Agile Software Architecture

Christine Miyachi
Principle Systems Software Engineer, Xerox Corporation
cmiyachi@alum.mit.edu

Abstract

When the Agile software development became popular in software development communities, some engineers thought that software architecture would die out. But no matter what, a software project will have an architecture, whether documented intentionally or created on the fly. Software Architecture should enable Agile development methods by itself being Agile. But how? In this paper, I will discuss my experience of doing architecture in an Agile way and in a non-Agile way. Have an architecture that follows the Agile principles is key to obtaining success in an Agile development environment.

Keywords: Agile, Software, Architecture

Introduction

When one thinks of architecture several images may come to mind: stable buildings, arches that never fall. Dictionary.com defines architecture as "the action or process of building; construction". When one thinks of agility, they might think of quickness or the ability to change. Dictionary.com defines agile as "marked by an ability to think quickly; mentally acute or aware".

The two words don't seem to fit together: Agile Architecture. This paper will outline how they actually do fit together in software development.

How can something made to be stable but quick to change?

Software Architecture is defined differently from building architecture. The standard on software architecture - IEEE 1471-2000 – says (underlined terms are further defined in the standard):

> Software architecture is the **fundamental organization** of a **system**, embodied in its **components**, their **relationships** to each other and the environment, and the **principles** governing its design and evolution.

Agile Software is defined by a Manifesto[1]:

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

This Manifesto was developed in response to heavy-weight methods which were defined by complicated processes and measurements. To the critics, it appeared that a lot of work was going into regulation versus actually producing working systems. The creators of the Manifesto wanted to get back to building systems; several went on to create light-weight processes based on this Manifesto such as Extreme Programming (XP) and SCRUM.

The Agile Manifesto is backed up by twelve principles[1]:

- Customer satisfaction by rapid delivery of useful software
- Welcome changing requirements, even late in development.
- Working software is delivered frequently (weeks rather than months)
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Close, daily cooperation between businesspeople and developers
- Face-to-face conversation is the best form of communication (co-location)
- Projects are built around motivated individuals, who should be trusted
- Continuous attention to technical excellence and good design
- Simplicity
- Self-organizing teams
- Regular adaptation to changing circumstances

The Cost of Change

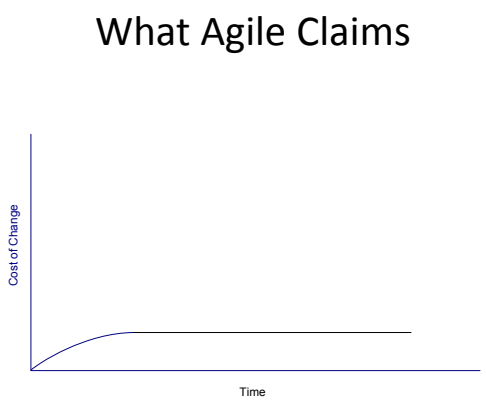
I can attest to these principles as I have been using Agile processes with my development teams for years now. We have been able to deliver more, with higher quality, and we continue to improve.

And yet, how do we manage Agile Architecture? Can the "fundamental organization" of a system be regularly changed? How do the creators of the Agile Manifesto address this?

In 2001, I was on a business trip with my System Design and Management (SDM) class from MIT. We were in Ireland at a web software development company who prided themselves in using Agile methods. Most of the students were not in the software industry and one student ask "How will you manage long term maintenance of this product if you just working in two week iterations? That is, do you look at anything long term?" The leader in question shrugged his shoulders and said "No – it's a question and we don't know how we will deal with that".

What Agile experts claim is that with software, the cost of changing levels out over time:

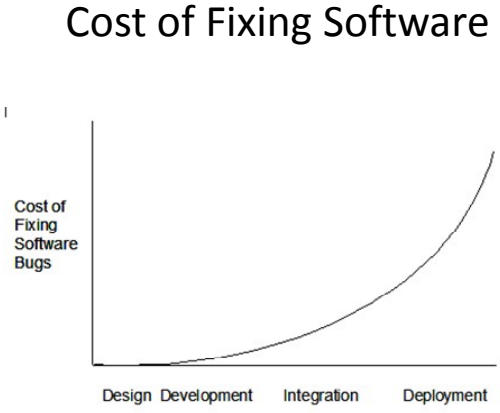
DOI: 10.1145/1943371.1943388 http://doi.acm.org/10.1145/1943371.1943388



That is, if you build your development process to accept change, the cost of changing levels out. Building software is more malleable than building hardware, they claim.

While this may be true with pure software systems or smaller software systems, most software systems today are large and complex, despite the push to get them simpler.

In the organizations I have worked for, it's expensive to change the software especially once it is shipped. Our cost of change curve looks like this:



And it becomes even more expensive if we were to change the fundamental organization of the system or the software architecture.

In many software systems, agile or not, the software architecture is what Grady Booch calls "accidental"[2]. This is probably what the architecture was at that company's product I visited in Ireland. No matter what a team does, an architecture is created. It can be done consciously in the beginning or unconsciously through simply doing iterations and small decisions that form a large decision that becomes the software architecture. Scott Ambler backs this up by saying "Every system has an architecture. BUT, it may not necessarily have architectural models describing that architecture".[3]

In cases like this the software architecture is not documented. This can be very problematic because people move on and no one understands the original intent of the system. Software archeology can be difficult to accomplish.

On the flip side, I have been on projects that are cut in stone. The architecture and design of the system is set early on and never changed in response to a changing business environment. Eventually the project is cancelled and though we have some impressive documentation (called BDUF or Big Documentation Up Front in the Agile world) we have no working system. Eventually these projects got cancelled. Customers don't buy design documentation from us, they buy working software. Scott Ambler says "**You should beware ivory tower architectures**. An ivory tower architecture is one that is often developed by an architect or architectural team in relative isolation to the day-to-day development activities of your project team(s). …. Ivory tower architectures are often beautiful things, usually well-documented with lots of fancy diagrams and wonderful vision statements proclaiming them to be your salvation.[3] The documentation for this project was indeed perfection. But perfection didn't get us a working system.

It is worth it to mention that there is a difference between Software Architecture and Software Design. There are overlaps between the two, but one way to think about it: Software Architecture consists of those key decisions that if changed later on, cost the business. Software Design consist of other decisions that can be changed later in response to business needs.

A Case Study: Making an Architectural Decision

On one system I worked on in the past, we had to decide what underlying infrastructure we were going to use as a communications mechanism between a client and server. Would it be web services, a lightweight HTTP mechanism, a file mechanism, or something else? We need to make this decision carefully as the structure would be around for a long time, and all of our future work would be built upon it. Several business people were involved in making this decision as it affected their ability to market our product.

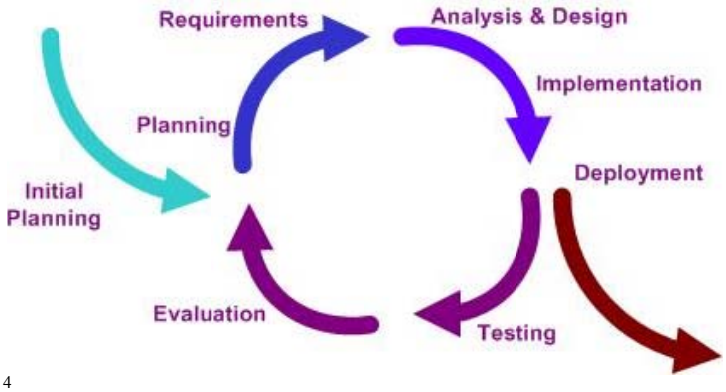
This is an example of an architectural decision. We created a set of requirements, weighted those requirements, and rated each solution against our requirements. When our decision was questioned we were able to return to these documents and review what we had done. Had the business climate changed and our requirements or their weightings changed? Had a new solution emerged? Had we learned something about a solution that changed our ratings?

If any of that had changed, we might have had to change our architectural decision. However, if we did, it would cost our business and the change would not have been made easily. There would have been a cost to this change, especially if it happened later rather than sooner.

Iterative Development

One way to mitigate this risk is to follow the Agile Manifesto of creating working systems. After we made this decision, we set about to immediately verify it an implementation. Some individuals on the team were concerned that our choice would not be robust enough. In delivering this initial system we used iterative development:

DOI: 10.1145/1943371.1943388 http://doi.acm.org/10.1145/1943371.1943388

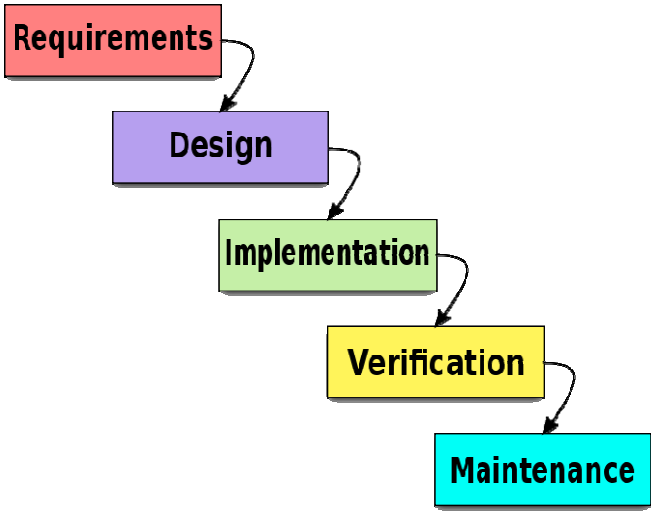


4

In this methodology, requirements analysis, design, and architecture are done each iteration. A final system is complete after many iterations. This methodology is typically part of lean or agile software methodologies, like SCRUM.

To build our system we had to do several iterations to finally verify our choice.

Contrast this to a non-agile process like the waterfall method:



5

In this methodology, all requirement analysis, design, and architecture are done up front. This is an example of BDUF (Big Documentation Up Front). As I said earlier, in large projects I have worked on, we have never gotten out of the Requirements and Design phase before the project was cancelled.

So what happened? We found out that our original system wasn't as robust as we wanted and we decided to go with a more robust but heavier-weight (needed more memory and disk space) system. Because we had verified the architecture early on in the design we were able to change this key decision.

Summary of Agile Architecture

I believe Software Architecture can and should be Agile. That said, at some point early on in the project, the key architectural decisions must be made. Here are my principles for Agile Software Development:

1. Determine what key decisions must be made early on. If a decision can wait without delaying the project, then let it wait.
2. Get all stakeholders in the decision involved and come up with a solution as team. There will be tradeoffs to be made. Sometimes consensus isn't possible and the lead architect will have to make a decision.
3. Document the requirements, weight the requirements, and rate the solutions against each requirement. Clearly state the tradeoffs of using one solution over another.
4. Verify the architecture as soon as possible by implementing a working system.
5. Make changes if necessary as early as possible.
6. Do not create an accidental architecture – it usually isn't as good as a well thought out one.

A web site dedicated to agile software architecture has created a set of objectives the Agile Manifesto for software architecture. It says:[6]

- Deliver working solutions
- Maximize stakeholder value
- Find solutions which meet the goals of all stakeholders
- Enable the next effort
- Manage change and complexity

These objectives are dependent on the first objective to produce a working solution.

Agile Architecture is based on the Agile Manifesto with a set of objects and principles that are in line with Agile principles. Successful projects using Agile development have Agile Architecture.

References

[1] http://agilemanifesto.org/
[2] http://www.informit.com/articles/article.aspx?p=471929
[3] http://www.agilemodeling.com/essays/agileArchitecture.htm
[4] http://en.wikipedia.org/wiki/File:Iterative_development_model_V2.jpg
[5] http://en.wikipedia.org/wiki/File:Waterfall_model.svg

[6] http://www.agilearchitect.org/agile/principles.htm

Author

Chris Miyachi has over 25 years of experience working for both start ups and large corporations. She has a blog she writes weekly about software architecture: http://abstractsoftware.blogspot.com/ . She is currently a Principle Systems Engineer and Architect at Xerox Corporation, holding several patents. Chris graduated from the University of Rochester with a BS in Electrical Engineering and from MIT twice: the first, a master's degree in Technology and Policy/ Electrical Engineering and Computer Science and the second masters in Systems Design and Management. Her full resume is at http://home.comcast.net/~cmiyachi/.

DOI: 10.1145/1943371.1943388 http://doi.acm.org/10.1145/1943371.1943388