2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

How Much Up-Front? A Grounded Theory of Agile Architecture

Michael Waterman*, James Noble†, George Allan‡
*Specialised Architecture Services Ltd, Wellington, New Zealand
mike@specarc.co.nz
†School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
James.Noble@ecs.vuw.ac.nz
‡Email: drgeorgeallan@gmail.com

***Abstract*—The tension between software architecture and agility is not well understood by agile practitioners or researchers. If an agile software team spends too little time designing architecture up-front then the team faces increased risk and higher chance of failure; if the team spends too much time the delivery of value to the customer is delayed, and responding to change can become extremely difficult. This paper presents a grounded theory of agile architecture that describes how agile software teams answer the question of how much up-front architecture design effort is enough. This theory, based on grounded theory research involving 44 participants, presents six forces that affect the team's context and five strategies that teams use to help them determine how much effort they should put into up-front design.**

I. Introduction

Software architecture is the high-level structure and organisation of a software system [1]. Because architecture defines system-level properties, it is difficult to change after development has started [2], causing a conflict with agile development's central goal of better delivering value through responding to change [3], [4].

To maximise agility, agile developers often avoid or minimise architectural planning [5], because architecture planning is often seen as delivering little immediate value to the customer [6]. Too little planning however may lead to an *accidental architecture* [7] that has not been carefully thought through, and may lead to the team spending a lot of time fixing architecture problems and not enough time delivering functionality (value). An accidental architecture can potentially lead to gradual failure of the project. On the other hand too much architecture planning will at best delay the start of development, and at worst lead to expensive architectural rework if the requirements change significantly. Up-front architecture design effort is therefore a trade-off between the need for architectural support [8] and agility. This conflict does not yet have a satisfactory solution [9].

Many agile methodologies recommend some architecture planning [10], but there is little guidance as to which architecture decisions should be made up-front [6], and what factors influence those decisions.

Many agile developers deal with this absence in guidance by designing 'just enough' architecture up-front to enable them to start development, with the rest being completed during development as required [6]. How much is just enough depends on context, which includes technical and environmental factors such as the organisation and the domain [6], as well as social factors [11] such as the background and experience of the architects. A particular system may have more than one architectural solution [12], [13], and two architects are likely to produce different architectures for the same problem with the same boundaries [11]. It is therefore difficult to determine in advance how much 'just enough' is.

There has been little research on the relationship between software architecture and agile development to date [14]. This lack of research does not mean that it is not an important issue: at the XP2010 conference, how much architectural effort to expend was rated as the second-equal most burning question facing agile practitioners [15].

This paper addresses this problem by presenting "a grounded theory of agile architecture," a high-level descriptive theory that explains how teams determine how much architecture to design up-front. A team's up-front effort depends on five agile architecture *strategies* that a team may choose; which strategies a team chooses depend on the context of the team and the system being built. Context is characterised by

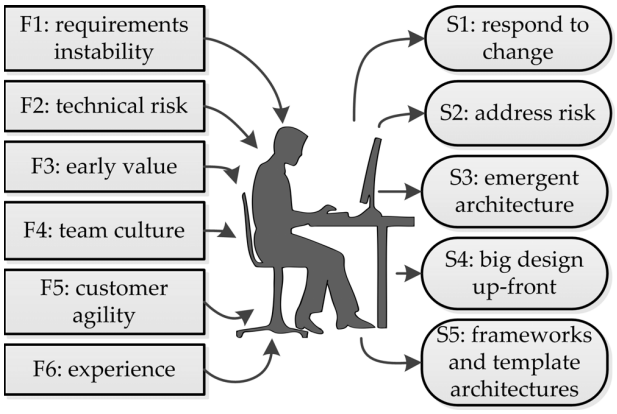

Fig. 1. The forces and strategies that comprise the theory of agile architecture

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.54

six *forces* that a team must consider when designing an agile architecture. The forces and strategies are listed in figure 1.

This theory was derived using grounded theory, a systematic and rigorous methodology [16] that allows researchers to develop a *substantive* theory (based on data) that is a "formal, testable explanation of some events that includes explanations of how things relate to each other" [17]. The theory is more than a description of experiences or perspectives [18] based on a compilation of anecdotes, expert opinions and experience reports: it is a well-codified and systematically-generated set of propositions [19], [20].

This paper is based on research published in Waterman's PhD thesis [21].

The rest of this paper is as follows: section II defines some of the terminology used in the paper, and section III presents the research methodology. Section IV describes the agile architecture forces, and section V describes the agile architecture strategies. Section VI discusses the relationships between the forces and the strategies, and the theory in context of related work. Section VII discusses the limitations of the theory. Finally, section VIII concludes the paper.

II. Definitions

We define *software architecture* as "the set of significant decisions about the high level structure and the behaviour of a system" [22], where 'significant' is measured by the cost of change [2]. In other words, architecture comprises the planning and design decisions that are made up-front because they are difficult to change once development has started. *Architecting* is the process of making architectural decisions [23], and may include research, analysis, modelling, verification and the creation of architectural artefacts. *Architects* make the architectural decisions; in agile software development, this is often the whole team through collaboration.

We define *agility* from a conceptual perspective, based on an abridged version of Conboy's definition [24]: "[Agility is] a software development team's ability to create change, respond to change and learn from change so that it can better deliver value." Adolph used a similar definition [25]. This definition avoids specifying particular methodologies and practices, and refers simply to the team's ability to deliver value more quickly by responding to change and by improving its processes.

We define an *agile architecture* as an architecture that satisfies the definition of agility by being able to be easily modified in response to changing requirements, is tolerant of change, and is incrementally and iteratively designed – the product of an agile development process.

III. Research Methodology

This research used the grounded theory methodology [26]. Qualitative research methodologies such as grounded theory are used to investigate people, interactions and processes, and architecture is very dependent on the architects and the development teams themselves. 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 [14], an inductive strategy that develops a new hypothesis was more suitable for this research. We selected grounded theory because it allowed us to develop a high-level theory based on a broad range of participants.

A. Data Collection

We collected data primarily through face-to-face semi-structured interviews with agile practitioners who design or use architecture, or who are otherwise architecture stakeholders. The average interview length was 70 minutes.

We asked participants to select a project that they had been involved with to discuss during the interview. Types of projects varied hugely, from green field 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. Systems varied from highly mission-critical systems such as flight control and health record management, to business critical systems such as banking and retail, through to largely non-critical administration and entertainment broadcast systems. We also obtained documentation, such as software architecture documents, which were able to confirm the decisions made and why they were made, and copies of architecture models, which provided overviews of the architectures. Additional data in the form of discussions by email and telephone clarified earlier interviews and documentation where necessary.

B. Research Participants

We interviewed 44 participants in 37 interviews. Participants were gathered through industry contacts, agile interest groups and through direct contact with relevant organisations. Almost all participants were very experienced architects, senior developers, team leaders and development managers with at least six years' experience (twenty years was not uncommon), and most were also very experienced in agile development. Organisation types included independent software vendors (ISVs), government departments, mass-market product developers and sole contractors. Different types of agile development were included, with most participants using Scrum; other methods included XP, Lean and bespoke methods. Most participants adapted their processes to some extent to suit their team or customer's requirements. The inclusion of this range of participants and systems enabled the research to include the effects of different factors on architecture decision making.

To maintain confidentiality, we refer to the participants using labels P1 to P37, reflecting the interview numbers. Where there are several participants in a single interview, we give the labels alphabetic suffixes, such as P23a, P23b and P23c. A summary of participants and their projects is listed in Table I on the following page.

C. Data Analysis

The first step of grounded theory data analysis is open coding, which can begin as soon as the first data are obtained.

TABLE I
SUMMARY OF PARTICIPANT DETAILS

	Role	Experience in role	Organisation type	Domain	Agile methods	Team size or no. of teams	System description
P1	Developer	> 15 years	Government agency	Health	Single developer	1 team member	Web-based, .NET
P2	Developer/ architect	20 years	ISV	E-commerce	Scrum	3 team members	.NET, cloud-based
P3	Devopment manager	6 years (agile)	ISV	Personnel	Scrum	3 teams	Web-based, .NET,
P4	Director of architecture	> 20 years	ISV	Digital archiving	Scrum	5 developers	Java, rich client, suite of standalone tools
P5	Coach/dev. manager	20 years	Start-up	Entertainment	Scrum/ kanban	Various	Various
P6	Man. Dir./ lead dev.	20 years	Service provider	Telecoms	Iterative	1–3 developers	Suite of standalone applications
P7	Business analyst	5–6 years	ISV	Telecoms	Scrum	12 team members	Suite of web-based services
P8	Lead developer	6 years	ISV	Digital archiving	Scrum	4–14 team members	Ruby on Rails, Java back-end
P9	Developer	40 years	Financial services	Telecoms	Bespoke	2–24 team members	Web-based system
P10	Coach	25 years	Hardware and services	Transport	Scrum/XP	500–800 developers	Large distributed web-based system
P11a	Development manager	10 years	Government	Government services	Scrum	8 team members	Web-based, .NET
P11b	Architect	15+ years					
P12	Senior developer	10 years	Financial services	Financial services	Scrum	6–7 developers	Web-based, .NET
P13	Architect	16 years	ISV	Medical	Scrum	12 team members	Monolithic .NET app
P14	Architect	10 years	ISV	Animal health	Scrum	6–8 team members	.NET, large GIS component
P15	Customer	4 years	Start-up service provider	Retail (electricity)	Scrum	7 developers	Ruby On Rails
P16a	CEO/chief engineer	> 10 years	ISV	Retail (health)	XP	5 team members	Ruby On Rails
P16b	Head of engineering	10 years					
P17	Manager/coach	> 10 years	Government	Statistics	Scrum	6 dev + admin	Web-based, PHP using DAO pattern
P18	Dev. manager	10 years	Multinat. hardware vendor	Health	Scrum	15 team members	Web-based, Java platform
P19	Dev. manager	6–7 years	Start-up service provider	Retail (travel)	Lean	4 developers	PHP/Symfony, Javascript/Backbone
P20	Coach and trainer	20 years	Independent consultant	N/A	Scrum	N/A	N/A
P21a–P21d	Manager/coach, architect, team leader, team leader	20 years, N/A, N/A, N/A	ISV	Retail (publishing)	Scrum	3 teams; 40 total	.NET, Websphere Commerce, SAP, others.
P22	Senior manager	14 years	ISV	Contact management/marketing	Scrum/XP	More than 40 total	.NET
P23a-P23c	Engineering manager, product lead, team lead	N/A, 6 years, N/A	Service provider	Pharmaceutical	Bespoke	3 teams	Various web based, client/server
P24	Customer	> 10 years	Start-up service provider	Retail (electricity)	Scrum	7 developers	Ruby On Rails web applications
P25	Team lead	10 years	ISV	Banking	Scrum	1 team	.NET, single tier web
P26	Team lead	8 years	ISV	Water management	Scrum	8 team members	.NET, web based, 7 tier
P27	CEO/coach	16 years	Start-up service provider	Retail (electricity)	Scrum	7 developers	Ruby On Rails
P28	Technical lead	13 years	Service provider	Broadcasting	Scrum	42 team members	Python with Django, CMSs for multiple websites
P29	Development manager	20 years	Banking	Banking	Kanban	20 team members	Web based, interface to mainframe
P30	Consulting architect	25 years	Service provider	Telecoms	Scrum	7 team members	Python with Django and Twisted, NoSQL
P31	Enterprise architect	25 years	Government	Transport	Bespoke	7 team members	Web services, SOA using .NET/WCF
P32	Software dev. director	> 15 years	ISV	Government	FDD/kanban	N/A	N/A
P33	Product architecture team leader	15 years	Medical service providers	Medical	Kanban/Scrum	One team per product stream	Multiple product streams; SOA
P34	Development unit manager	15 years					
P35	Design Engineer	N/A					
P36	Development unit manager	12 years					
P37	Consultant/ freelance software developer/architect	13 years	Service provider	Broadcasting	Bespoke/XP	15–16 team members	Java/embedded

In open coding, phenomena in the data are methodically identified and labelled using a code that summarises the meaning of the data [27]. As open coding progresses, emerging codes are compared with earlier codes; codes with related themes are aggregated into higher levels of abstraction called *concepts*. This process, called *constant comparison* [28], continues at the concept level, with similar concepts being aggregated into a third level of abstraction called *categories*. Categories are the highest conceptual elements of grounded theory analysis; a grounded theory research project may have hundreds of different codes but will typically have no more than four or five categories [20]. The relationships between the categories are analysed and focused using *selective coding*; a dominant category emerges as the *core category*, which is central to the emerging theory. This theory is a "formal, testable explanation of some events that includes explanations of how things relate to one another" [17]. Throughout the analysis process, *memos* – free form notes ranging anywhere in size from a sentence to several pages – are written to record thoughts and ideas about developing relationships between codes, concepts and categories, and to aid the development of the theory [29].

Grounded theory uses iteration to ensure a wide coverage of the factors that may affect the emerging theory [27]: later data collection is dependent on the results of earlier analysis. Data collection and analysis continue until *saturation* is reached, which occurs when no new insights are learned, and all variations and negative cases can be explained [30].

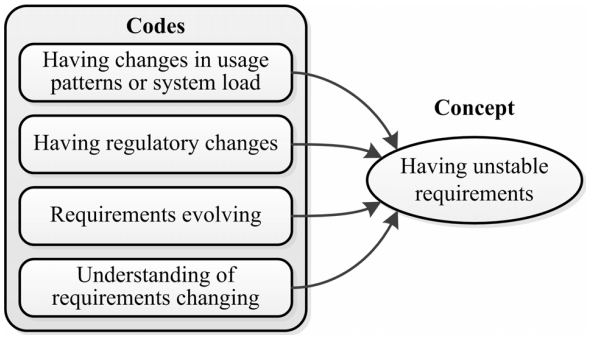

Fig. 2. An example of a concept emerging from its codes

We can illustrate the grounded theory process with an example from this research. One participant commented that they had regular tax law changes that meant regular changes to their requirements:

> *"You've got your taxation changes coming in on specific dates throughout the year, so those are generally around our release dates, because we have to stay compliant with that." (P3, development manager)*

We coded this as 'having regulatory changes.'

Similarly another participant commented on the pharmaceu-

tical regulations that affected his company's product:

> *"The regulations keep changing every six months." (P23, senior manager)*

We also coded this as 'having regulatory changes.' Codes that had similar themes to this example included 'having changes in usage patterns or system load,' 'requirements evolving' and 'understanding of requirements changing.' We combined these similar codes into a concept called 'having unstable requirements,' as shown in figure 2.

A full discussion of the methodology is presented in Waterman's PhD thesis [21].

IV. Agile Architecture Forces

This research has found six forces that make up the team's and the system's *context*, the set of conditions that can affect how the team designs the agile architecture. The six forces are F1 (REQUIREMENTS INSTABILITY), F2 (TECHNICAL RISK), F3 (EARLY VALUE), F4 (TEAM CULTURE), F5 (CUSTOMER AGILITY) and F6 (EXPERIENCE).

F1: Requirements Instability

F1 (REQUIREMENTS INSTABILITY) refers to the effect that unstable requirements have on up-front planning. Being able to respond to changing requirements is one of the underlying values of the Agile Manifesto [31] and is central to our conceptual definition of agility (see section II). All participants in this research reported unstable requirements to some extent, whether they were part of team working on a relatively stable redevelopment project or whether they were part of a start-up in a highly dynamic environment:

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

Unstable requirements are caused by *incomplete requirements* and by *changing requirements*. Incomplete requirements are caused by the customer not initially knowing what they want, or coming up with new ideas about what they want during development. In some instances requirements are fairly stable, but it is not possible for the team to develop a complete understanding of those requirements up-front; a full understanding comes later during development. As an example, P13's team was replacing an old system with a new system that was functionally very similar. They were not able to understand the intricacies of the system before starting development:

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

Requirements may also change frequently during development, due to the customer changing their mind about what they have already requested or by changing usage patterns. Often it is impossible for a team or their customer to know how their system will be used once it goes live; if the system is commercially successful, usage is likely to be higher than anticipated, leading to the performance suffering unless the system is redesigned to ensure it can maintain the required performance levels:

> *"[Planning up-front] assumes you know to begin with the usage patterns that your system is going to be put through... and you don't. You have to play it out in real life." (P10, coach)*

In a business environment, any delays caused by detailed requirements gathering and planning increase the chance that requirements will change:

> *"[If you put too much detail into your requirements] there's a fat chance that by the time you get around to starting the work your world has changed." (P3, development manager)*

Teams therefore only define the high-level requirements up-front. Detailed requirements are gathered as needed; any requirements not immediately being implemented are left undefined, because any additional time spent on requirements gathering is wasted when requirements change. By avoiding detailed requirements gathering, analysis and architecture design, software engineers can start development and demonstrating the product to the customer early in development, and therefore can get feedback from the customer early.

F2: Technical Risk

F2 (TECHNICAL RISK) describes the effect that exposure to potentially negative outcome has on a team's up-front effort. Risk is caused by complex architecture, as described by the participants:

> *"Complexity in terms of how complicated the code and the solution underneath it are going to be does influence how much planning we're going to do." (P33, product architecture team leader)*

Participants described architecture complexity as being caused by one or more of three things: challenging or demanding architecturally significant requirements (ASRs), by having many integration points with other systems, and by involving legacy systems. ASRs are the requirements that drive and constrain a system's architecture [32], and consist primarily of the qualities, or non-functional requirements, such as performance, security and reliability. Challenging ASRs are difficult to design for, and may lead to many trade-offs. Participants described how challenging requirements lead to a complex architecture:

> *"Highly demanding non-functional requirements are in my mind a direct driver of complexity and will require more effort to address, particularly as there are trade-offs between them – for example, performance versus security." (P31, enterprise architect)*

Legacy systems are those that are no longer being 'engineered' but rather are simply patched or hacked as requirements change [33], [34]. Legacy systems have increased complexity:

> *"Systems become more complex with age. Just the burden of code – entropy over time and all that." (P32, software development director)*

Good engineering practices such as simplicity, modularity and high cohesion are eroded, and continuing to develop – or even interfacing with – these entropic legacy systems are a source of complexity that requires more up-front exploration or experimentation to ensure that integration succeeds.

Participants also identified integration points, or interfaces to external systems, as a major source of complexity in the systems being developed, particularly when the other systems are legacy or are built from different technologies. Integration with other systems require data and communications to be mapped between the systems, adding to the up-front effort to ensure integration is possible with the technologies being used. For example:

> *"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)*

Teams must reduce the risk to a suitable level. The amount of risk reduction, and hence architecture effort, depends on the team's and the customer's appetite for risk.

While complexity is a direct cause of risk, size is not if it does not also increase complexity:

> *"If we have size that just extends the time, it's of little concern to us [architecturally]. It's just a slightly larger backlog, management overhead." (P32, software development director)*

A team mitigates risk through using the strategy S2 (ADDRESS RISK), described in section V.

F3: Early Value

F3 (EARLY VALUE) refers to a customer's need to gain value from a system or product being built (rather than simply provide feedback) before all functionality has been implemented, perhaps in the form of a *minimum viable product* [35]. Early value is frequently required by businesses operating in a dynamic commercial environment who cannot wait for the full product to be developed:

> *"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)*

Teams that deliver early value must reduce the time to the first release by spending less time on up-front architecture design. They achieve this by reducing the *planning horizon* – how far ahead the team considers (high level) requirements for the purpose of architecture planning. In its extreme, teams do no planning ahead, using S3 (EMERGENT ARCHITECTURE), described in section V, to start delivering value as soon as possible.

No up-front design increases the overall effort because the architecture must be evolved with each iteration, and hence cost of development increases. With each iteration, the team has to consider if the existing architecture is suitable for the current set of requirements being implemented; if it is not, then the architecture must be redesigned. Overall, there is likely to be more redesign than if a longer planning horizon is used with more initial design. For example, as the product grows and the user base grows, the architecture that was designed for day one will no longer be suitable and will need to be redesigned – a redesign that may not have occurred with a larger planning horizon. While "cost is always a concern" (P10), participants accepted this increase in cost, because it occurs later when the customer has cash flow from early adopters to pay for that cost:

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

and

> *"If we needed to go to a million [end users] [...] we'd have to rewrite swathes of the software – you absolutely would have to. But it's a problem you can have once you've got a million users and you've got a million users worth of revenue." (P27, CEO/agile coach)*

– particularly important for start-ups with limited cash.

F4: Team Culture

F4 (TEAM CULTURE) describes the effect that a team's culture has on its agility and the effort it puts into up-front planning. A culture that is people-focused and collaborative is a very important factor in a team's ability to communicate. For example:

> *"There was a very good culture; [...] we could be full and frank in our discussions and planning and nobody would get too offended." (P12, senior developer)*

Many participants also commented on the importance of trust in an agile culture:

> *"And the most important thing when it comes to agile is trust." (P15, customer)*

A team without a trusting people-focused and collaborative culture has to rely on documentation for communication and formal plans, and hence requires more up-front effort to guide development.

The size of the team affects its ability to communicate; small teams are able to collaborate better, while large teams require more structure and more up-front planning, and hence are less agile:

> *"And the team has been going, '[we're] too big, can't communicate, hate the meetings.' [...] It definitely takes more effort and negotiation in iteration zero. I don't think it changed the complexity of the architecture or the way we attack the architecture – it's just comms time." (P29, development manager)*

The agile experience of team members also affects up-front planning, because a people-focused and collaborative culture does not come into being instantaneously – it comes with experience and practice as the team members become more experienced working together:

> *"It [agile] is a continuous journey, people have to get accustomed to the culture." (P20, coach and trainer)*

A team new to agile – particularly if it consists of developers from a traditional plan-driven background – is likely to struggle to be successful without a predefined architectural plan, but as the team becomes more experienced and develops an agile mind-set, it will become more comfortable working without the guidance of up-front plans:

> *[Inexperienced developers] will find it very difficult to start without having a concrete design in place. The culture will be different. They always want to follow what is already laid out. [...] An experienced set of developers or members in the team will make it easier to actually [form and evolve] the design." (P22, senior manager)*

F5: Customer Agility

F5 (CUSTOMER AGILITY) describes the culture of the customer's organisation and the huge impact that it has on the amount of up-front architecture design a team does. A customer must have an agile culture that is similar to the team's culture, whether the team is in-house or an ISV (independent software vendor), for the team to be truly agile. A highly-agile team will not fit in well with a heavyweight process-oriented organisation that prefers planning and formal communication.

Like trust within a team, trust between the company and the team is important to help the company become part of the team and break down formal processes, improving agility:

> *"We've become the same team. That removes a lot of the tension, streamlines the process massively. [...] What we started finding is that our customers then start breaking down their own processes, and start making processes [agile] for us." (P27, CEO/coach)*

Highly agile customers do not require their development teams to produce excessive documentation or plans, and do not need fixed budget approval for fixed scope delivery.

Conversely, it is difficult for an agile team to operate in a non-agile process-driven environment: a customer that does not buy in to the agile mind set greatly reduces the team's ability to be agile. This can happen in many ways. For example, a non-agile customer that prefers a 'command and control' management style will impose their own processes upon the team.

P32 commented that some of their planning was simply because their customer expected it and preferred to see the developers perform traditional planning:

> *"We have learned that a lot of the planning we've done up-front has been more a planning art or a planning play or some sort of production because there's some air of respectability around it. It's necessary – people demand it." (P32, software development director)*

Another way in which a non-agile customer can negatively affect a team's agility is by taking on a 'benediction' role, in which it 'blesses' or approves the team's architectural decisions prior to development. A customer may want to ensure the decisions are compatible with a larger system, with company policy, or they may simply want comfort in the quality of the design decisions.

Many non-agile customers prefer the accountability of a fixed price contract with a fixed scope and fixed delivery dates – perhaps with penalties if milestones are not met:

> *"They've mandated quite a draconian liquidated damages kind of risk contract, about [what happens] if you miss a milestone." (P32, software development director)*

For a team, these contracts mean investing significant amounts of time up-front to map out in detail how much work is required to deliver the customer's list of requirements.

Other reasons for a customer not buying into the agile mind set include needing advance budget approval from their CFO or Board, for which they need a fixed scope, and simply being unable to commit to ongoing time with the development team. Agile developers solve this latter problem by spending more time planning to compensate for the lack of feedback, and hence are less agile.

F6: Experience

F6 (EXPERIENCE) describes the impact that an experienced architect's tacit knowledge and implicit decision-making ability has on the time that an agile team spends on up-front design. Experienced architects have breadth of knowledge; they are more likely to be aware of suitable options for implementing a solution and better understand what will work and what will not. Hence they can make better decisions:

> *"Figuring out whether there's something out there appropriate that already does it – that sort of thing – that's where experience and knowledge really come into play." (P36, development unit manager)*

While generally important for all software development methods, experience is more important in agile development because the tacit knowledge and implicit decision-making that come with experience supports agile development's reduced process and documentation, and reduces the up-front effort:

> *"You implement certain patterns without thinking [...] you've done this kind of pattern for solving this kind of a problem, without even thinking that this is the way that you are going." (P16b, head of engineering)*

while inexperienced architects rely more on explicit decisions that are written down and which need more effort in the form of proofs of concept, experiments (spikes) and research.

It is also important for developers to know the technology being used. Knowledge or experience in the technology helps the team to speed up design and avoid the technology's weaknesses:

> *"The architect we had working on this worked on another project or two using this framework, plus also other portal ones, and has definite opinions on pitfalls to avoid." (P7, business analyst)*

If a team does not have the required experience, they may have to gain that knowledge through research, or they may bring in someone with suitable experience to join the team.

V. Agile Architecture Strategies

The theory of agile architecture consists of six strategies that teams may choose from in response to the forces described above, and which help the teams determine how much architecture to design up-front. The strategies are S1 (RESPOND TO CHANGE), S2 (ADDRESS RISK), S3 (EMERGENT ARCHITECTURE), S4 (BIG DESIGN UP-FRONT) and S5 (USE FRAMEWORKS AND TEMPLATE ARCHITECTURES).

S1: Respond to Change

A team's ability to use S1 (RESPOND TO CHANGE) is directly related to how agile it is. S1 increases the architecture's agility by increasing its modifiability and its tolerance of change, and allows the team to ensure the architecture continuously represents the best solution to the problem as it evolves:

> *"The key thing is [the architecture] is not going to be frozen. It's not going to be documented and put it in a glass case and hung on the wall and [we] say, 'that's the architecture, let's look at it and keep developing' – no, that's not it." (P22, senior manager)*

Teams proactively review their architecture to ensure it stays up to date:

> *"...so it's very much evolving and living with the system in response to, what have we learned, what are the things we need to do now?" (P10, coach)*

Agile teams use S1 because of F1 (REQUIREMENTS INSTABILITY). There are five tactics that teams can use to implement the strategy and design an agile architecture: *keep designs simple*, *prove the architecture with code iteratively*, *use good design practices*, *delay decision-making*, and *plan for options*.

Keeping designs simple means only designing for what is immediately required: no gold plating and no designing for what *might* be required or for what can be deferred. Proving the architecture with code iteratively means testing a design by building it and testing it in real life rather than through up-front analysis, and refining the design if it proves to be unsuitable. This tactic can be used once development has started. Using good design practices, such as separation of concerns, is important in all development whether agile or not, but is particularly important in agile development because it makes it easier to modify the architecture as requirements evolve. Delaying decision-making means not making architecture decisions too early; waiting until sufficient information on the requirements is known so that there is less likelihood of the decisions needing to be changed. Planning for options means building in generality and avoiding making decisions that are unnecessarily constrained and which may close off possible future requirements without significant refactoring.

Three of these tactics, keeping the design simple, proving the architecture with code iteratively and following good design practices, increase the modifiability of the architecture so that when requirements change or become known the architecture can be easily updated. The other two tactics, delaying decisions and planning for options, increase the architecture's tolerance of uncertainty (its resilience to change), so that any changes to the requirements have less impact on the architecture. Keeping the design simple, proving the architecture with code and delaying decisions all reduce up-front effort. Following good design practices and planning for options may slightly increase the up-front architecture effort, but will most likely decrease overall effort. The tactics are summarised in Table II.

TABLE II
A COMPARISON OF THE 'RESPOND TO CHANGE' TACTICS

Tactic	**Impact on responsiveness to change**	**Reduces up-front effort?**
Keep designs simple	Increases modifiability	Yes
Prove the architecture with code iteratively	Increases modifiability	Yes
Use good design practices	Increases modifiability	No
Delay decision making	Increases tolerance of change	Yes
Plan for options	Increases tolerance of change	No

S2: Address Risk

S2 (ADDRESS RISK) reduces the impact of risk before it causes problems, and is usually done up-front, particularly for risk relating to system-wide decisions (for example, risk in selecting the technology stack or top-level styles). Using S2, a team designs the architecture in sufficient detail that it is comfortable that it is actually possible to build the system with the required ASRs with a satisfactory level of risk:

> *"That's essentially what you're doing in the technical design/planning phase, you're trying to reduce the risk of the whole thing going off the rails [...] It's very much a risk-based process." (P36, development unit manager)*

More TECHNICAL RISK (F2) means more up-front architecture design is required, meaning the team is less able to use S1; to reduce risk, a team must sacrifice some of the team's ability to respond to change. The team can find a balance between S1 and S2 by doing sufficient up-front design to reduce risk to a satisfactory level, and delaying decisions where the impact of risk is low. The higher the impact of the risk, the more important it is to mitigate that risk early. For example, P33–P36's medical system had clinical and security risks that meant the design had to undergo additional architectural scrutiny:

> *"Anything that is deemed to relate to either a clinical risk or a security risk is actually assessed by a separate independent team, who will tell you how bad it [the risk] is." (P16a, CEO/chief engineer)*

while P2's customer was willing to risk a small financial loss in certain circumstances to get a cheaper system with less planning:

> *"The customer has said to us he is quite willing to trade the risk of accidentally redeeming the same voucher*

twice, once in a blue moon [...] so if two different people in different geographies within a tenth of a second of each other try to redeem the same voucher there's lowish odds that they'll redeem it twice." (P2, developer/architect)

Teams can reduce risk through the use of research, modelling and analysis, performing experiments (spikes) or by building a 'walking skeleton' [36] of the system.

S3: Emergent Architecture

S3 (EMERGENT ARCHITECTURE) produces an architecture in which the team makes only the minimum architecture decisions up-front, such as selecting the technology stack and the highest level architectural styles and patterns. In some instances these minimum decisions will be implicit or will have already been made (and can therefore be considered as constraints), in which case the architecture is totally emergent.

When using S3, the team only considers the requirements that are immediately needed for its design, ignoring even high-level requirements that are to be implemented in the longer term. S3 helps ensure the design is the simplest it can be, and the product being built can be released to market as quickly as possible, hence satisfying a need for EARLY VALUE (F3). For example, P29 looked no further than a few weeks ahead:

"We're doing bugger all [practically no up-front design] actually. Most of the time we're working a couple of iterations ahead, we're looking at the design, things that might have to go through to committee, so we tend not to plan a year or two out – we're planning a few weeks out." (P29, development manager)

S3 is likely to be used when developing a minimum viable product, or MVP (see F3). If the system has demanding architecturally significant requirements (ASRs) or unique requirements, it may need a more complex solution that requires bespoke components or multiple frameworks, and more up-front design to address risk (S2). Hence F2 precludes an emergent design.

S4: Big Design Up-Front

S4 (BIG DESIGN UP-FRONT) requires that the team acquires a full set of requirements and completes a full architecture design before development starts. There are no emergent design decisions, although the architecture may evolve during development. S4 is undesirable in agile development because it reduces the architecture's ability to use S1 (RESPOND TO CHANGE) by increasing the time to the first opportunity for feedback, increasing the chance that decisions will need to be changed later, and increasing the chance of over-engineering.

While S4 may be considered the case of addressing risk (S2) taken to the extreme, in reality the use of S4 is driven primarily by an absence of CUSTOMER AGILITY (F5) rather than the presence of TECHNICAL RISK (F2):

"There's a definite need to estimate [the total cost] and there's a definite need to give confidence on the functional scope at a big level." (P32, software development director)

The up-front design in S4 is sufficient to satisfy the non-agile customer: either sufficient to prove that the team knows how to solve the problem before starting, sufficient that the team can estimate the cost of the system for the given requirements for a competitive tender, or sufficient that they are able to complete the design without ongoing interaction with their customer:

"There's a definite need to estimate and there's a definite need to give confidence on the functional scope at a big level." (P32, software development director)

While the team using S4 cannot fully implement S1 – for example, they cannot use the 'delay decisions' tactic – they may be able to use other S1 tactics, such as good design practices and planning for options so that they can still evolve their architecture as requirements change. Not being able to delay decisions, however, will compromise their ability to be agile:

"So if we're going to have to do a heavy architecture which plans for a year or two or five years into the future on every one of those experiments, we're screwed. We cannot be agile." (P29, development manager)

Larger independent software vendors (ISVs) often use S4 because their customers are more likely to be larger process-driven organisations who require more financial accountability. Larger ISVs therefore often struggle to become as agile as smaller organisations.

S5: Using Frameworks and Template Architectures

S5 (USE FRAMEWORKS AND TEMPLATE ARCHITECTURES) is the use of software frameworks, and template and reference architectures sourced from particular framework vendors for use with those frameworks. Frameworks such as .NET, Hibernate and Ruby on Rails include default architectural patterns which constrain the systems to these patterns.

S4 provides the benefit of standard solutions to standard problems, which means that software engineers do not need to make as many architectural decisions, and can greatly reduce the effort required to design a system and get it up and running:

"So we don't have architectural discussions – we don't need to – the problem's been solved [in the framework]. Don't try to solve it again. So we have very, very little discussion." (P27, CEO/coach)

Frameworks – particularly those that follow the 'convention over configuration' paradigm – also greatly reduce the complexity of the architecture because many of the architectural decisions are embedded in the framework, and hence architectural changes can be made with a lot less effort. What used to be considered architecture decisions in the past can now sometimes be considered design (non-architecture) decisions:

"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)

which means that fewer decisions have to be set in stone:

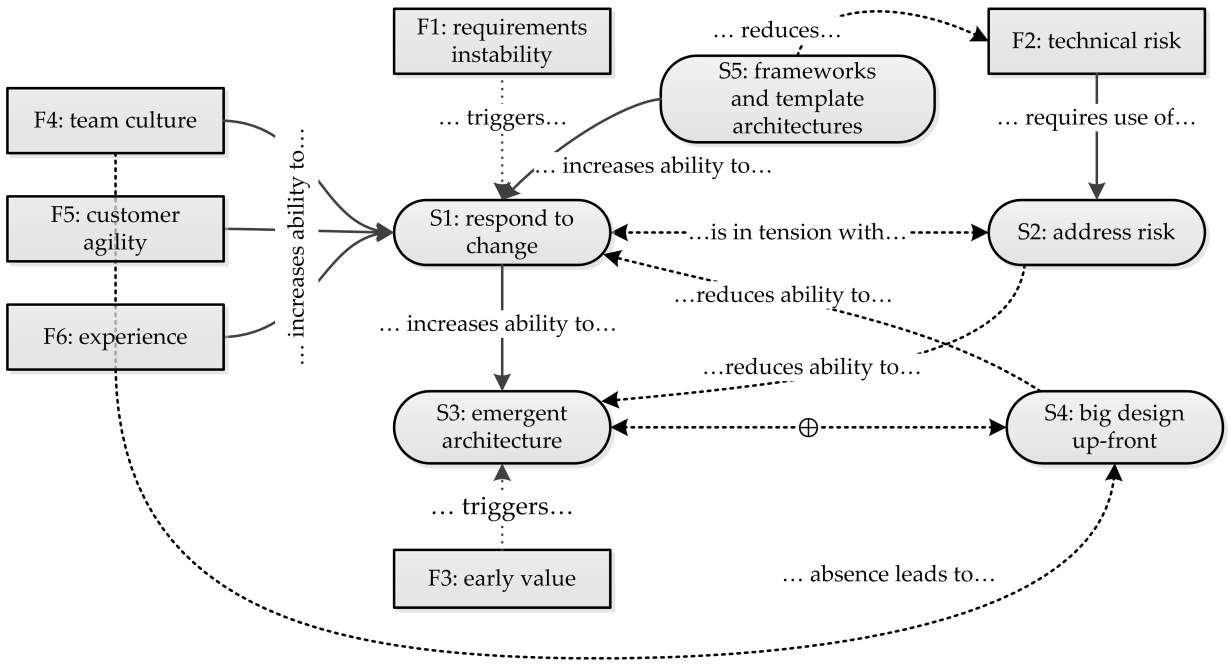

Fig. 3. The relationships between the forces and strategies. The rectangular boxes represent forces, and the round-cornered boxes strategies. Arrows represent dependencies or causal relationships: a change in the independent force or strategy causes either a positive change (solid line) or a negative change (dashed line) in the dependent force or strategy. A dotted line represents a trigger dependence: the presence of the force is a trigger for the corresponding strategy. The symbol $\oplus$ represents mutual exclusion.

"Those [structural] decisions can be very emergent nowadays; I don't think they're nearly as intractable." (P29, development manager)

While frameworks and templates are hugely beneficial to all development methods, the ability to change architecture decisions more easily is most useful to agile methods.

Despite their immense importance and ubiquity, teams must be aware that frameworks cannot always provide a complete solution. If the problem is not standard, if the requirements are critical and the required architecture is sufficiently complex or unique, there may be no suitable frameworks or existing libraries that a team can use to implement a design, either as a whole or in part. In these situations a team needs to design and build bespoke components or libraries and, for those components, miss out on the benefits that frameworks can provide. Having to design bespoke components increases the complexity of the architecture decisions that need to be made and increases the technical risk – and hence increases the use of S2 (ADDRESS RISK). S5 can be used at the same time as any of the other strategies, S1–S4.

VI. Discussion

A. Relationships Between Forces and Strategies

Figure 3 shows the relationships between the forces and the strategies. A team's use of S1 (RESPOND TO CHANGE) is triggered by the presence of F1 (REQUIREMENTS INSTABILITY). A team's agility, and thus its ability to use the tactics of S1, is increased by F4 (TEAM CULTURE), F5 (CUSTOMER AGILITY) and F6 (EXPERIENCE). Agility is not directly affected by F1; rather, less stability may motivate the team to improve its agility through improving F4, F5 and F6 so that it is better able to respond to change.

S1 is in tension with S2 (ADDRESS RISK), which must be used to address F2 (TECHNICAL RISK), and hence S1 and S2 must be in balance. An extremely agile team that does not need to use S2 can, when triggered by F3 (the need for EARLY VALUE), reduce its up-front effort to the point where it is using S3 (EMERGENT ARCHITECTURE). On the other hand, a team with low levels of F4, F5 and F6 may have to use S4 (BIG DESIGN UP-FRONT), which will reduce its ability to use S1.

S5 (USE FRAMEWORKS AND TEMPLATE ARCHITECTURES) can be used to significantly reduce development and design effort; in particular it significantly reduces risk and up-front effort for standard problems. A team successfully using S3 would typically be using S5 to build a low risk system.

B. Relationships With Related Work

Abrahamsson, Ali Babar and Kruchten listed eight factors that affect up-front architecture design [6]; one of these is *rate of change* (of requirements), which corresponds to F1 (REQUIREMENTS INSTABILITY). This research, however, has found that F1 does not affect the team's up-front design effort; rather, the reverse is true: the team's agility, and thus its ability to use S1 (RESPOND TO CHANGE), impacts its ability to reduce up-front effort and its ability to respond to unstable requirements.

F2 (TECHNICAL RISK) is caused by complexity, and causes a team to use S2 (ADDRESS RISK), increasing up-front effort. Abrahamsson et al. identified *size* as a factor that affects up-front design [6]. This research has found that size is not a direct factor in up-front effort; rather, complexity is, although size is an attribute of complexity [37]. Size is possibly considered a proxy for complexity. Abrahamsson et al. listed three other factors that can affect F2 and hence S2: *stable architecture*, which is how well defined the architecture can be at the start of development, *age of system*, which is reflected in the legacy component of F2, and *criticality*, which affects the customer and team's tolerance of risk [6]. Fairbanks proposed a non-agile-specific method using only risk to determine which architecture decisions should be made up-front [13]. Boehm and Turner also discussed up-front architecture design being used to reduce risk [38]. In an interview the lead for the NASA software architecture review board described spending more effort on areas where there are more challenging ASRs and hence more risk [39]. We explored complexity and risk and its impact on up-front effort in an earlier paper [40].

F3 (EARLY VALUE) is a trigger for S3 (EMERGENT ARCHITECTURE). Another factor suggested by Abrahamsson et al. as affecting up-front design effort was the *business model* [6], of which the customer's need for F3 is a part. We described the effects of early value in an earlier paper [40].

F4 (TEAM CULTURE) lies at the heart of agile methodologies. A team culture that is people-focused and collaborative greatly reduces the time required to relay information [41] and hence reduces the feedback cycle. A team with a highly agile culture has team members who are physically close together, are trusting and amicable, and use face-to-face communication instead of written documentation [41]. Abrahamsson et al.'s *team distribution* factor affects the team's ability to communicate [6]. The importance of team culture on agility is well recognised in the literature [41], [38], [42], as is trust [41], [43], [44].

F5 (CUSTOMER AGILITY) has similar effects to Abrahamsson et al.'s *business model* and *governance* factors [6], which both impact the customer. Small organisations are often more able to provide an agile environment, while large organisations often prefer heavy-weight processes and are non-learning [45], [46], preferring extensive planning and formal communication [41]. Hence large organisations are often not very agile.

F6 (EXPERIENCE) is important in agile methodologies [38], as it is in other methodologies [47]. Dreyfus and Dreyfus explained that as a learner moves from being a novice to becoming a master, decision-making changes from requiring analysis to being intuitive [48] which speeds up the decision-making process and hence increases agility.

Kruchten, Obbink and Stafford called frameworks *pre-cooked* architectures [1] because much the architecture is defined within the framework. Mirakhorli and Cleland-Huang also noted the benefits of *reference* architectures [39]. The use of frameworks and reference architectures, corresponding to S5 (USE FRAMEWORKS AND TEMPLATE ARCHITECTURES), make agile development more effective because they simplify the design and reduce the architectural (and development) effort required [40], [49]. This simplified design and reduced effort increases the team's ability to use S1 (RESPOND TO CHANGE), reduces F2 (TECHNICAL RISK), and hence reduces the need to use S2 (ADDRESS RISK). Cervantes, Velasco-Elizondo and Kazman also noted that the framework functionality may need to be extended if it does not provide the required functionality out of the box [50]. This need increases risk and effort, requiring increased use of S2.

VII. LIMITATIONS

Like any grounded theory study, the result is only applicable to the domain and context being studied [30], and therefore cannot be assumed to be applicable to other contexts, or in general. The result is, to some extent, dependent on the research participants selected for the research and how they described their experiences.

We took a number of steps to prevent threats to the validity of the results. The first step was to minimise bias being introduced by similar participants (architecture experts) all taking a common perspective by including a number of non-architecting participants, such as customers and business analysts. Participants from different roles in the same team or organisation were also included to help negate any personal bias. The second step was to collect data in the form of documentation to back up data obtained from interviews, which helped prevent bias being introduced through only collecting data in one form. Thirdly, feedback on the emerging results was obtained directly from participants and from conference audiences to help validate the results.

We have evaluated the full theory [21] using qualitative research criteria proposed by Lincoln and Guba [51], Miles and Huberman [52] and Creswell [53]. Criteria included trustworthiness, originality, resonance and usefulness.

VIII. CONCLUSION

The aim of agile software development is to increase the delivery of value through the ability to respond to changing requirements. To increase agility, up-front effort is reduced, so that the customer can start providing feedback earlier. Reducing the up-front design too much, however, could lead to an accidental architecture that does not support the team's ability to develop functionality and fails to meet requirements. To maximise agility, a team must find an appropriate trade-off between a full up-front architecture design and a totally emergent design. This paper presented a grounded theory of agile architecture that describes how teams determine how much architecture they design up-front. The theory includes five strategies that teams use to determine how much architecture to design up-front. The strategies are chosen according to the context of the team and the system it is building, which is described by six forces.

ACKNOWLEDGEMENTS

We are grateful to Victoria University of Wellington for providing funding for this research, and to Ewan Tempero and Rashina Hoda for their invaluable feedback on this paper.

REFERENCES

[1] P. Kruchten, H. Obbink, and J. Stafford, "The past, present, and future for software architecture," *IEEE Software*, vol. 23, no. 2, pp. 22–30, Mar.–Apr. 2006.

[2] G. Booch, "Architectural organizational patterns," *IEEE Software*, vol. 25, no. 3, pp. 18–19, May–Jun. 2008.

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

[4] L. Williams and A. Cockburn, "Agile software development: it's about feedback and change," *IEEE Computer*, vol. 36, no. 6, pp. 39–43, Jun. 2003.

[5] P. Kruchten, "Agility and architecture: an oxymoron?" *SAC 21 Workshop: Software Architecture Challenges in the 21st Century*, 2009. [Online]. Available: http://csse.usc.edu/csse/event/2009/Arch-Workshop/presentations/Kruchten

[6] P. Abrahamsson, M. A. Babar, and P. Kruchten, "Agility and architecture: Can they coexist?" *IEEE Software*, vol. 27, no. 2, pp. 16–22, Mar.–Apr. 2010.

[7] G. Booch, "The accidental architecture," *IEEE Software*, vol. 23, no. 3, pp. 9–11, May–Jun. 2006.

[8] B. Boehm, "Architecting: How much and when?" in *Making Software*, A. Oram and G. Wilson, Eds. O'Reilly, 2011.

[9] F. Buschmann and K. Henney, "Architecture and agility: Married, divorced, or just good friends?" *IEEE Software*, vol. 30, no. 2, pp. 80–82, Mar.–Apr. 2013.

[10] G. Booch, "An architectural oxymoron," *IEEE Software*, vol. 27, no. 5, p. 96, Sep.–Oct. 2010.

[11] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., ser. SEI Series in Software Engineering. Addison-Wesley, 2003.

[12] G. Booch, "The irrelevance of architecture," *IEEE Software*, vol. 24, no. 3, pp. 10–11, May–Jun. 2007.

[13] G. Fairbanks, *Just Enough Software Architecture: A Risk Driven Approach*. Marshall and Brainerd, 2010.

[14] 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.

[15] S. Freudenberg and H. Sharp, "The top 10 burning research questions from practitioners," *IEEE Software*, vol. 27, no. 5, pp. 8–9, Sep.–Oct. 2010.

[16] G. Allan, "The legitimacy of Grounded Theory," *European Conference on Research Methods (keynote address)*, July 2006.

[17] W. G. Zikmund, B. J. Babin, J. C. Carr, and M. Griffin, *Business Research Methods*, 8th ed. South-Western Cengage Learning, 2010.

[18] P. N. Stern and C. Porr, *Essentials of Accessible Grounded Theory*. Left Coast Press, 2011.

[19] B. G. Glaser and J. Holton, "Remodeling grounded theory," in *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*, vol. 5, no. 2, 2004.

[20] B. G. Glaser, *The Grounded Theory Perspective III: Theoretical Coding*. Sociology Press, 2005.

[21] M. G. 'Waterman, "Reconciling agility and architecture: a theory of agile architecture," Ph.D. dissertation, Victoria University of Wellington, 2014.

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

[23] P. Eeles and P. Cripps, *The Process of Software Architecting*. Pearson Education, 2009.

[24] K. Conboy, "Agility from first principles: Reconstructing the concept of agility in information systems development," *Information Systems Research*, vol. 20, no. 3, pp. 329–354, 2009. [Online]. Available: http://isr.journal.informs.org/content/20/3/329.abstract

[25] S. Adolph, "What lessons can the agile community learn from a maverick fighter pilot?" in *Agile Conference (AGILE '06)*, 2006.

[26] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine de Gruyter, 1967.

[27] G. Allan, "A critique of using grounded theory as a research method," *Electronic Journal of Business Research Methods*, vol. 2, no. 1, July 2003. [Online]. Available: http://www.ejbrm.com/vol2/v2-i1/issue1-art1-allan.pdf

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

[29] M. Waterman, J. Noble, and G. Allan, "How much architecture? Reducing the up-front effort," in *Agile India 2012*, Feb. 2012, pp. 56–59.

[30] K. Charmaz, *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. SAGE Publications Ltd, 2006.

[31] K. Beck *et al.*, "Agile manifesto," Available at http://agilemanifesto.org/, 2001.

[32] J. Cleland-Huang, R. S. Hanmer, S. Supakkul, and M. Mirakhorli, "The twin peaks of requirements and architecture," *IEEE Software*, vol. 30, no. 2, pp. 24–29, Mar.–Apr. 2013.

[33] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sept. 1980.

[34] L. McGovern, "What is legacy code?" 2008. [Online]. Available: http://www.flickspin.com/en/software_development/what_is_legacy_code

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

[36] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., 1996.

[37] P. Kruchten, "Complexity made simple," *Proceedings of the Canadian Engineering Education Association*, 2012.

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

[39] M. Mirakhorli and J. Cleland-Huang, "Traversing the twin peaks," *IEEE Software*, vol. 30, no. 2, pp. 30–36, Mar.–Apr. 2013.

[40] M. Waterman, J. Noble, and G. Allan, "The effect of complexity and value on architecture planning in agile software development," in *Agile Processes in Software Engineering and Extreme Programming (XP '13)*. Springer, 2013.

[41] A. Cockburn and J. Highsmith, "Agile software development: the people factor," *IEEE Computer*, vol. 34, no. 11, pp. 131–133, Nov. 2001.

[42] E. Whitworth and R. Biddle, "The social nature of agile teams," in *Agile Conference (AGILE '07)*, 2007, pp. 26–36.

[43] S. Dorairaj, J. Noble, and P. Malik, "Understanding the importance of trust in distributed agile projects: A practical perspective," in *Agile Processes in Software Engineering and Extreme Programming (XP '10)*. Springer, 2010, pp. 172–177.

[44] J. A. Highsmith, *Adaptive Software Development: a Collaborative Approach to Managing Complex Systems*. Dorset House, New York, 2000.

[45] P. M. Senge, *The fifth discipline: the art and practice of the learning organization*. Doubleday/Currency, 1990.

[46] R. Hoda, J. Noble, and S. Marshall, "Agile undercover: When customers don't collaborate," in *Agile Processes in Software Engineering and Extreme Programming (XP '10)*. Springer, 2010, vol. 48, pp. 73–87.

[47] A. Cockburn, *Agile Software Development: The Cooperative Game*, 2nd ed. Addison-Wesley, 2007.

[48] S. E. Dreyfus and H. L. Dreyfus, "A five-stage model of the mental activities involved in directed skill acquisition," DTIC Document, Tech. Rep., 1980.

[49] D. Spinellis, "Agility Drivers," *IEEE Software*, vol. 28, no. 4, p. 96, July–Aug. 2011.

[50] H. Cervantes, P. Velasco-Elizondo, and R. Kazman, "A principled way to use frameworks in architecture design," *IEEE Software*, vol. 30, no. 2, pp. 46–53, Mar.–Apr. 2013.

[51] Y. S. Lincoln and E. G. Guba, "But is it rigorous? Trustworthiness and authenticity in naturalistic evaluation," *New Directions for Program Evaluation*, vol. 1986, no. 30, pp. 73–84, 1986.

[52] M. B. Miles and A. M. Huberman, *Qualitative Data Analysis: An Expanded Sourcebook*, 2nd ed. SAGE Publications, Inc., 1994.

[53] J. W. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*, 4th ed. SAGE, 2014.