Architecture vs Agile: competition or co-operation?

John Grundy

Swinburne University of Technology

Until recently, conventional wisdom has held that Software Architecture design and Agile development methods are somehow "incompatible", or at least they generally work at cross-purposes (Nord and Tomayko, 2006). Software architecture design has usually been seen by many in the agile community as a prime example of the major agile anti-pattern of "big design up front". On the other hand, agile methods have been seen by many of those focusing on the discipline of software architecture as lacking sufficient forethought, rigor and far too dependent on "emergent" architectures (a suitable one which may never actually emerge). In my view, there is both a degree of truth and a substantial amount of falsehood in these somewhat extreme viewpoints. Hence the time seems ripe for a book exploring leading research and practice in an emerging field of "agile software architecture", and charting a path for incorporating the best of both worlds in our engineering of complex software systems.

In this foreword I briefly sketch the background of each approach and the anti-agile, anti-software architecture viewpoints of both camps, as they seem to have become known. I deliberately do this in a provocative and all-or-nothing way, mainly to set the scene for the variety of very sensible, balanced approaches contained in this book. I hope to seed in the reader's mind both the traditional motivation of each approach and how these viewpoints of two either-or, mutually exclusive approaches to complex software systems engineering came about. I do hope that it is apparent that I myself believe in the real benefits of both approaches and they are certainly in no way incompatible – agile software architecting - or architecting for agile, if you prefer that viewpoint – is both a viable concept and arguably the way to approach the current practice of software engineering.

Software Architecture - the "Traditional" View

The concept of "software architecture" – both from a theoretical viewpoint as a means to capturing key software system structural characteristics (Shaw & Garlan, 1996) and practical techniques to develop and describe (Kruchten, 1995; Bass et al, 2003) – emerged in the early to mid 1980s in response to the growing complexity and diversity of software systems. Practitioners and researchers knew implicitly the concept of a "software architecture" existed in all but the most trivial systems. Software architecture incorporated elements including, but not limited to, human machine interfaces, databases, servers, networks, machines, a variety of element inter-connections, many diverse element properties, and a variety of further structural and behavioral sub-divisions (thread management, proxies, synchronization, concurrency, real-time support, replication, redundancy, security enforcement, etc). Describing and reasoning about these elements of a system became increasingly important in order to engineer effective solutions, with special purpose "Architecture Description Languages" as well as a wide variety of architecture modeling profiles for the UML. Software architecting includes defining an architecture from various perspectives and levels of abstraction, reasoning about the architecture's various properties, ensuring the architecture is realizable by a suitable implementation which will meet system requirements, and evolving and integrating complex architectures.

A number of reusable "architecture patterns" (Bass et al, 2003) have emerged, some addressing quite detailed concerns e.g. concurrency management in complex systems, while others much larger scale organizational concerns e.g. multi-tier architectures. This allowed a body of knowledge around software architecture to emerge, allowing practitioners to leverage best-practice solutions for common problems and researchers to study both the qualities of systems in use and to look for improvements in software architectures and architecture engineering processes.

The position of "software architecting" in the software development lifecycle was (and still is) somewhat more challenging to define. Architecture describes the solution space of a system and hence traditionally is though of as an early part of the design phase (Bass et al, 2003; Kruchten, 1995). Much work has gone into developing processes to support architecting complex systems, modeling architectures, and refining and linking architectural elements into detailed designs and implementations. Typically one would identify and capture requirements, both functional and non-functional, and then attempt to define a software architecture that meets these requirements.

However, as all practitioners know, this is far easier said than done for many real world systems. Different architectural solutions themselves come with many constraints on which requirements can be met and how they are met, particularly non-functional requirements. Over-constrained requirements may easily describe a system that has no suitable architectural realization. Many software applications are in fact "systems of systems" with substantive parts of the application already existent and incorporating complex, existent software architecture that must be incorporated. In addition, architectural decisions heavily influence requirements and co-evolution of requirements and architecture is becoming a common approach (Avgeriou et al, 2011). Software architectural development as a top-down process is hence under considerable question.

Agile Methods – the "Traditional" View

The focus in the 1980s and 90s on extensive up-front design of complex systems, development of complex modeling tools and processes, and focus on large investment on architectural definition (among other software artifacts) were seen by many to have some severe disadvantages (Beck et al, 2001). Some of the major ones identified included over-investment in design and wasted investment in over-engineering solutions, inability to incorporate poorly defined and/or rapidly changing requirements, inability to change architectures and implementations if they proved unsuitable, and lack of a human focus (both customer and practitioner) in development processes and methods. In response a variety of "agile methods" were developed and became highly popular in the early to mid 2000s. One of my favorites and one that I think exemplifies the type is Kent Beck's eXtreme Programming (XP) (Beck, 1999).

XP is one of many agile methods that attempts to address these problems all the way from underlying philosophy to pragmatic deployed techniques. Teams comprise both customers and software practitioners. Generalist roles are favored over specialization. Frequent iterations deliver usable software to customers ensuring rapid feedback and continuous value delivery. Requirements are sourced from focused user-stories and a backlog and planning game prioritizes requirements, tolerating rapid evolution and maximizing value of development effort. Test-driven development ensures requirements are made tangible and precise via executable tests. In each iteration enough work is done to pass these tests but no more, avoiding over-engineering. Supporting practices including 40 hour week, pair programming and customer on site avoid developer burn-out, support risk mitigation and shared ownership, and facilitate human-centric knowledge transfer.

A number of agile approaches to developing a "software architecture" exist, though most treat architecture as an "emergent" characteristic of systems. Rather than the harshly criticized “big design up front” architecting approaches of other methodologies, spikes and refactoring are used to test potential solutions and continuously refine architectural elements in a more bottom-up way. Architectural spikes in particular give a mechanism for identifying architectural deficiencies and experimenting with practical solutions. Refactoring, whether small scale or larger scale, is incorporated into iterations to counter “bad smells”, which include architectural-related problems including performance, reliability, maintainability, portability and understandability. These are almost always tackled on a need-to basis, rather than explicitly as an upfront, forward-looking investment (though they of course may bring such advantages).

Software Architecture – Strengths and Weaknesses w.r.t. Agility

Upfront software architecting of complex systems has a number of key advantages (Abrahamsson et al, 2010). Very complex systems typically have very complex architectures, many components of which may be “fixed” as they come from third party systems incorporated into the new whole. Understanding and validating a challenging set of requirements may necessitate modeling and reasoning with a variety of architectural solutions, many of which may be infeasible due to highly constrained requirements. Some requirements may need to be traded off against others to even make the overall system as a whole feasible. It has been found in many situations to be much better to do this in advance of a large code base and complex architectural solution to try and refactor (Abrahamsson et al, 2010). It is much easier to scope resourcing and costing of systems when a software architecture documenting key components exists upfront. This includes costing non-software components (networks, hardware) as well as necessary third party software licenses, configuration and maintenance.

A major criticism of upfront architecting is the potential for over-engineering and thus over-investment in capacity that may never be used. In fact, a similar criticism could be leveled in that it all too often results in an under-scoped architecture and thus under-investing in required infrastructure, one of the major drivers in the move to elastic and pay-as-you-go cloud computing (Grundy et al, 2012). Another major criticism is the inability to adapt to potentially large requirements changes as customers re-prioritize their requirements as they gain experience with parts of the delivered system (Beck et al, 2001). Upfront design implies at least some broad requirements – functional and non-functional – that are consistent across the project lifespan. The relationship between requirements and software architecture has indeed become one of mutual influence and evolution (Avgeriou et al, 2011).

Agile – Strengths and Weaknesses w.r.t. Software Architecture

A big plus of agile methods is their inherent tolerance and in fact encouragement of highly iterative, changeable requirements, focusing on delivering working, valuable software for customers. Almost all impediments to requirements change are removed, and in fact many agile project planning methods explicitly encourage reconsideration of requirements and priorities at each iteration review, the mostly widely known and practiced being SCRUM (Schwaber, 2009). Architectural characteristics of the system can be explored using spikes and parts found wanting refactored appropriately. Minimizing architectural changes by focusing on test-driven development – incorporating appropriate tests for performance, scaling and reliability – goes a long way to avoiding redundant, poorly fitting and costly over-engineered solutions.

While every system has a software architecture, whether designed-in or emergent, experience has shown that achieving a suitable complex software architecture for large-scale systems is challenging with agile methods. The divide-and-conquer approach used by most agile methods works reasonably well for small and some medium-sized systems with simple architectures. It is much more problematic for large-scale system architectures and for systems incorporating existent (and possibly evolving!) software architectures (Abrahamsson et al, 2010). Test-driven development can be very challenging when software really needs to exist in order to be able to define and formulate appropriate tests for non-functional requirements. Spikes and refactoring support small system agile architecting but struggle to scale to large-scale or even medium-scale architecture evolution. Some projects even find iteration sequences become one whole refactoring exercise after another, in order to try and massively re-engineer a system whose emergent architecture has become untenable.

Bringing the Two Together – Agile Architecting or Architecting for Agile?

Is there a middle-ground? Can agile techniques sensibly incorporate appropriate levels of software architecture exploration, definition and reasoning, before extensive code bases using an inappropriate architecture are developed? Can software architecture definition become more "agile", deferring some or even most work until requirements are clarified as develop unfolds? Do some systems best benefit from some form of big design up front architecting but can then adopt more agile approaches using this architecture? On the face of it, some of these seem counter-intuitive and certainly go against the concepts of most agile methods and software architecture design methods.

However, I think there is much to be gained leveraging strengths from each approach to mitigate the discovered weaknesses in the other. Incorporating software architecture modeling, analysis and validation in "architectural spikes" does not seem at all unreasonable. This may include fleshing out user stories that help to surface a variety of non-functional requirements. It may include developing a variety of tests to validate these requirements are met. If a system incorporates substantive existing system architecture, exploring interaction with interfaces and whether the composite system meets requirements by appropriate test-driven development seems eminently sensible early-phase, high-priority work. Incorporating software architecture-related stories as priority measures in planning games and SCRUM-based project management also seems compatible with both underlying conceptual models and practical techniques. Emerging toolsets for architecture engineering, particularly focusing on analyzing non-functional properties, would seem to well support and fit agile practices.

Incorporating agile principles into software architecting processes and techniques also does not seem an impossible ask, whether or not the rest of a project uses agile methods. Iterative refinement of an architecture including some form of user stories surfacing architectural requirements; defining tests based on these requirements; rapid prototyping to exercise these tests; and pair-based architecture modeling and analysis could all draw form demonstrated advantages of agile approaches. A similar discussion emerges when trying to identify how to leverage design patterns and agile methods, user-centered design and agile methods, and model-driven engineering and agile methods (Nord and Tomayko, 2006; McInerney and Maurer, 2005; Dybå and Dingsøyr, 2008). In each area, a number of research and practice projects are exploring how the benefits of agile methods might be brought to these more "traditional" approaches to software engineering, and how agile approaches might incorporate well-known benefits of patterns, UCD and MDE.

Looking Ahead

Incorporating at least some rigorous software architecting techniques and tools into agile approaches appears, to me at least, to be necessary for successfully engineering many non-trivial systems. Systems made up of architectures from diverse solutions with very stringent requirements, particularly challenging non-functional ones, really need careful look-before-you-leap solutions. This is particularly so when parts of the new system or components under development may adversely impact existing systems e.g. introduce security holes, privacy breaches, or adversely impact performance, reliability or robustness. Applying a variety of agile techniques – and the philosophy of agile – to software architecting also seems highly worthwhile. Ultimately the purpose of software development is to delivery high quality, on-time and on-budget software to customers, allowing for some sensible future enhancements. A blend of agile focus on delivery, human-centric support for customers and developers, incorporating dynamic requirements, and where possible avoid over-documenting and over-engineering exercises all seem of benefit to software architecture practice.

This book goes a long way to realizing these trends of agile architecting and architecting for agile. Chapters include a focus on refactoring architectures, tailoring SCRUM to support more agile architecture practices, supporting an approach of continuous architecture analysis, and conducting architecture design within an agile process. Complementary chapters include analysis of the emergent architecture concept, driving agile practices using architecture requirements and practices, and mitigating architecture problems found in many conventional agile practices.

Three interesting works address other topical areas of software engineering: engineering highly adaptive systems, cloud applications and security engineering. Each of these areas has received increasing attention from the research and practice communities. In my view, all could benefit from the balanced application of software architecture engineering and agile practices described in these chapters.

I do hope that you enjoy this book as much as I have in reading over the contributions. Happy agile software architecting!

References

Abrahamsson, P., Babar, M.A., Kruchten, P., Agility and architecture – can they co-exist?, IEEE Software 27 (2), 2010.

Avgeriou, P., Grundy, J., Hall, J.G., Lago, P., Mistrík, I. Relating software requirements and architectures, Springer, 2011.

Bass, L., Clements, P., Kazman, R., Software architecture in practice, Angus & Robertson, 2003.

Beck et al, Manifesto for Agile Software Development, http://agilemanifesto.org/, 2001.

Beck, K. Embracing change with extreme programming, Computer 32 (10), 1999.

Dybå, T. and Dingsøyr, T., Empirical studies of agile software development: A systematic review, Information and Software Technology 50, August 2008.

Garlan, D. and Shaw, M. Software architecture: perspectives on an emerging discipline, Angus & Robertson, 1996.

Grundy, J., Kaefer, G., Keong, J., Liu, A. Software Engineering for the Cloud, IEEE Software 29 (2), 2012.

Kruchten, P. The 4+ 1 view model of architecture, IEEE Software 12 (6), 1995.

McInerney, P. and Maurer, F., UCD in agile projects: dream team or oadd couple?, Interactions 12 (6), 2005.

Nord, R.L., Tomayko, J.E., Software architecture-centric methods and agile development, IEEE Software 23 (2), 2006.

Schwaber, K., Agile Project Management with SCRUM, O'Reily, 2009.