ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/224118841

Agility and Architecture: Can They Coexist?

Article *in* IEEE Software · May 2010

DOI: 10.1109/MS.2010.36 · Source: IEEE Xplore

CITATIONS
137

READS
2,361

3 authors:

Pekka Abrahamsson
University of Jyväskylä
266 PUBLICATIONS **7,934** CITATIONS
SEE PROFILE

Muhammad Ali Babar
University of Adelaide
361 PUBLICATIONS **7,645** CITATIONS
SEE PROFILE

Philippe Kruchten
University of British Columbia - Vancouver
289 PUBLICATIONS **12,663** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Software Development Methods View project

Adversarial Machine learning View project

All content following this page was uploaded by Philippe Kruchten on 26 May 2014.

The user has requested enhancement of the downloaded file.

Agility and Architecture—Can they coexist?

Philippe Kruchten
University of British Columbia
Vancouver BC, Canada
pbk@ece.ubc.ca
Office +1 604 827 5654
Fax : +1 604 822 5949

Abstract: Software architecture is taking a bad rap with many agile proponents; big up-front design, massive documentation, smell of waterfall, it is pictured as a non-agile practice, something we do not want to even consider; though everybody want to be called an architect. However, certain classes of system, ignoring architectural issues too long "hit a wall" and collapse by lack of an architectural focus. Agile architecture: a paradox, an oxymoron, two totally incompatible approaches? In this paper we review the real issues at stake, past the rhetoric and posturing, and we suggest that the two cultures can coexist and support each other, where appropriate.

Keywords : software architecture, agile development methods

1. Paradox, oxymoron, incompatibility?

Jim Highsmith (2002) defines agility as "the ability of an organization to both create and respond to change in order to profit in a turbulent business environment." And Sanjiv Augustine (2005, p. 21) notes that the common characteristics of agile software development methods (XP, Scrum, FDD, Lean, Crystal, etc.) are: iterative and incremental lifecycle, focus on small releases, collocated teams, a planning strategy based on two levels: release plan driven by a feature or product backlog, and an iteration plan handling a task backlog. They all also more or less adhere to the values of the "agile manifesto" (Agile Alliance, 2001).

The Rational Unified Process® or RUP® (IBM, 2007) defines software architecture as the "set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed together with their behavior as specified in the collaboration among those elements, the composition of these elements into progressively larger subsystems, the architectural style that guides this organization, these elements and their interfaces, their collaborations, and their composition. Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technological constraints and tradeoffs, and aesthetics" (Kruchten, 1998, p. 80).

So far so good. But agilistas world-wide continue to perceive and describe software architecture a something of the past; they equate it with Big Up-Front Design

(BUFD) (a bad thing), leading to massive amount of documentation, implementing features YAGNI (You Ain't Gonna Need It). They see a low value of architectural design, a metaphor should suffice in most cases (Beck, 2000), and the architecture should emerge gradually sprint after sprint, as a result of succession of small refactoring.

Conversely, in places where architectural practices are well developed, agile practices are often seen as amateurish, unproven, limited to very small web-based socio-technical systems (Kruchten, 2007).

The tension seems to lie on the axis of *adaptation versus anticipation*, where agile methods want to be resolutely adaptive: deciding at the "last responsible moment" or when changes occur, and they perceive software architecture to be pushing too much on the anticipation side: planning too much in advance. And where architects may be tempted to anticipate too much, too early.

2. A field story: hitting the wall

A large international financial corporation wants to re-implement a large legacy IT system, result of years of evolutions, acquisitions, and integrations, several millions ines of Cobol, C, C++, RPG. It decides to proceed with agile methods, hire the proper consultants, get the training, and starts reim-plementing. The team is large, but collocated and has direct access to the internal "customers" (users of the legacy system), as well as the legacy code and documentation. Things proceed pretty fast at the beginning, with every 2 weeks some prototype demonstrating actual valuable progress. After some 6 months or so, things become harder. The team has now grown to about 50 people. The refactoring driven by new functionality added to the system hardly fit in one iteration, and are destabilizing everyone. Little visible progress is done, though everybody is really busy. Internal crisis ensue, turn over increases dramatically, lots of finger pointing. The system comes to a complete halt. Like marathonians after km 40, they have "hit a wall". Focusing solely on user stories valuable for their users upstairs, they completely neglected any architectural activities, and having accumulated a rather large code base, some of the decisions they have now to make are pretty dramatic. The blame goes to the agile methods. The project is restarted with more conservative approaches, and a stronger focus on mitigating technical risks, in particular putting in place some minimal architectural skeleton.

3. The real issues

The issues of trying to understand the apparent conflict and reconcile the two sides are in multiple levels: semantics (what do you really mean by software architecture), scope (how much architecture need to be done), lifecycle (when), role (by whom), documentation (how does architecture manifest itself), methods (how to proceed), value and cost (what value does it bring anyhow?)

3.1 Semantics

What do we mean by "software architecture"? I have given above the RUP definition. It is about the important stuff, the decisions that will be taken early and will be the hardest to undo, to modify, to remove (Fowler, 2004). Software architecture is about design, decisions that will be binding in the final product. But we witness a certain dilution of the term architecture.

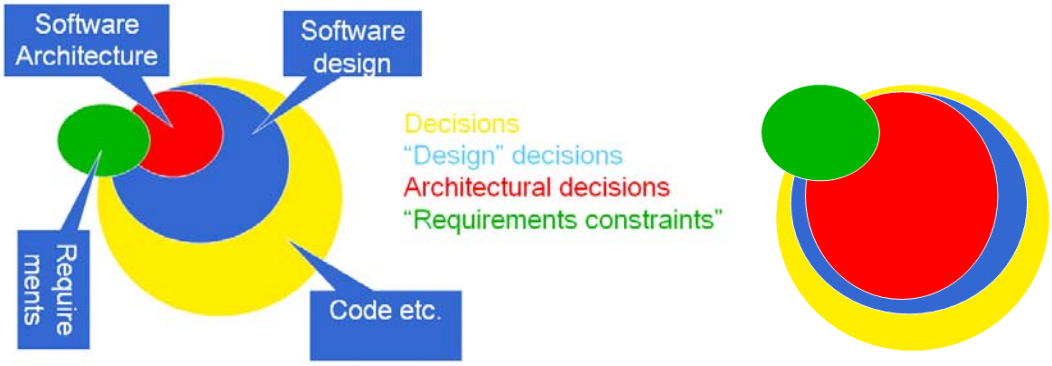

Fig. 1—Decisions and when architecture equates design

If the yellow circle represent all decisions that will be made for a software system, a subset will be labeled "design decisions" (blue), leaving a large amount of decision at the level of programming. In turn a very small subset of these design decisions will be architecturally significant (red), i.e., will be overarching, long-lived, very hard to change. Note that some decisions are made "upstream" in the form of requirements constraints (green).

Unfortunately, more and more, we see the decision landscape look more like the figure on the right, where not much distinction is left between design and architecture (blue equals red). Mary Shaw had warned us a long time ago: "Do not dilute the meaning of the term architecture by applying it to everything in sight" (cited by Garlan, 1995). Not all design decisions are architectural. Very few are, actually, and in many projects as we will see below, they are already taken on day one.

3.2 Scope

How much architectural activity will you need to have on your project? Here the response it a loud "it depends". It fluctuates widely depending on the *context* of the software project. By 'context' I mean the environment of the project: the organization, the domain, etc, and then the specific factors of a given project.

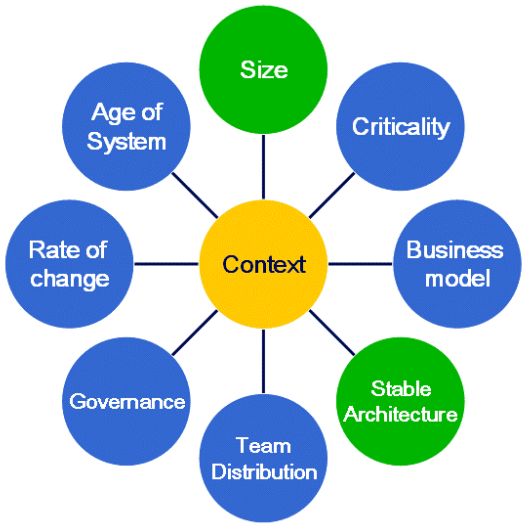

Fig. 3 – Factors making up the context of a project

1. Size

The overall size of the system under development is by far the greatest factor, as it will drive in turn the size of the team, the number of teams, the needs for communication and coordination between teams, the impact of changes, etc.

2. Stable architecture

Is there an implicit, obvious, de facto architecture already in place at the start of the project? Most projects are not novel enough to require a lot of architectural effort. They follow commonly accepted patterns in their respective domain. Many of the key architectural decisions are done in the first few days, by choice of middleware, operating system, languages, etc.

3. Business model

What is the money flow? Are you developing an internal system, a commercial product, a bespoke system on contract for a customer, a component of a large system involving many different parties?

4. Team distribution

Linked sometimes to the size of the project, how many teams are involved and are not collocated? This increases the need for more explicit communication and coordination of decisions, as well as more stable interfaces between teams, and between the software components that they are responsible for.

5. Rate of change

Though agile methods are “embracing changes”, not all domains and system experience a very rapid pace of change in their environment. How stable is your business environment and how much risks (and unknowns) are you facing?

6. Age of system

Are we looking at the evolution of a large legacy system, bringing in turn many hidden assumptions regarding the architecture, or the creation of a new system with fewer constraints?

7. Criticality

How many people die or are hurt if the system fail? Documentation needs increase dramatically to satisfy external agencies who will want to make sure the safety of the public is assured.

8. Governance

How are project started, terminated, who decide what happens when things go wrong?

In the “sweet spot” of agile project, in my experience, little architectural activities need to take place in most projects. But this still lives room for the large, complicated, projects where significant architectural efforts will be needed. Many agilistas have not seen such projects (yet). Not that agile practices cannot be applied to such projects, but they need

some adjustments to fit their practices to the specific circumstances: criticality, distribution, governance, business model, etc.

3.3 Lifecycle

When will architectural activities take place? As much as we'd love to defer decisions to the last responsible moment, for architectural decisions, because many of other design decisions will depend on them, they have to take place in the early phase of the project, or at least well identified. This does not necessarily mean Big Up-Front Design, where all sorts of design take place early, in a void, not driven by any immediate needs. Also we'd love to see the architecture gradually emerge as the result of successive refactoring, but for large novel systems, this may need a bit of anticipation. You Ain't Going to Need It (YAGNI) requires at least that you identify the "it"; same for deferring the decision to the last responsible moment implies that you know what it is that you are deferring. Putting in place a robust architecture is the purpose of the Elaboration phase in the RUP lifecycle (figure 4). This is driven by functional needs, not a pure architectural exercise in a void.

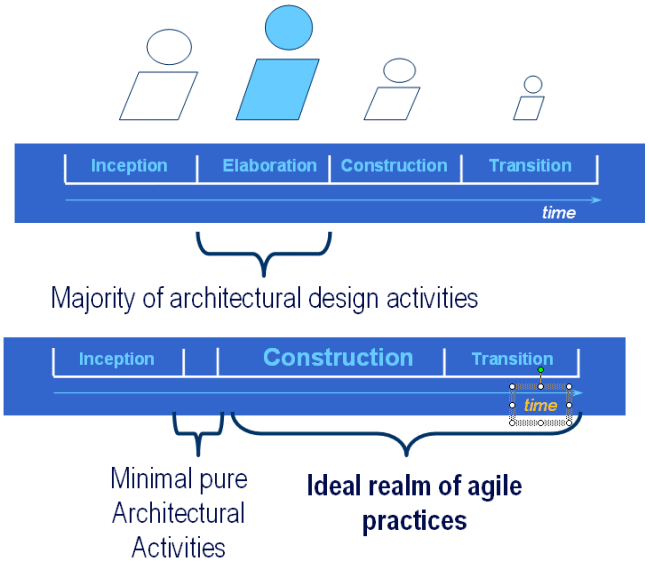

Fig.4 – Architectural activities in the RUP lifecycle are mostly in elaboration phase

And if an architecture is in place on day one, the elaboration phase is basically empty, or takes one quick iteration to validate (fig. 4). The key criterion to enter the construction phase is: have we mitigated all our technical risks? And *not*: have we finished the design?

When some significant architectural activities must happen, the early iterations (sprints) will integrate enough technical elements and focus more on architectural validation than delivering functionality (as shown in figure 5).

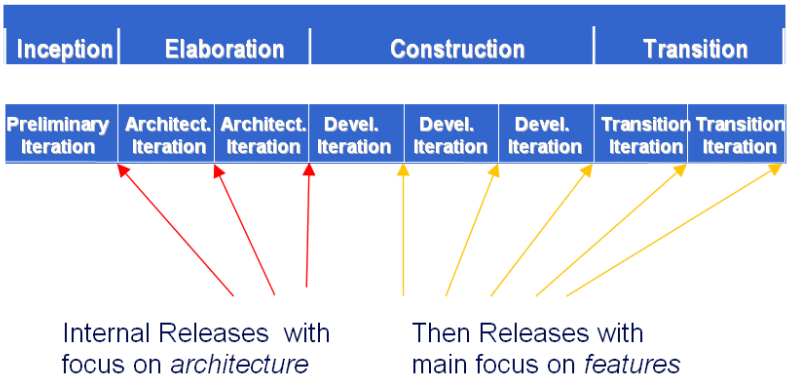

Fig.5 – change of focus through the lifecycle

They key here is not to focus on doing all the design (and implementation) of the architecture up-front, but on reaching rapidly a point where enough architectural elements are in place to enable multiple teams to proceed rapidly without too much interference between them or changes with dramatic effect. Figure 6 shows how an large software development organization (from 5 to 150 people) evolved through the RUP lifecycle. The groups in green operate very naturally using agile practices. The other one provide more of a "glue" and coordination function.

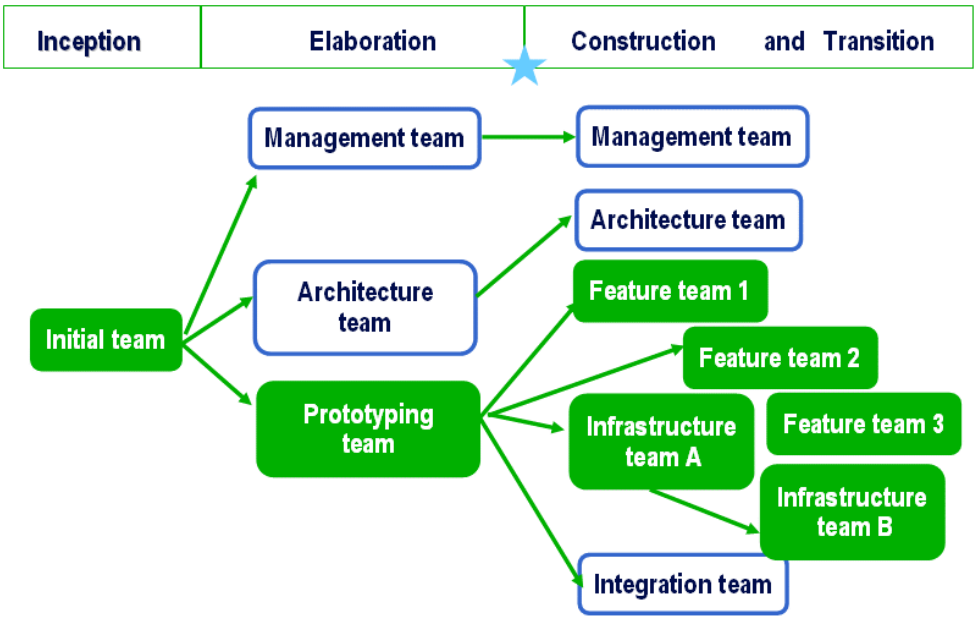

Fig. 6 – Evolution of large organizations; from (Kruchten, 2004)

Agile approaches put emphasis on delivering value, but there is also in some contexts a large amount of unknowns and of risks that must be addressed early, if only to discover early that certain approaches are unfeasible, do not scale up, rely on immature technologies, assume skills or productivity not available, etc. Many of the technical risks are mitigated by some architectural design and prototyping. But not all projects are facing such risks.

3.4 Role

Who are the architects? While on a small systems everyone may contribute to the role, on larger systems the architect(s) will play a combination of roles (Kruchten, 1999; Mills, 1985; Rechtin, 1994):

1. *Visionary*, shaping the system
2. *Designer*, making important technical choices
3. *Communicator*, playing go-between multiples stakeholders and groups
4. *Troubleshooter*, giving direction when things get hard or wrong
5. *Herald*, being the window of the project to the outside world
6. And even sometimes *Janitor*, cleaning up the mess behind project managers, product managers or developers….

These often partition into 2 types of individuals:

1. *Maker and keeper of big decisions*, bringing technological changes, focusing on external coordination, more requirements-facing, gatekeeper
2. *Mentor, troubleshooter, prototyper*, implementing elements of the architecture, focusing on intense internal communication, more code-facing.

These correspond to *Architectus Reloadus*, and *Architectus Aryzus* in (Fowler, 2003), and their respective ecological niche on fig. is architecture team and prototyping team, the *architectus aryzus* becoming naturally technical leads of the various teams down the line.

3.5 Documentation

How is architecture communicated to the various parties who need to know about it? Her again there is a wide spectrum of possibilities:

- An *architectural prototype*: the architecture is defined in the code
- A *metaphor* or a set of metaphors with carefully selected names to provides a good narrative to explain who the system is organized and how it works (Beck, 2000)
- A large component or class *diagram* on the wall
- A *software architecture document*, as suggested in RUP (IBM, 2007) or defined eloquently by (Paul Clements, et al., 2002; P. Clements, Ivers, Little, Nord, & Stafford, 2003).
- An architecture specific tool
- Or a combination of these.

The key idea is *communication*: choosing the best vehicle in the context of the project. The goal is not documentation for the sake of it. It will again depend much on the context (see figure 3).

3.6 Methods

How are we identifying and resolving architectural issues? Many agile approaches are rather silent on the topic. A pity, as a study of 5 architectural methods (Hofmeister, et al., 2007), shows that they all have underlying an iterative pattern (see figure) not too different from Scrum (Schwaber & Beedle, 2002): Identification of architectural issue (from requirements: stories, other constraints, assets, etc.), incremental design, validation by prototyping, and evaluation by testing, review or otherwise.

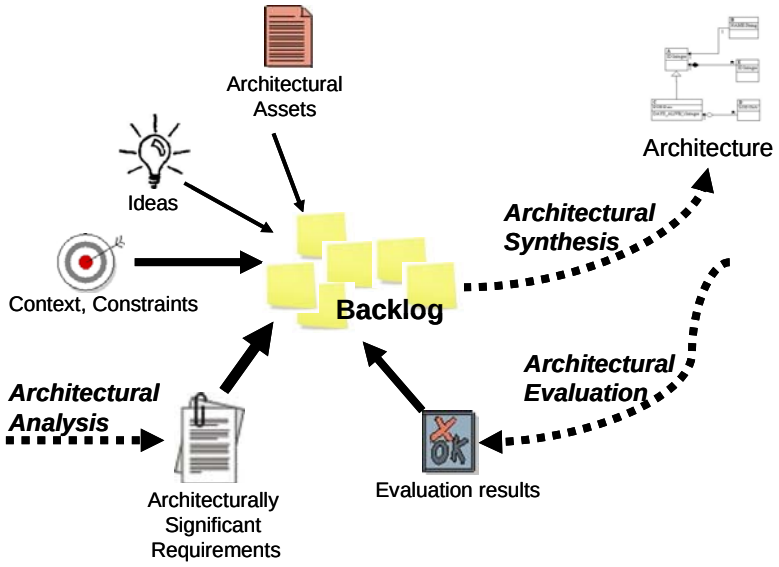

Fig. 7-- Pattern at the core of 5 architectural methods (Hofmeister, et al., 2007, p. 114)

Here the issue seems to be mostly ignorance of architectural practices by a large segment of software developers, architects included.

3.7 Value and cost

Agile approaches strive to deliver business value, "early and often." Though not all domains and business context allow doing this, it is still a useful way to pace the project, by defining small increments, leading to internal releases. However, if the value for the business can be defined in relative terms (priorities) or even in absolute terms by financial analysis (Denne & Cleland-Huang, 2004b), the value of architecture is not externally visible, and some of the necessary architectural thinking will not occur until too late to efficiently alter the course of the project. This will be the case mostly for large, novel systems. On non trivial systems, there is a balance to strike between business value and technical risk (Fowler, 2004). Scrum allow for provision for technical stories or activities in the backlog. Though the cost of the architectural effort can bed known, much of its value is realized much later in the project. An approach like the Incremental Funding Method (Denne & Cleland-Huang, 2004a) allows to cast the right compromise, without falling into the trap of Big Up-Front Design.

4. Planning

For system development that require some non trivial of architectural work, the planning could follow the zipper pattern. From requirements and constraints, functional and non functional, separate architecturally significant issues and purely functional issues. Establish dependencies between them; this is the "architectural analysis" step of fig. ^. Create additional stories for architectural issues and risks. Derive their value from the functional elements that depend on it. All the architectural issues do not need to be resolved up-front, but at least they are now visible in the backlog. Temporary, simple strategies can be put in place to allow early development of some functional aspects, while retaining in the project memory of further actions that may have to take place later: design, refactoring, validation.

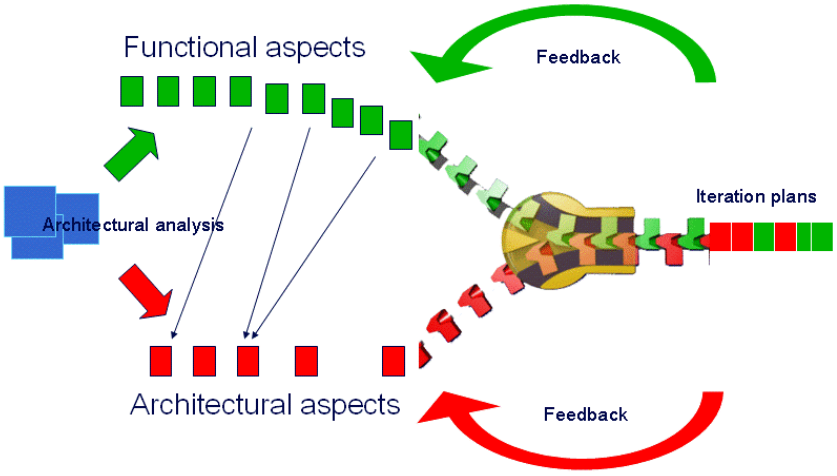

Fig. 8 – Zipper metaphor: Weaving functional and architectural activities

5. A conflict of cultures

Spencer-Oatey (2000, p. 4) defines culture as "a fuzzy set of attitudes, beliefs, behavioural norms, and basic assumptions and values that are shared by a group of people, and that influence each member's behaviour and his/her interpretations of the 'meaning' of other people's behaviour." Agility is more a culture than an engineering method and probably to a lesser extent is the less visible community of software architecture. They both have developed, based on specific beliefs, a set of values ('agile manifesto'), and behaviours (methods, practices, artifacts), sometimes also rituals and jargon that tend to detract outsiders (Kruchten, 2007). As for any clash of two cultures, the members of one culture go through several phases (Ting-Toomey, 1999, pp. 157-158):

- *Ethnocentrism*: my culture is right, is the reference by which other cultures must be judged, others are bad, etc. The attitude is defensive, dismissive, coercive.
- *Ethnorelativism*: leading to acceptance of other view-points, values, and possibly attempts to coexistence and integration. It may guide the definition of new behaviours, and from this new values and possibly evolution of the beliefs.

I personally stand with one foot in each of these two cultures: architecture and agility, and like other ethnic cultures, I see two parties not really understanding the real issues at hand, stopping at a very shallow, caricatural view of the "other culture", not understanding enough of the surroundings, beliefs, values of the other one, and stopping very quickly at judging behaviours.

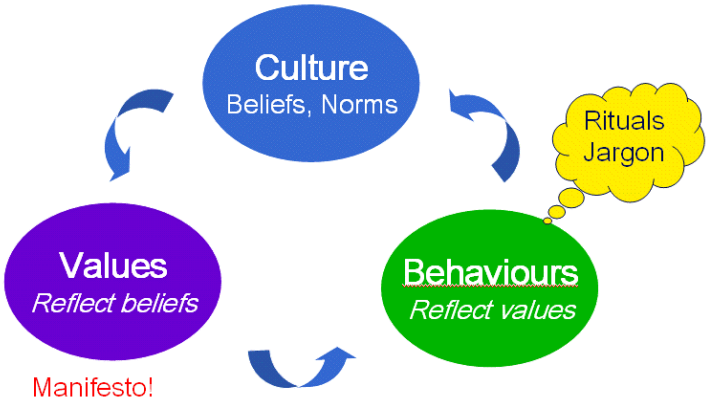

Fig. 9 – Agility (or architecture) as a culture, after (Thomsett, 2007)

6. Burying the hatchet

What lessons could both these cultures take away?

- Understand the context. There is a vast array of software development situation, and although agile practices "out of the box" address many of these, there are outliers for which we need to understand. What is the size, domain, age of the system? What are the business model, degree of novelty, hence of risk? How critical must the system be? How many parties will be involved?
- Define architecture clearly: the scope of the word, the role and responsibility of the architect. Do not assume a tacit, implicit understanding (Kruchten, 1999).
- Define an *architecture owner*, like there should be a product owner and a project leader. But do not let the architects lock themselves in some ivory tower, polishing the ultimate architecture for an improbably future system. Architect or architects are part of the development group.
- Exploit architecture to better *communicate and coordinate* between various parties, in particular multiple distributed teams, if any. Define how it will be represented, based on the need-to-know of the various aprties.
- Use important, critical, valuable functionality to *identify architectural issues*, and assess them. Understand interdependencies between technical architectural issues, and user visible functionality, so as to weave them appropriately over time (zipper metaphor).
- Understand when it is appropriate to *freeze the architecture* to provide the necessary stability for developers to finish a product release, and what amount of technical debt is then accumulated.
- Keep track of *unresolved architectural issues*, either in the backlog or in the risks. Deferring decisions to the last responsible moment does not equate to ignoring them, but may be adding risks to be managed like other risks in the project.

In the movie *The Matrix Reloaded* (Wachowski & Wachowski, 2003), the Architect says: "The first matrix I designed was quite naturally perfect…. a triumph equaled only by its monumental failure. I have since come to understand that the answer eluded me because it required a lesser mind, or perhaps a mind less bound by the parameters of perfection."

References

Agile Alliance (2001). *Manifesto for Agile Software Development.* Retrieved June 24, 2008, from http://agilemanifesto.org/

Augustine, S. (2005). *Managing Agile Projects*. Upper Saddle River, N.J.: Prentice Hall.

Beck, K. (2000). *Extreme Programming Explained: Embrace Change.* Boston: Addison-Wesley.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., et al. (2002). *Documenting Software Architectures: Views and Beyond*. Boston: Addison-Wesley.

Clements, P., Ivers, J., Little, R., Nord, R., & Stafford, J. (2003). *Documenting Software Architectures in an Agile World* (Vol. CMU/SEI-2003-TN-023). Pittsburgh: Software Engineering Institute.

Denne, M., & Cleland-Huang, J. (2004a). The Incremental Funding Method: Data-Driven Software Development. *IEEE Software, 21*(3), 39-47.

Denne, M., & Cleland-Huang, J. (2004b). *Software by Numbers: Low-Risk, High-Return Development*. Upper Saddle River, N.J.: Prentice Hall.

Fowler, M. (2003). *Who needs an architect? IEEE Software, 20*(4), 2-4.

Fowler, M. (2004). *Is Design Dead?* Retrieved June 24, 2008 from http://martinfowler.com/articles/designDead.html

Garlan, D. (1995). First International Workshop on Architectures for Software Systems Workshop Summary. *Software Engineering Notes, 20*(3), 84-89.

Highsmith, J. A. (2002). *Agile software development ecosystems*. Boston: Addison-Wesley.

Hofmeister, C., Kruchten, P., Nord, R., Obbink, H., Ran, A., & America, P. (2007). A General Model of Software Architecture Design derived from Five Industrial Approaches. *Journal of Systems & Software, 80*(1), 106-126.

IBM (2007). *Rational Unified Process.* Retrieved June 24, 2008 from http://www-306.ibm.com/software/awdtools/rup/?S_TACT=105AGY59&S_CMP=WIKI&ca=dtl-08rupsite

Kruchten, P. (1998). *The Rational Unified Process--An Introduction* (1 ed.). Boston, MA: Addison-Wesley.

Kruchten, P. (1999). The Software Architect, and the Software Architecture Team. In P. Donohue (Ed.), *Software Architecture* (pp. 565-583). Boston: Kluwer Academic Publishers.

Kruchten, P. (2004). Scaling down projects to meet the Agile sweet spot. *The Rational Edge,* (August 2004). Retrieved June 24, 2008 from http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/aug04/5558.html

Kruchten, P. (2007). Voyage in the Agile Memeplex: Agility, Agilese, Agilitis, Agilology. *ACM Queue, 5*(5), 38-44.

Mills, J. A. (1985). A Pragmatic View of the System Architect. *Comm. ACM, 28*(7), 708-717.

Rechtin, E. (1994). *The Systems Architect: Specialty, Role and Responsibility.* Paper presented at the 1994 NCOSE Symposium, San Jose, CA.

Schwaber, K., & Beedle, M. (2002). *Agile Software Development with SCRUM*. Upper Saddle River, NJ: Prentice-Hall.

Spencer-Oatey, H. (2000). *Culturally Speaking: Managing Rapport through Talk across Cultures*. New York: Cassel.
Thomsett, R. (2007). Project Management Cultures: the Hidden Challenge. *Agile Product and Project Management, 8*(7).
Ting-Toomey, S. (1999). *Communicating acrosss cultures*. New-York: The Guilford Press.
Wachowski, A., & Wachowski, L. (Directors) (2003). *The Matrix Reloaded.* Warner Bros.

Philippe Kruchten is a professor of software engineering at the University of British Columbia in Vancouver, Canada. He is a founding member of the IFIP working group 2.10 on software architecture, and the co-founder and chair of Agile Vancouver--an agile method fan club. In a previous life he led the development of the Rational Unified Process, which was more agile in its intent that practitioners have made of it, and which embodied an architectural method.

View publication stats