Agile Development

We have worked in the library system for a combined total of almost twenty years. Individually or together we have designed sections of or written code for Voyager, the USDA Economics and Statistics System, Illiad, the Libray Gateway and its predecessor, the Mann Gateway, and MyContents, and most recently, the MathArc project. The scope of the projects have cut across organization boundaries, at Cornell, with the US Goverrntment, and internationally. We have seen and studied many software project management methods, learning new technologies and techniques in a constant effort to improve our effectivenes.

We appreciate being here today because we have learned a lot from some of you, working side by side on some of our projects, or working under your supervision or management. We're happy to be able to share with you some of the project development techniques we are using in the MathArc project.

These techniques challenge some of the ways we've worked in the past. Instead of designing the system and trying to envision all the possible uses and risks and functions before we begin building working software, we're solving problems in small batches, not looking very far ahead, designing as we go. In project management terms, we're not relying on thorough system requirements or requrements documentation; we're using what the agile development approach calls "user stories." Second, instead of dividing the labor of developing the system into discrete units of functionality, one programmer to one unit, we're using a technique called "Pair Programming," in which two of us share the repsonsibility for the code we write, the documents we produce, and the presentations we give. And finally, instead of separating the design and implementation of the system from the testing of the system, we're writing the tests first and using them to help us design and implement the system.

These practices seem counter-intuitive. It may sound as if it's impossible to build a good system using them. Gathering requirements and designing systems up front seems more professional, seems more rigorous, more logical. User stories and test first design techniques sound as if we're giving up control. It sounds arbitrary, as if we're working on solutions to specific problems that can't possibly grow together into a coherent system. How can we possibly know what we're doing? And pair programming, isn't that inefficient? Two people sitting in front of one screen with one keyboard. How can that possibly be worth the time and money? We recognize these questions because we raised them ourselves, long before we started to use the techniques. We'll be talking about how these techniques produce robust sytems, generalizable systems, efficiently and quickly.

The standard project development techniques that we have used in the past rely on some variation of the waterfall method: Requirements Gathering, Big Design Up Front, plan the building process thoroughly, assign implementation tasks to individuals, build the whole system, test it, and finally deliver it to users. The Standish Group, in their latest Chaos Report, for 2004, state that 66% of IT projects fail in the United States. That's not a good return on investment.

The same report says that the rate of success, 34%, is twice as high as it was ten years earlier. The Standish Group attribute the growth to the agile techniques we're using. We believe that, using these techniques, we can reduce the amount of time it takes to build a system, reduce the number of bugs, make the system more generalizable, getting more out of our development time, produce better internal documentation, communicate better with the system stakeholders, and last, but not at all least, create built-in maintenance procedures that will reduce the cost of the system over the years.

We started to work together in Mann Library ITS in 1997. There was no formal process for designing a system. Projects were top-heavy with committee members discussing requirements. Sometimes our colleagues talked about the projects, developing an idea of what they wanted, for months before handing the projects over to us. We sometimes had less time to build the systems than they had spent talking about them. There was very little room for error. Tight schedules led to poor design decisions, numerous bugs. We couldn't go back to make changes easily, even if we thought the changes would produce a better system. We saw systems go into production before they were ready. Morale was low.

We first tried to introduce some better structure into the design process by studying the Unified Modeling Language, UML, and employing the Rational Unified Process in the projects we worked on. We hoped UML would improve communication between the librarians who initiated the projects and the programmers who did the design and coding. It did, but we were still using the UML and the Rational Process in a development process that relied on a lot of up-front planning and control.

We added iterative development to the process. We chunked up the designs we had made into what we thought were coherent groups of tasks that produced systems with growing layers of complexity. We made risk diagrams to prioritize the chunks so that we wouldn't cherry-pick the easy functions. We wrote time-lines to fit the tasks into the weeks or months until the end of the project. Our work and our projects improved.

At about this time, a group of techniques emerged, gathered into what was named the Agile Development process. They became popular topics of discussion on software development and programming lists. We started to talk about them among ourselves, too. We knew we wanted to try them, but they were radical enough that we couldn't easily introduce them into our projects or the programming culture we were working in.

Now, in the final year of the MathArc project, the year planned as being the year for implementation of the system, we find ourselves working together in an environment

Claims about how agile development can help an organization

How Stories work

Decisions of managers

Scope

Priority

Composition of realeases

Dates of releases

Decisions by technical people

Estimates

Consequences

Process

Detailed scheduling

Pair Programming

Collective ownership

Coding stadards

Advantages we've found:

- Synergy. We don't have identical skill sets. We can do more together than we could do separately.
- We find defects early. Catch each other's mistakes.
- We support each other through the day's energy cycles.
- We generate more design ideas, building off each other's ideas. Two heads are better than one.

Test-driven Development

Simple design

Refactoring

Continuous integration

Conclusion

These practices support each other.

How we believe the projected schedule has changed.