Reconciling Architecture and Agility: What Matters?

Michael Waterman, James Noble and George Allan
School of Engineering and Computer Science,
Victoria University of Wellington,
New Zealand
Email: waterman@ecs.vuw.ac.nz, kjx@ecs.vuw.ac.nz, george.allan@ecs.vuw.ac.nz

***Abstract*—A key feature of agile software development is its prioritisation of responding to changing requirements over planning ahead. If an agile development team spends too much time planning and designing architecture then responding to change will be extremely costly, while not doing enough architectural design puts the project at risk of failure. The team must therefore find a balance. This Grounded Theory research has identified four contexts that can affect that balance: unstable requirements, early value, team experience and non agile environments. Agile teams can address these contexts through the use of one or more of five strategies: use modern vendor frameworks, do no up-front planning, plan for options, address technical challenges, and do a full up-front design. System complexity is also a strong factor in how much architecture effort is required; size does not have a direct impact.**

I. Introduction

A software architecture represents the high-level structure and behaviour of a system [1] and can be difficult to change once development has started [2]. Architectural planning is about planning ahead – getting the design of the system right and avoiding costly refactoring. On the other hand, one of the key features of agile software development is the ability to respond to changing requirements in preference to planning ahead [3]. There is therefore a tension between up-front architecture design and agile methods.

This paper presents results from ongoing research that is investigating the relationship between up-front architecture design and agile software development – that is, 'how much architecture?' Specifically, this paper presents results from qualitative research into what affects how much architecture development teams design up-front.

Following this introduction, section II discusses the problem of the tension between up-front architecture design and agile software development, and the research gap. Section III describes the research methodology used (Grounded Theory). Sections IV to VII present the findings of this research and section VIII discusses the results in context of the literature. Finally section IX concludes the paper and outlines the future direction of this research.

II. Background

Kruchten defined architecture as "the set of significant decisions about the high level structure and the behaviour of the system" [1]; Booch extended this by noting 'significant' can be measured by the cost of change [2]. Boehm [4] included not just the high level design decisions but all front-end activities in his definition of architecture.

Delivering *value* to the customer and other stakeholders lies at the heart of being agile; many of the twelve principles of the agile manifesto directly relate to delivering value earlier and faster [3]. Scrum and XP maximise value by prioritising iteration tasks according to business priorities [5], [6], and Lean places high importance on value streams [7].

Agile developers often pay insufficient attention to architecture [8], with the architecture being either neglected entirely or only implicitly defined, which introduces a tension with architecture's need for planning up-front (the 'big design up-front', or BDUF, when taken to its extreme). On one hand too much architecture will lead to expensive architectural refactoring if the requirements change significantly, while on the other hand too little architecture may lead to an *accidental architecture* [9] – one that has not been carefully thought through and may lead to gradual failure of the project.

Agile methodologies generally advise developers to deal with this tension through a compromise – by designing just enough architecture to start development ('just enough design up-front') with the rest being completed during development as required [10]. Agile methods, however, are silent on how to identify the significant decisions that should be made up-front, how to perform incremental architecture design and how to validate architectural features [11]. The level of up-front architectural design depends heavily on context. Context includes environmental factors such as the organisation and the domain, as well as specific context factors such as size, criticality, business model, stable architecture, team distribution, governance, rate of change and age of system [11]. More than this however; context also includes social influences [12], such as the background and experience of the architects. Booch and Fairbanks noted that a particular system may have more than a single correct architecture [13], [14], and two architects are likely to produce different architectures for the same problem with the same boundaries [12]. Taylor described architecture as being as much about 'soft' (subjective) factors as it is about objective design [15].

There has been very little empirical research on the relationship between software architecture and agile development to date [16]. Breivold et al. performed a survey of the literature and concluded that studies have been small, diverging, and in some cases, performed in an artificial setting [16]. Dybå and Dingsøyr also noted the need for more knowledge of software development in general, particularly through empirical studies [17]. This paper presents results from ongoing research that helps to address this gap, by investigating the factors that have the greatest effect on how much architecture that agile architects and developers plan and design up-front.

III. Research method

This research into up-front architecture uses the qualitative Grounded Theory methodology [18], [19]. Qualitative methods are used to investigate people, interactions and processes [20]. As noted above, architecture is very dependent on the architects themselves and the development teams. Qualitative research is generally *inductive* – it develops theory from the research, unlike *deductive* research which aims to prove (or disprove) a hypothesis or hypotheses. Because of the scarcity of literature on the relationship between architecture and agile methods [16], an inductive methodology that will develop a new hypothesis is more suitable for this research. Grounded Theory was selected because it allows the researcher to develop a *substantive* theory that explains the processes observed in a range of cases [21].

The key steps of Grounded Theory are *open coding* the data to identify points of research interest; *constant comparison* to derive *concepts* from the codes and *categories* from the concepts; using these categories and their inter-relationships to find a *core category* that is central to the emerging theory; *memos* are written to develop the relationships between codes, concepts and categories, and to aid our understanding of their importance in this research; and *selective coding* to focus the core category [22]. Grounded Theory uses iteration to ensure a wide coverage of the factors that may affect the emerging theory [22]: later data collection is dependent on the results of earlier analysis.

For example in his interview, participant 'P10' commented on the difficulty of planning up-front:

> *"The problem with planning up-front is it assumes you know to begin with the usage patterns that your system is going to be put through." (P10, agile coach)*

This was coded as 'can't do full design up-front' because he was referring to his teams' inability to design in advance without knowing the usage patterns.

The emerging codes were constantly compared with existing codes from this and earlier interviews [23]. Codes that had similar themes to this example included 'avoiding design where requirements may change', 'avoiding overengineering', 'can't do full design up-front', 'delaying decision making as long as possible', 'design for change' and 'smallest amount of up-front design'. These codes were combined into a concept called 'affected by changing and unknown requirements'. Figure 1 shows the relationship between the underlying codes and 'affected by changing and unknown requirements'. This method was previously described in Waterman, Noble and Allan [24].

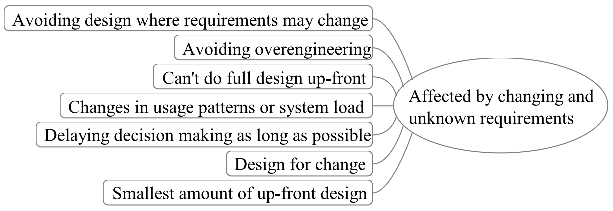
Avoiding design where requirements may change
Avoiding overengineering
Can't do full design up-front
Changes in usage patterns or system load
Delaying decision making as long as possible
Design for change
Smallest amount of up-front design
Affected by changing and unknown requirements

Fig. 1. An example of a concept emerging from its codes

In this research, data was collected primarily through semi-structured interviews with agile practitioners who design or use architecture, or are otherwise architecture stakeholders. Participants were typically architects, developers, project leaders/managers and customers, and are all involved with business-type applications.

Twenty nine interviews have been analysed to date. Participants were gathered through industry contacts, agile interest groups and through direct contact with relevant organisations. Almost all participants were very experienced developers, and most were also very experienced in agile development. Organisation types vary from development consultancies, government departments, mass-market product developers and single contractors. Different types of agile development are included, with most participants using Scrum and others using XP, Lean or bespoke methods. Most participants described adapted their systems to some extent to suit their team or customer's requirements. The inclusion of this range of participants and systems enables the research to include the effects of different factors on architecture decision making.

Participants were asked to select a project that they had been involved with to discuss in the interview. Types of projects varied hugely, from green fields to system redevelopment, from standalone systems to multi-team enterprise systems, and from start-up service providers and ongoing mass market product development to bespoke business systems. Documentation is also obtained where possible to corroborate the interview data. To maintain confidentiality, the participants are referred to using codes P1 to P29. A summary of participants and their projects are summarised in table I.

This paper focuses on the results that have emerged to date. An earlier paper [24] captures the effects of architectural frameworks and templates, and the architects' experience, on the amount of up-front effort required in architectural design.

IV. Findings

There are a number of different situations which affect the level of architecture that teams plan up-front; these are referred to in this paper as the **contexts** of the project or programme of work. The contexts that have emerged in this research are the need to deliver value by responding to *unstable requirements*, the need to deliver *early value*, the *team's experience* in agile development and a *non-agile environment*. The presence of one or more of the first three contexts will mean a team uses strategies that offer less up-front architecture, while the

	Role	Organisation type	Domain	Agile methods	Team size	Duration	System description
P1	Developer	Government agency	Health	Single developer	1	6 months	Web-based, .NET
P2	Dev./architect	Start-up	E-commerce	Scrum	3	Ongoing	.NET, cloud-based
P3	Dev. manager	Vendor	Human resources	Scrum	3 teams	Ongoing	Web-based, .NET,
P4	Director of architecture	Government dept.	Digital archiving	Scrum	5 developers	Ongoing	Open source, rich client, suite of standalone tools
P5	Coach/dev. manager	Start-up	Entertainment	Scrum/kanban	Various	N/A	Various
P6	Man. Dir./ lead dev.	Vendor	Telecoms	Informal, iterative	1–3 developers	Ongoing	Suite of standalone applications
P7	BA	Telecoms operator	Telecoms	Scrum	12 total	1 year+	Suite of web-based services
P8	Lead developer	Government dept.	Digital archiving	Scrum	4–14 total	1 year+	Ruby on Rails, Java back-end
P9	Developer	Financial services	Telecoms	Bespoke	2–24 total	3 years	Web-based system
P10	Coach	Multinat. hardware vendor	Transport	Scrum/XP	500–800 total	Several years	Large distributed web-based system
P11	Architect	Government	Government services	Scrum	8 total	Several years	Web-based services, .NET
P12	Senior developer	Service provider	Financial services	Scrum	6–7 developers	7 months	.NET, suite of web-based applications
P13	Architect	Government	Health	Scrum	12 total	4 years	Monolithic .NET app
P14	Architect	Government	Animal health	Scrum	6–8	18 months	.NET, large GIS component
P15	Customer	Start-up service provider	Retail (electricity)	Scrum	7 developers	Ongoing (3 years)	Ruby On Rails
P16	CEO/Chief engineer	Start-up	Retail (health)	XP	5 total	5 months	Ruby On Rails
P17	Manager/Coach	Government	Statistics	Scrum	6 dev + admin	2–3 years	Web-based, PHP using DAO pattern
P18	Dev. manager	Multinat. hardware vendor	Health	Scrum	15 total	Ongoing (>2 years)	Web-based, Java platform
P19	Dev. manager	Start-up service provider	Retail (travel)	Lean	4 developers	Ongoing (<1 year)	PHP/Symfony, Javascript/Backbone
P20	Coach and trainer	Independent consultant	N/A	Scrum	N/A	N/A	N/A
P21	Manager/Coach	Service provider	Retail (publishing)	Scrum	3 teams; 40 total	Several years	.NET, Websphere Commerce, SAP, others.
P22	Senior manager	Service provider	Contact management/marketing	Scrum/XP	More than 40 total	N/A	.NET
P23	Senior manager	Vendor	Pharmaceutical	Own methods	3 teams	Ongoing	Various web based, client/server
P24	Customer	Start-up service provider	Retail (electricity)	Scrum	7 developers	Ongoing (3 years)	Ruby On Rails web applications
P25	Team lead	Service provider	Banking	Scrum	1 team	Ongoing	.NET, single tier web
P26	Team lead	Government	Water management	Scrum	8 total	1 year	.NET, web based, 7 tier
P27	CEO/founder/ coach	Start-up service provider	Retail (electricity)	Scrum	7 developers	Ongoing (3 years)	Ruby On Rails
P28	Technical lead	Service provider	Broadcasting	Scrum	42 total	N/A	Django, CMSs for multiple websites
P29	Dev. manager	Banking	Banking	Kanban	20 total	Ongoing	Web based, AJAX, interface to mainframe

TABLE I
PARTICIPANT SUMMARY

presence of the latter means a more up-front architecture strategy (and often overrides the first three).

Agile development teams respond to these contexts with a number of **strategies**: *using vendor frameworks*, and either doing *no up-front architecture*, doing a high level architecture design and *planning for change*, doing a high level architecture design and *addressing the technical challenges*, or doing a *full up-front architecture*. The latter four all lie on a sliding scale of the level of up-front architecture, although are not necessarily mutually exclusive. As well as the contexts all affecting where the development team lies on this scale, the system complexity also has a major impact.

The relationship between these contexts and strategies is shown in figure 2. The contexts, strategies and the effects of complexity are described in more detail in the following sections.

V. Contexts

Each of the contexts that affect how much up-front architecture is described below.

A. *The unstable requirements context*

The *unstable requirements* context refers to the need for agile development teams to be able to adapt to changing or unknown requirements. Participants all reported requirements

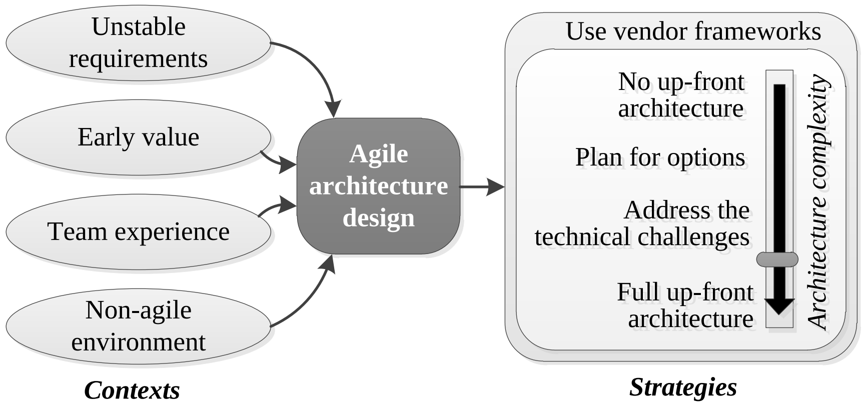

Fig. 2. Contexts and strategies for up-front architecture

changing or evolving to some extent, from the team working on a relatively stable redevelopment project to the start-up with a change-driven business plan:

"Even within a week there's a lot of fluidity about [the customer]" (P27, CEO/founder/agile coach)

There are a number of reasons requirements may change, such as business priorities changing:

"...there are many occasions where something really important last week has been swapped out for something that's even more important now" (P15, customer)

or they become known or better understood:

"I don't know if the actual requirements ever changed but our understanding of them changed enormously" (P13, architect)

or usage patterns that may be either initially unknown or change.

By starting development and demonstrating the product to the customer early in development, developers are able to get feedback from the customer early, and increase value by responding to that feedback:

"So we have the quick feedback cycle, and every week [the Project Owners] tell us what's been working, what these [end users] are asking for, so that's where get a lot of our feedback from." (P22, senior manager)

B. The early value context

For many businesses, particularly those building new mass market services or products, value is measured in terms of cash flow. Delaying the initial release by spending too much time planning will cost the business through delayed revenue, or worse, through missed opportunity:

"Today they've got an opportunity for a business idea that might make them some money – if they don't pounce on it it's gone regardless of how clever they think they are." (P26, team lead)

and

"If they [build] the big system, then they will never reach their end customer and make their money."

A number of participants described their goals as releasing a *minimum viable product* [25], or MVP, which is a marketing strategy in which a product with the barest minimum of features possible is released as early as possible, so that they can attract the early adopters who provide early feedback and start to generate cash flow, which can pay for future development. The developers of an MVP typically have little idea of how its requirements will evolve prior to receiving feedback.

C. The team experience context

The *team experience* context refers primarily to developers who have the ability to follow good architecture design principles, and who are familiar with the technologies they are using and therefore can make use of tacit knowledge and make decisions implicitly. Experienced agile developers are also comfortable with not having to follow plans when developing. Developer experience and ability therefore have an important impact on amount of up-front architecture.

The effects of developer experience was described in more detail in a paper published earlier from this research [24].

D. The non-agile environment context

While the previous contexts all require a reduced up-front architecture, the *non-agile environment* context requires an increased up-front architecture. A number of participants were developing for organisations that required assurance on what was being delivered for the money spent – and therefore needed more up-front architecture:

"They [the customer] need to go to the Board with some answers, and they need to go to the Board with some numbers... the 'we need to understand the end game before we start'-type approach" (P26, team lead)

Some customers required accountability or a record of decisions made through the delivery of an architecture document:

"A lot of it is just proving to the customer you've thought about what you're doing and being able to give it to someone for independent QA and they can say, yes, it looks ok or they can weigh it or whatever they need to do." (P14, solutions architect)

VI. Agile architecture strategies

This section presents five architectural strategies adopted by agile teams.

*A. **The vendor framework strategy***

The use of modern vendor frameworks, such as .NET, the Java platform and Ruby on Rails, are used by development teams to greatly reduce the amount of up-front architecture and development effort required, particularly in web-based business applications. Modern frameworks reduce the up-front effort required to build a system, and they enable architectural changes to be made a lot easier:

> *"What used to be architectural decisions ten years ago can now almost be considered design decisions because the tools allow you to change these things more easily." (P4, director of architecture)*

Similarly, P29 noted that with modern frameworks architecture problems have become much smaller:

> *"Most of the architecture problems you are dealing with appear to me to be small, most of the big backbone problems of web development or of relational database development or network development have been around for quite a long time, there are fairly well known paradigms and patterns and we're not really having to build a new wheel." (P29, development manager)*

P27 noted that the architecture of their system was easily planned and implemented with their chosen framework:

> *"I won't say we don't have [architectural] discussions, but this stuff has been done a thousand... it's been done a million times. On sites fifty times bigger. There is no discussion to be had really." (P27, CEO/founder/agile coach)*

The *use vendor frameworks* strategy is crucial in the *unstable requirements* and *early value* contexts to reduce up-front and rework effort. If the wrong frameworks are selected however then the benefits will be lost. P28, a technical lead, believed "fear of doing up-front design has become a dogma" in his teams, and so the teams actively avoided planning to the point where they had chosen their framework because it was an easy decision, rather than because it was best for the business. As a result all their development time was being used to try and make the framework work, rather than deliver features.

Developers must be aware that the frameworks do not always provide a solution to all parts of their system. This is described below in the *address the technical challenges* strategy.

The benefits of frameworks and template architectures were described in a paper published earlier from this research [24].

*B. **The no up-front architecture strategy***

At one end of the up-front architecture scale is the *no up-front architecture* strategy, which is used by the teams whose systems fall entirely within the boundaries of their frameworks and have little technical risk. In these instances very little up-front architectural effort is required beyond selecting the frameworks and platforms; the majority of architectural decisions are made during development. Because the team believes it is able to refactor the architecture – whether up-front or emergent – to cope with whatever requirements may arise in the future, the team does not need to make any allowance for requirements beyond what is being built today:

> *"We built a prototype based on some wireframes that [the customer] had done. Got it out there, and it worked [...] and then largely on that back of that they said, ok, let's take it to the next step, have you got any resources available, so we did that, then take it to the next step, take it to the next step, and all of a sudden we are here today." (P27, agile coach/founder)*

The *no up-front architecture* strategy is therefore particularly useful for delivering value early, such as when building MVPs. When faced with a decision of either doing more architectural planning activities up-front (and delaying the release) or delaying architectural activities (and releasing as early as possible with less functionality), the teams consider which option will provide the most value to the business:

> *"You can't really go ahead and say that you're architecting, you're architecting, you're architecting, and you don't release anything for a month. That doesn't make any sense to me. " (P16, CEO/chief engineer)*

and

> *"Minimum viable product. Generally about six weeks in you go into production, as opposed to nine months." (P17, manager/agile coach)*

While delivering value early is most important to start-ups, it are not exclusively for start-ups. P29 was a development manager with a national bank whose architectural decisions are sanctioned by a central architecture committee; even with this environment, P29 believed the no up-front architecture was crucial:

> *"We want to be able to put in the smallest simplest minimum viable [marketing] experiment, prove an assumption, beef it up if we want to, or follow it wherever it goes. That means the entire architecture is going to be emergent [...] so if we're going to have to do a heavy architecture which plans for a year or two or five years on every one of those experiments – we're screwed. We can't be agile." (P29, development manager)*

Because the developers build only for today and accept they may need to refactor the architecture, they are prioritising customer value over minimising the overall cost of architectural effort:

> *"Maybe it'll cost a lot more to replace it a year later, but you already have some business..." (P22, senior manager)*

and

> *"[Designing for a million users] is a problem you can have once you've got a million users and you've got a million users worth of revenue..." (P27, CEO/founder/agile coach)*

The *team experience* context is important for a successful *no up-front architecture* strategy, because experienced developers are more comfortable designing and working without an architecture designed in advance and are more able to build change-friendly architectures:

"You need to have experienced developers in place, otherwise it's very difficult for them to think in that model [of not designing up-front]. [Inexperienced developers] will find it very difficult to start without having a concrete design in place." (P22, senior manager)

and

"I do believe in the team having enough brains to be able to think on the fly, and if they're building simple emergent modular design as they go, then every bolt-on, every change should only affect one encapsulated piece." (P29, development manager)

C. *The planning for options strategy*

The *planning for options* strategy is the next step up from the *no up-front architecture* strategy in which teams complete only high level architectures up-front. They delay architectural decisions until the last possible moment, and they design for change. They use a project scope or high level road map to plan for options.

The high level architecture is "just enough that people could start [building] stuff" (P13, architect):

"You define the top-level architecture, the interaction between the components. Just design the top level architecture but don't get into the detail of the individual components." (P20, agile coach/trainer)

As well only doing high level architecture up-front, developers should avoid designing to perfection because if requirements change then this effort is wasted:

"...We'd gone down several rabbit holes in the past [...] you tend to over-architect things – as developers, we want perfection. That's the killer." (P19, development manager)

Teams also delay making decisions until needed for development:

"We were making concrete decisions as late as possible." (P25, team lead)

P17 described an architectural point of no return, which in his case was when the cost of starting again due to an architectural error became higher than the cost of working around that error:

"There is a point of no return with architecture. Somewhere between eight weeks and nine months is that point of no return! Where the cost of refactoring that core architecture is too great." (P17, manager/agile coach)

Like the *no up-front architecture* strategy, teams designed to allow change. Good design practices, such as separation of concerns and modularity are important to facilitate change.

Knowing the scope of the overall project allows teams to guide its architectural decisions, to ensure it doesn't close off options as the requirements evolve.

"[The developer] could do the work and spit out the output but they started saying, hang on, we can see all this little individual stuff, but we can't actually work out where we're going, what's actually the bigger picture. And so this is where we said we'd have these more formal planning sessions." (P15, customer)

Teams plan for options using techniques such as including extra levels of indirection and leaving space for extra components. Planning for future possibilities is against the principles of simplicity and YAGNI ("you ain't gonna need it"), because it means that the developers are considering possible requirements and ensuring that those requirements are not closed off, even though those options are not designed or built.

Examples of designing for the future include P2, P10 and P23:

"We had dotted lines where we could insert caches for additional performance." (P2, developer/architect)

and

"What you need to develop instead is the ability to be flexible [...] so rather than attempting to be too forward looking, we'll add an indirection here because it'll be extensible" (P10, agile coach).

and

"But basically when we design something, we keep in mind the permutations and combinations that can come up for a particular functionality." (P23, senior manager)

P9, a developer on a team building a telecommunications billing system, described this as "bounding the design" where the team considers the extreme set of requirements they might need to develop for:

"...how many carriers are there in the world, ever? What's the rate with which new carriers are being added, what's the rate with which new phones are being added, how much traffic could we ever envision, what do we have to do to accommodate that, and so this was all done up-front to see to it that we could adapt" ... "we didn't have to find the exact answer, we had to find that an answer we could live with existed." (P9, developer)

The *planning for options* strategy is used when the team understands the high level scope or development roadmap, and wishes to reduce its architectural refactoring efforts. While an important strategy for the *unstable requirements* and *early value* contexts, it is not as suitable for an MVP because an MVP would not normally have a defined scope. The *no up-front architecture* strategy may be a better option in this case.

D. *The address the technical challenges strategy*

The *address the technical challenges* strategy shares many characteristics of the *planning for options* strategy, such as only doing enough architecture planning to start development. This strategy recognises that the vendor frameworks being used may not eliminate all technical risk: there may be technical challenges, such as there being unique parts to the system (which cannot be implemented using framework components), are critical to meeting the systems non-functional requirements (such as performance), or when the technology is new or unknown. In these instances the team must do extra up-front architecture planning to address the technical challenges and ensure risks are suitably mitigated. The team prioritizes architecture work:

> *"So what you're doing in prioritising work is you're de-risking the system as rapidly as possible" (P10, agile coach)*

Developers identify risk and focus their architectural efforts to reduce risk using techniques such as research, modelling and experimentation (spikes or prototyping) [14]:

> *"I think our decisions on two of the key bits of technology, the object relational mapping and the data binding, were discussed in quite a lot of detail and we had pros and cons written down and so in that sense were trying to think about what the risks might be, things that might go wrong in the future, will this be hard to maintain, will this be hard to support, that sort of thing." (P13, architect)*

The key requirement for P12's system was to be able to process its data in less than 1 second. The team had to understand whether or not this was technically possible, and thus had to perform more detailed design:

> *"One of the things we tried to identify up-front was, how long does it currently take? And of those subsystems, how long does each of those subsystems take, in order to find out where we're going to focus our time." (P12, senior developer)*

On a number of occasions participants had to build their own components when the framework's off the shelf components were not able to meet requirements.

Participants frequently noted that often the only sure technique for proving if something works is through experimentation: "you have to get your feet wet, your hands dirty" (P7, business analyst):

> *"You can estimate till the cows come home, before actually trying it out in the flesh, and when you try it out in the flesh then you learn uh-oh, all sorts of things are [causing problems]." (P10, agile coach)*

Even if requirements are known in detail, the team still has to start building the system to prove the technology works or a solution exists:

> *"We've easily spent three weeks working through the various issues [with the technology]. And those are the kinds of things that you can't predict up-front."..." The architecture is evolving as we're going through this learning exercise." (P7, business analyst)*

Experiments can be used to explore the performance of a system:

> *"I said [to the customer], well, we see quite a risk here in performance in this particular area. Are you willing to spend some money to reduce this risk? And he said, how can we do that? And I said, we could knock something up in a day which we could then give death to see if it scales and he went, if you could do it in six hours I'd be up for it. And we said, ok, six hours, we'll knock that bit out, that's not important, and we did it." (P2, developer/architect)*

or to explore an unknown or new technology:

> *"Whoops, we know that is coming up, we don't quite know how to deal with it – let's have a spike to explore the domain, and find out what the optimal countermeasure is at this time, and have it in place." (P10, agile coach)*

or to compare options:

> *"the choice between alternatives was done by quite a lot of prototyping; we had both things going and did quite a lot of analysis and discussion on the pros and cons of each of them." (P13, architect)*

In an earlier example above, P2 ran an experiment to test capability of the architecture; on the other hand, P17 and P21 had not done any analysis or experiments, and ran into difficulties when the load exceeded the designed capabilities of their systems. Both required major architectural rework. P21 talks about their solution:

> *"What we did was we didn't try to optimise the backend, we tried to optimise more at the frontend. So instead of using heavy handed Javascript we moved to Web 2.0 kind-of thing. And that gave us some gain in optimising information in how we submit the pages, rather than using a synchronous call, to get the level of information on the page." (P21, manager/coach)*

In hindsight, P17 would have done extra analysis up-front, but would not necessarily have changed the architecture:

> *"If we'd actually done some performance testing beyond [the little that we did], all it would have given us is a bit more forewarning [to the overload problem, but] I'm not sure we would have made that many different decisions, we wouldn't have delivered on the features at those early stages. [Doing more analysis] probably would have hindered us as well, because we wouldn't have been able to be so adaptive.*

That is, changing the architecture in response to results of the testing would have contravened the YAGNI principle.

The *address the technical challenges* strategy is used where up-front architectural effort is required to address the technical risks in the system – where there are components that are not part of the framework, are critical to meeting requirements, or where the technology is new or unknown. The latter challenge, new or unknown technology, suggests an absence of the *team experience* context, which was noted as important for the success of the *no up-front architecture* strategy.

*E. **The full up-front architecture strategy***

The *full up-front* strategy is used by teams that work in a *non-agile environment* context, where the customer requires the team to do more up-front requirements gathering and up-front architecture planning than they might otherwise prefer.

P11, whose agile team is part of a non-agile corporate environment, described getting their architectural decisions approved by their business prior to development:

"We have to do a document, that's our details and design document. So we have a discussion, document the design, do a review, you have to sign it off usually." (P11, architect)

P7's team was also working in a non-agile environment – their customer was a reluctant agile participant that had its own team of enterprise architects who had to approve all architectural decisions:

"The customer's got architects galore [...] there are architects that look at it end to end, and there are architects who look across to across, so there is a lot of consulting and socialisation going on. And those architects expect – following the [client's] normal process – after high level requirements there will be an architecture, [which] will be decided and written up, low level requirements will come along and then there will be a design written up, and then they will start building." (P7, business analyst)

P14 was required to supply to the customer a full architecture document recording decisions made. He noted how much extra detail is contained in this document:

"I could have written a five or ten page document that would have got a lot of those key aspects across that people could have read without going to forty or fifty pages." (P14, solutions architect)

The non-agile customer may also affect the ability of the team to be fully agile. If the customer is not able to commit to regular show-and-tell sessions and provide feedback to the team, then the team may prefer to do more of a big up-front architecture where the customer only has to commit to time during a scoping or envisioning phase, releasing the customer from many of their ongoing obligations with the team.

While teams who used the *full up-front architecture* strategy designed their architecture up-front, they did not necessarily have every last detail of the requirements or design fixed. P6 noted:

"If you designed every last detail you'd spend a year on the spec and life would move on." (P6, lead developer and managing director)

The *full up-front architecture* strategy therefore does not mean a traditional, non-agile process. Some teams talk about putting on a traditional front to the customer, but still develop using an agile process, and still plan for change [26]:

"From a reporting to the customer point of view, when we first sign up with them, we absolutely report it in waterfall. But internally, we run it like this [agile]. Which is hard!" (P27, CEO/founder/agile coach)

This strategy is therefore able to be used in the *unstable requirements* context. It is typically not useful in an *early value* context; the non-agile customer who requires an up-front architecture design is generally not interested in deploying MVPs into the market. The full up-front architecture can therefore be viewed as a more up-front version of the *plan for options* and *address the technical challenges* strategies, in which the extent of the architecture is dictated by the *non-agile environment* context. The *team experience* context is less important than the other strategies, as inexperienced team members can rely on the up-front architecture design to support development.

VII. System complexity

While the contexts are important in determining the architecture strategy, the complexity of the system is the most important determinant of how much architecture a team plans. Architectural complexity is determined by the effort required to select the appropriate frameworks and tools that will be used to develop the solution, and solve the critical, unknown, unique or otherwise technically challenging problems of the system (as described above in the *address the technical challenges* strategy), plus complete other architectural-related activities:

"In the initial one week we do a couple of things. Even though we don't do up-front design we choose the technologies we're going to use [...]. And also identify the critical parts of the system which are the complex parts, the technically risky parts of the system [...] So those kinds of things we identify." (P16, CEO/chief engineer)

and

"[Complexity] definitely affects how much prework you might have to do to make sure you've got all your automation environments, your continuous integration, your continuous delivery set up, so you can actually put in the first bit of code and run it through that system, and if it's really horribly complex and you've got to request all sorts of bits of infrastructure from all over the show to get it to work then it definitely slows down iteration zero." (P29, development manager)

P14 explains another source of complexity in modern systems:

"Today's systems tend to be more interconnected – they have a lot more interfaces to external systems than older systems which are typically standalone. They have a lot higher level of complexity for the same sized system." (P14, solutions architect)

Complexity determines how much architectural effort is required within each strategy.

While size is sometimes considered to be a factor in determining how much up-front architecture [4], this research suggests that complexity is a more important factor: a large but non-complex system may require a lot less up-front effort than a small but complex system. A number of participants commented on the relationship between size and complexity and their relationship with up-front architectural effort:

"In my experience, the complexity of an organisation's systems landscape has a greater influence on the amount of fore-thought required than the budget or size of any particular initiative" (P10, agile coach)

P29, whose teams build systems with highly emergent architectures, also noted that the start-up phase (iteration zero) did not depend on size of the project:

"It could have been a very small thing that created a big iteration zero." (P29, development manager)

Conversely, a large system with a lot of functionality may require very little up-front effort if its architecture is easily implemented with the standard components of the frameworks being used and has little risk:

> *"We talk to a lot of systems, we interface with a lot of systems, we've got customer web requests coming in, we've got iPhone requests coming in, from a software point of view there's a lot of moving parts. The [functionality] is very, very complex – but the physical architecture itself that it sits on is nice and standard [...] it's a just well adopted Ruby On Rails stack. We deliberately try not to do anything different. Go with what's proven, go with what works." (P27, CEO/founder/agile coach)*

Avoiding complexity has allowed P27 to use the *no up-front architecture* strategy.

Thus in agile development where modern vendor frameworks are used, complexity is a more important factor in determining the up-front architectural effort required than project size.

VIII. Discussion

This study of agile practitioners has found that developers use five broad strategies to determine their level of up-front architecture effort. The first strategy is to make use of modern vendor frameworks where possible. These frameworks greatly reduce the architectural and development effort required [24]. The remaining four strategies vary in their levels of up-front architecture.

The *no up-front architecture* strategy can be used where the system fits within the boundaries of the framework and the team has skilled developers who are able to build an emergent architecture that follows good design principles. This strategy strictly follows the YAGNI principle, considering only the requirements that are known today, and relies on refactoring when requirements change. The no up-front architecture strategy is good for maximising customer value when new products or services are being developed – perhaps when requirements are highly unstable and when minimum viable products are required. Providing value is a higher priority than reducing the overall architecture cost, which may increase due to the team only architecting for today, even when a longer term evolution of the architecture can be forecast. This strategy of providing value differs from Boehm's analysis of the overall cost of delivery of the system [4] in which the overall cost of delivery is minimised to determine the sweet spot of up-front architecture planning. While "cost is always a consideration" (P10) in agile development, Boehm's analysis does not consider the value gained from early delivery of functionality. Poort and van Vliet's proposed method [27] also included minimising cost as a goal of architecture design; they claimed that stakeholder value is implicit in the presence of the solution's goals and business requirements. This assumption is not appropriate for agile development however, because firstly it assumes that the solution's goals and business requirements do not change after development has started, and secondly, like Boehm's analysis, it assumes that there is no value in delivering functionality to the end user before development of the entire system has been completed.

The *planning for options* strategy allows the team to consider the overall scope or road map of the system being developed to reduce architectural refactoring. Developers design the minimum architecture required to start development, and consider future possibilities within their designs, so that changing requirements can be accommodated with minimal rework. This strategy implicitly relies on the scope not changing, and thus requires more up-front work than the minimum suggested by YAGNI.

The *address the technical challenges* strategy is required where the framework being used does not provide all the functionality required, requiring components to be built from scratch, or where certain components are critical to the success of the system, or where the technology is new or unknown to the team. In these instances there is a level of risk which must be reduced to a satisfactory level, generally through experiments or analysis, before development can proceed. Fairbanks [14] and Poort and van Vliet [27] proposed risk-based approaches to architecting in which the risks are assessed and prioritised, with the more significant risks being addressed up-front, and leaving the risks where the benefit achieved does not justify the effort.

The *full up-front architecture* strategy is used where teams are working in a non-agile environment, perhaps being required by their customer to document in detail what will be delivered and what the price will be, or perhaps accounting for architectural decisions made. Teams must complete an up-front design that is more detailed than the team would otherwise produce. Requirements are typically unstable, so it is still preferable for the team to use agile development methods than traditional methods.

A team may select different strategies for different parts of a system being developed, and different degrees of each may be used.

Complexity is an important factor in determining how much up-front architecture planning is required, where complexity is particularly related to the technical challenges of the system, such as building components that are not provided as part of the framework.

On the other hand, system size is not a good indicator of up-front effort. Boehm's analysis [4] presented a clear relationship between the optimal level of up-front design and application size, due to the diseconomies of scale of software development. The reason for the difference between Boehm's result and these findings may be because of the types of software systems used by the participants in this research, with participants using modern development frameworks which reduce the design effort required for standard problems and are not subject to the same diseconomies of scale relationship.

Abrahamsson, Babar and Kruchten [11] listed a number of factors that they suggested can affect the level of up-front planning: age of the system, rate of change, governance, team distribution, stable architecture and business model. These factors were not defined, but are either included within the boundaries of the contexts described in this paper or evidence has not been found to support them.

Boehm and Turner [28] identified a number of factors that determined whether a team should use an agile or a traditional method when building a system: personnel, culture, dynamism (of requirements), team size and criticality. The latter two are also used to distinguish between the members of the Crystal family of methods [29]. While not necessarily related to up-front architecture, it is useful to consider them; however little or no evidence has been found to support any relationship with up-front architecture.

IX. Conclusion

This paper presents research findings into how agile developers manage up-front architecture design. We found four broad contexts that affect how much up-front architecture: *unstable requirements*, in which requirements change or are initially unknown; *delivering early value*, in which teams release the system early before development has finished – particularly in the form of minimum viable products; *team experience*, in which experienced developers are better able to cope with emergent architectures; and *non-agile environments*, in which the customer requires additional up-front effort to meet their own needs.

Teams choose some or all of five strategies when designing architecture: teams almost exclusively select the *use vendor frameworks* strategy, which allows them to greatly reduce the architectural effort for standard functionality. Then with different degrees of up-front architecture are the *no up-front architecture* strategy, in which the team builds a totally emergent architecture solely for current requirements; the *plan for options* strategy, in which the team uses the project scope to ensure potential requirements can be built without major refactoring; the *address the technical challenges* strategy, in which the team runs experiments to reduce risk to a satisfactory level; and *the full up-front architecture* strategy, in which the team's customer requires a full up-front architecture so they can approve decisions made or approve funding. These strategies are not all mutually exclusive and can be used to varying degrees as required.

The complexity of the system has an important impact on the level of up-front architecture planning required. On the other hand the size of the system does not appear to directly affect the up-front planning.

These results should give agile development teams guidance in what affects how much up-front architecture planning teams do, and in determining which up-front planning strategies can be used. We plan to continue this research with additional interviews to further explore these relationships and develop a deeper understanding of what affects how much effort agile development teams put into up-front architectural planning.

References

[1] P. Kruchten, *The Rational Unified process – an Introduction*. Addison Wesley, 1998.

[2] G. Booch, "Handbook of Software Architecture (Blog)." http://www.handbookofsoftwarearchitecture.com/index.jsp?page=Blog&part=2006, March 2006.

[3] K. Beck *et al.*, "Agile Manifesto." http://agilemanifesto.org/, 2001.

[4] B. Boehm, "Architecting: how much and when?," in *Making software* (A. Oram and G. Wilson, eds.), O'Reilly, 2011.

[5] P. Deemer, G. Benefield, C. Larman, and B. Vodde, "The Scrum Primer." http://assets.scrumtraininginstitute.com/downloads/1/scrumprimer121.pdf, 2010.

[6] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2 ed., 2005.

[7] J. O. Coplien and G. Bjørnvig, *Lean Architecture for Agile Software Development*. John Wiley and Sons, Ltd, 2010.

[8] P. Kruchten, "Agility and Architecture: an Oxymoron?," *SAC 21 Workshop: Software Architecture Challenges in the 21st Century*, 2009.

[9] G. Booch, "The Accidental Architecture," *Software, IEEE*, vol. 23, pp. 9–11, May–Jun. 2006.

[10] S. W. Ambler, "Agile Architecture: Strategies for Scaling Agile Development." http://www.agilemodeling.com/essays/agileArchitecture.htm.

[11] P. Abrahamsson, M. A. Babar, and P. Kruchten, "Agility and Architecture: Can They Coexist?," *IEEE Software*, vol. 27, Mar.–Apr. 2010.

[12] L. Bass, P. Clements, and R. Kazman, *Software Architecture in practice*. SEI Series in software engineering, Addison-Wesley, 2 ed., 2003.

[13] G. Booch, "The Irrelevance of Architecture," *IEEE Software*, vol. 24, pp. 10–11, May–Jun. 2007.

[14] G. Fairbanks, *Just enough software architecture: A risk driven approach*. Marshall and Brainerd, 2010.

[15] P. R. Taylor, *The Situated Software Architect*. PhD thesis, Monash University, Dec. 2007.

[16] H. P. Breivold, D. Sundmark, P. Wallin, and S. Larson, "What does research say about agile and architecture?," *Fifth International Conference on Software Engineering Advances*, 2010.

[17] T. Dybå and T. Dingsøyr, "What Do We Know about Agile Software Development?," *Software, IEEE*, vol. 26, pp. 6–9, Sep.–Oct. 2009.

[18] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory*. Aldine de Gruyter, 1967.

[19] B. G. Glaser, *Theoretical Sensitivity*. The Sociology Press, 1978.

[20] R. Hoda, J. Noble, and S. Marshall, "Organizing self-organizing teams," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering – Volume 1*, ICSE '10, (New York, NY, USA), pp. 285–294, ACM, 2010.

[21] A. Strauss and J. Corbin, "Grounded Theory Methodology," in *Handbook of Qualitative Research* (N. K. Denzin and Y. S. Lincoln, eds.), Sage Publications, Inc, 1994.

[22] G. Allan, "A Critique of Using Grounded Theory as a Research Method," *Electronic Journal of Business Research Methods*, vol. 2, July 2003.

[23] A. Bryman, *Social Research Methods*. Oxford University Press, 3 ed., 2008.

[24] M. Waterman, J. Noble, and G. Allan, "How Much Architecture? Reducing the Up-Front Effort," in *Agile India 2012*, pp. 56–59, Feb. 2012.

[25] E. Ries, *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Publishing Group, 2011.

[26] R. Hoda, J. Noble, and S. Marshall, "Agile undercover: When customers don't collaborate," in *Agile Processes in Software Engineering and Extreme Programming* (A. Sillitti, A. Martin, X. Wang, E. Whitworth, W. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, eds.), vol. 48 of *Lecture Notes in Business Information Processing*, pp. 73–87, Springer Berlin Heidelberg, 2010.

[27] E. R. Poort and H. van Vliet, "Architecting as a risk- and cost management discipline," in *WICSA 2011*, pp. 2–11, IEEE Computer Society, 2011.

[28] B. Boehm and R. Turner, "Using Risk to Balance Agile and Plan-Driven Methods," *Computer*, vol. 36, no. 6, pp. 57–66, 2003.

[29] A. Cockburn, *Agile software development: the cooperative game*. Upper Saddle River, NJ: Addison-Wesley, 2 ed., 2007.