertions: short, done

- Short cycles address changing need, effort waste, unforeseen road blocks
- Agile helps firewall failures
- Cycles must release something usable, "done"

Before I go on to some of the more cursory features of Agile, I want to reiterate two important points about working incrementally.

First, cycles should be short. I can’t tell you how short because that really depends on a lot of factors. The risk of changing needs, the risk of wasted effort, the risk of unforeseen road blocks all factor in. Agile won’t prevent failure, it will help you prevent a cascade of failure.

The other important thing is that each cycle must represent something that can be considered usable. Sometimes when we say done we mean sort-of-ready.
Defining Agile

On the outer skirts of the Agile solar system there are some important concepts that you may or may not find in all variants. These all play on the central themes we have already covered.

Teams, individuals, respect

- Giving team members freedom
- Limit solo work

Teams are important. Thinking about how you make a successful team is baked into many Agile methodologies.

It might be as simple as making sure that individuals on the team have reasonable freedoms, such as estimating or negotiating their own work. Not to plug the NCSU crowd too much, but I have heard there is a “Planning Poker” presentation later today and that is one way some Agile methodologies recommend.

Some methodologies go so far as to say no-solo work is ever done. Extreme Programming for example advocates Pair Programming only.

A more moderate approach is to encourage pair and small group activities based on what makes sense, and what feels effective.

Pair activities give team members the chance to grow closer. It is also a good way for people to develop their technical skills, leadership skills, and mentoring skills. Everyone on the team has strengths. A truly Agile approach highlights, harnesses, and rewards the unique value each team member brings.

Emphasizing pair activities encourages cooperation without the need for exercises in design-by-committee.
Defining Agile

Short meetings
- "Stand Up" meetings
- What you did yesterday
- What you will do today
- Importance of co-located teams

Cycles are not the only thing that should be short.
Meetings should be as short as practical. Several Agile variants recommend "stand up" meetings, 15 minute daily meetings where the team updates each other (and the project manager) on what they did yesterday and verify what they should be doing today.

Co-location of the team is also widely viewed to be important. This directly counter-indicates the outsourcing craze that has swept the corporate world.

Defining Agile

Agile and outsourcing don't mix
- divisions caused by outsourced work, federated efforts
- avoid crossing lines and creating interdependencies

Some Agile methodologies try to adapt to outsourcing within the scope of a project, but most strongly discourage it.

I am not saying Agile means you can't outsource or federate work, only that you have to be careful about allowing such resources to straddle project lines.

These types of disjointed efforts and relationships create significant risk.

Defining Agile

Prioritizing work
- Capturing what is important in requirements
- Deliverable work, versus artifacts of work
- Keep things simple

Finally, prioritizing work and value cannot be emphasized enough.
If something is really a need, it should be captured as a requirement and quantified in some manner so that it can be prioritized.

It is important to draw a clear distinction between things like documenting the product and documenting the process. One has clear value to an end-user, the other has circumstantial value to the team.

In order to be Agile, we can't allow ourselves to get bogged down in little details. When it comes to prioritizing, don't nitpick. A three to five tier scale should be more than enough.
To give you a better picture of how Agile plays out I will run through an example of how my team works in a cycle.

Please keep in mind this is one way to do Agile. Our way is a light version of a popular method called Scrumm.

We always begin with planning. Sometimes there are new needs that come from a product team (representing users). Other times these are unaddressed needs from previous cycles.

We put the needs into some collection or repository for all known task. This exhaustive wish list is known as the backlog.

If there is any task that might need doing, and it isn't in the backlog, we add it.
We prioritize the backlog based on what we know and what our product team advises.

We consider what things have to be done before we can start work on others. What things will deliver the most value?

Now, out of all of the things that are in the backlog we need to select a number of tasks.

As a personal rule of thumb I like to suggest that cycles and tasks should be scoped in length and complexity so that for each team member, there are seven or fewer tasks per cycle.

Another general rule is that as much as possible, no task in a cycle depend on another. Sometimes this is unavoidable.

Performing a task takes time. In order to plan a cycle all of the tasks must be estimated and the result must be a total that the team can realistically accomplish.

Another general rule I like to use is that the average time per task in a cycle should be less than 8 hours. It is important to recognize that people can’t be expected to spend 100% of their time in a given day on a single task and that switching tasks has an associated cost.

It is also important to consider that estimates depend on who is working on them.
Agile for Software Development

Focus on doing
- Plan only for the cycle
- Transitioning from six week cycles to three

Ideally we spend very little time in a given cycle meeting and planning what we will do in a cycle.

Most of the time in a cycle should be focused on "doing".

Initially, my team worked in six week cycles. The first week was focused on planning, but often included some programming too.

Recently we have started trying shorter cycles. To make this work we try to overlap planning for the next cycle with wrapping up the current cycle. We take on fewer tasks and work more collaboratively.

Agile for Software Development

Deciding what to do and what to delay
- Trivial things can be tacked on, but
- Don't let surprise tasks ambush productivity

Working in an Agile team, like anything else involves decision making. Agile doesn't have much to say about work at the task level.

If you are scheduled to do something, you do it and will often want to record your effort. We have software that I will show shortly for tracking what needs to be done, what is known about the task, what we did, and what is finished.

When you run into something unknown or unanticipated there are guidelines in place to prevent team members from being ambushed.

Agile for Software Development

Expect the unexpected
- Adapting means making compromises
- Decide to backlog or escalate
- Team decides when individual can't

There is an expectation that unexpected things will occasionally happen. When they do the individual or team has choices.

If it seems trivial for an individual to handle, taking care of the problem is always an option.

If it seems, or becomes too time consuming the team or project manager should decide how this new task should be prioritized. If it is critical then the compromise is to move some other task or tasks into the backlog. Otherwise the new task is backlogged.
To reiterate, working on a project is not manufacturing, we are not making widgets.

The kinds of unexpected things we deal with are not "standard deviations".

Unlike a model kit, the projects we are talking about don't have pre-fabricated parts as such. They also do not come with well defined instructions.

Meeting for a short time each day helps to ensure that no one diverts too much time to putting out fires and jeopardizes the team's ability to successfully complete a deliverable product for the cycle.

Team members can sometimes negotiate backlogging new requests themselves. Unless something really is critical and there is no reasonable workaround the default behavior should be to send things to the backlog before interrupting and renegotiating the work for a cycle.

Requirements are stated in the form of a verifiable fact, or have some verifiable facts as a component. Clear and specific requirements make sure that the team doesn't get to the end of a cycle only to find there is more work to be done.

There are different ways to quantify done-ness. The person working on a task might think it is done, but they may have forgotten some component. It is also possible that some critical part of a need was not captured in a requirement.
Testing in some form is the benchmark for done-ness.

A number of Agile methodologies advocate automated testing. Test Driven Development is a way to accomplish this. The idea here is to write some test-harness that reflects the requirements. Initially, the tests should all fail. When the tests all succeed, the task is done.

There are three challenges with Test Driven Development.

The first is learning how to do it. It isn’t something our team has much expertise with yet. As you might imagine it is a topic that could fill an entire presentation.

The second problem is that not all requirements are easy to test automatically. There are ways to design systems that increase the testability, but the idea of “full automation” remains very pie-in-the-sky.

Finally, hinging completeness on automated testing assumes that the tests themselves are correct, and they reflect the needs accurately.

Alternatively, my team tends to focus on a two-step process for task completeness.

The person doing the task tests the functionality manually, they put themselves in the role of a user and verify that the stated need is met.

As often and early as is practical, we release work to a testing server and try to get actual users to verify things work as intended.

This process relies on tools to ensure the cost of making changes available for testing is low. I will talk about these tools shortly.
Agile for Software Development

Cost of delivery, driving it down

- Waterfall: test to release
- Agile: release to test

The cost of delivery is an important consideration in Agile. In a traditional methodology delivery of incomplete products is a waste to be avoided. The resources needed to apply quality assurance to anything released can be high. One could say that you test to release.

Agile approaches this from the other angle. The tools we use focus on keeping the cost of delivery down so we can do it often. We release to test. Automated build tools, dedicated testing servers, and version control repositories all help with this.

Agile for Software Development

Reflecting

- What worked well this cycle
- What didn’t work well
- Suggest changes, but don’t implement them immediately

At the end of a cycle, after everything that has been done is released, we spend a little time reflecting on how things went. This is not a venting session, but it is good to be honest and talk about what worked well and what didn’t.

This is the best time to make suggestions to a change in the process, though it is reasonable to expect that the team will wait a cycle to think about and plan for drastic changes, even if everyone is on board.

It is important to note that "lessons indicated" doesn’t automatically mean "lessons learned".

Aside: Projects Fail!

I'm sure many of you are aware of the factor risk plays in projects. Depending on who you ask and how you define failure, anywhere from fifty to eighty percent (or more) of software projects fail.

If you define success in terms of fully meeting the requirements set out on schedule and on cost, most numbers I could find broke down into approximately 15% total failures, 40% completely successful, and 45% falling into the challenged zone. These challenged projects eventually complete either late, over cost, or both.
The first thing an Agile team will want is a way to track needs.

Even before moving to Agile our department used a tool called Jira, by Atlassian.

Jira is a reporting and tracking tool. It can be setup to allow users to report issues or requests, such as bugs in software. These requests are organized into projects and can be given priority.

Within a project, requests can be assigned to be addressed in a version. Complicated requests can be broken down into sub-requests, which can be assigned to different versions if needed.

Requests in Jira can be estimated, and assigned to different individuals who can log work and pass the request to others.

Our installation of Jira also uses a plug-in called Greenhopper which adds some additional Agile functionality.

In a version we can drag and drop tasks from "not started" to "in progress" to "done" and logged work is charted out to help us keep visual track of progress.

Other Agile groups use tools like Basecamp, or have physical boards in their office to track projects and progress.

Jira can also integrate with other services. It can send e-mails to developers based on some basic workflow rules, such as when work is assigned to them or a comment is made on a task assigned to them.

Using the unique identifier assigned by Jira, programmers on our team can easily associate any work the commit to our code repository with one or more requests.

Jira looks at the repository nightly and updates a view of what files were edited to accomplish a task. This can help the team detangle things if something they started on won’t be included in the next release.
A code repository is important for software projects. It is like a content management system, but for code. Our team currently uses Subversion (SVN).

Like Jira, our repository is organized by project. The repository has three basic components: a file system, a database, and a program that manages everything.

Each project has a sub-folder called trunk, that is the current up-to-date state of the project. The version in progress as it were. Every version of every file has a unique number so snapshots of each file and the whole project can be generated based on any single committed change.

Projects can have other sub-folders. One kind is branches. Branches are used when significant independent efforts are going on in parallel and there might eventually be a need to merge changes back into the trunk or to other branches.

For example, let's say you have a product with a version that people are using, 1.5. You want to undertake a major effort to create a new version, 2.0. It could take a while to have a stable version of 2.0 and you might end up having to release a version 1.6 to address some critical bugs in the meanwhile.

If all of the work goes into the trunk, it could be hard to keep the changes for 2.0 separate from 1.6.

Projects can also have tags. Our team tags every version as it is released to a production server. This ensures that if anything goes wrong and we have already started working on the next version, we know exactly what the code in production looks like.

Trunk, branches, and tags are all just folders. They can be browsed as such, but you wouldn't want to mess with the files directly since that can cause things to get out of synch.

The program and database run by Subversion allow the team to use tools to safely keep copies of those folders on their personal computers and on the server.
Agile Tools

Subversion

- Manages files, is not a file editor
- Primarily a command line tool
- Plug-ins ans stand-alone applications available

Using Subversion is certainly a lot safer than just copying files around and sharing a networked drive. When people are working together on a project conflicting changes are bound to happen.

Subversion allows teams two important affordances. First, Subversion is not an editor, it keeps files in sync. People on your team can use different file editors. Our team all uses some variation of Eclipse because we are programmers, but some people in other groups use tools like Bare Bones Edit which also has a Subversion plug-in.

Your editor of choice doesn't have to have a Subversion plug-in. There are Subversion tools that work in a stand-alone manner as well.

In addition to being free to use different editors, Subversion will help the team to solve conflicting changes.

If I work on a file, and no one else has touched it: I can do a “commit”. The same is true if I add a new file to the project or delete one no one has edited.

If someone else has edited a file that I have not touched, Subversion can tell me I need to get the updated version. Making sure you do this regularly will help avoid conflicts.

When you do have two people working on the same file, there are tools that will show you both versions side-by-side and even let you quickly copy or reject whole sections at once.

Once you feel that you have successfully resolved the conflicts by copying other people’s changes and keeping your own it is important to test to make sure your local copy still works as expected.

If it does, you mark your local copy as “merged” and Subversion then knows that even though there are conflicting changes still in the repository, you have addressed them. Now you can “commit” your changes as normal.

The great thing about a code repository is that even when you commit work, all of the old versions are retained.
Agile Tools

At the center of everything my team does is an Integrated Development Environment, the swiss army knife of software development. We use Eclipse or sometimes Zend Studio, which is a modified version designed more for PHP.

Eclipse has plug-ins for Subversion. It has features that allow it to automatically download software libraries needed by a project, automate building, and step through debugging.

The editor can catch common programming mistakes and help you correct them.

Agile Tools

Soft tools, techniques

- Stand up meetings
- Short and to the point

Not all of our tools are technology based. Some are more like techniques.

I've already mentioned Extreme Programming once, and it is one of the more infamous Agile methods. I'm not a fan of Extreme Programming itself, but it does have some important tactics.

The first is the daily "stand up" meeting. We try not to get mired in the details of how we will do things, or how to solve problems. The stand-up meeting can help spawn those discussions later including only the relevant people.

Agile Tools

Pair Programming

- Practice of only coding in pairs
- Not for everyone
- Production bottleneck

Another important technique that we like to use, though not as ubiquitously as is prescribed in Extreme Programming is Pair Programming.

Pair programming is the practice of coding only in pairs. Both people sit at the same desk and there is never any code written that doesn't involve the active participation of less than two people.

Some people really like this way of programming, but as you might imagine it doesn't work well for the stereotypical programmer, and in the worst case scenario it can become a production bottleneck.
Agile Tools

Pair Programming

- Collaboration cuts down on mistakes
- Builds teamwork and cross trains team

Because you have two pairs of eyes, when things go well Pair Programming is pretty effective. It is especially good when you have programmers with complementary skill sets, different levels of expertise, and a general willingness to listen and compromise.

Pair programming is a good way to build teamwork, and when you pair newer programmers with those that are more experienced it not only yields good code, it also facilitates the cross training of your staff.

---

Agile Tools

Harnessing experts for Pair Programming

- Specialists speed up the work for other team members
- Helps less familiar members learn from peers

Our team spends most of it's coding time programming solo, but everyone is the expert in at least one product and our specialists each have focus in specific technologies.

I, for example, know the basics of Java, but I'm not the Java expert. All of our Java applications are build using frameworks. Our other specialist knows the Frameworks inside-and-out, so when I'm not sure how to do something, I explain what is needed and he walks me through the proper way to build that in the framework.

Our team members are encouraged to seek out pair programming when ever it makes the most sense.

---

Agile Tools

Harnessing frameworks

- Reusable software component
- Works in the opposite way of a library
  - Libraries "plug-in" to your code
  - Frameworks give your code a place to plug-in

The subject of Frameworks brings up another important tactic with Agile: efficiency through reuse.

Frameworks are reusable software tools, much like a software library, or an API. They are used to speed up development by providing some functionality out-of-the-box.

What makes Frameworks different is the paradigm in which they work with what is produced locally. In a traditional Library or API, your code is the primary driving force. You are essentially plugging the outsourced work into your product.

Frameworks generally work in reverse, you are basically plugging your code into an already mostly working system.
Mores and Taboo

Nothing prevents disaster, but you can learn from it

- Lessons indicated
- Lessons learned

A final note on Agile from my team:

All of the tools and techniques in the world won't prevent disasters from happening. This could be a personal disaster, such as a bad hard drive. It could be a system wide failure, such as new version going out with an unexpected flaw.

The team works together, under the direction of the manager to react to disasters when they happen and to also try to anticipate likely disasters ahead of time.

Mores and Taboo

- No releases on Fridays
- Prefer morning releases to afternoon releases
- Push to testing servers (and test) before any push to production

Over time, we have learned things that tend to cause and prevent disasters. As a result we have a number of unwritten rules that we make sure new team members understand.

They are un-written because they are not set in stone. Sometimes someone will break them with the expectation that they will have to deal with the consequences.

We don't release new versions to production on Fridays. As much as possible we try to release things in the morning.

We test things on a testing server before rolling them out to production, even when it seems like a trivial change.

Mores and Taboo

- Tag versions when they are released to production
- Don't take on too many tasks at once
- Commit often
- Don't commit broken code
- Update often
- All rules can be questioned

We tag any version that gets released into production.

We try to work on code for a single ticket at a time so we can commit to the repository frequently.

We try to always commit and log work before leaving for the day.

We don't, however, commit code that is more broken than what is already there.

We update local copies before starting in the morning to reduce the number of conflicts that will need to be merged.

Most importantly, any of these established rules can be questioned. There will come a time when one or another no longer makes sense.
Mouse Trap (Never Works)

Projects and processes that are overly complex will suffer setbacks.

The point of Agile is to avoid this. If things start to feel too complex, scale back.

How Higher Education Is Different

Agile is from Mars
- Reaction to heavier methodologies
- Predictability over adaptability

As I’ve mentioned, Agile came from the corporate world. It was a reaction to the more complex and process-heavy methodologies at the time.

You can read books like The Mythical Man Month to get a sense for the climate at the time. There was an increased effort to try and drive software efforts efficiently, treating it like a manufacturing process.

Rather than consider adaptability, the common theme of older methodologies is to try to make success in software development a predictable, reproducible quality.

How Higher Education Is Different

Small and large companies collide
- Small companies couldn’t adopt heavy process
- That didn’t stop them from succeeding while small

Smaller businesses on the other hand could not possibly hope to employ a lot of the techniques that came out of these efforts. The metric-heavy recommendations made it impractical.

Two things happened. Eventually a few people in charge of large enterprise efforts got smart and realized more of the same wouldn’t produce better results, they needed something different.
How Higher Education Is Different

Smaller companies did fine without heavy process
Until they tried to scale

Smaller businesses could succeed at software development with similar success rates as large companies. Despite the disparity in resources, there was some factor balancing the equation out.

The smaller companies were challenged in that they needed some kind of process to scale successfully.

Ad-hoc process worked well for them with a handful of employees, but as efforts scaled up, something resembling the rigid processes that worked for bigger groups became more necessary.

How Higher Education Is Different

Competition and adaptive advantage are the driving forces behind the development of Agile

What lead large and small companies alike to come up with Agile was competition.

The adaptive properties became useful for teams in large companies only partly because the teams were thinking differently about how they worked and delivered results. Equally important was that the whole organization could think differently about how they worked with the software developers to get more effective results.

Competitive advantage by being adaptable makes sense. It was something organizations could rally behind.

How Higher Education Is Different

Competition and adaptation in higher education

This is one place higher education is different. We certainly do compete, but not in the same ways as the corporate world.

That is not to say adaptation isn't necessary. Because we have such constrained budgets, things can change rapidly.

We all like to think that our institutions are bastions of innovation too. In some ways we are more free to innovate than teams in the corporate world, even when we may have to do it on a much smaller budget.

Competition is a value in higher education, but possibly not as pivotal as it was in companies moving to more agile practices.
How Higher Education Is Different

Limited scale of higher education

- Cost limitations, outsourcing at the enterprise level
- Serving specialized local needs better with Agile development

In many cases we are not faced with the same questions of scale as the corporate world is.

Sure, we can talk about "enterprise" level systems and at the level of an entire university this is considerable, but most of the things we do locally are at the smaller scale.

There is nothing wrong with outsourcing on the larger scale of things. Even when we are working with purchased solutions and services there is room for innovation, and we'll talk about that a bit later with applying Agile to other aspects of IT.

The sweet spot for software development in Higher Education is at the smaller scale. That is where you find the more specialized needs.

How Higher Education Is Different

Different team-project mapping

- Industry focuses on one team per project or several teams to one project
- Higher education has overlapping teams, multiple projects

One way that my team in particular is different than anything I have read about is that we are a single team working on many different products. The most typical focus in the corporate world is to have multiple teams and divide efforts along the lines of distinct products.

Most Agile literature is written as if the team is focused on a single product over a series of cycles. They might even be working within a particular feature set of a larger application where other teams handle other aspects of the same software.

In this way our team is almost like a collection of different overlapping teams, and we each have multiple cycles going on at any given time.

How Higher Education Is Different

Narrow focus is rare for a higher education team

- Make this trait an advantage
- Agile firewalls projects from failure across cycles
- Broad focus diffuses the risk in any single project

This all goes back to budgets and scale. In higher education we probably rarely have the luxury (or problem depending on how you like to work) of being so narrowly focused.

Far from hurting our efforts, this approach does have one key advantage that could be argued to be more Agile.

Projects sometimes fail. Agile helps us firewall off cycles from each other so the amount of waste is contained in smaller time-boxes, but the strategy of working on multiple projects at once creates another opportunity.
How Higher Education Is Different

- Imagine how it would feel to have a month long project canceled
- Then a three week project
- Finally, only one of several projects fails

How would you feel if the only thing you were working on for a whole month failed, or was canceled?

Now imagine it is the only thing you were working on for three weeks.

Instead imagine something that was one of two or three things you were working on. Failure hurts, but when we have other things to hold on to and be proud of it isn't that bad.

The nice thing about Agile is it challenges us to think about things like success and failure in the light of changing goals and priorities.

How Higher Education Is Different

Agile reminds us of a common sense approach

- Use what works
- Avoid being sold a solution

Like I have said before, Agile will seem like common sense to some people because if you have been working effectively you may have already intuitively discovered the value in doing some of these things.

The important thing in considering Agile is there are a lot of different variations and there are also a lot of people out there trying to sell something.

Extending Agile

- Books on Agile for software development abound
- Enterprise Unified Process is on that explores Agile for the whole IT enterprise

Agile for software development is a pretty well covered topic. There are literally dozens of books I could recommend on the subject.

What there is a lot less of is advice on applying principles from Agile to other types of efforts.

The one book I know about on the subject is "The Enterprise Unified Process".
Extending Agile

- Agile has proven itself to our team
- Are other things we do a good match?
- There are some similarities in work across the enterprise.

Having used an agile process for a couple of years I can say it has definite value.

So Agile is not just something that sounds neat, it has worked in our case even though there might be other things that would also work.

Looking at the kind of work developers do, and comparing that to other IT professionals lends some credibility to the idea that Agile might be in some ways applicable to the broader IT enterprise.

Extending Agile

- Communication is still key
- Teamwork is central because skill sets vary
- Project based work
- Not everything in IT is a project

IT professionals, generally speaking, need to communicate effectively. They also have to work with other IT individuals because there is a diverse range of skill-sets needed.

We are all working on project type work at least part of the time.

If we are not working on a schedule then at the very least many of the things we do are "as soon as possible".

Extending Agile

- Defining Agile for project list tasks only.
- Some project features are different
  - Working in batches
  - Homogeneous results

Our desktop support team is looking into Agile. Their primary focus is applying the principles to their more project oriented tasks: setting up and refreshing hardware and software in our computer labs, deploying batches of new hardware to staff, and other such efforts.

Exploring, documenting, and learning these steps is certainly part of the process so there is some creative work going in the project.

One key difference is that many of these projects involve some repetitive tasks and a group of homogeneous results.

Whether this difference will be an issue remains to be seen.
The desktop support team has been meeting on a weekly basis for a while now, but this meeting tends to run at least an hour long.

Like the development team, the desktop support team will switch to meeting in shorter daily intervals so that progress and issues can be more easily tracked.

Desktop support has not decided what length of cycle they will be starting with, but shorter cycles seem to make more sense for their work. Currently they expect anything from a week to three weeks.

They have decided they do not want to use Jira for tracking. Instead they plan to use a physical board to track progress.

This choice made more sense to them because they anticipate fewer tickets per cycle and shorter cycles.

The desktop support team is only going to attempt to use Agile for project based work at this time, so less of each individuals time will be represented in the tracking mechanism as compared to the software development team.

The Enterprise Unified Process (EUP) is a book that talks about how an organization might extend Agile into the broader enterprise.

It has some good ideas, but I must first point out that the IBM Rational Unified Process (RUP) it is based on is process heavy. There are Agile variants of RUP, but IBM does not claim RUP is Agile.

RUP has some good ideas, some that are especially useful for my team because we work on multiple projects in parallel.
Extending Agile

Project Configuration from RUP

- Rather than define process amorphously, process can be templated.
- The process is configured on a per project basis.

"Project Configuration" is one such concept. The idea is that just as you might configure software for local use, you can have a single process template the organization uses across all projects and then plug-in configuration options to best suit the project's needs.

We do that to a certain extent, each person on our team for example has a known number of hours per week they are anticipated to be available for programming tasks. When we decide what products we will take on in a cycle and who will work on what, we keep those numbers in mind.

Extending Agile

- Heavy RUP is a bad fit
- Concepts from RUP and EUP fit however
- Both RUP and EUP are designed to be adaptable, variations are OK

Overall, RUP is very "heavy" feeling. The way it defines cycles seem more like phases from Waterfall. It tends to chart out efforts in long spans like Waterfall did.

Fortunately the pieces introduced by Enterprise Unified Process (EUP) seem like they might plug into lighter variants of Agile just as well.

Both RUP and EUP divide all of the different tasks involved in the work of getting things done into "Disciplines".

Extending Agile

EUP defines new disciplines for the broader IT enterprise

- Mainly focused on Operations and Support for now
- Touch and Hold approach the customer communication

EUP defines several new disciplines on top of RUP that are interesting to our department. They mirror the way we like to view and organize tasks across the many things we do.

The only discipline we are looking to put into the Agile fold for now however is the "Operations and Support" discipline.

One idea from the book on EUP focuses on improving the customer experience and simplifying resolution tracking with a "Touch and Hold" approach as opposed to the "Escalation" model that you may have been frustrated with before.
Extending Agile

- Touch and hold optimizes by centralizing tracking
- Fewer hand offs for fewer points of failure

The touch and hold method attempts to keep customers interacting with the same support person as much as possible on a single issue. This helps centralize tracking responsibilities.

The support staff will coordinate with other tiers of support and specialists.

EUP also advocates finding the most useful types of work to time track, and not worrying with other less important types.

This includes time that is easier to track, such as work logged on request tickets.

Extending Agile

EUP talks little about iterative work for Operations and Support staff

- Our exploration is ongoing
- We will continue to search for new ways to better deliver value to our customers

EUP doesn't offer any insights into how working iteratively for projects might be different for Operations and Support staff.

Our department will continue exploring opportunities in applying Agile to aspects of the IT enterprise. Working iteratively has benefits for the staff performing tasks and the people being served.

Thank You

Time for questions

Thanks for attending this session. I left time for questions, so I hope you came up with something good.