Reconciling agility and architecture: a theory of agile architecture

by

Michael Grant Waterman

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy

Victoria University of Wellington
2014

Abstract

The purpose of agile software development is to enable the software development team to respond to change and learn from change so that it can better deliver value to its customer. If an agile software development team spends too much time planning and designing architecture up-front then the delivery of value to the customer is delayed or otherwise compromised, and responding to change can become extremely difficult. Not doing enough architecture design increases exposure to risk and increases the chance of failure. The balance between architecture and agility is not well understood by agile practitioners or researchers.

This thesis is based on grounded theory research involving 44 participants from 36 organisations, all working in agile software development and who are either experienced in architecture design or are closely involved with architecture. The thesis presents a theory that describes how agile software teams design an agile architecture with reduced up-front design and which is able to respond to change, helping teams find a balance between architecture and agility.

The theory describes six forces that affect the agility of the architecture and up-front design, and five strategies that teams use in response to those forces to determine how much effort they put into up-front design.

Understanding these forces and strategies helps agile teams to determine how much up-front design is appropriate in their contexts.

Acknowledgments

I would like to thank Professor James Noble and Dr. George Allan for their guidance in helping me complete this learning experience; my participants, on whose contribution this thesis is based; and Jo, Isaac and Celeste, who shared the journey with me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Software development is not a relay sport" (Simon Brown [57])

This thesis presents a theory of agile architecture that describes how teams design an architecture that is modifiable and tolerant of change, and explains how teams determine how much architecture they design up-front.

1.1 The balance between architecture and agility

The purpose of agile software development is to better deliver value to the customer by being able to manage change in the environment – particularly requirements – during development.

Agile methods use a feedback loop in which the customer provides feedback to the team about changed or new requirements, and the development team responds to that change by adapting the system. The more agile the team, the shorter the feedback loop: the quicker the customer can provide feedback, and the quicker the team can respond to that feedback. Conversely, a less agile team has a longer feedback loop.

The values and principles of the Agile Manifesto discourage excessive planning because planning increases the time before the customer can start providing feedback, and makes it more likely the team will need to make changes to the plans when requirements change.

Software architecture is the set of design decisions that the expert developers of the development team need to understand, and are difficult to change during development. Architecture is important for satisfying the architecturally significant requirements (ASRs), consisting mainly of the quality attributes such as performance, security and scalability, and is particularly important when there is a small solution space, difficult requirements or a high risk of failure. Because they are difficult to change, architectural decisions are made early, typically before the development of the functional requirements begins.

To increase agility, the planning time – including up-front architecture design – is reduced, so that the customer can start providing feedback earlier. This leads to a more emergent and implicit architecture, but reducing the amount of architecture design too far could lead to an accidental architecture which fails to meet ASRs, and requires more time to fix problems. If the team spends too much time fixing problems then its ability to respond to change and hence its agility is reduced. Thus, less architecture design does not necessarily mean more agility.

To maximise agility, a team must find a suitable balance or trade-off between a full up-front architecture design and a totally emergent design.

1.2 The research problem

The trade-off between architecture and agility is not well understood. In practice, there are a number of schemes that specify up-front architecture design in agile development, such as extreme programming's system metaphor [26], iteration zero [153], the research 'spike'[181], Agile Modeling [18], risk-based design [91] and Real Options [32]. These schemes generally offer little guidance as to which decisions should be made up-front.

There has been very little empirical research on the relationship between software architecture and agile development to date, yet it has become an important issue.

The study presented in this thesis helps to fill this research gap. The research used grounded theory, a qualitative research strategy that generates theory inductively, to explore and explain how practitioners treat the relationship between architecture and agile in practice. The result of the research is a theory, 'a theory of agile architecture,' which explains how much effort agile software teams put into architecture design, with a focus on how teams determine which architecture decisions to make up-front and what influences those decisions.

1.3 Structure of thesis

The structure of this thesis is as follows:

Chapter 1: Introduction has provided a rationale for the research and this introduction to the thesis.

Chapter 2: Background provides a background to the problem. The chapter introduces software engineering and the reason for the advancement of agile software development methodologies. The chapter then describes agile methodologies and what it means to be agile, and describes software architecture and why architecture is important. The chapter then discusses architecture in agile development: how architecture and agile affect each other and what the research gap is. The chapter closes by presenting the research objective.

Chapter 3: Research design describes the research design. It presents the research approach, and discusses a number of potential research strategies. It then describes the selected research strategy, grounded theory, and the procedures used to implement it in this research.

Chapter 4: Data collection and analysis describes how the data was collected and analysed using the grounded theory methodology. The chapter describes how the theory was developed, providing a link between the description of the research design in Chapter 3 and the presentation of the findings in Chapter 5.

Chapter 5: A theory of agile architecture introduces the findings of the research: a theory of agile architecture. It gives an overview of each of the main parts of the theory: the forces, which make up the team's and system's context; the strategies, which teams choose in response to the context; and the agile architect.

Chapter 6: The agile architecture forces is the second findings chapter. It discusses in more detail the forces that make up the project's context. The forces affect the strategies that a team chooses when designing their agile architecture, described in the next chapter.

Chapter 7: The agile architecture strategies is the third findings chapter. It discusses in more detail the strategies that a team uses to help design an architecture that is able to respond to change and to determine how much architecture to design up-front.

Chapter 8: The agile architect is the fourth and final findings chapter. It discusses who makes architecture decisions in an agile project, and what other roles the architects have in an agile team.

Chapter 9: Discussion discusses the theory of agile architecture from an external perspective. It reflects on the presentation of the theory, its boundaries and on how to evaluate the theory. It then discusses the theory in the context of the literature.

Chapter 10: Conclusion The final chapter concludes this research. It outlines the thesis's contributions and implications for practice, discusses the theory's limitations and suggestions for future work. It closes with a conclusion to the thesis.

Chapter 2

Background

"The hardest thing is what does architecture mean in agile? Agile architecture, it's almost an oxymoron." (P35, design engineer)

This chapter provides a background to agile development and software architecture, and presents the research objective. This research uses grounded theory; the justification for grounded theory is provided in Chapter 3. Significantly, grounded theory does not have a major literature review before the research begins [104]. Instead, an initial *minor literature review* provides the background to the research. The minor literature review should be sufficient for the researcher to understand the research problem and the knowledge gap, but not enough to give the researcher preconceived ideas or theories about the solution to the problem being investigated. The *major literature review* starts in the later stages of the research, and relates the findings to the literature. Both the minor and the major literature reviews contribute to this chapter, and provide a background to the research problem. Section 3.3.6 explains the role of the literature review in more detail.

In this chapter, section 2.1 introduces software engineering and the problems that software engineers had with traditional plan-driven methodologies that led to the advancement of agile software development methodologies. Agile software development is described and defined in section 2.2.

Software architecture is introduced in section 2.3, and the balance needed between architecture and agility is described in section 2.4.

The chapter concludes with the research objective, presented in section 2.5.

2.1 Early software engineering and the empirical process control model

Software engineering is a relatively recent engineering discipline that originated at two NATO software engineering conferences held in the late 1960s [157, 172]. At these conferences the term *software engineering* was used to imply a need for software development to have theoretical foundation and practical disciplines like those found in other branches of engineering [172].

The earliest methodologies to emerge were modelled on traditional engineering disciplines, which were often based on *defined process control models* [199, 230]. In a defined process, every part of the work is completely understood and the design completed in advance of manufacturing. The product can be built sequentially using pre-defined phases and standardised tasks, and if necessary can use a *Fordist*-style production system [193] to produce identical products multiple times [230]. To manage the project's risks, the designers attempt to anticipate specification variation and potential problems, and thus require large quantities of documentation to be handed over to the development team (often referred to being "thrown over the wall," alluding to the one-way relationship between the designers and the developers) for implementation. In software, these defined processes are the traditional plan-driven *big design up-front* (BDUF) methodologies such as the unmodified waterfall method [195] and formal processes aligned with the CMM (capability maturity model)/CMMI (capability maturity model integration) [67]. The postmodern software development viewpoint describes these early software development methods as having modernist characteristics [193] because their structured development methods assumed everything can be clearly understood in advance. The third issue of the *IEEE Software* magazine (1984) [124] features the photo of a car assembly line on its cover, shown in Figure 2.1, consistent with a common view of software development at the time.

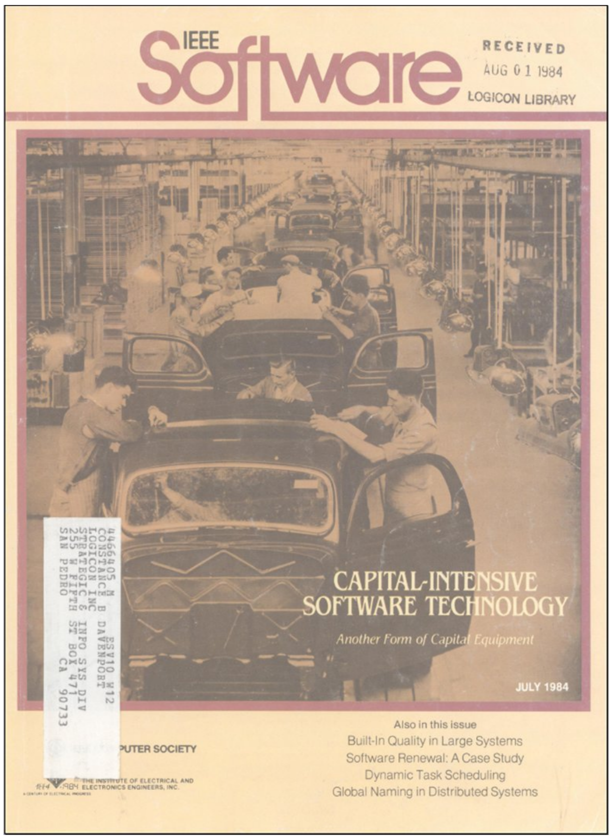
IEEE Software
RECEIVED
AUG 01 1984
LOGICON LIBRARY
CAPITAL-INTENSIVE SOFTWARE TECHNOLOGY
Another Form of Capital Equipment
JULY 1984
Also in this issue
Built-In Quality in Large Systems
Software Renewal: A Case Study
Dynamic Task Scheduling
Global Naming in Distributed Systems
PUTER SOCIETY
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

Figure 2.1: Cover of *IEEE Software 1*, 3

More recently software engineers have recognised that often software cannot be thought of as a defined process because the specifications often cannot be fully understood in advance; development is frequently subject to change both in requirements and in the implementation technology [230]. Parnas and Clements (1996) summed up this realisation:

- "In most cases the people who commission the building of a software system do not know exactly what they want and are unable to tell us what they know.
- "Even if we knew the requirements [...] many of the details only become known to us as we progress in the implementation [...]
- "Even if we knew all of the relevant facts before we started, experience shows that human beings are unable to comprehend fully the plethora of details that must be taken into account in order to design and build a correct system [...]
- "Even if we could master all of the detail needed, all but the most trivial projects are subject to change for external reasons" [180].

Rather than a defined process, software development is often more suited to an *empirical process control model* [199, 230] which requires interaction between design and manufacturing through frequent inspection, feedback and adaptation.

This recognition of software development as an empirical process has led to the advancement of *agile development methodologies*. Agile development methodologies closely involve the customer, with the specifications being developed simultaneously with the product. Agile development methodologies are able to respond to rapidly changing requirements, and purportedly require less effort, produce software of higher quality, and improve customer satisfaction [41]. Beedle (2011) believed that in the future agile development methodologies will simply be called 'software development' [29] – that is, will become the prevailing development methodologies.

2.2 Agile development

Agile development methods are based on a philosophy and a set of principles that allow the development team to manage changing and unknown requirements. Agile methods frequently use an *iterative and incremental development* (IID) approach [31] where development takes place iteratively and incrementally in parallel with requirements gathering. To manage change, agile methods are light on defined processes [2] and discourage detailed planning. Rather, only a small amount of planning should be included, with most of the architecture design emerging during development [27]. Agile methods attempt to improve customer satisfaction through high levels of *customer involvement*, improve the quality of code through practices such as *test-driven development* and improve productivity through *self-organising* and motivated teams [41].

This section describes the Agile Manifesto, which defines the agile philosophy (section 2.2.1), lists some of the main agile methodologies (section 2.2.2) and defines agility (section 2.2.3). The section then discusses the agile feedback cycle (section 2.2.4), the agile characteristic of responding to change (section 2.2.5) and finally learning from change (section 2.2.6).

2.2.1 The Agile Manifesto

The Agile Manifesto (2001) [28] introduced the software world to the term *agile*, bringing what were then known as lightweight methodologies together under one umbrella with a common philosophy. The Manifesto does not define any methodologies or practices itself, but rather outlines a philosophy in the form of a set of values and principles that methodologies and agile software engineers adhere to.

The agile values

The introduction to the Agile Manifesto lists four values:

> “We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - “Individuals and interactions over processes and tools
> - “Working software over comprehensive documentation
> - “Customer collaboration over contract negotiation
> - “Responding to change over following a plan.
>
> “That is, while there is value in the items on the right, we value the items on the left more.” [28]

The items on the right – processes and tools, comprehensive documentation, contract negotiation, and following a plan – are all characteristics of traditional defined process methodologies that are based on knowing the requirements in advance, so that a development team’s architects could design the system in full before handing over to the team’s developers. In contrast, the items on the left – individuals and interactions, working software, customer collaboration and responding to change – emphasise communication, collaboration, flexibility, and – perhaps most importantly – working software, the end product.

It is worth emphasising the philosophy does not put *no* value in the items on the right; it simply puts more value in the items on the left. This is a misconception with agile methodologies [166].

The agile principles

In addition to the four values, the Agile Manifesto lists twelve principles:

> “We follow these principles:
>
> - “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

- “Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.
- “Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
- “Business people and developers must work together daily throughout the project.
- “Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- “The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- “Working software is the primary measure of progress.
- “Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- “Continuous attention to technical excellence and good design enhances agility.
- “Simplicity – the art of maximizing the amount of work not done – is essential.
- “The best architectures, requirements, and designs emerge from self-organizing teams.
- “At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.” [28]

The main themes of these principles are the value of working software, the ability to cope with changing requirements, the importance the team members and the customer collaborating and communicating, and the importance of the quality of the work performed. There should be no big design up-front, because a system designed in detail makes it harder to deliver software early (contravening the first principle), welcome change (contravening the second principle), and risks doing work that will not be required when requirements change (contravening the tenth principle). Architects should not be separate from the team: the architecture should be designed by the development team itself (in line with the eleventh principle), because the team best understands how the architecture should be implemented.

2.2.2 Mainstream agile methodologies

The Agile Manifesto itself does not define any practices or methodologies. Rather, there are a number of mainstream agile development methodologies that implement the agile practices, with differing degrees of agility [41]. The most popular mainstream agile methods (sometimes called 'commercial' methodologies [3, 74]) include *Scrum* [199], *extreme programming* (XP) [26] and *lean development* [150].

Generally each methodology has its own focus with its own set of practices and principles; some overlap but others do not. Teams sometimes mix and match the practices they choose to follow – for example, the technical aspects of XP are often used with the management aspects of Scrum (for example [186]), and pair programming is often dropped in XP development [156].

The most popular agile methodologies and their key practices are outlined in the following sections.

Scrum

Scrum is a management methodology that deals with how the development is managed, such as using self-organising teams and the management of requirements [88, 159, 226]. Scrum does not define the engineering processes that the developers use to actually build the software.

To allow for changes in requirements, development is structured into fixed length iterations called *sprints*. Sprints last no longer than one month. Each sprint should produce code that is *potentially shippable* – tested and integrated. At the end of each sprint, the team holds a *sprint review* in which the team reviews the product and demonstrates it to the customer. The team also holds a *sprint retrospective* in which the team reviews, and if necessary adapts, its process and environment.

The requirements are kept in a prioritised list called a *product backlog*, and those to be implemented are chosen by the team on a sprint-by-sprint basis – the *sprint backlog*. Requirements cannot change during a sprint; any changes must wait for the next sprint. The *product owner*, a customer representative on the team, is responsible for keeping the backlog up to date. Choosing the requirements to be developed on a sprint-by-sprint basis, and taking into account customer feedback and suggestions, allows the development to *inspect and adapt* to changing requirements.

The team is *self-organising*: it manages itself with a very high degree of autonomy and accountability. Every day the team examines its progress in a *scrum* (or *stand up meeting*) and reviews its progress. The team itself decides what it will commit to, and decides how to achieve that commitment. There is no specific mention of architecture in Scrum, although many teams include a *sprint zero* phase at the start of development, which is dedicated to defining a high-level design and does not produce deliverable software.

The canonical Scrum guide is considered to be Schwaber and Beedle (2002) [199], although there are a number of guides that summarise Scrum's rules, such as [83] and [200].

Figure 2.2 summarises the key steps and definitions in Scrum.

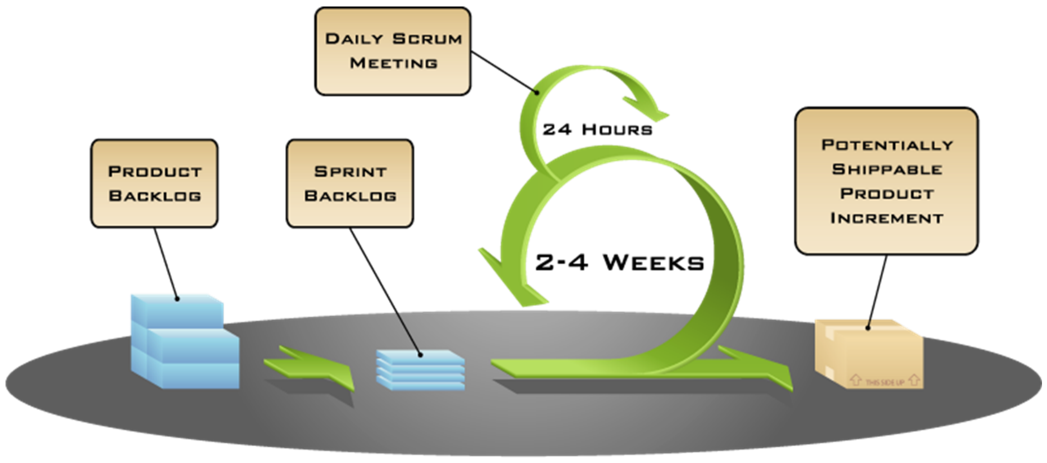

Figure 2.2: The Scrum process. Figure ©Mountain Goat Software; used under creative commons licence.

Extreme programming

Extreme programming (XP) is an agile methodology that is based on five *values*: simplicity, communication, feedback, courage and respect [27]. These values are fairly abstract, and are supported by *practices* that give specific guidance on how to use XP [27].

The practices are the things that the team will be doing each day [27], and are given purpose by the values. A number of the practices relate to the management of the project, and are very similar to the practices of Scrum, such as small regular releases, there being a customer representative remaining on-site during development, self-organising teams and requirements being defined in advance of each iteration. There are also a number of technical practices, such as *pair programming* – two developers per keyboard and screen; *test driven development* – every unit of development must pass a pre-defined test before it can be accepted; and *constant refactoring* the code design – to prevent the quality of the code degenerating through an evolving design [27]. *Continuous integration* means the team merges development streams and produces a working copy of the software several times a day.

The original XP book by Beck (2000) contained a *system metaphor* which "helps everyone on the project understand the basic elements and their relationships" [26] and represents the early architecture design.

An emergent architecture design (where architecture design takes place during development) coupled with *refactoring* (changing the system to improve its structure without changing its external behaviour [95]) is preferable to up-front design wherever possible. Up-front design is only allowed if required to ensure success of the software. Beck (2005) described refactoring as "a sensible, low-risk way to develop software" [27], even though some consider refactoring to potentially increase cost [36] (at least up to a certain point [69]) and create unnecessary overhead [210]. Refactoring (including refactoring at the architectural level, when required) ensures the system always meets its requirements at that point in time, including the quality requirements.

Lean software development

Lean software development is a software version of lean production that was originated by Toyota as the Toyota Production System ("the Toyota Way") in the mid-1900s [79]. Poppendieck and Poppendieck (2003) [184] brought lean to the software world and is considered the definitive lean software development guide.

Many of the lean development principles follow the agile philosophy laid out in the Agile Manifesto. For example, lean emphasises continuous process improvement through the reduction of waste and bottlenecks (the agile principle of simplicity), delaying decisions until the last minute (to ensure as much is known about the inputs as possible when the decision is made) and increasing speed of delivery (to give the team more opportunity to delay decisions) [184]. Coplien and Bjørnvig (2010) highlighted a number of differences between lean and other agile methodologies [79], which are presented in Table 2.1, although some of these differences, such as bringing decisions forward, are at odds with Poppendieck and Poppendieck (2003) [184], and appear to be derived from a comparison with lean production rather than lean software development. Ballard (2010) described lean production as making decisions at the 'last responsible moment'; the last responsible moment requires careful planning so that the lead-time to that decision can be properly understood [24].

Lean	**Agile**
Thinking and doing	Do-inspect-adapt
Feed-forward and feedback (design for change and respond to change)	Feedback (react to change)
High throughput	Low latency
Planning and responding	Reacting
Focus on process	Focus on people
Teams (working as a unit)	Individuals (and interactions)
Complicated systems	Complex systems
Embrace standards	Inspect and adapt
Rework in design adds value, in making is waste	Minimize up-front work of any kind and rework code to get quality
Bring decisions forward (decision structure matrices)	Defer decisions (to the last responsible moment)

Table 2.1: Differences between lean development and the agile philosophy, from Coplien and Bjørnvig (2010) [79].

Other agile methods

Other agile development methods include:

Crystal Light, a family of methods developed by Manifesto signatory Alistair Cockburn [69]. Crystal focuses on people, and has different versions for different team sizes (named after colours: Crystal Clear, Crystal Yellow, Crystal Orange, Red, Magenta, Blue and so on). There are also variations that take into account system criticality and project priorities.

Adaptive Software Development (ASD), which focuses mainly on the problems of developing complex, large systems [2, 116]. ASD encourages incremental and iterative development with constant prototyping and a focus on components rather than tasks. The ASD process involves cycles of three steps: *speculate*, which involves planning for the cycle, *collaborate*, the development phase, in which there may be a number of tasks under concurrent development, and *learn*, which involves quality-reviews in which the functionality to date is reviewed with the customer.

Feature driven development (FDD) is a method that focuses on the design and building phases [2, 178]. As the name suggests, planning is driven by the features to be included. FDD follows these five steps: develop an overall model, write the features list, plan by feature, design by feature and build by feature [2]. The last two steps are iterative, with the functionality developed in each iteration being reviewed by the users and sponsors of the system for validity and completeness [2].

Kanban is related to lean and just-in-time production [21]. Kanban does not prescribe a large methodology; it revolves around reducing the amount of work in progress through limiting the number of tasks in the 'to do' list and the 'in progress' list. Like lean, Kanban aims to make processes more efficient. The Kanban philosophy includes making decisions at the last moment, strict prioritisation and short lead times.

2.2.3 Defining agility

This section discusses a number of definitions of agility, and presents a definition adopted in this thesis.

There is little agreement on how to define agility or on what form a definition should take. Some definitions disagree with each other [74, 189], and others are vague. Two possible ways to define agility are listed by Kruchten (2007) [140]: by enumeration (as one of a set of defined methodologies, such as XP, Scrum, and feature-driven development), or by some predicate (such as the degree of adherence to the Manifesto). Attempts at defining agility by practitioners and consultants often take an empirical perspective based on experiences in a particular context [146] rather than taking a theoretical or conceptual perspective [74]. Different types of definitions follow.

Agility as adherence to the Agile Manifesto

A simple way to define agility is by adherence to the values and principles laid out in the Agile Manifesto. Authors who defined agility this way include Coplien and Bjørnvig (2010) [79]:

> "By Agile, we mean the values held up by the Agile Manifesto."

– and the practitioners Caneiro and Barazi (2010) [62]:

> "Today, the Agile Manifesto is widely regarded as the canonical definition of agile development."

Lindvall et al. included adherence to the Manifesto as part of their definition [155]:

> "Agile methods are: iterative, incremental, self-organising, emergent. All Agile methods follow the four values and twelve principles of the Agile Manifesto."

Adherence to the Manifesto is not a suitable definition for agility. Many consider the Manifesto's principles vague [15, 76, 112], and repetitive [112]. Séguin, Tremblay and Bagane (2012) determined that seven of the twelve principles did not qualify as software engineering principles [202]; one of the five that did qualify only did so after being reworded. Conboy and Fitzgerald (2004) noted the Manifesto has no basis in management theory [76].

Agility as a set of methodologies

Agility is sometimes defined as the set of methodologies that claim to be agile, such as Scrum, XP, lean development, Crystal and feature driven development. Poppendieck and Poppendieck (2003) [184] and Jacobs (2006) [128] used such definitions.

Defining agility as a set of methodologies is not satisfactory because not only do agile methodologies vary greatly in values and practices [221], but some define engineering practices (such as XP), some define management practices (such as Scrum), and some define philosophical principles (such as lean development) [74]. Nor does this definition provide guidance on how to determine if a new methodology is agile.

Agility as a set of practices

Another definition of agility specifies certain practices or methods [155, 228], such as having iterations, daily meetings and frequent customer demonstrations. According to these definitions, if a development team is not following the specified practices then it is not agile [70, 162]. Some have devised tests or metrics for agility based on adherence to these practices, such as the 'Nokia Test' [215], which tests for a team's adherence to the Scrum practices, or the 4-DAT test [188], which tests a methodology's adherence to a particular set of practices and values.

Agile as a set of practices is not a good definition, because simply performing a set of practices does not necessarily make a team or methodology agile – one can use agile practices and still not be agile [75, 146, 162, 206]. In particular, teams need to be aware that agile methodologies must be customised to a team and its context [206]: “one size fits nobody” [5], “one size fits one” [72] and “context is key” [140]. Cockburn and Highsmith (2001) wrote that “every process must be selected, tailored, and adapted to the individuals on a particular project team” to make the most of each individual’s and each team’s strengths [72]. Indeed, adapting the process is a central feature of Cockburn’s Crystal methodologies [69].

Like the previous definition, this definition does not provide guidance on how to determine if new practices are agile or not.

Agility as a general concept

Defining agility as a general concept considers the purpose of agility, rather than focusing on operational matters such as methodologies and practices [74]. The purpose of software, like any product or service, is to provide value in some form to the customer; delivering value is the reason for the production of the software, and agile is a means to that end. The first Agile Manifesto principle recognises this: “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software” [28]. In an unpredictable postmodern world, determining how to maximise value (or how to deliver value the most efficiently) cannot be determined in advance, so software development – specifically, agile software development – benefits from an empirical process control model (section 2.1). An empirical process allows a development team to respond to the changing environment [230] by involving the customer in the development process. Managing change is one of the core values of the Agile Manifesto: “[We have come to value] responding to change over following a plan” [28].

Ultimately, the purpose of agile software development’s values and principles (such as those defined in the Manifesto) and practices (such as those defined in specific agile methodologies) is to support this need to respond to change to increase the delivery of customer value.

Various Manifesto authors have highlighted the importance of responding to change. Highsmith and Cockburn (2001) wrote that the ultimate purpose of agility was "creating and responding to change" [118], and explained that teams cannot be agile if they have long iteration cycles (long periods between customer feedback). *Creating* change means that agility encourages the customer to continually redefine the product so that it provides them with more value – a process that is contrary to traditional plan-driven development methods. *Responding* to change means that the product being developed is changed to take into account changing requirements. The title of Williams and Cockburn's paper, *Agile software development: it's about feedback and change* [230], echoed this purpose. "Embrace change" is also part of the title of Kent Beck's original XP book [26].

A number of authors defined agility based on the need to adapt to change [6, 74, 117, 133, 146]. For example, Highsmith (2002) defined agility as:

> "Agility is the ability to both create and respond to change in order to profit in a turbulent business environment." [117]

Highsmith also noted that agility is only an advantage when comparing competitors – "a copper mining company doesn't need to be as agile as a biotechnology firm" – and that an agile organisation must balance structure and flexibility – "if everything changes all the time, forward motion becomes problematic." [117]

Conboy (2009) derived a similar but more extensive definition of agility from first principles, considering the definitions of agility, flexibility and leanness from other disciplines:

> "[Agility is] the continual readiness of an ISD [information systems development] method to rapidly or inherently create change, proactively or reactively embrace change, and learn from change while contributing to perceived customer value

> (economy, quality, and simplicity), through its collective components and relationships with its environment.” [74]

As well as creating change and embracing (or responding to) change, this definition includes *learning* from change so that a team adapts its process to better manage change, to try to maximise the rate at which it delivers customer value. In other words, improve both the product and the process.

Adolph (2006) used the ‘maverick fighter pilot’ tale of US Air Force Colonel John Boyd to define agility:

> “A project is agile if it is able to execute its reorienting and action-taking cycle faster than the changes occurring in its environment” [6].

Kruchten favoured this definition [146].

These general definitions of agility address the weakness of plan-driven methodologies in unstable environments: because plan-driven methodologies are unable to welcome change during development, teams are unable to provide full value to their customers.

This thesis uses a definition of agility that is derived from these general concept definitions and reflects the benefit that empirical process development methods provide:

> DEFINITION OF AGILITY
>
> 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.

The most important part of this definition is the ability to respond to change. Creating change is pointless if the team cannot then respond to that change, and learning from change means improving the ability of the team to respond to change.

Lean development is considered agile by this definition. Not all authors include lean as an agile methodology: Coplien and Bjørnvig (2010) [79], for example, outlined what they considered the differences were (listed in Table 2.1).

This definition of agility also means that software engineers do not necessarily have to use one of the recognised agile methodologies (section 2.2.2) to be agile. Indeed, software engineers and teams who have never even heard of the term 'agile' can be agile if they satisfy this definition.

2.2.4 The agile feedback cycle

Responding to change requires agile development methodologies to have a feedback loop in which the customer provides feedback to the development team about any required changes to the product, and then addresses those required changes. A shorter period means a team is more agile; a longer period means less agility – Highsmith and Cockburn (2001) noted that teams cannot be agile if they have long iteration cycles [118], and the central theme of Ries's *Lean Startup* book is reducing the length of the feedback cycle [192]:

> "We need to focus our energies on minimising the total time through this [build – measure – learn] feedback loop." [192]

Lean development also stresses the importance of many short feedback (learning) cycles [150, 184].

Boehm's well-known studies show that the cost of fixing a defect rises exponentially the longer the period until detection [34]; Ambler used this to demonstrate the effect that detecting requirements defects (that is, responding to requirements that change or are otherwise not what there were initially thought to be) sooner has on cost [12], using the model shown in Figure 2.3. In this figure we can compare the extremes of customer feedback: the cost of finding requirements defects when using active stakeholder participation (frequent feedback) in the bottom left is orders of magnitude lower than the cost of finding requirements defects when using traditional acceptance testing (no feedback until development has been completed), top right.

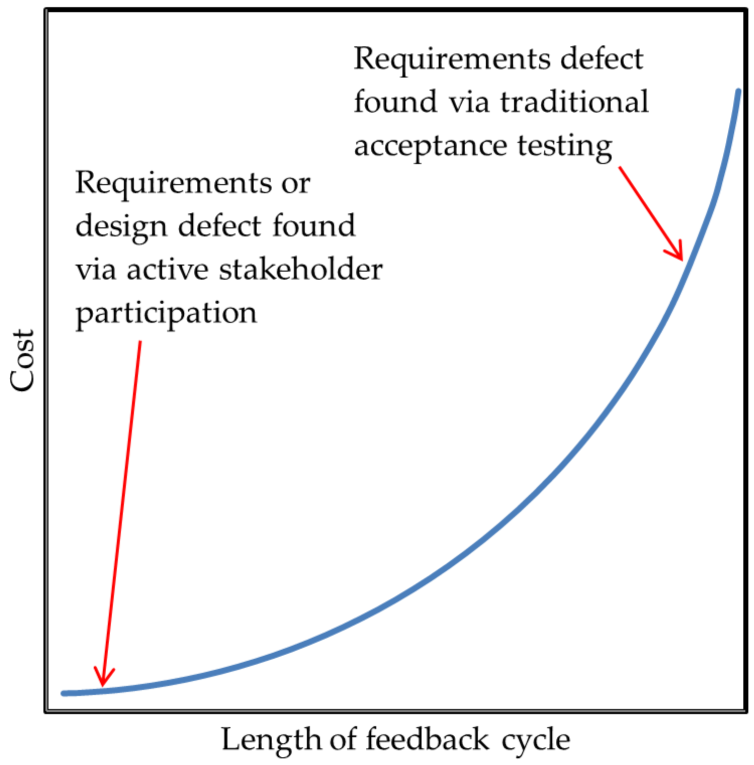

Figure 2.3: The effect of the feedback cycle on the cost of finding defects, after Ambler [12]

With iteration-based agile methodologies such as XP and Scrum, the period of the feedback loop is the iteration length, with acceptance testing at the end of each XP iteration [27] and sprint reviews at the end of each Scrum sprint [199]. XP suggests a one- to three-week iteration [27], while Scrum prefers one to four weeks [83, 201]. In lean development, acceptance tests are run throughout development [184].

It is not enough to simply increase the frequency of feedback and make more emergent decisions to become more agile: the team must also be able to effectively respond to change, as described below.

2.2.5 Responding to change

A team can more effectively respond to change if it can make decisions more rapidly and reduce the elapsed time between making a decision and seeing the result of that decision [72].

Many of the practices, principles and values of the agile methodologies support the teams in reducing this cost and time. For example, these activities or features reduce the feedback cycle time:

- *Continuous integration* [27] allows the team to immediately see how each developer's work affects the application, reduces the time required to find flaws, and gives the opportunity for more frequent customer feedback.
- *A customer representative being a part of the team* [200] allows collaboration while defining and prioritising requirements, and the opportunity for more frequent demonstrations.
- *Test driven development* [27] allows testing to be completed faster, find problems earlier and hence provide the opportunity for earlier customer demonstrations and earlier feedback.

These activities or features improve the team's ability to respond to change:

- *Minimising documentation and simplicity* [28] helps the team to better respond to change by reducing the unnecessary non-code 'baggage.'
- *Simplicity is essential* [28] means simpler code and less waste, making the system easier to change.
- *Collective ownership of code* [27] means better code quality and better knowledge of the system; anyone can make changes without having to wait for the single owner of the code to become available.

These activities or features support the more efficient delivery of value:

- *Prioritise requirements* [27, 199] to ensure the most valuable requirements are implemented first.

- *Eliminate waste* [79, 184] to ensure that the team works as efficiently as possible.

- A *people-focused, collaborative culture* [72] to reduce the feedback cycle by moving people physically closer together, replacing written documentation with face-to-face communication, improving the team's 'amicability' and shortening the time required to relay information.

While these practices and activities increase agility, it should also be noted that reducing the feedback cycle can have a cost: where a team sits on the agility scale is a trade-off that depends on how much they can benefit from this cost. For example, pair programming is an important practice of XP [27], but it has the cost of two developers sitting at one keyboard. An effective agile team will reap the benefit of this additional cost, but an ineffective team may have the personnel department frowning at the wasted costs. Revisiting Adolph's and Highsmith's definitions of agility [6, 117] in section 2.2.3, it is sufficient for an agile team to respond to change faster than the environment can change: in a slow-changing environment, being highly agile is unnecessary.

2.2.6 Learning from change

Part of the definition of agility is learning how to better respond to change: *adapting* the process or methodology so that the team can respond to change faster, and hence deliver value more efficiently. To adapt, teams must become learning organisations [203] that can judge their environment, and change their processes to fit their context [116, 120]. This is sometimes referred to as the *learning loop* [116]. Highsmith (2000) called learning the *adaptive software development cycle*, distinguishing it from a normal feedback cycle because of its complex human interactions [116]. Most agile methodologies include the ability to adapt the process. For example, XP encourages local adaptation: "Adopting XP does not mean I get to decide how you develop" [27]. Lean development refers to adaptation as *kaizen* [150], a Japanese term meaning continuous improvement. Customising and adapting the process is a core element of the Crystal family of methodologies [69], called *on-the-fly tuning*. Even Scrum, which has a very strict definition of what can be called Scrum, requires developers to *inspect and adapt* [83, 216] their process through the use of *sprint retrospectives* (as distinct from the *sprint review* which is used to inspect and adapt the product), as long as any changes to the process are contained within the Scrum framework [201]. (Scrum co-inventor Ken Schwaber recognised that the Scrum framework could limit a team's agility and proposed the name 'Scrum and' to signify versions of Scrum adapted beyond the defined Scrum framework [198].)

2.3 Software architecture

Software design is "the activity of determining what the parts of some larger whole should be, and how those parts will fit together" [223]. Most authors describe two levels of design in a system: the *architecture* – the whole-system design decisions – and the *design* [91, 205] – the decisions about the algorithms and data structures. The distinction between architecture and design is not always clear. Reeves (1992) described the levels of design as overlapping [190] , and Hollander (2010) described it as a continuum:

> "It is futile to try to distinguish between architecture and design! It's a continuum, the coarser being architecture, the fine-grained being design." [123]

This section defines software architecture and describes some of its characteristics.

2.3.1 Defining software architecture

There are many definitions of software architecture [208], none of which is universally accepted [97]. Definitions vary according to their focus, from those that have a technical focus, to those that consider the value of the decisions made, and to those that take a practical perspective by defining architecture according to its impact on the development team. Examples of each type are discussed below.

Technical definitions

Technical definitions define software architecture by specifying an architecture as the high-level design of the system being built. These definitions usually mention the software system's high-level concepts or structure and include the specification of its elements or components, and how those elements communicate or interact. One such definition is the popular (but recently superceded) ANSI/IEEE Std 1471:2000 definition:

> "[Software architecture is] the fundamental organisation of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." [126]

There are many similar and related definitions [25, 138, 205, 207]. These definitions are independent of the designers and development team, and have the disadvantage of potentially meaning different things to different people according to their context [61]. Fowler (2003), quoting colleague Ralph Johnson, even described these definitions as 'bogus' because "there is no highest level concept of a system. Customers have a different concept than developers. Customers do not care at all about the structure of significant components." [97]

Ambiguity may be introduced because larger systems are built from smaller subsystems [48, 77], each of which have their own fundamental organisation or high level structure.

The 1471 definition has been superceded by the recent ANSI/IEC/IEEE Std 42010:2011 definition:

> "[Software architecture is] the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution." [127]

– which addresses a number of concerns about the 1471 definition. For example, the term 'concept' introduced in the new definition implies an architecture requires human understanding.

Value-based definitions

Another type of definition of architecture takes a value-based approach. Booch (2008) referred to the *cost* of making changes to design decisions:

> "A system's architecture is defined by its significant design decisions, where in my experience, 'significant' is measured by the cost of change." [46]

Architecture decisions are 'significant' because they have wide reaching effects, and correcting defects requires large amounts of effort [1, 175]. This type of definition implies that architecture decisions should be made as early as possible and that changes to architecture should be avoided or minimised.

Fowler (2003) quoted a similar definition:

> "Architecture is the set of design decisions that must be made early in a project." [97]

– and Brown (2013) summarised architecture decisions as those decisions that cannot be reversed without considerable effort:

> "...they're the things that you can't easily refactor in an afternoon." [57]

The extent of the cost of such architectural change has been quantified by Boehm (2002):

> "For very large systems, our Pareto analysis of rework costs [...] indicated that the 20 percent of the problems causing 80 percent of the rework came largely from "architecture-breakers," such as architecture discontinuities to accommodate performance, fault-tolerance, or security problems, in which no amount of refactoring could put Humpty Dumpty back together again." [36]

The cost of change is an important consideration in agile software development, because agile encourages change (section 2.2.3).

Practical definitions

Definitions with a more practical focus define architecture by its relationship with the team rather than its role in the system. Fowler, again quoting Johnson, provided a definition that gives practical guidance to the development team by describing architecture as the senior team members' shared understanding of the system design:

> "In most successful software projects, the expert developers working on that project have a shared understanding of the system design. This shared understanding is called 'architecture.' This understanding includes how the system is divided into components and how the components interact through interfaces." [97]

Their definition also refers to the level of detail that is important to the system being built:

> "These components are usually composed of smaller components, but the architecture only includes the components and interfaces that are understood by all the developers." [97]

In other words,

> "architecture is about the important stuff. Whatever that is." [97]

Kruchten, Obbink and Stafford (2006) presented a similar definition that refers to the properties that are best considered at the system level:

> "[Software architecture involves:]
>
> - "the structure and organization by which modern system components and subsystems interact to form systems, and
> - "the properties of systems that can best be designed and analyzed at the system level." [148]

These definitions can include architectural decisions that are, by nature, low-level and embedded within the code [91], rather than being purely high-level. They can also include decisions such as the programming language which would not be considered architectural using a technical definition [97].

Boehm (2011) included *all* planning activities in his definition of architecture:

> " 'Architecting' refers to [...] the overall set of concurrent front-end activities (site surveys, operations analysis, needs and opportunities analysis, economic analysis, requirements and architecture definition, planning and scheduling, verifying and validating feasibility evidence) that are key to creating and sustaining a successful building or software project." [37]

– although this research does not consider these other activities.

For the purpose of this research, a definition that considers the economic impact of architecture and the impact of the architecture on the team (and vice versa) is most useful, because the cost of change and the role of the development team are highly pertinent to agile software development. This research is not investigating specific architecture decisions or the technical aspects of architecture, and therefore does not require a definition that takes a technical perspective.

The definition of software architecture used in this thesis therefore considers both the impact of change and recognises the role of the team:

DEFINITION OF SOFTWARE ARCHITECTURE

A software architecture is the set of design decisions of a software system that the expert developers of the development team need to understand, and are difficult (and hence expensive) to change during development.

2.3.2 Architecture artefacts

An architecture is a set of *decisions*, rather than a tangible product:

> "[An architecture] is in the human mind. An architecture may exist without ever being written down." [127]

Artefacts may be used to record or express those decisions, whether they are in textual form, are hand-sketched, use a formal architecture description language such as UML, or use some combination of those forms.

Artefacts are the physical output of the architectural design process: they may include the description of the documentation of the architecture and the decision-making process that lead to the architecture [50]. This may include models of the system to be built and documentation such as the *software architecture document* (SAD):

Software architecture models

Architecture models are abstract representations of the system, and can range from the informal use of paper, cards or whiteboard diagrams through to a rigorous and formal process using a specialised modeling language and software. Ambler (2009) compiled an extensive list of model types [19]; one popular formal modeling language is the Unified Modeling Language (UML), a standardised modeling language originally developed by Rumbaugh, Jacobson and Booch [196], while Kruchten's 4+1 view model [137] creates four (plus one) views that represent the system from different perspectives.

Software architecture documents (SADs)

SADs provide a written record of the architecture, and may include the requirements specifications, an explanation of the drivers of the architecture (justification of the decisions made) and justification of how the architecture satisfies the drivers [50, 54]. The SAD therefore has a broader scope than an architecture model.

The *requirements specification* is a description of the behaviour of the system, and includes both functional requirements (what the system is to be built for) and quality attributes (or *architecturally significant requirements*, ASRs) [50]. ASRs are discussed further below in section 2.3.4.

The SAD may also provide guidance to the development team in the form of principles and conventions. For example the architecture may define strategies for dealing with error handling, memory management and data storage, as described by Booch (1994): "it is important to explicitly design these policies; otherwise we will see developers inventing ad hoc solutions to common problems, thus ruining our strategic architecture through software rot." [43]

Bosch (2000) gave three other uses for an architecture artefact [50]:

- early *assessment of the quality attributes*: the artefact allows the design to be assessed to determine whether or not quality attributes (or ASRs) are met. The relationship between architecture and ASRs is discussed in more detail in section 2.3.4.

- *communication between stakeholders*: the artefact allows trade-offs to be discussed and the architecture to be agreed upon.
- the artefact defines the *shared components* for product lines.

2.3.3 The human aspect of architecture

The definition of architecture at the end of section 2.3.1 is notable in that it refers to the developers' understanding of the design decisions and thus recognises the human and social aspect of architecture. *People* design architectures – quoting Booch (2010):

> "Architecting is a social issue because people conceive of, build, evolve, and use such systems." [49]

– and it is people who decide what is considered architectural. According to Fowler (2003):

> "...architecture is a social construct (well, software is too, but architecture is even more so) because it doesn't just depend on the software, but on what part of the software is considered important by group consensus." [97]

Similarly, Buschmann and Henney (2013) described architecture as an expression of knowledge [61], and the recent ISO/IEC/IEEE 42010 definition, significantly, includes the word 'concept', implying human understanding:

> "[Architecture is] the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution." [127]

Taylor (2007) noted architecture is more a product of the members of the development team themselves and human factors than the product of processes, methods or frameworks:

> "The design of software architecture by professional software engineers and architects (or the quality of software architecture) is driven more by an individual designer or design leader than by individual or collective adherence to externalised methods, processes or frameworks." [218]

and:

> "Secondly, in many cases (in the author's experience) software design outcomes are shaped as much by factors and forces in the immediate design situation as they are by objective constraints (such as problem requirements, platform or product constraints). This second observation suggests that 'human' factors have as much weight in the design of quality software architecture as do methodological or technological ones." [218]

The human factors that affect architecture design include the background, experience, intuition and judgement of the architects [23, 25]. Indeed, Booch (2007) and Fairbanks (2010) noted that a particular system may have more than a single correct architecture [45, 91], and two architects are likely to produce different architectures for the same problem with the same boundaries [25].

2.3.4 The importance of architecture

An important characteristic of architecture is that it is critical to the software's ability to satisfy the system's *architecturally significant requirements* (ASRs)) [66, 177]. ASRs consist mainly of non-functional requirements, or quality attributes, such as performance, security, scalability and maintainability [50, 65, 90, 91], which are very difficult to implement during development. The Software Engineering Institute (SEI) noted that:

> "The architecture is the primary carrier of system qualities, such as performance, modifiability, and security, none of which can be achieved without a unifying architectural vision." [207]

For example, an architecture may use a layered style for maintainability or portability, or it may use a particular database technology for efficiency [20].

Having an architecture that is not well designed is a major source of technical risk [138], as the development teams work around the poorly designed architecture to meet ASRs. Architecture planning reduces technical risk to a level that is acceptable to the team and stakeholders [91, 182]; without an architecture explicitly designed to meet the required ASRs, it is difficult to determine if the ASRs can be met without testing the completed system [50].

On the other hand the *functional requirements* have a much smaller impact on the architecture, and are largely left to the developers to implement [90]. Fairbanks (2010) described quality and functional requirements as being largely orthogonal, because architecture and functionality can be mixed-and-matched to a certain extent [91]. He used the example of a system that may be built with the same functionality using either a 3-tier architecture or as a peer-to-peer system, both of which offer quite different qualities. Booch (2007) used a metaphor of a lift [45]: it does not matter to the passenger whether the lift is software-enabled or entirely electromechanical – or even powered by a team of well-trained elephants – as long as the passenger is delivered to the correct floor in a suitable timeframe.

Other reasons for having an well-defined architecture [50, 91, 138] include providing a framework for development, communicating and providing a common understanding between stakeholders, and enabling reuse. Architecture can also be used to provide a basis for project management activities [138], such as cost estimation and risk management [58].

While Broy and Reussner (2010) described architecture as being "valuable and useful in nearly all phases of the software lifecycle" [58], having a well-defined architecture is more important in some situations than others. Both Shaw and Garlan (1996) [205] and Fairbanks (2010) [91] noted that architecture design becomes more of a significant issue as the size and complexity of the software system increases. Fairbanks described five specific cases when architecture is important [91]:

- Small solution space – when it is difficult to come up with a solution that works
- High failure risk – when the probability or impact of failure is high (such as medical equipment)
- Difficult quality attributes – when the quality attributes (ASRs) are difficult to meet
- New domain – when the architect team has little experience in the domain of the software
- Product line – when the architecture is going to be used in a number of products.

2.4 The balance between architecture and agility

In section 2.2.3, agility was defined as the ability to respond to changing requirements to better deliver value:

> 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.

In addition to the period of the feedback loop affecting agility (section 2.2.4), the planning phase – how much time is required between project kick-off and development starting – also affects agility. Figure 2.4 shows the feedback cycle with the planning phase.

The planning phase, often called *iteration zero* or *sprint zero* [153, 174], usually includes architecture design. In section 2.3.1 architecture was defined as the design decisions that affect the system as a whole and

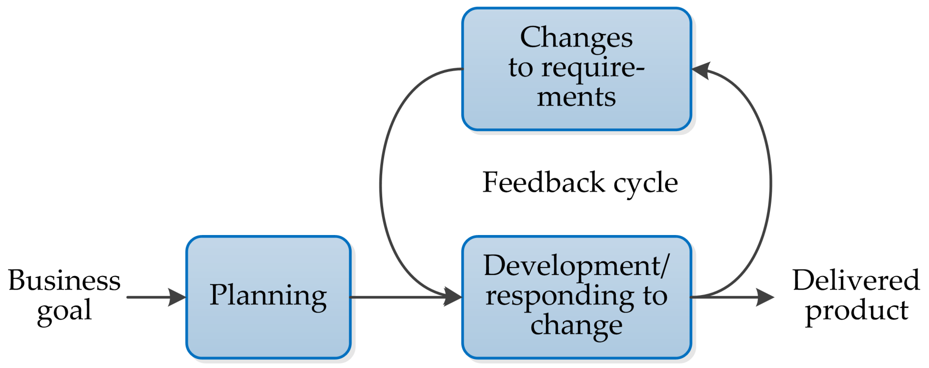

Figure 2.4: The feedback cycle, including planning. The arrows show the order of the activities.

are difficult to change; because of this, architecture design decisions are generally made in this up-front phase, along with other planning activities. In plan-driven development, all architecture design takes place up-front.

The term 'up-front' refers to architecture design effort that takes place before the implementation of functional requirements ('development') commences. Up-front architecture design effort includes architectural activities such as research, modelling and experiments, and includes any development required to verify the architecture design. Verification may be, for example, in the form of throwaway prototypes, in which code written does not go into production, or skeletal systems, which do become production code but provide minimal functionality (see section 2.4.2). Once development has started for the purpose of implementing functional requirements rather than verifying architectural decisions, the up-front phase is complete and the development phase has begun. Any architectural decisions made during this phase are emergent decisions (see the following section).

Boehm (2002) described agile methods as fitting on "a spectrum of increasing level of emphasis on plans" [36]. Figure 2.5 illustrates this spectrum, with no planning, such as *hacking* or *cowboy coding* (referring to undisciplined development with no process and no planning [197]), at one extreme and the big design up-front with an ironbound contract at the other extreme.

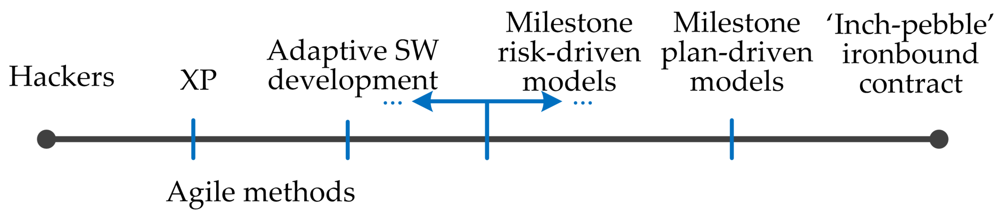
Hackers
XP
Adaptive SW development
Milestone risk-driven models
Milestone plan-driven models
'Inch-pebble' ironbound contract
Agile methods

Figure 2.5: The planning spectrum, after Boehm (2002) [36]. Unplanned and undisciplined hacking occupies the extreme left, while micromanaged milestone ('inch-pebble') planning occupies the extreme right.

The more planning and up-front architecture design a team does, the longer it takes before the customer is able to start providing feedback, and the less agile the team is. At the other extreme is the totally unplanned design, the *no up-front design*, which agile software engineers may equate with being agile [1, 174], because they see architecture and agility as an 'all or nothing' dichotomy: architecture design belongs to plan-driven development methodologies [1, 174]. Instead of up-front planning, these agile software engineers rely on refactoring to maintain code quality.

There are two reasons software engineers may consider agile development to be incompatible with architecture design: first, by definition, architecture is difficult to change, and hence is difficult to change in response to changing requirements, and second, because architecture has little impact on the functional requirements, it appears to deliver little value to the customer [142, 223], and hence is often given lower priority than functionality. Booch (2007) described software architecture as being largely irrelevant to its end users [45].

The no up-front design approach is a naive reaction according to Hollander (2010) [123]. The no up-front design approach results in an *emergent architecture* in which all architecture decisions are emergent; the design evolves from the "aggregation of a bunch of ad-hoc tactical decisions" [96] – potentially leading to a *big ball of mud* architectural pattern [94] or an *accidental architecture* [44] – rather than being explicitly thought-out and designed up-front. An emergent architecture is typically one where all decisions are *implicit* – the decisions are made without any conscious effort or decision-making process and there is no document that describes design, but rather the code needs to be inspected to discover the design [223]. Problems caused by this aggregation of ad-hoc decisions can cause projects to not satisfy their ASRs and hence gradually fail – in the words of Booch (2009), "die the death of a thousand cuts" or be "nibbled to death by ducks" [47]. If not a straight out failure, they leave a project exposed to the risk of missed milestones and exceeding budget constraints.

Cowboy coding is an example of development that produces an emergent architecture. Software built by cowboy coders is frequently unmaintainable and of poor quality [78]. Cowboy coders ultimately spend more time fixing code [78], making it more difficult and ultimately potentially impossible to respond to change, no matter how often the customer provides feedback. In this sense, cowboy coding certainly cannot be described as being agile.

Thus while a customer may be able to provide feedback earlier, less design does not necessarily make the team more agile.

The amount of planning an agile team requires is a balance or trade-off between the two extremes: a team must do some architecture design to ensure the team has sufficient architectural guidance for it to proceed with development, but without premature commitment [48]. Further architecture decisions are made during the iterative development as required.

Finding this balance is not well understood. Agile methods are silent on how to determine how much architecture design a team requires.

Section 2.4.1 introduces terminology for when architecture decisions are made, and section 2.4.2 discusses various methods used in practice for recommending how much up-front design a team should do.

2.4.1 When architecture decisions are made

Architecture decisions are often made *up-front*, but may also be made after development begins; architecture decisions made after development begins comprise an *evolving architecture*. An evolving architecture consists of *emergent* decisions, in which the architecture emerges as architecture design decisions are made, and *refactored* decisions, in which previous architecture decisions (whether up-front or emergent) are changed.

Whether up-front or emergent, architecture decisions may be either *explicit* (the decisions are consciously or intentionally defined or documented using architectural design processes) or *implicit* (the decisions are made without any conscious effort or decision-making process, in isolation from other decisions and without regard to the overall architecture) [44, 46]. Refactored decisions are always explicit.

The relationship between up-front, evolving and emergent decisions and explicit and implicit decisions is shown in Figure 2.6.

2.4.2 Architecture and agility in practice

There are a number of proposed schemes for architecture design in agile development, which are a compromise between plan-driven big up-front designs and no up-front emergent designs. Many of these schemes allow software engineers to do just enough design to advance development without premature commitment [48]. In general, the planning of the project includes some up-front design [153, 174], with ongoing architectural refinement during the iterative development. Examples of schemes that implement architecture in agile development are described below.

Even when up-front design is included, there is little guidance as to how much architecture to design [1].

XP and the system metaphor: Architecture in an XP project is replaced by the *system metaphor*, which is a 'story' on the system as a whole [26, 163], and

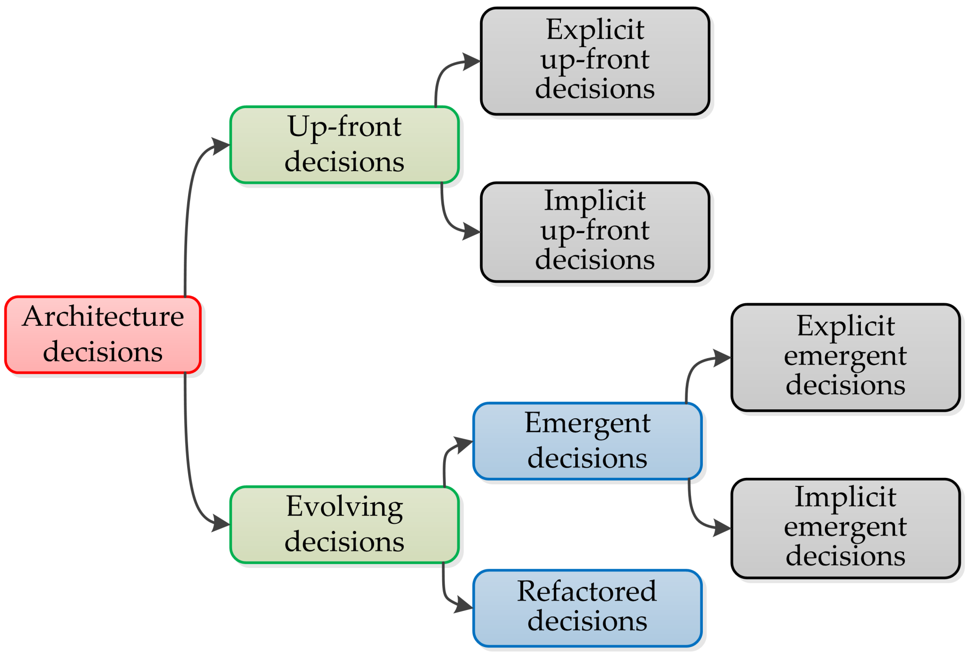
Architecture decisions
Up-front decisions
Explicit up-front decisions
Implicit up-front decisions
Evolving decisions
Emergent decisions
Explicit emergent decisions
Implicit emergent decisions
Refactored decisions

Figure 2.6: Relationship between up-front, evolving and emergent decisions, and explicit and implicit decisions. Arrows show subsets of decisions.

is used to ensure that everyone on the team has a coherent understanding of what they are building.

The system metaphor is poorly understood and frequently ignored in XP [220]. XP creator Kent Beck omitted the system metaphor from the second edition of his book [27], with no replacement practice. It is now often known as “the lost XP practice” [89].

The architecture spike: An *architecture spike* is an investigation or experiment that allows the software engineers to explore an unknown aspect of a requirement or user story, so that they can determine a satisfactory solution and estimate how much time to allocate to it [181]. A spike is often a prototype that will be thrown away, built with just enough detail for the software engineers to obtain a solution to the unknown.

The architecture spike will often occur during iteration zero, and can lead to an architecture in the form of, for example, a system metaphor [18] or a walking skeleton [181] (both described elsewhere in this section).

Risk-based: Fairbanks (2010) proposed a risk-based approach [91] where the architect evaluates and ranks the developmental risks, and mitigates them in order of rank. This approach uses three steps: identifying risks, mitigating the risk through architecture design, and evaluating the level of risk reduction.

Architects only design when there is an associated risk. They should focus on project-threatening risks and not waste time on low-impact risks and low-impact techniques:

> “When you are unconcerned about security risks, spend no time on security design. However, when performance is a project-threatening risk, work on it until you are reasonably sure that performance will be okay.” [91]

Agile modeling: Agile modeling (or 'Agile Model Driven Development') is an agile methodology that uses models to support architecture design [18]. Agile modeling requires "just enough high-level modeling at the beginning of a project to understand the scope and potential architecture of the system" [18], with additional modeling at the start of each subsequent iteration. The initial modeling takes place in iteration zero, and its aim is to give the developers a basic understanding of the high-level requirements and of a potential architectural solution [18]. The length of iteration zero depends on the size of the project.

Real options: 'Real options' theory lets developers delay decision-making (and hence design) until more information is known [32, 161]. Based on this is 'responsibility-driven architecture' [32] in which architecture decisions are mapped out in the order in which they need to be made, from up-front to late in development. For external parties that need sign-off on the architecture (for example, for assurance the application was developed with due process) it proposes incremental sign-off that follows the decision making. Real options assume there are three decisions that a team can make: "a 'right decision,' a 'wrong decision,' and 'no decision' " [161]. If a team does not have enough information to know whether a decision is the right one or the wrong one, the best decision is to not make that decision until information (gathered by the team during development) is available and makes the correct decision clear.

Related to delaying decisions is including a range of options for architectural direction, which can be narrowed down as development progresses and solutions become apparent [156].

Skeletal system: The *skeletal system* (*walking skeleton* [71] or *vertical slice* [123]) is the first incarnation of the system that links the main architectural components. The skeletal system would typically implement a small amount of end-to-end functionality [71] that acts as a frame for further development.

Architectural backlog: Madison (2010) also proposed a scheme that uses an initial architecture phase and an ongoing architecture development phase, with the architects being part of the ongoing iteration process [156]. He maintains separate logical backlogs for architectural concerns and development concerns.

2.4.3 Architecture, agility and context

The methods for designing architecture in agile development methodologies described above all attempt to reduce the level of up-front design but still maintain the benefits that architecture brings. Many of these methods are silent on how to identify the ASRs that should be included in iteration zero, how to perform incremental architecture design, and how to validate architectural features [1]. Determining the most appropriate amount of architecture is not a simple matter and depends on context [1] – which includes, among other things, the organisation, the domain, the project size, how stable the architecture is and the business model. Possible factors that make up the context are shown in Figure 2.7.

A focus of this research is to investigate what the architecture trade-off depends on, and when architecture design takes place.

2.5 The research objective

There has been very little empirical research on the relationship between software architecture and agile development to date [52]. Just prior to the start of this research, Breivold et al. published the results of a survey of the literature and concluded that studies have been small, diverging, and in some cases, performed in an artificial setting [52]. Dybå and Dingsøyr also noted the need for more knowledge of software development in general, particularly through empirical studies [88]. This lack of research does not mean that it is not an important issue: at the XP2010 conference in

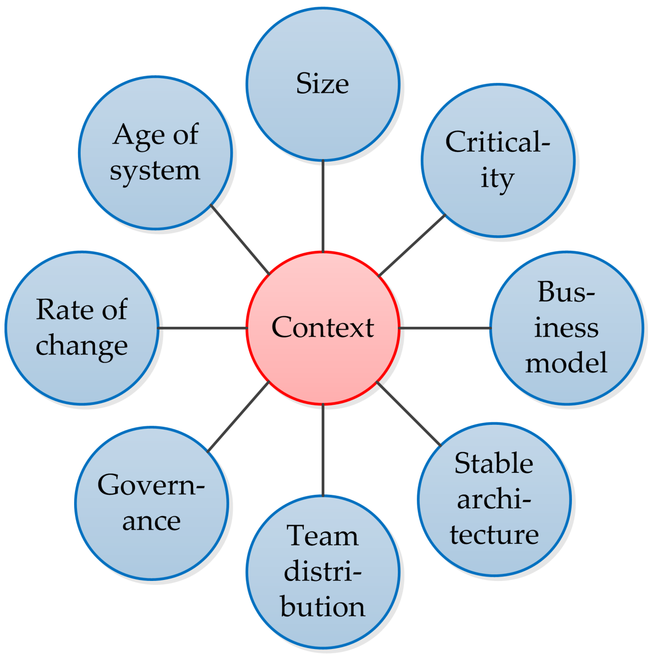

Figure 2.7: The factors that make up a project's context, after Abrahamsson, Ali Babar and Kruchten (2010) [1] and Kruchten (2013) [146]

Norway, how to determine how much architecture design is enough was rated as the second-equal most burning question of more than sixty that agile practitioners face [99] (Table 2.2).

This research will help to address this gap. The goal of this research is presented as an objective rather than as a hypothesis; defining a hypothesis is very difficult due to the paucity of prior research.

RESEARCH OBJECTIVE

The objective of this research is to explore how much effort agile software teams put into architecture design, with a focus on how teams determine which architecture decisions to make up-front and what influences those decisions.

This research will address the question "how do agile teams determine how much architecture to design up-front?"

Priority	Research issue
1	Agile and large projects
2=	What factors can break self-organisation?
2=	Do teams really need to always be collocated to collaborate effectively?
2=	Architecture and agile – how much design is enough for different classes of problem?
5=	Hard facts on costs of distribution
5=	The correlation between release length and success rate
5=	What metrics can we use with minimal side-effects?
8=	Distributed agile and trust – what happens [at] around 8–12 weeks?
8=	Statistics and data about how much money/time is saved by agile
8=	Sociological studies – what were the personalities in successful/failed agile teams?

Table 2.2: The top ten burning research questions faced by practitioners, from Freudenberg and Sharp (2010) [99].

Chapter 3

Research design

"We think that larger studies, based on defined metrics, performed in the industrial domain, are necessary in order to increase our understanding of how agile and architecture interrelate." [52].

This chapter presents the research design. Section 3.1 discusses the research position and perspective, and determines a qualitative strategy is the most appropriate. Section 3.2 presents a number of possible qualitative research strategies, selecting grounded theory as the most suitable. Section 3.3 outlines the research methodology: the techniques and practices of grounded theory. Section 3.4 discusses the role of the researcher, and section 3.5 closes the chapter with a discussion on how ethical issues and threats to validity were treated.

3.1 Adopting an appropriate research strategy

There are a number of considerations that must be taken into account before selecting an empirical research strategy [113]: the relationship between theory and research, epistemological issues, ontological issues, and issues of qualitative versus quantitative research [59]. This section describes these considerations.

3.1.1 Relationship between theory and research

Two approaches can be taken when considering the relationship between theory and research: research can either be *deductive*, in which the research is used to prove (or disprove) a hypothesis or hypotheses, or it can be *inductive*, in which a theory is derived from the research [59] (Figure 3.1).

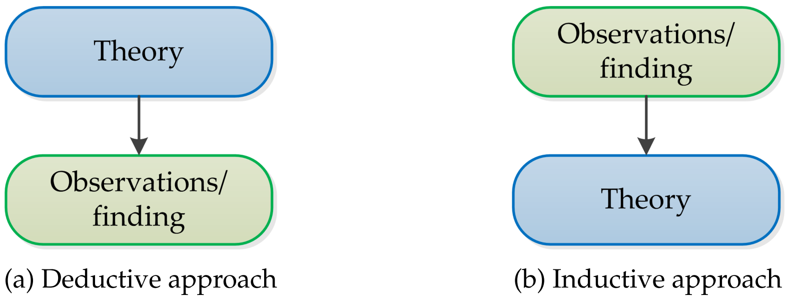

Figure 3.1: Deductive and inductive approaches to the relationship between theory and research, after Bryman (2008) [59]

The goal of this research is in the form of an objective (section 2.5), noting that defining a hypothesis would be difficult. An inductive approach is the most appropriate, with theory being developed from the research guided by the objective. The scope of the objective is very broad, and hence the theory is likely to be a general macro-level (high-level) theory [81] that explains broad, overall insights into many phenomena [227].

3.1.2 Epistemological considerations

Epistemological issues concern the question of what is regarded as acceptable knowledge in a discipline [59] and how *theory* is defined. A central issue to research of this nature is whether the social world (based on the experience, knowledge and judgement of agile team members) can be studied using the same principles and procedures as the natural sciences.

The three main epistemological positions are *positivism*, *realism* and *interpretivism*. Positivism is the position that believes knowledge is obtained using measurable observations from a neutral perspective [64, 168], and is typically used in scientific research when proving hypotheses. Bryman (2008) described positivism as entailing five principles:

> 1. "*Phenomenalism*: only phenomena and hence knowledge confirmed by the senses can genuinely be warranted as knowledge.
> 2. "*Deductivism*: the purpose of theory is to generate hypotheses that can be tested and that will thereby allow explanations of laws to be assessed.
> 3. "*Inductivism*: knowledge is arrived at through the gathering of facts that provide the basis for laws.
> 4. "*Objective*: Science must (and presumably can) be conducted in a way that is value free.
> 5. "There is a clear distinction between scientific statements and normative statements (implying a norm or a standard) and a belief that the former are the true domain of the scientist.
>
> "This last principle is implied by the first because the truth or otherwise of normative statements cannot be confirmed by the senses." [59]

From these principles it can be seen that positivism has elements of both deductive and inductive research.

Related to positivism is *realism*, which believes that there is a reality that is separate from our observations or descriptions of it [59]. Realism can be one of two types: *empirical realism* or *critical realism*. Empirical realism asserts that reality can be understood through the use of appropriate methods, and critical realism asserts that understanding the real world requires the identification of the structures that generate that world. The identification of the structures allows researchers to change them. Unlike positivism, critical realism accepts that the structures may not be detected by the senses.

In contrast to positivism and realism is *interpretivism*, which is based on *understanding* and *meaning*, and *interpreting* the data and its context. The key difference between interpretivism and positivism is the former requires the research to interpret and *understand* the behaviour of the actors from their perspective [64], rather than the neutral perspective of the latter. If positivism tries to *explain* human behaviour, then interpretivism tries to *understand* human behaviour (the philosophy of *hermeneutics*).

In the context of this research, the effort an agile team puts into architecture design cannot be mapped out in advance independently of the team: architecture design depends on the experience, knowledge and judgement of the architects and the development team, the way that the agile team works together and how it follows the agile philosophy and principles, and what the team members themselves believe the appropriate amount of architecture for the minimum effort will be for their particular circumstances. This research will investigate and interpret these factors and related data to determine how much effort teams put into up-front architecture design, and what affects that effort, and thus requires an interpretivist perspective.

3.1.3 Ontological considerations

Ontology relates to the nature of social world: whether the world is external to those participating in the research, or whether they are a part of the world. There are two ontological positions: *objectivism*, in which social entities are considered objective entities that have a reality external to the participants, and *constructivism*, in which social entities are considered social constructions built up from the perceptions and actions of the participants [59]. Objectivism is a position where phenomena are independent of the actors; they are beyond our reach or influence. For example, an organisation may be considered an object that has rules and regulations, procedures and structure. The individuals within that organisation conform to the organisation's rules, its hierarchy and its social order. The organisation's features are independent of the individuals within it.

Constructivism, on the other hand, refers to social phenomena that are defined by the actors, and continue to be shaped by them. The entities in this research are agile software development teams. Agile teams are not plan-driven; they are self-organising and self-committing (see section 2.2): the work to be completed by the team is decided by the team. Therefore the teams and hence the systems they are building are shaped by the team members, and evolve with them as their skills develop and team members come and go. The success of the teams reflect the abilities and the performances of the team members.

A constructivist ontology is therefore the most appropriate for this research.

3.1.4 Qualitative and quantitative research

Empirical research can be categorised into two types: *qualitative* and *quantitative*. The data from qualitative research comes from words, and the data for quantitative research comes from numbers – the quantification and measurement of data and analysis [187].

Qualitative and quantitative research strategies are strongly linked to the considerations discussed above – research and theory, the epistemological perspective and the ontological position [59], as summarised in Table 3.1.

This research is inductive, uses an interpretivist position and constructivist ontology, and therefore a qualitative strategy is the most appropriate.

	Quantitative	**Qualitative**
Principal orientation to the role of theory in relation to research:	Deductive; testing of theory	Inductive; generation of theory
Epistemological orientation:	Natural science model; in particular positivism	Interpretivism
Ontological orientation:	Objectivism	Constructivism

Table 3.1: Fundamental differences between quantitative and qualitative research strategies [59]

3.2 Adopting an appropriate research strategy

There are a number of popular qualitative research strategies. These include *action research*, in which the solution to a problem of the participant is developed and tested with that participant, *case study* research, a detailed and intensive analysis of a small number of cases, *ethnography*, which requires researcher participation for an extended period of time, and *grounded theory*, which generates theory out of the research data using iterative data collection and analysis.

This section describes these strategies and their suitability for this research. Grounded theory is chosen as the most appropriate strategy.

3.2.1 Action research

Action research is a strategy that sets out to influence or change the subject of the research [194]. The purpose is often to develop a solution to a problem that the participant has, in a collaboration between the researcher and participant. The steps of action research are defining a concern, planning a change to address that concern, implementing the change, then observing the effects of the change, and repeating if necessary [135].

While action research would be a suitable strategy for researching architecture design in agile development, it does not meet the broad, high-level goals of the research as defined by its objective. Action research requires a significant time commitment from the participant or participants, which is often a difficulty for participants working in commercial organisations.

Therefore action research was considered not to be suitable for this research.

3.2.2 Case study research

Case study research is used when the researcher is looking for a deep understanding of the phenomenon being explored [232], and involves detailed and intensive analysis of a small number of related cases [81, 194] within a particular context (or situations, organisations or groups).

Case study research usually requires data collection from multiple sources, such as observations, interviews, documents and audiovisual material [80].

Case study research would be ideal for exploring in detail the effects of team members on architecture design in one, two or three agile software teams, but does not produce a high-level theory as suggested by the research objective. Like action research, it also requires a significant time commitment from participants, which again may be difficult for teams in a commercial environment.

Case study research was therefore considered not to be the most appropriate for this research.

3.2.3 Ethnography

Ethnography requires researcher participation for an extended period of time with a particular cultural group [80]. The purpose of ethnography is to describe and interpret the culture, social structure and way of life of the group (its values, behaviours, beliefs and language) [187, 194]. The researcher is immersed in the group, and participates in its daily activities.

While ethnography has been used successfully for agile software development research [204], it is not considered suitable for this research because like case study research, does not generate a theory, whereas the aim of this research is a high-level theory that requires a much broader range of groups.

3.2.4 Grounded theory

Grounded theory is a strategy that generates theory out of data rather than describing phenomenon: "grounded theory discovers theory from data systematically obtained from social research" [111].

Grounded theory is useful for exploring new areas of research or new perspectives where there is no prior theory available, or when existing theory is incomplete or was developed for different populations [81].

Grounded theory involves gathering data from a large number of individuals (authors suggesting various ranges from 20 to 60 may be sufficient [80, 81, 212]), who have all experienced the phenomenon being explored, and who are selected to ensure wide coverage of all the aspects of the phenomenon.

Grounded theory has been described as the most popular strategy for conducting qualitative research [11, 59, 219]. Despite its popularity, grounded theory has a number of limitations. These include the need for researchers to put aside their awareness of relevant related theories until late in the analysis, the inability to tightly define the research question and process to be used in advance (such as defining the sample size), the time required to transcribe interviews, whether or not grounded theory actually generates a theory, the vagueness of the grounded theory terminology and procedures, an over-emphasis on coding data, and the presence of competing grounded theory variations [11, 59]. These limitations are either accepted characteristics of grounded theory or can be addressed while designing the research or performing the analysis.

Grounded theory was selected for this research because it most closely fits the goals of the research. Grounded theory is also becoming increasingly popular in the field of software engineering, and in particular agile software development (for example, [7, 84, 93, 119, 160]). The prior work to develop the use of grounded theory in software engineering means there are better resources and support for its use in this field.

As well as being a research strategy, grounded theory also has a specific set of procedures for systematically and rigorously deriving theory from data (explained below).

3.3 The grounded theory methodology

Grounded theory is both a research strategy (described in the previous section) and, unlike other strategies, a well-defined set of techniques and procedures for systematically and rigorously analysing data and developing theory [64, 111, 187]. Grounded theory was developed in 1967 by Glaser and Strauss [111] to address a lack of sociological theory-generating qualitative research strategies at that time [64, 194].

The theory produced by grounded theory is "a set of integrated conceptual hypotheses systematically generated" [109] which produces a "formal, testable explanation of some events that includes explanations of how things relate to one another" [233]. The theory is not a description or representation of experiences or perspectives [212]. A grounded theory can be presented as a well-codified set of propositions or as a running theoretical discussion [108]. Because it is based on observations (such as interviews), it is described as being *grounded* in the data being studied. The theory is called a *substantive* theory because it is 'from the data' – empirical [108] – and is only appropriate to the study conducted. The theory can be extended with further grounded theory research or combined with the results of other grounded theory research in other substantive areas, in which case it may become part of a *formal* or general theory [10, 80, 104].

It is important that the researcher does not have a preconceived hypothesis about the subject under study as this may bias and influence analysis, leading the researcher to prejudge or 'force' the results and miss important factors that may affect the emergent theory [9, 64].

Grounded theory is highly iterative: analysis gives direction to subsequent data collection to ensure a wide coverage of all variations.

3.3.1 Versions of grounded theory

There are broadly two versions of grounded theory, both originating in the classic 1967 grounded theory book by Barney Glaser and Anselm Strauss [111], and which arose from a disagreement between the two authors. The first and original version of grounded theory, *Glaserian* (or *classic*) grounded theory [109], is described in Glaser and Strauss (1967) [111] and Glaser (1978) [104]. The second type, *Straussian* grounded theory [211], is described primarily in Strauss and Corbin (1998) [213].

Glaserian grounded theory is a more emergent or 'pure' grounded theory that Glaser claimed is more in line with grounded theory's original intention of generating substantive theory [194]. Straussian grounded theory has a more defined procedure [7, 81] which puts more weight on the researcher's preconceived views [11], but which Glaser claimed gives it more opportunity to force the results [105].

A third and less common version of grounded theory is Charmaz's *constructivist* grounded theory [64]. Constructivist grounded theory is based on the belief that traditional grounded theory methodologies actually take more of an objectivist ontological orientation, and hence takes an orientation that is more constructivist [64]. Constructivist grounded theory researchers believe that their role is to create the theory rather than be a conduit for the research process (that is, 'discover' the theory [111]), and thus the theory is dependent on the researcher's view.

While Straussian grounded theory often appeals to software engineering researchers because of the more defined procedures [7], this research uses Glaserian because of the experience and resources that this school's agile research group has in Glaserian grounded theory [10, 84, 119].

3.3.2 The research question

Grounded theory is an inductive research strategy, and hence does not start with a specific preconceived question or hypothesis [104]. Glaser considered preconceived problems as risking 'filtering' the data according to those problems rather than letting the problems emerge from the data (from the participants) [109, 212]. A question for a grounded theory research proposal is therefore general; it is vague and global [212], and is used to simply guide the area of research [104].

The objective for this research (defined in section 2.5 and repeated here) is very general and is suitable for a grounded theory objective: "The objective of this research is to explore how much effort agile software teams put into architecture design, with a focus on how teams determine which architecture decisions to make up-front and what influences those decisions."

3.3.3 Data collection

Data for qualitative research can be gathered in a number of ways, including interviews, observations and documentation. Interviews are the most common method of data collection in grounded theory [9, 194]; types of interviews are *structured* interviews, *semi-structured* interviews and *unstructured* interviews [168].

Semi-structured interviews best suit this type of research because they are based on a set of *open questions* that can be re-ordered or added to investigate issues further and draw out more information if required [59, 168]. Questions evolve as the emerging theory points to new directions for investigation and emerging gaps [111].

Open questions allow the participant to respond however they wish, according to their own knowledge and experience, enriching the data and enabling new areas to be explored [59].

On the other hand structured interviews follow a strict set of *closed questions* to ensure each participant receives the same stimulus, and are often used for surveys. These methods are not considered suitable for this research because they restrict the answers to choices the interviewer has already selected and are not useful for exploring new areas. Unstructured interviews proceed like everyday conversations [168] and provide the interview participant with significant freedom to discuss the issues that are important to them [82]. The interviewer provides limited prompting and is guided only by a list of topics or issues.

It is impossible to determine in advance exactly how many participants are required; data collection continues until *theoretical saturation* [111], at which point the data yields no further insights. As noted above, sample sizes typically vary from 20 to 60.

The participants in this research were agile practitioners who have experience with architecture design (such as architects and senior developers) or have experienced the impact of architecture (such as team managers and customers). I did not interview those without knowledge or awareness of the architecture.

I selected participants through existing industry contacts, industry user groups, email lists and snowball sampling [194]: participants were asked to suggest further participants. The location of participants was not restricted: I selected participants from New Zealand and other countries as travel permitted.

The semi-structured interviews were all face-to-face rather than by telephone or video conference to maximise rapport with the participants, and so that body language could be observed and nuances picked up.

Where possible, I interviewed more than one participant from a single project to get multiple viewpoints.

Interviews covered seven broad topics: the participant, the agile project and the development methodology to provide background, and the architecture, success factors, problems faced, and other issues to uncover architectural concerns.

Questions in early interviews were very general to ensure coverage of all possible themes. As the research progressed and as themes started to emerge that required further investigation through more specific questions, the question schedule evolved and become more focused. This evolution is discussed in more detail in section 4.1.2. The initial and final question schedules are included in Appendix B.

All interviews were recorded and transcribed. While some researchers – notably Glaser [104] – prefer not to record interviews, analysing the data as they gather it, recording and transcribing ensured accuracy and completeness, and also allowed the participant to confirm the transcript was an accurate record of the interview, which was part of the application to the Human Ethics Committee for approval to do this research (see Appendix A). All transcribing was completed personally, which, while time consuming, meant that analysing could start immediately during transcription, as suggested by Kvale (2009) [149], rather than having to wait for a third party to finish transcribing.

3.3.4 Data analysis

This section describes the steps and processes of Glaserian grounded theory that were used in this research. These steps are not prescriptive: they provide an explanation of the methodology rather than a 'codification' of procedures [187]. How I applied these steps in this research is explained in Chapter 4.

The main goal of grounded theory analysis is to produce a high-level abstraction called the *core category* which accounts for the key concerns and variations in the data [187]. This high-level abstraction is achieved with two intertwined activities: *coding*, which is used at a number of levels to create higher levels of abstraction, and *constant comparison*, which finds and relates concepts at each level of abstraction. Both activities play important roles in generating theory.

Open coding

The first activity of data analysis is *open coding*, which the researcher uses to raise the data to a first level of abstraction. The researcher identifies events, or *incidents* [104], in the data, and assigns them codes (or labels) that reflect what is happening, and thus are analytical rather than descriptive [187]. To show a code represents a process, it is often (or includes) a *gerund* [104] (a present participle of a verb ending in *-ing*). The objective of coding is to conceptualise the data, rather than to summarise the data or restate the data in a different way [187].

The coded incidents can be single words, phrases, lines, sentences or paragraphs [9], depending on the type and richness of the data [105]: looking for codes line-by-line in the data is useful for interviews [104], word-by-word is useful for documents and other ephemera, and incident-by-incident is useful for observations [64]. Using smaller blocks of data can ensure more rich theory but using blocks that are too small is very time consuming and can cause the research to lose focus [9]. Codes are not predefined; they are defined to fit the data as analysis progresses, to ensure the direction of the analysis is not restricted in any way [104].

Constant comparison

Constant comparison is used to generate a theory that is integrated, consistent, plausible and close to the data [111]. There are three levels of constant comparison: incidents are compared with incidents, *concepts* (recurring patterns of social behaviour [7]) are compared with incidents, and concepts are compared with concepts [109].

Comparing incidents with incidents starts at the same time as open coding starts. As incidents are coded, the researcher compares each new incident with previously coded incidents to establish the similarities and differences between the incidents [104]. Related incidents are coded with the same codes; the groups of related incidents and their similarities and conditions become generated concepts and hypotheses [104, 109].

As coding and constant comparison progress, the researcher compares generated concepts with new incidents as they are coded to generate new theoretical properties of the concept [104]. Using constant comparison, concepts evolve as new incidents are added and as themes emerge [10].

When a concept leads to no new insights and no longer evolves, it has become *saturated*; the code becomes known as a *substantive code* [104] (that is, a code based on data), and the incidents, using grounded theory's concept-indicator model [104], become known as *indicators* [104].

Finally, concepts are compared with other concepts; relationships between the concepts are explored which develop themes that eventually lead to the theory.

Categories and the core category

During the concept-to-concept stage of constant comparison, some concepts will become more important and more central to the emerging themes, and thus become *categories* [109, 111]. Related concepts may be aggregated under the umbrella of a category [9], which expands to describe the broader overarching concept.

A research project should have no more than four or five categories, otherwise the research may lose focus [10].

Eventually, a *core category* will emerge. The core category is one category that appears to account for most of the variation around the problem that is the focus of the study [109]. The researcher links other categories to the core category, either as conceptual properties or sub-categories of the core category or as related categories.

The characteristics of the core category are it is central (related to as many other categories and their properties as possible), it reoccurs frequently in the data, it takes longer to saturate than other categories because it relates to so many other categories, it relates easily and meaningfully with the other categories, and it has clear and grabbing implication for formal theory [104].

Dimensions are theoretical codes that emerge from a category or property [108].

Theoretical sampling

While the procedures above are described as the 'steps' of grounded theory, they are not a sequential process. Analysis is iterative, occurring simultaneously with data gathering.

Grounded theory recommends that each interview is analysed before the next participant is interviewed: this allows the interview questioning to evolve to investigate different themes, to ensure holes in the data do not develop, to uncover bias, and, as noted above, allow the results of the analysis to guide future interviews using theoretical sampling [167, 168].

As the theory evolves and starts to emerge, the researcher explicitly searches for data that fills in gaps in the data and brings the categories to saturation, a procedure known as *maximum variation sampling* [167] or *theoretical sampling* [111]. Theoretical sampling guides the researcher to groups that are of interest to the research, which can be used to collect data to help develop underdeveloped themes, and find and explain outlier cases. (Grounded theory does not attempt to obtain statistical significance from the data [167].)

While grounded theory is described as being inductive, theoretical sampling is a deductive activity, with the emerging theory driving the data collection and analysis to further develop that theory [59, 104].

Selective coding

Once the core category has emerged, open coding ceases and is replaced with *selective coding*.

Like open coding, selective coding is a form of substantive coding – it codes the data – but only codes to the core category [104], and only codes incidents that further develop the core category, to further explain the emerging theory and its variations. Selective coding identifies the recurring problem [212] and is used with theoretical sampling to ensure that only data that is relevant and useful is collected and analysed.

Theoretical coding

Once the core category has reached saturation, *theoretical coding* may be used to conceptualise the relationships between the substantive codes (concepts, properties and categories) and the core category [104]. (Unlike substantive codes which come from comparing incidents in the data.) Theoretical codes help provide a framework for the emerging theory, putting back together the data that was split apart by open coding [194], or, in Glaser's words, "weave the fractured story back together again" [104].

Glaser defined a number of theoretical coding families that provide a structure for the theoretical codes, with eighteen in Glaser (1978) [104], nine in Glaser (1998) [106], and a further 23 in Glaser (2005) [108]. Examples of coding families include the '6 Cs', which identifies the properties of a core category as causes, contexts, contingencies, consequences, covariances and conditions [104], 'basics,' which describe a basic social process (a process describing social change [106] such as "cultivating, defaulting, centering, highlighting or becoming" [110]), and 'causal,' which describes cause and effect relationships [108].

Figure 3.2 summarises the relationships between grounded theory concepts and the different types of coding. The coded data (incidents) are labelled during open coding; the open code labels are known as indicators.

The indicators evolve during coding, and become substantive codes once saturated. Related substantive codes are grouped together into higher levels of abstraction, becoming categories or properties of categories (sub-categories). One category emerges as a core category; once this happens, open coding stops and is replaced with selective coding, which focuses on the emergent core category. Finally, theoretical coding finds the relationships between the categories and sub-categories. In contrast to substantive coding, theoretical coding does not deal with data; rather, coding is based on the memos (see below) attached to the categories and sub-categories.

Memos and sorting

Memos are notes that the researcher writes in parallel to data analysis [109] to aid the generation of concepts and categories [59]. Memos are the important stepping stone between analysis and writing up the theory [64] and can be used to explain concepts where the code itself is not sufficient, to describe the relationships between concepts, to record ideas that the researcher has during analysis, and to force the researcher to develop theoretical codes [104]. They record thoughts, comparisons and connections; they are used for thinking aloud and help give direction to future analysis [64]. Memos vary in length: anywhere, for example, from a brief sentence to a five page article. Memos are crucial to the development and the writing of the theory [109], and should be written throughout the analysis process, starting with the emergence of the first concepts.

Glaser (1978) listed five benefits of the ideas that develop through writing memos;

- "It raises the data to a conceptualisation level.
- "It develops the properties of each category which begin to define it operationally.
- "It presents hypotheses about connections between categories and/or their properties.

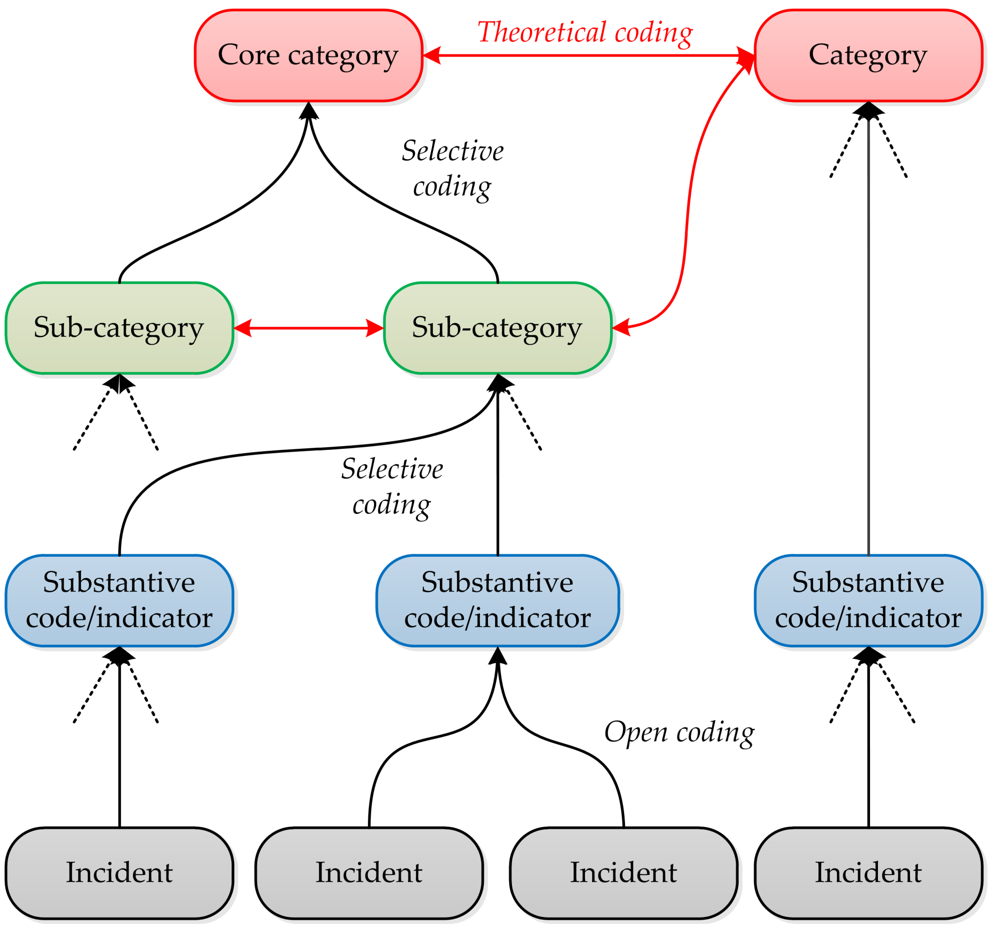

Figure 3.2: The grounded theory coding process. Black arrows show substantive coding (open coding and selective coding) towards higher levels of abstraction of the data: abstracting incidents (grey) to substantive codes (blue) and to then sub-categories (green) and categories (red). Red arrows show theoretical coding between categories and sub-categories.

- "It begins to integrate these connections with clusters of other categories to generate the theory.

- "It begins to locate the emerging theory with other theories with potentially more or less relevance." [104]

Once the core category has saturated and data collection and analysis has nearly been completed, the next step is to *sort* the memos. Sorting memos integrates the categories theoretically [64], creating a "theoretical outline, or conceptual framework, for the full articulation of the grounded theory through an integrated set of hypothesis" [109]. Sorting memos can spark new ideas and new memos.

Some of Glaser's important rules of sorting are [104]:

- Sorting starts anywhere with the memos; sorting finds its own start, beginning and end.

- Theory is generated for the core variable, so each category and property should only be sorted on how it relates to the core category. If a concept does not relate to the core category then it is not included in the theory.

- If there are two core variables, one should be demoted so that sorting focuses on one core variable: the theory relates to one core variable only.

- Constant comparison between the memos may introduce new ideas and hence new memos.

Glaser (2005) recommended physically sorting the memos on a surface such as a table, using first the memo labels and then the memos themselves [108] to better grasp the relationships between memos.

Writing the theory

The final step of grounded theory research is to write up the theory using the theoretical outline developed using the theoretical codes and the sorted memos [104]. Each sorted memo may become a section or a subsection of the draft write-up of the theory [64].

3.3.5 Emergence and forcing

The explicit purpose of grounded theory is to generate theory entirely from the data [111].

This purpose has several very important implications: firstly, the researcher must not have any preconceived ideas about the findings; any preconceived ideas may cause the researcher to subconsciously filter the data and analysis, leading to concepts that are not connected to the data and missing important issues [105]. Secondly, the researcher must not *force* the data out of impatience; codes, concepts and categories must be allowed to emerge from the data. The codes must be emergent and 'earn' their way in [104]; they must not be predefined. While coding, the researcher methodically codes line-by-line, asking the neutral question "what is happening here?" [104] rather than looking for evidence of some idea that they may already have. To avoid preconceiving ideas, the researcher avoids a full literature review prior to research.

3.3.6 The role of the literature review

Literature is treated differently in grounded theory from other research methodologies, and is based on the principle that the theory emerges from the data, rather than from preconceived ideas, studies or theory [104] (see above).

When using grounded theory, there is no major literature review before research begins. Instead, the initial literature review is a minor review whose purpose is no more than to provide the researcher with an understanding of the research problem and to determine that the problem has not already been solved [212]. An exhaustive review may lead to the researcher to forming premature ideas about the problem that would unduly influence (or contaminate [105]) the emergence of the theory from the data and "violate the basic premise of grounded theory" [109].

Once the theoretical directions of the research have formed, the researcher can start the main literature review. At this stage the literature is treated as any other data to feed into the analysis process [187] – "all is data" [104]. The purpose is to relate the emerging theory to previous research, and to identify the contributions of this research [212].

The main literature review fulfils the role of *theoretical integration* [222], comparing the generated theory with other research, and is included in the discussion chapter, Chapter 9.

The initial literature review introduces this research, and thus forms the basis of Chapter 2.

3.4 The role of the researcher

Because this research takes an interpretivist position, the researcher is important to the interpretation of the data, although is also a source of bias [81, 212].

I have a Bachelor of Electrical and Electronic Engineering (First Class Honours) and a Master of Electrical and Electronic Engineering, both from the University of Canterbury in Christchurch in New Zealand. For these degrees, my main interests were communications and software engineering. In the time between completing my Master's degree and commencing this doctoral research I worked in the telecommunications field, spending more than ten years as a strategy consultant. In contrast, this research allowed me to pursue my other academic interest, software engineering.

My background fitted this research well: being new to the software industry but at the same time having an understanding of the technical and commercial worlds means that I could act as a neutral bystander, with neither preconceived ideas to affect my ability to explore different issues or explain observations [104]. My background helped me to take the role of 'student' and avoid the risk of interview participants perceiving me as judging their actions and decisions [64, 212], which could lead them to not being open about their experiences, particularly when discussing projects that did not go well. Thus my background helped minimise researcher bias and reduce participant bias [194].

3.5 Ethical issues and threats to validity

This section describes how I designed the research to address ethical issues and threats to the research's validity.

3.5.1 Ethical issues

This research considered ethical concerns such as those listed by Miles and Huberman (1994) [167] and Robson (2002) [194], relating to privacy and confidentiality, informed consent, rights, fairness, sensitivity, and benefits.

One of the most important issues was the privacy of the participants. No names of individuals or companies, or any other information that might identify the participants or companies (including specific project details) were used. If participants were confident that their identity, company or project will not be identified, they would be more likely to be open and free with information during interviews.

All participants were given an information sheet that summarised the purpose of this research to ensure they had a full understanding of what the research entailed and what their contribution would be. They had to provide written consent of their participation, and were given the opportunity to read transcripts of the interviews afterwards so that they could confirm they are an accurate record of what was discussed. They were also given the opportunity to withdraw if they wished.

This study received approval from the Victoria University Human Ethics Committee (see Appendix A).

3.5.2 Threats to validity

Threats to validity may come from a number of sources. Participants may not have described their experiences accurately; their recollections may have been incorrect or they may have missed important details about their experiences. Participants were all volunteers who wanted to contribute to the research; they identified themselves as being agile, and hence were typically enthusiastic agile supporters who were interested in agile research and theory about agile processes. They were therefore more likely to be experienced and have higher-than-average agile skills; practitioners with average and less-than-average skills were less interested in sharing experiences and are therefore probably under-represented in the research. Below average practitioners and practitioners who are not sure about their agility may have different issues and problems from above-average practitioners – although, on the other hand, the abilities of above-average practitioners may enable them to better understand the true causes of issues they face.

It is implicitly assumed that the experiences described by participants can be used to make recommendations to other teams on how to design an agile architecture (see the implications for practice in section 10.2). While it is possible that every participant 'got it wrong,' leading to inappropriate implications, this is unlikely. Some participants getting it wrong should be picked up through theoretical sampling (section 3.3.4), which ensures varying cases are explored and explained. The interview questions gave participants the opportunity to describe anything they would do differently in hindsight, so if the participants were aware that they 'got it wrong' then they could describe what they should have done. Several participants did indeed use this opportunity to describe what they would do differently. Given the experience of the participants and the number of participants, the chance that *all* teams got it wrong, and were unaware that they got it wrong, is very small.

Other threats to validity include:

- an inaccurate or incomplete *description* of the data [194]. This can be prevented by recording interviews, or by taking high-quality notes. I recorded all interviews in this research.
- an invalid *interpretation* of the data, which can be prevented by providing a clear chain of evidence of the research findings that prove the validity of the interpretations [194]. The chain of evidence for this research is provided in the analysis chapter (Chapter 4).
- not considering alternative explanations or understandings of the phenomena that lead to the *theory* [194]. Alternative explanations and understandings that did not support the emerging theory were actively sought as part of the theoretical sampling process.

Tactics that minimise threats to validation are *data triangulation* (using multiple sources of data), *member checking* (findings were sent to participants for verification and comment), *audit trail* (all transcripts, notes and a research journal were kept) and *negative cases/outliers* (negative cases were presented and explained in the theory) [81, 167, 194].

Glaser (2002) recommends not asking participants to review the theory:

> "They may or may not understand the theory, or even like the theory if they do understand it. Many do not understand the summary benefit of concepts that go beyond description to a transcending bigger picture. GT [grounded theory] is generated from much data, of which many participants may be empirically unaware. GT is applicable to the participants as an explanation

> of the preponderance of their ongoing behaviour which is how they are resolving their main concern, which they may not be aware of conceptually, if at all. It is just what they do! GT is not their voice: it is a generated abstraction from their doings and their meanings that are taken as data for the conceptual generation." [107]

Member checking was therefore used with caution.

Chapter 4

Data collection and analysis

"I've never been in a project where we've got to the end and gone, 'yep, what we wrote on day one was primo, and we had it right.' " *(P24, customer)*

This chapter describes how the grounded theory methodology, described in section 3.3, was used to generate theory from this research. Section 4.1 describes the data collection and summarises the participant details, and section 4.2 describes the data analysis using the grounded theory methodology.

4.1 Data collection

This section describes the participants in the research and how I collected the data.

4.1.1 Participants

This research included forty four participants in thirty seven first interviews and six follow-up interviews. Participants were mainly architects, senior developers, team leads and managers; there were also a smaller number of customers, business analysts and agile coaches.

First interviews were labelled uniquely P1 to P37; subsequent interviews were numbered with suffixes (such as P6#2 and P6#3).

In most instances participants were from different companies and worked on different projects, although in a number of instances there were multiple participants from the same company who were interviewed separately (such as P15 and P24, and P33 to P36).

I did not seek participants who used specific agile methodologies. I did not believe that participants using different methodologies would affect the findings, as long as the methodologies participants used satisfied the definition of agility in section 2.2.3.

On four occasions there were multiple participants in a single interview. I did not request multiple participants: on one occasion, the participants changed the schedule from two individual interviews to one shared interview at the start of the session, while on other occasions the lead participant (with whom I had set up the meeting) brought in extra participants to provide additional viewpoints. The advantages of group interviews are they are an efficient method for gathering the views of a lot of participants, they provide natural quality control on the data, and it is easy to gauge consistent and shared views. The disadvantages are more difficult facilitation, a risk of domination by one or two people, and not having sufficient time to get everyone's responses to the questions [194]. While having sufficient time to interview each participant separately would have been preferable, I believe that the benefits outweighed the disadvantages in these cases. Different participants in the same interview were not given unique numbers, but were differentiated by appending different letters (such as P11a and P11b).

Nearly half of the interviews took place in my home town of Wellington in New Zealand, while another eight participants were interviewed in other cities within New Zealand: Dunedin, Christchurch and Auckland. The remaining data gathering took place in the United Kingdom, Sweden, Austria and India. The participants themselves were from (and discussed projects based in) New Zealand, Australia, India, the United Kingdom, the United States and Brazil.

As the research progressed, theoretical sampling meant more specific types of participants were recruited where possible to fill in gaps in the emerging theory. For example, P33–P36 were recruited because architects working in product line-type systems were under-represented.

Recruiting participants was generally more difficult and time consuming than expected; suitable candidates were not always available to meet face-to-face at the time I was interviewing and, particularly initially, some potential participants ruled themselves out because they did not think they were suitably qualified (they did not think they were expert enough). For example, one large independent software vendor (ISV) only offered P4, their most senior architect ('Director of Architecture'), as a participant when other architects may also have made useful contributions. I overcame this to some extent by emphasising during recruitment that questions were not technical, and that everybody's opinions were valuable. In some instances confidentiality was a problem; one participant requested I sign a non-disclosure agreement so that he could more openly share his experiences.

Table 4.1 contains a summary of each participant, including their role (or job title), how much experience they had in professional software development, the type of organisation they worked for (for example, ISV or a government department), and the domain they worked in (or the domain their sample project was from), such as telecommunications, healthcare or finance.

Table 4.2 provides information about the team and the project the participant was involved with. The table includes the agile methods that the team used, the size of the team, the duration of the project, and a very high-level description of the system or architecture that they were building or designing.

	Participant role	**Experience**	**Organisation type**	**Domain**
P1	Developer	>15 years	Government agency	Health
P2	Developer/ architect	20 years	ISV (independent software vendor)	E-commerce
P3	Development manager	6 years (agile)	ISV	Personnel
P4	Director of architecture	More than 20 years	ISV	Digital archiving
P5	Coach/dev. manager	20 years	Start-up	Entertainment
P6	Managing dir./ lead developer	20 years	Service provider	Telecoms
P7	Business analyst	5–6 years	ISV	Telecoms
P8	Lead developer	6 years	ISV	Digital archiving
P9	Developer	40 years	Financial services	Telecoms
P10	Coach	25 years	Hardware & services	Transport
P11a P11b	Dev. manager Architect	10 years 15+ years	Government department	Government services
P12	Senior developer	10 years	Financial services	Financial services
P13	Architect	16 years	ISV	Medical
P14	Architect	10 years	ISV	Animal health
P15	Customer	4 years	Start-up service provider	Retail (electricity)
P16a P16b	CEO/chief engineer Head of engineering	>10 years 10 years	ISV	Retail (health)
P17	Manager/ coach	>10 years	Government dept.	Statistics

Table 4.1: Summary of participants

	Participant role	Experience	Organisation type	Domain
P18	Development manager	10 years	Multinational hardware vendor	Health
P19	Development manager	6–7 years	Start-up service provider	Retail (travel)
P20	Coach and trainer	20 years	Independent consultant	n/a
P21a	Manager/coach	20 years	ISV	Retail (publishing)
P21b	Architect	n/a		
P21c	Team leader	n/a		
P21d	Team leader	n/a		
P22	Senior manager	14 years	ISV	Contact management/marketing
P23a	Engineering manager	n/a	Service provider	Pharmaceutical
P23b	Product lead	6 years		
P23c	Team lead	n/a		
P24	Customer	>10 years	Start-up service provider	Retail (electricity)
P25	Team lead	10 years	ISV	Banking
P26	Team lead	8 years	ISV	Water management
P27	CEO/coach	16 years	Start-up service provider	Retail (electricity)
P28	Technical lead	13 years	Service provider	Broadcasting
P29	Dev. manager	20 years	Banking	Banking
P30	Consulting architect	25 years	Service provider	Telecoms
P31	Enterprise architect	25 years	Government agency	Transport

Table 4.1, cont.

	Participant role	**Experience**	**Organisation type**	**Domain**
P32	Software dev. director	>15 years	ISV	Government
P33	Product architecture team leader	15 years	Medical service provider	Medical
P34	Development unit manager	15 years		
P35	Design Engineer	n/a		
P36	Development unit manager	12 years		
P37	Consultant/free-lance software developer/ architect	13 years	Service provider	Broadcasting

Table 4.1, cont.

	Agile methods	**Team size/ no. of teams**	**Duration**	**System description**
P1	Single developer	1 team member	6 months	Web-based, .NET
P2	Scrum	3 team members	Ongoing	.NET, cloud-based
P3	Scrum	3 teams	Ongoing	Web-based, .NET
P4	Scrum	5 developers	Ongoing	Java, rich client, suite of standalone tools
P5	Scrum/kanban	Various	N/A	Various
P6	Iterative	1–3 developers	Ongoing	Suite of standalone applications
P7	Scrum	12 team members	1 year+	Suite of web-based services
P8	Scrum	4–14 team members	1 year+	Ruby on Rails with Java back-end
P9	Bespoke	2–24 team members	3 years	Web-based system
P10	Scrum/XP	500–800 developers	Several years	Large distributed web-based system
P11 a, b	Scrum	8 team members	Several years	Web-based, .NET
P12	Scrum	6–7 developers	7 months	Web-based, .NET
P13	Scrum	12 developers	4 years	Monolithic .NET app
P14	Scrum	6–8 team members	18 months	.NET, large GIS component
P15	Scrum	7 developers	Ongoing (3 years to date)	Ruby On Rails web application

Table 4.2: Summary of participants' agile methods and project details

	Agile methods	**Team size or no. of teams**	**Duration**	**System description**
P16 a, b	XP	5 team members	5 months	Ruby On Rails
P17	Scrum	6 developers + admin	2–3 years	Web-based, PHP using DAO pattern
P18	Scrum	15 team members	Ongoing (>2 years)	Web-based, Java platform
P19	Lean	4 developers	Ongoing (<1 year)	PHP/Symfony, Javascript/Backbone
P20	Scrum	N/A	N/A	N/A
P21 a–d	Scrum	3 teams with 40 total	Several years	.NET, Websphere Commerce, SAP, others.
P22	Scrum/XP	More than 40 total	N/A	.NET
P23 a–c	Own methods	3 teams	Ongoing	Various web-based, client/server
P24	Scrum	7 developers	Ongoing (3 years to date)	Ruby On Rails web application
P25	Scrum	1 team	Ongoing	.NET, single tier web
P26	Scrum	8 team members	1 year	.NET, web-based, 7 tier
P27	Scrum	7 developers	Ongoing (3 years to date)	Ruby On Rails web application
P28	Scrum	42 team members	N/A	Python, Django, CMSs for multiple websites
P29	Kanban	2 teams with 20 team members total	Ongoing	Web based, AJAX, interface to mainframe

Table 4.2, cont.

	Agile methods	**Team size/no. of teams**	**Duration**	**System description**
P30	Scrum	7 team members	2 years+	Python with Django and Twisted, NoSQL
P31	Bespoke	7 team members	13 week pilot	Web services, SOA using .NET/WCF
P32	FDD, kanban	N/A	N/A	N/A
P33–P36	Scrum, kanban	18 teams	Ongoing	Multiple products; SOA
P37	Bespoke/XP	15–20 team members	Ongoing	Java, embedded

Table 4.2, cont.

4.1.2 Interviews

The main source of data was face-to-face interviews. I did not know the vast majority of participants in advance of the interviews, so face-to-face interviews helped build rapport. All participants talked openly and freely about their experiences; I do not believe that telephone interviews or video conference interviews would have been nearly as successful.

As the research progressed and as themes started to emerge that required further investigation through more specific questions, the question schedule evolved and became more focused. The original question schedule used for the initial interviews in May 2010 was very general, and is included in Appendix B.

The schedule was updated for P12 in November 2011. The purpose of this update was to probe in more detail some of the questions that had proven to be the most significant, such as asking the participants to define the architecture levels and asking *how* and *why* they had made their architecture decisions.

The next update was for P27 in April 2012. This update included asking the participants what activities they did in the start-up phase, and how they determined what those activities were. It also included asking the participants if being agile affected the architecture.

The schedule was updated again for P29 in July 2012. This update included specifically asking what factors affect how much design is done up-front.

The final update was for P37 in May 2013. This update asked specific questions designed to test the validity of the findings that had emerged so far, such as what the impact of complexity and size on the up-front architecture design is, what the impact of frameworks is and what the impact of the type of product is. This final schedule is included in Appendix B.

At the same time as focusing on more specific issues, the questioning became more unstructured to allow the participants to fully lead the discussion around the issue, as recommended by Stern and Porr (2011) [212].

For example, I asked P1 the following general question, before any ideas about what affected up-front architecture design had emerged:

> *Interviewer: "Did the architecture change or evolve during development?"*

– while much later, when the research was focused on the core category, I asked P37 the following question:

> *Interviewer: "What I'm going to do, instead of going through these questions, I'm just going to ask you about what you think affects up-front architecture design effort – what determines how much you do, and if you can perhaps squeeze in these words: complexity and size, bespoke components and libraries, legacy integration, level of agility perhaps, type of project, experience."*

On a number of occasions, I asked participants to give second and even third interviews to explain statements they had given in their original interviews, or to give their views on ideas that emerged later.

The average length of the interviews was about one hour and ten minutes. Many of the early interviews were upwards of one and a half hours as participants discussed all issues (and more). Most of the final interviews were less than one hour as I focused on the important themes that were emerging from the research.

4.1.3 Data triangulation

In addition to interviews, where possible I used documentation and other written material provided by participants where possible to verify what was said in interviews. Material included records of architectural decisions (such as software architecture documents), which were able to confirm which architecture decisions were made and why, and copies of architecture models, which provided overviews of the architectures. This is *data triangulation* [81, 194] and is used to counteract participant bias and increase the reliability of the data.

Documentation was generally only available from participants who did more up-front design, such as P4, P7, P9, P14 and P17. The documentation typically defined the ASRs and the architecture, perhaps diagrammatically or by listing the architecture decisions made, and often confirmed why decisions or changes were made. I used the documentation to confirm the level of detail of architecture decision-making and how decisions were made. It was generally difficult to determine when decisions were made because the documents represented a snapshot in time, although some did record changes to decisions made.

While many grounded theory researchers also use observations as a data source, I did not use observations for this research. I interviewed most participants away from their offices, where observations were not possible. Most participants had also finished working on the projects they discussed, and no participant was in the up-front phase. Architecture design is an ongoing activity, and observations would be more suited to a research strategy with more sustained involvement (such as case study research) so that architecture decisions can be seen in context of the whole architecture and system being built. Observations would have been difficult to set up, would have been very time consuming and would not have provided data that was as rich as the interview data.

4.1.4 Saturation

Theoretical saturation determines when data collection stops. Recognising theoretical saturation can be difficult [9] and may be subjective [100]. Suddaby (2006) described recognising saturation as needing tacit knowledge rather than some a priori criteria [214]. Because of the broad objective of this research, it was sometimes difficult to know to what depth to take the study; I could have easily continued searching for more and more specific and richer data from more and more targeted participants, generating a theory with more and depth and more detail. More participants, however, would likely have resulted in an overly complex theory that lacked parsimony [227]. Furniss (2011) commented that saturation could be dependent on a "looming deadline" [100], and similarly, I declared the research saturated when the data gathering reached a natural plateau at a level of detail sufficient for a PhD thesis.

4.2 Data analysis

This section describes how I analysed the data using the steps of grounded theory (section 3.3.4), and shows how the theory emerged from the data.

4.2.1 Data analysis tools

I used the popular computer-based tool NVivo for the disciplined activities of managing the interview transcripts, open coding, for recording memos and for some constant comparison of codes and concepts.

Glaser is strongly opposed to the use of computers in grounded theory research because they can promote process over visibility of emerging theory, but according to Robson (2002) they do have certain benefits:

- "They provide an organised single location storage system for all stored material,
- "they give quick and easy access to material,
- "they can handle large amounts of data very quickly,
- "they force detailed consideration of all text in the database on a line-by-line (or similar) basis,
- "they help the development of consistent coding schemes." [194]

Similarly, the disadvantages are:

- "Proficiency in their use takes time and effort,
- "there may be difficulties in changing, or reluctance to change, categories of information once they have been established,
- "particular programs tend to impose specific approaches to data analysis." [194]

I avoided these disadvantages by only using NVivo for low level analysis.

Suddaby (2006) suggested that it is best to restrict computer-based tools to organising and coding [214]. Following this advice, I used various manual methods such as sticky notes, whiteboards and index cards for conceptual analysis such as finding the relationships between concepts and categories and sorting, which gave better visibility of the concepts and made it easier to see their relationships.

At no stage did I use any automatic analysis functionality such as automatic coding.

4.2.2 Open coding

Table 4.3 on the following pages shows a small selection of the coded data and codes and concepts that emerged from the interview with P29.

Data/incident	**Code/concept**
"...architects should be active coders in the team who are just seniors..."	*architect being über-developer and leader*
"...ideally that pool of seniors in your team act as a kind of proxy architecture committee, and we don't have to go to someone who's supposedly got the title sitting in an ivory tower, and has never actually built that thing in the last ten years because they've been thinking high-level..."	*senior developers making decisions*
"At an application level I think that those architects are a waste of time. I really don't think they know what they're talking about nowadays."	*architects out of touch*
"And I think that's immediately where they lose their value, if they don't have to code with the team, then they generally don't have the pain of the team and they don't know what the latest frameworks are doing, and maybe that's design, maybe that's architecture – that's where some of the debate comes in but when they're starting to look at what the best way of putting the fundamental construction backbone of whatever you're building in place... if you've never tried to flesh it out then you're never going to know if it's going to work. And I think there's a significant problem with places that separate those roles."	*architects out of touch*
"It means we have to go and justify ourselves. And so doing experiments is very hard."	*doing BUFD*
"That is where agile seems to be leading us in the lean world."	*customising process*

Table 4.3: The emergence of open codes from participant P29

Data/incident	**Code/concept**
"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."	*maximising agility, BDUF means not agile*
"...we want to be able to put in the smallest, simplest, minimum viable experiment, prove an assumption, beef it up if we want to or follow it wherever it goes, pivot and follow it wherever it goes. That means our entire architecture is going to be emergent based on where we want to go."	*maximising agility*
"Every experiment we have to make a call whether it's an A/B test or whether it's a single thing that's going to create a feedback loop and we're going to refactor it or adjust it or... so there's many different testing models – sometimes we use A/B, sometimes we use canary and sometimes we use more of a big-bang, here's an MVP, let's see how people react to it, hopefully adjust it... But that's a business decision"	*maximising agility; maximising value*
"We don't want to have to go through an architectural approval board for every one of those instances."	*spurning bureaucracy*
"...we've got a lot of smart design people who know more than the architects in our application space."	*being familiar with technology, having a capable team*
"...what we're trying to say to them is, 'look, we can actually go out with something that we haven't architected for future reuse for the next two years, because we're not sure whether it's even going to exist in two weeks.' "	*avoiding overengineering*

Table 4.3, cont.

As the analysis developed, some open codes in each interview became higher level concepts, and some concepts were aggregated into categories . Other codes and concepts did not progress the analysis any further.

Figure 4.1 shows codes written on sticky notes very early on in the analysis, after about ten interviews. The notes are being used to compare codes and relationships and to look for concepts. Some higher level codes representing concepts (red ink) are emerging, but there are very few relationships defined and there is very little order.

For later interviews, the priority for data collection and analysis was to consolidate earlier ideas and look for exceptions to earlier ideas; in particular the interview with P29 (Table 4.3) was focused on the relationship between complexity, size and up-front design. As a result of this priority, broader ideas – often consisting up to several sentences – were coded against each open code, rather than shorter phrases as was usual in earlier interviews.

4.2.3 Emerging concepts

A concept is the underlying meaning or pattern of an incident or group of incidents [105]. I labelled concepts with a high-level code and usually attached a memo to explain what was going on in the concept.

Figure 4.2 shows the codes written on sticky notes after about twenty interviews. This is still early in the analysis; concepts and relationships are starting to form but no definite categories have emerged. Groupings of notes indicate related concepts, while lines show potential relationships.

Table 4.4 below shows an example of how a high-level code and concept evolved from seven constituent codes. The seven constituent codes all had a similar theme, describing different parties making architecture decisions. Using constant comparison, they were aggregated into a concept *architecture decision makers* which described the different people and groups who make architecture decisions in agile software development. This table shows that

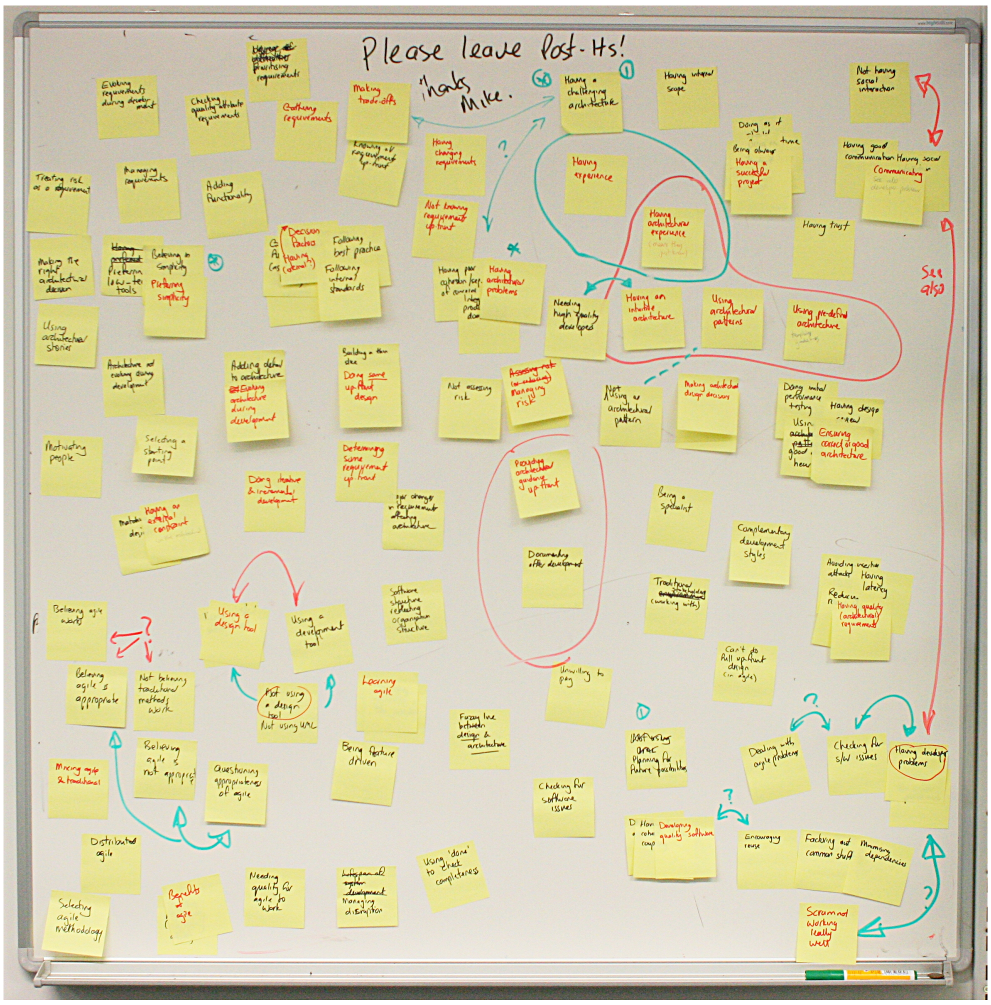
Please leave Post-Its!
thanks Mike.
Having a challenging architecture
Having unknown scope
Not having social interaction
Making trade-offs
Gathering requirements
Checking quality attribute requirements
Evolving requirements during development
Managing requirements
Adding functionality
Treating risk as a requirement
Having changing requirements
Having experience
Having architecture experience
Having good communication
Having trust
Making the right architectural decision
Using architecture stories
Following best practice
Following internal standards
Not knowing requirements up front
Needing high quality developers
Having an inflexible architecture
Using architectural patterns
Using pre-defined architecture
Adding detail to architecture
Doing some up-front design
Not assessing risk
Managing risk
Not using an architectural pattern
Motivating people
Selecting a starting point
Determining some requirements up front
Doing iterative & incremental development
Being a specialist
Complementary development styles
Documenting after development
Believing agile works
Using a design tool
Using a development tool
Software structure reflecting organisation structure
Not using a design tool
Not using UML
Learning agile
Believing agile is appropriate
Not believing traditional methods work
Believing agile is not appropriate
Questioning appropriateness of agile
Being feature driven
Checking for software issues
Unwilling to pay
Can't do full up-front design
Dealing with agile problems
Checking for s/w issues
Having design problems
Mixing agile & traditional
Distributed agile
Selecting agile methodology
Benefits of agile
Needing quality for agile to work
Managing disruption
Using 'done' to track completeness
Developing quality software
Encouraging reuse
Factoring out common stuff
Minimising dependencies
Scrum not working really well
See also

Figure 4.1: Open codes and concepts at an early stage of analysis

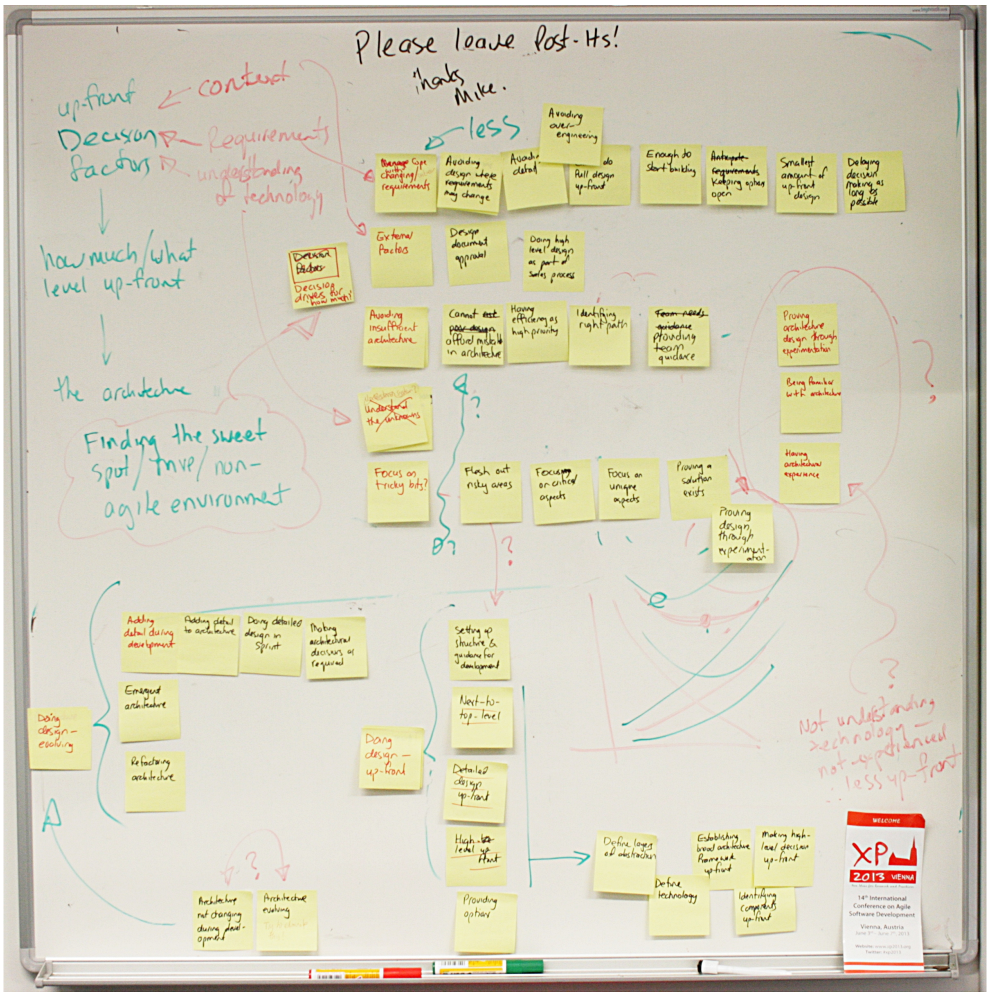
Please leave Post-Its!
thanks Mike.
up-front Decision factors
context
Requirements
understanding of technology
how much/what level up-front
the architecture
Finding the sweet spot / naive / non-agile environment
less
Avoiding over-engineering
Cope with changing/ requirements
Avoiding design where requirements may change
Avoiding detail
Full design up-front
Enough to start building
Keeping options open
Smallest amount of up-front design
Delaying decision making as long as possible
External Factors
Design document approval
Doing high level design as part of sales process
Decision drives for how much?
Avoiding insufficient architecture
Cannot afford mistakes in architecture
Having efficiency as high priority
Identifying right path
Providing team guidance
Proving architecture design through experimentation
Being familiar with architecture
Having architectural experience
Understand the unknowns
Focus on tricky bits?
Flesh out risky areas
Focusing on critical aspects
Focus on unique aspects
Proving a solution exists
Proving design through experimentation
Adding detail during development
Adding detail to architecture
Doing detailed design in Sprint
Making architectural decisions as required
Emergent architecture
Refactoring architecture
Doing design - evolving
Doing design - up-front
Setting up structure & guidance for development
Next-to-top-level
Detailed design up-front
High-level up-front
Providing options
Architecture not changing during development
Architecture evolving
Define layers of abstraction
Define technology
Establishing broad architecture framework up-front
Making high-level decisions up-front
Identifying components up-front
Not understanding technology - not experienced - less up-front
WELCOME
XP 2013 VIENNA
14th International Conference on Agile Software Development
Vienna, Austria

Figure 4.2: Open codes and concepts at an intermediate stage of analysis

Open code	**Source**
architecting as a group activity	P17, P22, P23, P29, P31, P35
customer architect making decisions	P21, P22, P26
disparate groups making architecture decisions	P8
having external design review	P2, P11, P14, P29, P33
having the last word on architecture decisions	P8, P12, P13, P17, P19, P22, P28, P30
senior developers making decisions	P29
whole team making decisions	P17, P28

Table 4.4: An example of the concept *architecture decision makers* evolving from constituent codes

P29 contributed to three of these codes, *architecting is a group activity*, *having external design review* and *senior developers making decisions*. The latter code is included in the sample list of coded data in Table 4.3 on page 89.

As low-level codes were aggregated into higher level codes and concepts through constant comparison, data was typically coded directly against the high level code. Any further contribution made by new data is captured in the memo that is attached to the concept (described in section 4.2.4). For example, the code and concept *guiding decisions* has a number of participants coded directly against it as well as constituent codes that were merged into it, shown in Table 4.5. Incidents were sometimes still coded against the constituent codes themselves after they were aggregated, because the codes were frequently moved around as the concepts evolved, and thus it was sometimes more appropriate to code against individual codes.

Typically, codes that emerged later in the analysis do not include data from early interviews, because the interviews were not revisited, to ensure analysis kept moving forward. Codes were sometimes also aggregated into more than one concept.

Open code	Source
providing guidance (concept)	P8, P10, P11, P12, P22, P30
creating a reference architecture for teams	P33
enforcing architectural principles	P4
enforcing coding guidelines	P4, P5, P10, P31
experienced practitioners providing guidance	P10
giving high level guidance	P3, P4, P5
providing architectural principles	P3, P4, P7, P10, P14
providing assistance to decision making	P4
teaching the team	P30
writing SAD in parallel to development	P4, P7, P10

Table 4.5: An example of the concept *providing guidance* with codes directly coded against it

4.2.4 Memos

Memos record ideas about concepts and categories, and the relationships between them, as they evolve. An extract from a memo, belonging to the concept *architecture decision makers* listed in Table 4.4 above, is:

> *Some customers need to review the team's architectural decisions, or even make the decisions on its behalf. There are many reasons for this: the customer does not want to give the team the power to make their own decisions (i.e. command and control or non-agile customer), they need to ensure the decisions are compatible with other systems (e.g. an enterprise system) (i.e. the team doesn't know the big picture), they don't trust the team (yet) (perhaps also a non-agile customer).*
>
> *Usually these reviews are before development starts, because the customer wants to ensure that the architecture is correct before development starts. This sort of customer is typically non-agile – they do not trust the team, they do not want to empower the team, and/or they*

require them to do a big part of the design up-front so they can do the review.

An agile team that has an agile customer (i.e. the customer has a culture compatible with its own) will either make its own architecture decisions, or will have team representatives on an architecture team that is familiar with the bigger picture (e.g. P10 with his communities of practice, P23 with his informal architecture meetings, P33–36 and their product architecture teams, P8 with leadership duties across two teams)
[...]

After theoretical coding and sorting, the memos were used to form the basis of the written theory.

4.2.5 Categories

Eight categories emerged from the data, which were candidates for being made the core category.

(a) The emergent categories

The eight categories that emerged from the data were:

- making architecture decisions
- architecting up-front (making up-front decisions)
- architecting during development (making emergent decisions)
- architecting strategies (agile architecture strategies)
- architecture decision makers
- architecture timing drivers
- non-architecting roles (other roles of the architect)
- defining requirements

Each category is, in turn, a concept that has been promoted and expanded to encompass a number of other concepts. For example, the category *making architecture decisions* describes how teams make architecture decisions: considering constraints, guiding decisions, mapping requirements, understanding the big picture, and so on. Figure 4.3 shows how the category was formed by promoting the lower-level concept of the same name *making architecture decisions* (itself an aggregation of the substantive codes *analysing*, *modelling*, *researching* and *experimenting* – not shown) to category level and merging in the six other related concepts shown in the figure.

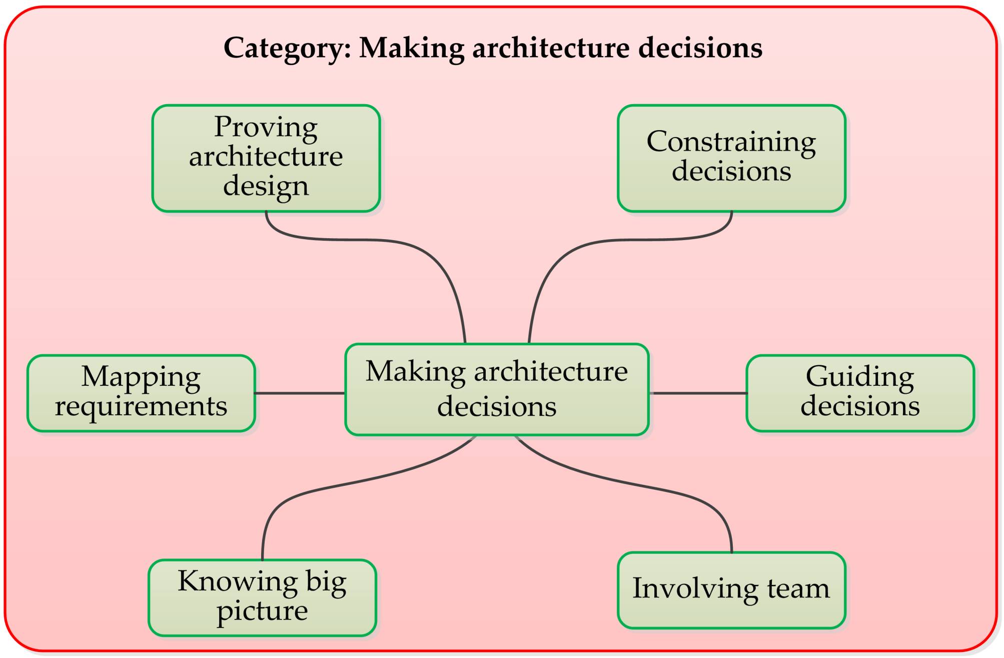
Category: Making architecture decisions
Proving architecture design
Constraining decisions
Mapping requirements
Making architecture decisions
Guiding decisions
Knowing big picture
Involving team

Figure 4.3: A category (red box) emerging from aggregated concepts (green boxes)

(b) Selecting the core category

A common theme among a number of these categories was *architecting*, representing the concept of designing an agile architecture. An agile architecture is one that allows changing requirements (more specifically, the architecturally significant requirements, or ASRs) to be more easily managed, by either being more easily modifiable or by being tolerant of change. I renamed the category, *making architecture decisions* to *architecting* and promoted it to core category. Four other categories that were closely related to *architecting* became properties of the core category: *architecting up-front*, *architecting during development*, *architecting strategies* and *architecture decision makers*. The three other categories, *architecture timing drivers*, *non-architecting roles* and *defining requirements* remained as separate categories, because they were not directly about making architecture decisions.

The categories are shown graphically in Figure 4.4.

(c) The relationship of the categories to the core category

The core category *architecting* describes the process of designing an agile architecture, and describes how some decisions are made up-front, before development starts.

The property of the core category *architecting strategies* represents the strategies that agile teams use to determine how much effort to put into up-front architecture design and how agile the architecture is.

Also in the core category, *architectural decision makers* – loosely called 'architects' or 'agile teams' in this research – represent those who perform the process of 'architecting' in an agile software development project, or who approve the decisions once they have been made. The architecture decision makers may be the whole team, may be a subset of the team, or may even be outside the team.

Outside the core category, *architecture timing drivers* describe the forces (using design pattern terminology [101]) that affect the strategies teams use

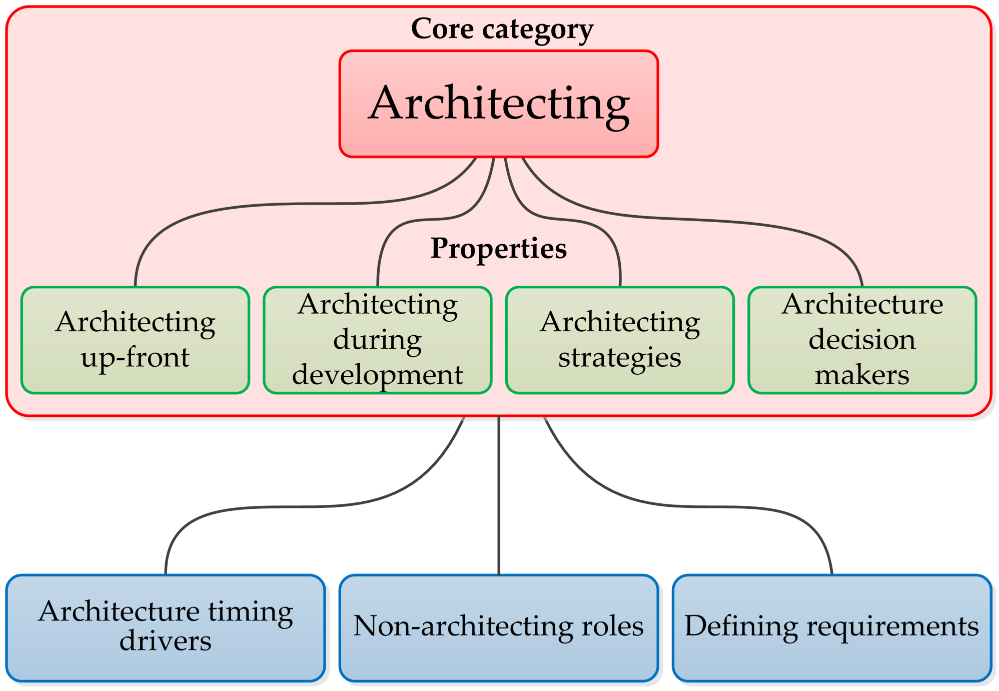

Figure 4.4: The core category (red), its sub-categories (properties) (green) and other categories (blue)

to determine how much architecture to design up-front. By recognising these forces, agile teams can determine how much effort to put into up-front architecture design and how much effort to put into emergent design.

The category *non-architecting roles* describes non-architecture decision-making roles that the architect in a team is often responsible for.

The category *defining requirements* describes the ASRs that must be defined up-front, before any up-front architecture design takes place.

4.2.6 Theoretical coding

It was not immediately clear which of Glaser's theoretical code family fitted these codes. Initially several seemed suitable, at least in part: *process* family (representing a process or processes), *type* family (representing types or classes of phenomenon), *strategy* family (representing a strategy or strategies), *models* family [104] (representing a model shown graphically) and *causal* family (representing cause and effect) [108].

Eventually it became clear that the *strategy* family was the best framework for the codes because it accounted for most of the relationships between categories, and that a central theme of the emergent theory was a set of strategies for designing an agile architecture and determining how much architecture to design up-front.

Figure 4.5 shows a representation of the categories and their relationships drawn on a whiteboard after theoretical coding.

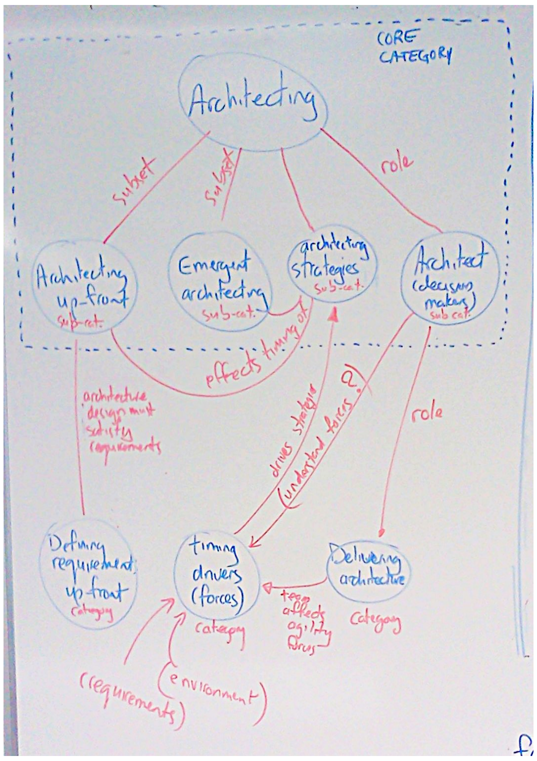
CORE CATEGORY
Architecting
subset
subset
role
Architecting up-front sub-cat.
Emergent architecting sub-cat.
architecting strategies sub-cat.
Architect (decision maker) sub cat.
effects timing of
architecture design must satisfy requirements
drives strategies
(understand forces?)
role
Defining requirements up-front category
timing drivers (forces) category
Delivering architecture category
team affects agility forces
(requirements)
(environment)

Figure 4.5: Categories after theoretical coding

*

The categories and their relationships to the core category, as defined by the theoretical coding, constitute the theory of agile architecture. This theory is the topic of chapters 5 to 8.

Chapter 5

A theory of agile architecture

“The key thing is [the architecture] is 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)

This chapter introduces a *theory of agile architecture* that describes how teams determine how much architecture to design up-front, and how they design an architecture that is able to manage change. An *agile architecture* is both an architecture that supports a team’s agility by being easily modifiable and tolerant of change, and the outcome of an agile process that has a more emergent design with a shorter planning period. Section 9.2 discusses the definition of agile architecture in further detail.

The theory of agile architecture is a high-level theory that explains how teams design an agile architecture, and how they decide how much design to do up-front and how much to leave emergent. The theory is *descriptive* because it describes the experiences of the research participants, rather than *prescriptive*, written as instructions to guide practitioners. While descriptive, it is implicitly assumed that these experiences are also applicable to other teams (see the discussion in section 10.3), and thus it is assumed the theory can be used by other practitioners to help them determine how much architecture to design up-front and how to design an agile architecture, without providing explicit instructions on the steps to take to design an agile architecture (such as in the form of a decision tree). The amount of up-front design effort varies between two extremes, from an entirely emergent architecture with no explicit architecture decision-making up-front to a full architecture designed up-front. Usually the architecture effort is somewhere between these two extremes, with some architecture decisions made up-front and some emergent. An agile team cannot simply make an ad hoc decision to put less effort into up-front architecture design and be sure of having sufficient architectural guidance. The extent to which a team can successfully reduce up-front architecture design effort depends on the system's *context*, consisting of *agile architecture forces*, the attributes that a team must consider when designing an agile architecture. The *agile architecture strategies* determine how a team designs the agile architecture and how much effort it puts into up-front design; which strategies a team uses depends on the forces.

Figure 5.1 shows the high-level relationship between the forces, strategies and architecture design. The agile team (*architect*) determines which *strategies* are used according to the *forces*. The strategies affect the *architecture design*: how much effort the team puts into *up-front architecture design decision* and how much into *emergent architecture design decisions*. Certain requirements – particularly *high-level architecturally significant requirements* – must also be defined up-front to enable the team to make any necessary up-front architecture decisions.

The theory of agile architecture is introduced in this chapter and described in more detail in chapters 6 to 8. In this chapter, section 5.1 defines context and the forces that make up context. Section 5.2 introduces the agile architecture strategies that teams use to design an agile architecture and which determine how much up-front architecture design they do. Section 5.3 discusses how designing an agile architecture affects the architecture design processes. Section 5.4 discusses gathering the project's requirements before development starts, and, finally, section 5.5 discusses the role of the architect in an agile development: who makes the architectural decisions and what other roles the architect plays.

Following this chapter, Chapter 6 describes each of the forces in more detail and chapter 7 describes each of the strategies. Chapter 8 discusses the agile architect role in more detail.

5.1 Context and agile architecture forces

The *context* of a software system is the set of conditions that affect the agile architecture and which teams must consider when designing an agile architecture. The conditions are defined by a set of agile architecture attributes or *forces*:

- F1: REQUIREMENTS INSTABILITY
- F2: TECHNICAL RISK
- F3: EARLY VALUE
- F4: TEAM CULTURE
- F5: CUSTOMER AGILITY
- F6: EXPERIENCE

REQUIREMENTS INSTABILITY (F1) refers to the project having some or all of its requirements undefined at the start of development or changing during development. This force is the main motivation for using agile development methods and for designing an agile architecture.

TECHNICAL RISK (F2) is the presence of technical risk – exposure to a potentially negative outcome because of uncertainty with the technology, the design or the system itself. Technical risk is often caused by architectural complexity brought about by demanding architecturally significant requirements or ASRs (section 2.3.4), and is mitigated by more up-front design to reduce uncertainty.

EARLY VALUE (F3) is the customer's need to derive commercial value from the system being developed before it would otherwise be ready, perhaps in the form of a minimum viable product (MVP) (section 6.3). To provide early value, a team reduces the planning horizon and spends less time on up-front architecture design.

TEAM CULTURE (F4), CUSTOMER AGILITY (F5) and EXPERIENCE (F6) all impact upon the team's agility, rendering it more or less able to design an agile architecture. F4 is a collaborative and people-focused culture based on trust. An agile team culture increases the team's ability to communicate rapidly and hence reduces the time needed to respond to change. F5 is an agile environment, such as the agility of the customer or the team's own organisation and any other stakeholders. As well as the team being agile, the environment must support the team's agility. F6 refers to the team's architectural and technical experience. Experienced team members are more able to use tacit knowledge to make faster and fewer explicit decisions, and so can reduce up-front effort and respond to change faster.

The impacts of forces F2, F4, F5 and F6 are continuously variable – they represent values on a continuum. F1 and F3, on the other hand, simply represent the presence or absence of the force: whether or not the requirements are unstable, and whether or not the customer requires early value, respectively.

The forces are described in detail in Chapter 6.

5.2 The agile architecture strategies

Teams use a selection of agile architecture strategies to determine how they design an agile architecture and to guide the team as to which architecture decisions should be made up-front and which should be left as emergent. Teams choose the most appropriate strategies to address the forces that make up the system's context.

The strategies are:

- S1: RESPOND TO CHANGE

- S2: ADDRESS RISK
- S3: EMERGENT ARCHITECTURE
- S4: BIG DESIGN UP-FRONT
- S5: USE FRAMEWORKS AND TEMPLATE ARCHITECTURES

RESPOND TO CHANGE (S1) is the key strategy for designing an agile architecture, which is modifiable and tolerant of change. Teams increase modifiability by keeping designs simple, proving the architecture with code iteratively and following good design practices. Teams increase the architecture's tolerance of change by delaying decisions and planning for options.

Up-front effort can be reduced by keeping design simple, proving the architecture with code iteratively, and by delaying decisions.

ADDRESS RISK (S2) is a strategy that uses more up-front design to mitigate technical risk as early as possible. S2 is in tension with S1's tactic of delaying decisions: teams have to balance mitigating risk with the architecture's ability to respond to change.

EMERGENT ARCHITECTURE (S3) is a strategy that results in an emergent architecture, with as few decisions made up-front as possible – typically simply selecting the technology stack and the top level architectural styles and patterns. S3 is a strategy that takes RESPOND TO CHANGE (S1) to the extreme, with all decisions delayed, whether or not the requirements are likely to change. S3 does not mitigate any risk up-front, and is therefore less suitable for complex systems with demanding ASRs.

BIG DESIGN UP-FRONT (S4) is a strategy in which most architecture decisions are made up-front. S4 is most likely used by agile teams that have a non-agile customer (or manager) who requires the team to complete the design before development starts. S4 reduces the team's ability to respond to change, and is mutually exclusive with S3 (EMERGENT ARCHITECTURE).

USE FRAMEWORKS AND TEMPLATE ARCHITECTURES (S5) reduces the effort required for architectural design by providing 'precooked' architectural solutions in the form of frameworks, templates, reference architectures and standard off-the-shelf libraries and plug-ins. Frameworks also reduce complexity and hence risk, and make it easier to change subsidiary architectural decisions. This is particularly important for agile development because the reduced effort allows a team to respond to change more quickly and hence become more agile.

The strategies are described in detail in Chapter 7.

5.3 Architecture design

No matter which strategies are used and how emergent the design is, there are certain architecture decisions that must be made up-front, before development starts; they cannot be changed later without major rework. Examples of decisions that must be made up-front include selecting the technology stacks and the highest level architectural patterns. These minimum up-front architecture decisions are described in the discussion on S3 (EMERGENT ARCHITECTURE) in section 7.3.

If the project is a green field (new) application, up-front decisions simply mean those that are made before any code to produce functionality is written; if the project is extending an existing application (such as developing for a new release) then up-front means any architecture decisions that are made for this release before development on features for this release starts; any earlier decisions and development act as constraints on the current project or are prior decisions that need to be reviewed and potentially updated.

Often, the architecture design practices and techniques (such as decomposition, partitioning and modelling) used in agile development are similar to those used in plan-driven methods; they are not discussed in any detail in this thesis. There are a couple of important differences however: firstly, the agile architecture is designed with responding to change in mind (through modifiability and being tolerant of change), and secondly, because many architecture decisions are emergent, teams often prove their architecture by building it and seeing if it works, rather than proving it in advance through analysis. This means less modelling and analysis when designing an agile architecture. For example, proving a design with code iteratively is a tactic of S1 where the first cut of a design is one that *could* work, not necessarily *will* work.

5.4 Up-front requirements

While it is usually very difficult to define all requirements in detail before starting development, a team would usually have an understanding of certain requirements up-front. At the highest level, teams must have an understanding of the business's vision or the problem the system is addressing, and must have the system-wide ASRs defined, so that the team can make the highest level architectural decisions. The more effort a team puts into up-front architecture design, the more requirements the team must know.

REQUIREMENTS INSTABILITY (F1) is related to up-front requirements. When requirements are unstable, the team uses RESPOND TO CHANGE (S1) to design its agile architecture. One of the tactics of S1 is to delay decisions until the last possible moment, when the requirements are better understood.

5.5 The agile architect

The role of the *architect* is that of architecture decision maker – designing the architecture and choosing the agile architecture strategies to be used.

5.5.1 The architecture decisions makers

In an agile development team, the architecture decision maker is usually a shared role, varying according to the needs of the team, their customer, the relationship between the team and the customer, the system being built, and system's relationship with other systems. Architecture decisions may be made with:

- WHOLE TEAM CONSENSUS
- APPROVAL WITHIN THE TEAM
- APPROVAL OUTSIDE THE TEAM
- DECISIONS MADE OUTSIDE THE TEAM

WHOLE TEAM CONSENSUS: Decisions may be made by team democracy, with the whole team agreeing on the architecture by consensus. This option of architecture decision-making is often used by very small teams where agreement is possible and where all software engineers are skilled and experienced enough to understand the architectural issues. Consensus has the advantage that the whole team has full understanding and buy-in of the architecture.

APPROVAL WITHIN THE TEAM: The whole team or most of the team is part of the architecture design process, but decisions may require approval within the team by a single team member who understands the business problem and the overall architecture, and can take a casting vote to sort out any disagreement between team members. Having a single team member approve design decisions is useful where consensus is not practical and where not all of the team is sufficiently skilled or experienced to make architecture decisions.

APPROVAL OUTSIDE THE TEAM: Teams that are developing a system that is part of a larger system – such as an enterprise system or a product line – may be required to have their decisions approved outside the team. The team still does the architecture design, but the design must be approved by architects outside the team. These external architects have an overview of the overall business objective and the overall system architecture that the team may not have.

DECISIONS MADE OUTSIDE THE TEAM: Finally, architecture decisions may be made outside the team, out of the hands of the team altogether. This most often happens when the team's customer is not agile and does not support the team's agility. Customers may also make architectural decisions such as the technology stack and high-level architectural styles when they specify certain requirements, such as requiring compatibility with existing systems or by selecting ISVs who are aligned with a particular technology vendor.

5.5.2 Other roles of the architect

A member of an agile team who makes architecture decisions is also an active member of the team; they often also play a number of other roles. These roles may include:

- being the TECHNICAL LEAD/'ÜBER DEVELOPER'
- knowing the BIG PICTURE
- creating a SHARED MINDSET
- creating and enforcing DEVELOPMENT GUIDELINES
- driving the DEVELOPMENT METHODOLOGY

Being a TECHNICAL LEAD/'ÜBER DEVELOPER': the architect is typically a senior software engineer, and is therefore able to solve the team's development problems and do the 'tricky bits.'

Knowing the BIG PICTURE: knowing how the system fits with the needs of the customer allows the architect to better understand the architecture that is needed. This role shares a lot in common with the business analyst role.

Creating a SHARED MINDSET: the architect needs to communicate the architecture to the team and other stakeholders, so that they have a good understanding of what is being built.

Creating and enforcing DEVELOPMENT GUIDELINES: development guidelines ensure consistent style and quality across the system.

Driving the DEVELOPMENT METHODOLOGY: architects understand the relationship between the architecture and the development process, and therefore may be able to perform an agile coaching role, driving the development process and methodology.

The architect role is described in more detail in Chapter 8.

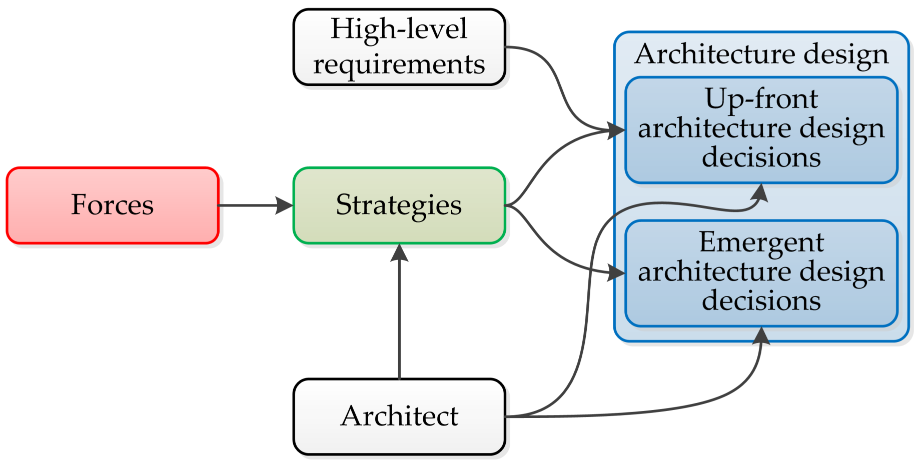
High-level requirements
Architecture design
Up-front architecture design decisions
Forces
Strategies
Emergent architecture design decisions
Architect

Figure 5.1: The relationships between the forces, strategies, requirements and architecture design. Arrows represent dependencies between the elements.

Chapter 6

The agile architecture forces

"It just really depends on the context" (P17, manager/coach)

The agile architecture forces describe attributes that agile teams must consider when designing an agile architecture and deciding how much effort to put into up-front architecture design. The particular combination of forces that acts upon the system forms that system's context.

There are six forces described in this thesis:

- F1: REQUIREMENTS INSTABILITY
- F2: TECHNICAL RISK
- F3: EARLY VALUE
- F4: TEAM CULTURE
- F5: CUSTOMER AGILITY
- F6: EXPERIENCE.

These forces may be external to the team, such as EARLY VALUE (F3), or they may be resultant of the team itself, such as EXPERIENCE (F6).

The relationships between the forces and the agile architecture strategies, the actions the teams take to address these forces and which affect architecture effort, are summarised in Figure 6.1. Each force and its relationships are described in this chapter, while the strategies are discussed in Chapter 7.

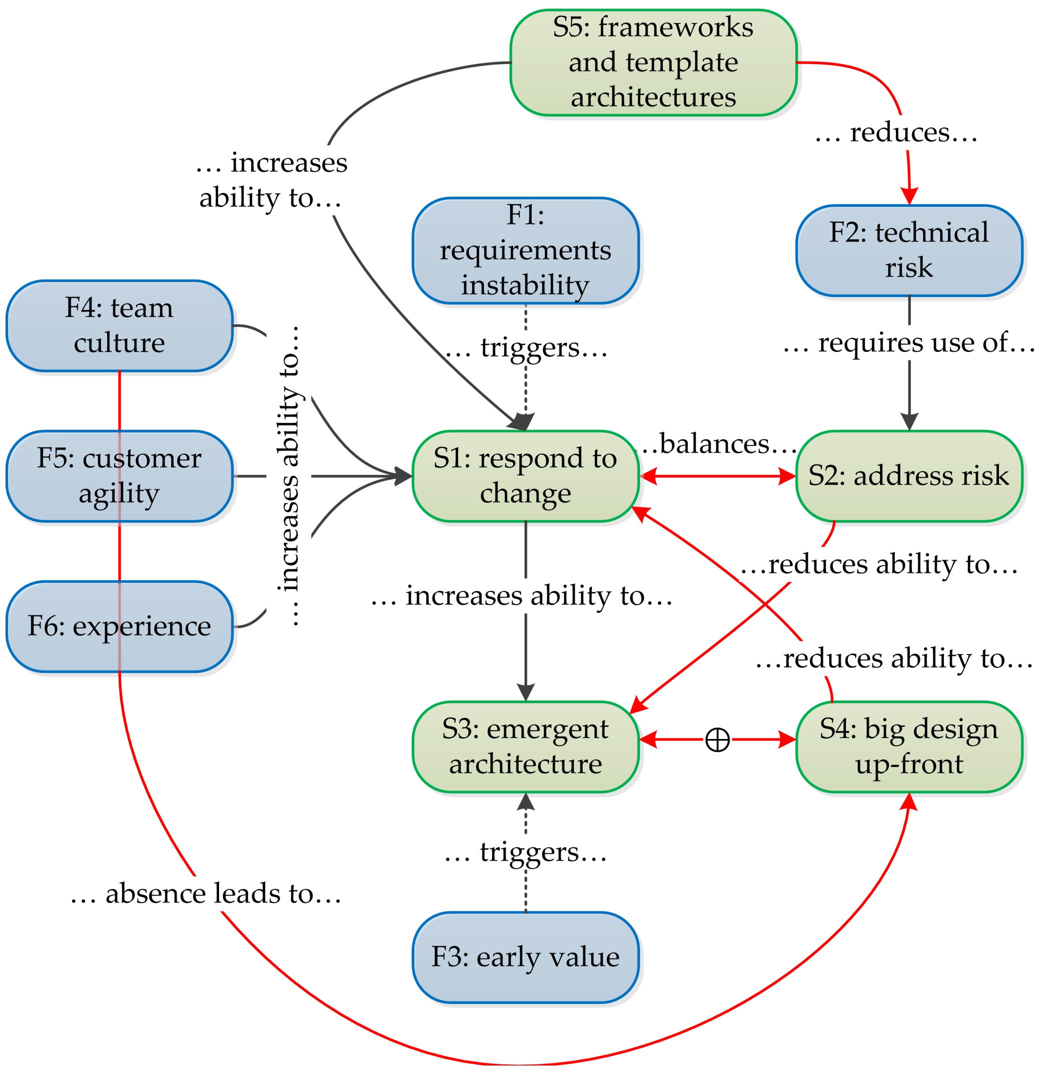

Figure 6.1: The relationships between the forces and strategies. The blue-coloured boxes represent forces, and the green-coloured boxes strategies. Arrows represent dependencies or causal relationships: a change in the independent force or strategy causes either a positive change (solid black line) or a negative change (solid red 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 ⊕ represents mutual exclusion.

6.1 F1: Requirements instability

DEFINITION OF F1: REQUIREMENTS INSTABILITY

Requirements instability is some or all of the requirements of the software system being unknown when development starts or changing during development.

F1 refers to unknown requirements or changing requirements. Teams with unstable requirements generally prefer to do less up-front design because that effort is wasted when ASRs change or become better understood.

Participants all reported unstable requirements to some extent, whether they were part of a team working on a relatively stable redevelopment project or a 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)*

Teams know and accept that requirements will change and evolve:

> *"It was almost as if one of the requirements was we need to be able to change the requirements." (P8, lead developer)*

Developers of business systems in particular are often subject to a continuously changing environment:

> *"Business is going on and new ideas are coming up and new things are being done." (P30, consulting architect)*

In fact, one organisation's whole business case revolved around continuously changing and improving:

> *"We market ourselves as [being] a whizzy online power company – you have to keep on being whizzy, you have to keep being new and exciting and changing." (P24, customer)*

Unstable requirements can be caused by *incomplete requirements* – perhaps where the customer has not decided what they want before development starts – and by *changing requirements* – where the customer changes their mind about what they want.

6.1.1 Incomplete requirements

Incomplete requirements are those that cannot be not fully defined prior to development starting. They are often caused by the customer not knowing what they want, or coming up with new ideas about what they, or their end users, want:

> *"So the customer would come up with the greatest [idea]... 'How about we do this? This is a great idea!' We'd go, 'yes, that's a great idea,' and add it to the backlog." (P17, manager/coach)*

Requirements may be fairly stable, but it may not be possible to develop a complete understanding of those requirements or what would be involved in implementing them until after development has started. A full understanding comes later during development:

> *"It would certainly be the case that they [the requirements] would often be better understood [as development progressed]." (P25, team lead)*

Frequently it is simply not possible for a team to have a full understanding of the requirements prior to starting development because the customer does not know what they want:

> *"And sometimes what happens is the customer doesn't know what they want, the product owner doesn't know, so we develop something and show [them]..." (P23a, engineering manager)*

– or they may not be able to clearly describe what they want:

> *"No matter what you write down on a piece of paper, until you actually see it in front of you, you cannot imagine it in enough detail to get it right." (P24, customer)*

and

> *"It was only after mocking something up, saying, 'we are pretty sure this is what you [the customer] are after,' and then going through that review process, maybe each week or something, that you'd really understand, 'no, we actually wanted that bit to be over there, or we wanted this feature in Flash'... and it was only through that [process] we were able to say, 'ok, now that we understand the finer detail we can actually build that and satisfy a series of tests with that, knock that one out then move on to the next bit.' " (P25, team lead)*

– or because they are ambiguous or not clearly defined:

> *"Well essentially they [the customer] threw every feature that they could remember from any of these fifty systems [that we were replacing] at us, and they weren't really well understood, really. [...] You overlay them and realise that some of them conflict and some things are just different ways of doing the same thing." (P30, consulting architect)*

Even if requirements can be defined up-front, they may change during development (section 6.1.2), wasting any earlier attempts to define requirements.

Some agile software engineers dislike the term 'project' because of its connotations of a well-defined scope with a fixed delivery date:

> *"I'm deliberately trying to eradicate the idea of a project; I think it's defunct in the agile world. I think that the idea of batch size is key and by doing large pieces of work that we think of as a 'project' we break one of the key [agile] paradigms." (P29, development manager)*

Rather, agile work could perhaps be viewed as a continuous stream or flow of work delivering regular value:

> *"We tend to think from a value-flow perspective, to say, how well do we understand the most valuable features, or feature sets, for the user community. Do we understand what's truly most valuable to them? And the answer to that is yes, we know what they care most about. They care about this, this and this." [...] "And essentially with these value streams you end up with a value network, and that gives you an idea of, if that is the flow of value, which way you should structure your work." (P10, coach)*

With this view of agile workflow, even defining a full set of requirements – much less defining a full set up-front – is often not possible:

> *"You [the interviewer] are framing the question as if 'the requirements' are a total set that can even be known." (P10, coach)*

(Note the term 'project' is also often used to loosely denote agile work flows, and is used in this way in this thesis.)

This situation is frequently true even when requirements are relatively stable. P13's team was redeveloping a system, with the new system being a very similar copy of the old system functionally. They were not able to understand the intricacies of the system and its work flow rules before starting development, even though they did not change much:

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

P13's team, working in the medical industry, was not able to understand the requirements in advance of development:

> *"[The] problem was there wasn't documented requirements for the old system, and – perhaps more problematically – is the detailed wrinkled little business rules. Their previous vendor said to us at*

> *one point, there's stuff in here, stuff in the logic that affects only two or three doctors out of the fourteen thousand that work in here, so there's little wrinkly bits of the logic, and originally the plan was that the [customer] would get their old vendor to document these rules. That never happened..." (P13, architect)*

This situation was made worse by the development team not being able to examine the old system due to data confidentiality...

> *"...and so that made it quite difficult!" (P13, architect)*

Demonstrating early and releasing frequently provides more opportunity for the customer and end users to determine the requirements at that time.

6.1.2 Changing requirements

Requirements frequently change during development, perhaps because the customer changes their mind or because of feedback from end users:

> *"An agency [customer] can do a U-turn and say, ok, I am going to do this from now on." (P23, engineering manager)*

and

> *"Maybe that one client that was yelling out loudest for a particular piece of functionality has moved on, or is happy, or God knows what." (P3, development manager)*

Sometimes changes are brought about by the addition of new features:

> *"So that's when the sales team will see an opportunity and they'll push for this change." (P23, engineering manager)*

A number of participants noted changes were frequently imposed upon them by third parties such as regulators:

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

Changing usage patterns can have a big impact on development. Higher usage patterns often mean performance suffers unless the system is redesigned to ensure it can maintain the required performance levels under that load. Often it is impossible for the team or their customer to know how their system will be used once it goes live:

> *"[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. You can't afford to have like a second life where, you know like Second Life the system, you deploy it virtually and see how people interact with it there, and only then deploy it in real life!" (P10, coach)*

If successful, usage of the system will be higher than anticipated:

> *"Since we were working with a more popular travel website, it turned out there was more traffic." (P21, manager/coach)*

and

> *"The new requirement was a higher volume of traffic." (P8, lead developer)*

Sometimes the requirements themselves do not change, but their priorities do:

> *"[Changing priorities] often mean that stuff that we thought we were going to do we ended up not doing, other things had to come in." (P7, business analyst)*

and

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

and

> *"They [the customer] have got to have this feature in at this time to coincide with a whole lot of marketing that's going to be on TV that nobody's told us about, and so we're going to do that now. That would totally throw the cat amongst the pigeons [in a non-agile methodology]. But you'd still have to react to that and fit things into your cycle." (P25, team lead)*

Most participants did not invest a lot of time on gathering requirements prior to development starting, because when requirements change this effort is wasted. Participants would therefore only define the highest level requirements, defining the details when they are needed.

Delays caused by gathering detailed requirements increases the chance those requirements will change:

> *"The moment that you do that [put too much detail into your requirements] you fall into the trap of trying to do all the requirements analysis upfront. [If you do] there's a fat chance that by the time you get around to starting the work your world has changed." (P3, development manager)*

By starting development and demonstrating the product to the customer early in development, developers are able to get feedback from the customer early:

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

and

> *"Because it's more important for the customer to see [a demonstration] within a week, rather than for us to go and spend a month or three more weeks building the same thing, but the customer not even being sure in the first place whether anyone actually wants the feature." (P16a, CEO/chief engineer)*

Early feedback from the customer means the team can increase delivered value by responding to that feedback.

*

Figure 6.2, a subset of Figure 6.1, shows the relationships between F1 and other forces and strategies. F1 motivates the use of RESPOND TO CHANGE (S1) – that is, having unstable requirements is a trigger for designing an agile architecture – and S1 is in turn required if the team wishes to design an EMERGENT ARCHITECTURE (S3). This relationship does not imply a relationship between the level of requirements instability and the level of a team's ability to respond to change.

On the other hand, the BIG DESIGN UP-FRONT (S4) has a negative impact on the team's ability to RESPOND TO CHANGE (S1).

6.2 F2: Technical risk

DEFINITION OF F2: TECHNICAL RISK

Technical risk refers to exposure to potentially negative outcomes caused by problems and uncertainty due to the technology suite, the architecture design or the system itself.

Risk is exposure to potential negative outcomes. Technical (or engineering) risk is the risk of failure caused by problems within the technology suite, the architecture design, or the system itself [17, 91].

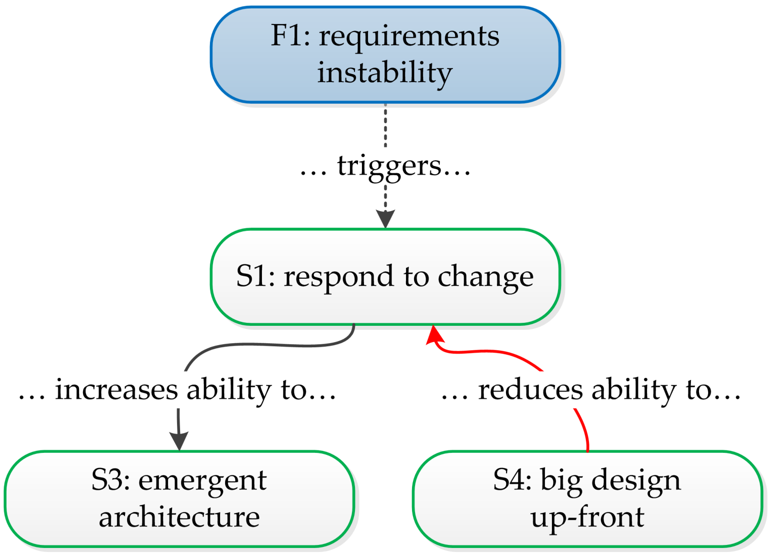

Figure 6.2: The relationships between REQUIREMENTS INSTABILITY (F1) and other forces and strategies.

Technical risk has a big impact on how much architecture design teams do, and is caused by having demanding architecturally significant requirements (ASRs, section 2.3.4) that lead to a complex architecture, by unique problems that have not previously been solved, and by unknown or new technology:

> *"If there's high complexity, integration, there's high risk." (P32, software development director)*

and

> *"Risk comes from complexity." (P33, product architecture team leader)*

and

> *"Is it something that's new? How well did we understand it as an organisation in that functional area?" (P33, product architecture team leader)*

The most significant contribution to risk is complexity, which has a direct impact on teams' up-front design effort:

> *"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)*

and

> *"Fundamentally though the smallest amount [of up-front architecture planning] possible is still quite a lot, so it really depends on how complex the thing is you're trying to build." (P36, development unit manager)*

Complexity is discussed in the remainder of this section.

6.2.1 Causes of complexity

Complexity can be caused by the system having demanding ASRs, by having many integration points and by having to interact with legacy systems.

> *"So if [the requirements] are architecturally significant, every single one of them, then that's complex." (P32, software development director)*

These causes are explored below.

Demanding ASRs

Demanding or challenging ASRs mean it is more critical to make correct architecture decisions, because there is a small solution space or because they require many trade-offs. Multiple stakeholders can also lead to demanding ASRs and trade-offs if they have conflicting needs. Demanding ASRs increase complexity and the technical risk. Participants described demanding ASRs leading to a complex architecture:

> *"Highly demanding NFRs [non-functional requirements, or ASRs] 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)*

and

> *"Demanding QAs [quality attributes] can put a lot of stress on a system's architecture and introduce considerable complexity without it necessarily being a 'big' system." (P31, enterprise architect)*

Demanding ASRs therefore increase the up-front architecture design effort required.

Legacy systems

Legacy systems are older systems that were created using now outdated techniques and technology [30], and are no longer being engineered or actively maintained but rather are simply patched as requirements change [165] without consideration of the technical debt being incurred [1]. These patches and technical debt add to the system's complexity [151]:

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

and

> *"The systems are quite antiquated, so we need to figure out how to interact with them." (P36, development unit manager)*

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

Integration

Participants 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:

> *"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)*

and

> *"[Complexity] is usually to do with the complexity of the interactions between the parts of the things in the domain [...] It's when you've got a lot of moving pieces in place that it becomes more complex. If it's a relatively small number of moving pieces, then it's not so complex." (P36, development unit manager)*

Integration with other systems require data and communications to be mapped between the systems, which adds to the up-front effort to ensure integration is possible with the technologies being used.

6.2.2 Size and complexity

The size of a system may be measured directly by using a metric such as lines of code, number of components or function points, or indirectly using a metric such as the project's budget or development time. The size of a system is frequently considered by the literature as a factor in determining how much up-front architectural effort is required (section 2.4.3). Participants in this research however did not consider size to be a direct factor in the amount of up-front design required:

> *"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)*

and

> *“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 system that does not have demanding ASRs, does not require a lot of integration and does not involve legacy systems is not likely to be complex, and therefore will require less up-front architecture effort, independent of size.

Specifically, a large system that can be implemented entirely using a framework’s standard components and libraries with an acceptable level of risk will require less effort than a similar sized complex system. For example, P27’s team was building a large system that could be implemented entirely using Ruby on Rails. They were therefore able to build the system with very little up-front design:

> *“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. [...] We don’t have architectural discussions – we don’t need to – the problem’s [already] been solved.” (P27, CEO/agile coach)*

Another participant, P26, described a .NET system that he built as having an ‘enterprise-grade architecture’ that was ‘too big’ for the system being built: it had more layers and levels of abstraction than required. Despite this extra size, he believed the extra complexity was minor, describing the additional up-front effort required for this larger architecture designed for a larger system as being minimal, with most of the extra effort coming during development when getting new team members up to speed with the architecture.

Conversely, even a small system may require a lot of planning if it is complex:

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

The impact of complexity is therefore more important than the impact of size on architecture effort.

*

The relationships between F2 and other forces and strategies are shown in Figure 6.3. The risk associated with complexity is mitigated with increased up-front design (ADDRESS RISK, S2). USE FRAMEWORKS AND TEMPLATE ARCHITECTURES (S5) reduces complexity, simplifying the design and reducing the risk and effort required for common problems. The figure also shows the three contributors to risk: complexity (caused by demanding ASRs, integration and legacy systems), having to build a unique solution, and using unknown (or unfamiliar) technology. Size is not a direct contributor.

6.3 F3: Early value

DEFINITION OF F3: EARLY VALUE

The early value force refers to a customer's need to gain value from a system or product by using that system before the system is commercially viable or complete.

F3 refers to a customer's need to gain value (rather than simply provide feedback) from a system or product's highest priority requirements

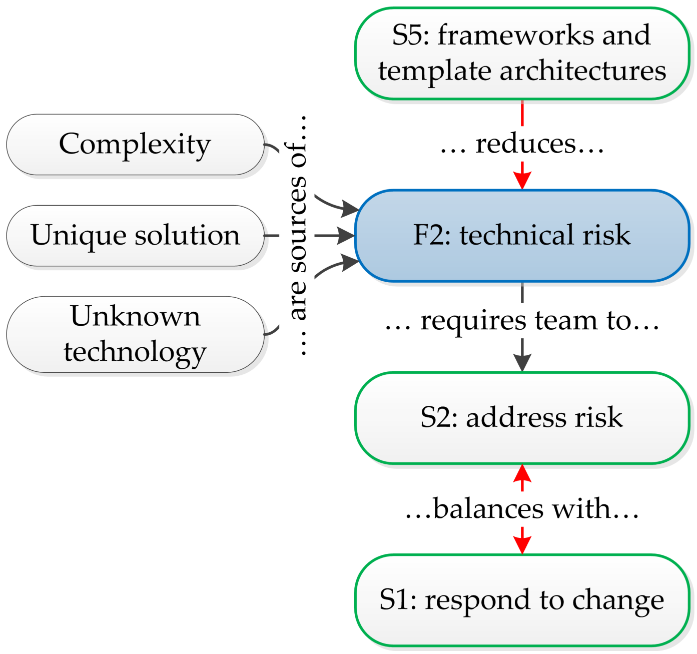

Figure 6.3: The relationships between TECHNICAL RISK (F2) and other forces and strategies

before enough functionality has been implemented to make the product commercially viable. Teams who deliver early value reduce the time to the first release and hence spend less time on up-front architecture design; the extreme case of reducing up-front architecture design is the EMERGENT ARCHITECTURE (S3) – perhaps as a working prototype or a *minimum viable product* (MVP) [192], a marketing experiment that has limited functionality and is designed to determine which features are desirable, rather than be a fully functional version of the software.

Early value is frequently required by businesses operating in a dynamic commercial environment whose business plan 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)*

and

> *"If they [build] the big system, then they will never reach their end customer and make their money." (P22, senior manager)*

Early value is also important for start-ups with limited cash who do not have the ability to pay for lengthy up-front architecture design and development prior to the first commercial release. If the product or service being developed is (or will be) a mass market product with many end-users (rather than an internal product), the customer can start generating value in the form of cash flow by releasing the product or service early to early adopters:

> *"We're a start-up. We didn't have twenty million dollars to spend on day one." [...] "So then you know it [the system] is not perfect, and you accept that it's not perfect, but then you say, well, that got me off the ground." (P15, customer)*

and

> *"Our business is based on first add value then the revenue comes in." (P16a, CEO/chief engineer)*

Teams reduce up-front design effort by reducing the planning horizon – how far ahead the team considers (high level) requirements for the purpose of architecture planning. In the extreme, the planning horizon is reduced to the current iteration only – the team does not make any allowance for requirements beyond the current iteration. In this case the architecture design is effectively totally emergent – an EMERGENT ARCHITECTURE (S3). F3 is a motivator, or trigger, for S3.

F3 is not usually relevant to systems that do not have paying end-users, such as internal and non-commercial systems.

*

Figure 6.4 shows the relationships between F3 and other forces and strategies. F3 triggers the team to build an EMERGENT ARCHITECTURE (S3). As noted above, for a team to build a system or product with an emergent design, the architecture should be highly responsive to change (RESPOND TO CHANGE, S1), and therefore must avoid risk that increases up-front design (ADDRESS RISK, S2); the system cannot both address risk and deliver early value (S2 and S3, respectively) at the same time. Thus a system being delivered early should have a simple and known solution, without demanding ASRs, many integration points and interaction with legacy systems. EMERGENT ARCHITECTURE (S3) and BIG DESIGN UP-FRONT (S4) are mutually exclusive.

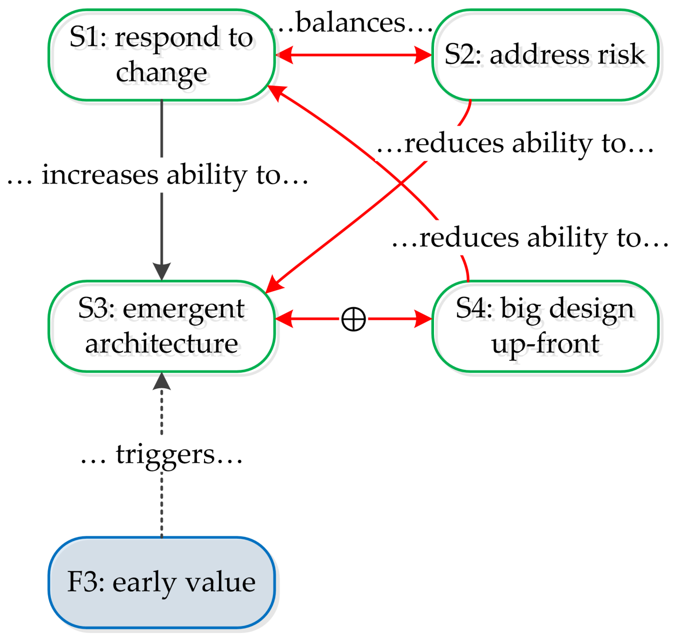

Figure 6.4: The relationships between EARLY VALUE (F3) and other forces and strategies

6.4 F4: Team culture

DEFINITION OF F4: TEAM CULTURE

The team culture force refers to the agile culture of the development team. A highly agile team culture helps the team become more agile (that is, respond to change faster), while a non-agile team culture impedes the team's agility.

A team culture that is people-focused and collaborative lies at the heart of agile methodologies, because it greatly reduces the feedback cycle by reducing the time required to relay information [72]. A team with a highly agile culture will almost certainly have team members who are physically close together, will use face-to-face communication instead of written documentation, and will have team members who are trusting and 'amicable' [72]. For example, P12's team had a very open and communicative culture:

> *"There was a very good culture; we would often go out after work and have a beer together. We got on very well. We could be full and frank in our discussions and planning and nobody would get too offended. Nothing that couldn't be sorted out over a beer after work." (P12, senior developer)*

A team's agility is determined by its ability to respond to change; a more agile team will have a shorter feedback cycle, and will do less planning to reduce the delay in getting initial feedback.

6.4.1 The people-focused and collaborative team culture

A people-focused and collaborative culture consists of team members who are able to work closely together and communicate freely with the customer:

> *"You need people who will ask questions, will welcome feedback, who are happy to get on Chat and literally have a chat, who aren't scared of talking to the client, who think really well." (P15, customer)*

P29 and P31 believed that team culture has a much greater impact on productivity than technical ability and knowledge of agile practices:

> *"But the actual productivity changer is the actual teaming thing." (P29, development manager)*

and

> *"You could have as much technical knowledge [as you like] about how to build something, but if you can't get the people working together then you may as well give up. You're not going to get the outcomes you expect. The technical aspects are dead easy to deal with." (P31, enterprise architect)*

Likewise, P30 saw communication skills as a key skill:

> *"One thing we were seeking was good written and verbal communication skills. The written skills I took as an indicator into the ability to think and express themselves clearly. Of course being able to talk to somebody is a fundamental skill!" (P30, consulting architect)*

In contrast, a poor agile culture is one where everyone is working as an individual with their assigned tasks and their own output. This can lead to misunderstandings, conflict and a blame culture when things go wrong:

> *"We tell people, don't worry about it, nobody's going to blame you, even if we make a mistake. [...] 'He changed my code, it was right, he changed my code and the defect came.' Why weren't we going and asking him, 'why did it cause the defect?' This is one of the cultural problems. [In agile] who caused it or whose mistake it is doesn't make a difference. There's no blame. If the defect is there, pick it up and fix it!" (P22, senior manager)*

To help avoid these problems, an agile team focuses on the output of the team as a whole rather than the output of the individual:

> *"That's sometimes a challenging dynamic to work with, [...] there is still in many places this emphasis on individual rather than team performance and assessment and reward." (P10, coach)*

A people-focused and collaborative team culture plays an important role in the team's agility.

6.4.2 Trust

Trust is hugely important for a people-oriented and collaborative culture:

> *"Trust is 99.9 per cent of [success]" (P15, customer)*

Trust holds the team together and enables it to work together and to produce team-oriented results:

> *"Trust is a really powerful glue for the fabric of the team... that's a fundamentally critical thing that you have to get right." (P30, consulting architect)*

and

> *"You need good people who have got buy-in to that [agile] process, trust, and I think you need to run a good tight technical ship." (P27, CEO/coach)*

Without trust, teams tend to operate as a collection of individuals, with little team focus and relying on more explicit forms of communication.

6.4.3 Team size

Collaboration works best in small teams, where every team member can be directly involved in all discussion; large teams require more formal communication methods, more structure and more planning than small teams [155]:

> *"I was talking about hundreds of people – you're not just going to throw them in a great hall and say, 'go do work.' You're going to have some sort of structure." (P10, coach)*

and

> *"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)*

and

> *"The thing that is problematic with bigger teams is establishing everyone on a common page, because there are more people to talk to, more ideas on the table, and so there's a cost in the communication of the team." (P29)*

More formal communication requires more time and effort, reducing the ability of the team to be responsive to change and hence its agility.

6.4.4 Agile experience

The agile experience of the team members can affect team agility and how much architecture design the team does: a people-focused and collaborative agile culture does not come instantaneously; it comes with experience and practice as the team becomes more experienced working together:

> *"You can learn agile [processes] but thinking agile is not so easy. It comes as part of culture. It's a continuous journey, people have to get accustomed to the culture." (P20, coach and trainer)*

Converting a whole company to agile and spreading the agile culture also takes time:

> *"You really need to have an agile mind-set throughout the organisation, and that takes years." (P3, development manager)*

A team new to agile will usually struggle to be successful without the guidance of a predefined architectural plan, but as the team becomes more experienced and develops an agile mind-set, team members should become more comfortable working in an unpredictable environment and less planning:

> *"It's very difficult for [inexperienced developers] to think in that model. The culture will be different. They will find it very difficult to start without having a concrete design in place. [...] 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)*

Conversely, one participant (P23) reported they would do more planning if they were more experienced in agile, but his team had come to agile from an ad hoc, process-free methodology background rather than a traditional plan-driven background. He believed more planning would have solved some of the technology problems they were facing.

*

Figure 6.5 shows the relationships between F4 and other forces and strategies. F4, along with being in an agile environment (F5, CUSTOMER AGILITY) and having architectural and technical experience (F6, EXPERIENCE), increases the team's ability to design an architecture that can RESPOND TO CHANGE (S1): a team is more able to respond to change if it has an agile team culture. On the other hand, a non-agile culture may increase the team's need to do a BIG DESIGN UP-FRONT (S4), which in turn has a negative impact on the team's ability to use S1.

S1 is a prerequisite of EMERGENT ARCHITECTURE (S3): for a team to design a totally emergent architecture, it must be highly agile.

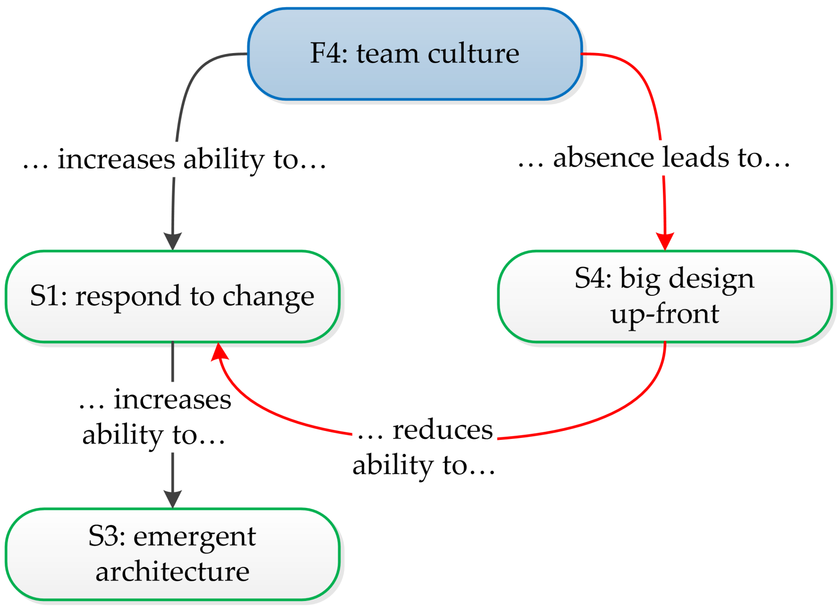

Figure 6.5: The relationships between TEAM CULTURE (F4) and other forces and strategies

6.5 F5: Customer agility

DEFINITION OF F5: CUSTOMER AGILITY

Customer agility refers to the relationship between the development team and its environment – the customer or manager and other stakeholders – and how that affects the team's agility and helps the team become more agile.

The team's environment, such as the customer's organisation, has an important impact on the team's ability to be agile. The customer needs to be sympathetic with and support the team's agile needs, such as its need for regular feedback and to be self-organising; a customer that is agile will greatly improve the team's agility:

"We've become the same team." (P27, CEO/coach)

– while a non-agile environment (such as a customer used to process and traditional styles of management) can hinder a team's ability to be agile:

> *"There is the expectation that, 'I've given you the requirements, I'm a busy person, go away and tell me when it's done.' " (P25, team lead)*

6.5.1 The relationship between the customer and the agile development team

Ideally there will be a good cultural match between the team and the rest of the customer organisation: as well as the team needing a people-focused, collaborative culture that encourages agile (F4, TEAM CULTURE), the customer organisation – whether an external customer or the organisation the team is part of – needs a similar culture that supports the team:

> *"We've become the same team. That removes a lot of the tension, streamlines the process massively." (P27, CEO/coach)*

Conversely, a highly-agile team will be less effective in a highly-process oriented organisation [203] that prefers extensive planning and formal communication [72]:

> *"I've always said customer organisations need to look at their vendors and determine the cultural compatibility." (P31, enterprise architect)*

Participants suggested large organisations tend to prefer more traditional process driven methods, while small organisations prefer to avoid process. Thus the relative sizes of the organisation is a strong indicator of whether or not their cultures match:

> *"Some places just cannot do a 'small scale' project, they just seem to have to do everything large scale. Likewise big corporates such as IBM or HP always seem to need a lot of process, however your*

> *average 'back of the garage' vendors will often struggle to do a lot of planning." (P14, architect)*

and

> *"Scale is one factor in [cultural compatibility]. So dealing with a 300 000-person multinational when you've got a thousand people, it's a David and Goliath... it's a bit like a mouse and an elephant. They've got different ways of behaving, different internal structure and all the rest of it. I think I would struggle to have confidence that a big vendor would actually be able to participate in [the agile approach] fully." (P31, enterprise architect)*

An example of a customer who had a culture highly compatible with the agile development team is that of participants P15, P24 and P27. The team's methodology is Scrum-based, modified so that changes could be made within their one-week sprint, to decrease the feedback cycle time. As noted above in section 6.1, the customer P24's business case revolved around continuous change, and the development team's highly agile culture was a good match for this business case. The development team had constant access to the customer, and vice versa, so they could discuss requirements whenever needed and could demonstrate new features of the software as required:

> *"Even Monique, who is our legal person, has full access to the guys [in the development team]" (P24, customer)*

The team and its customers collaborated very closely, to the point where in some instances specifications of the requirements were not written down at all; instead the team worked off verbal instructions and used early implementations to get feedback from the customer:

> *"Last Thursday in our sprint meeting at eleven o'clock in the morning, we [the customer] said it'd be really helpful if we could have a*

> *way to view future pricing. I want to know how those prices are going to change over the next twelve months, because every month has a different price. Five o'clock on Thursday – I didn't even know a Beetil [feature request] had been raised – I just thought we were going, that's a really cool thing to have, but we've got quite a lot going on so it's not going to happen. Five o'clock I get an email alert saying, 'how does this look?' 'Great! Can you just swap [the display] around?' 'Yep, sure, no worries!' And we got that released into test Monday afternoon. Our requirements were: 'hey, it'd be very nice if could see this!' " [...] "They've literally gone and done it, and done it in the developer environment and sent us the screen shot [rather create a mock-up]." (P24, customer)*

The team had such a close relationship with the customer the boundary between team and customer sometimes blurred:

> *"Sometimes it's hard to tell who the customer is, but ultimately it doesn't matter. Some of the developers are more passionate about this than my team." (P15, customer)*

This team had used S3 (EMERGENT ARCHITECTURE) successfully since its inception several years earlier.

Each of the participants P15, P24 and P27 from this project considered trust to be the most important success factor:

> *"I think that trust has been massive for us. [...] They [the customer] trust you and you deliver." (P27, CEO/coach)*

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

> *"Once you've built up to that level [of trust], 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)*

In contrast, P29 described a previous role where he had trouble getting his customer to commit to regular involvement. To compensate they spent more time on up-front design:

> *"So we'd try to drive out as many assumptions as possible at the start. And that was our answer to [the client organisation] not giving us customers. So we probably did a lot more architecture there up-front, on several big projects." (P29, development manager)*

P22 and P25 also had trouble getting feedback from their customer, leading to delays in the feedback cycle:

> *"When we [the team] say, 'oh, we want you [the customer] all the time, available,' they say, 'oh no, we cannot, we have a lot of other work to do also, we'll meet once a week'; we say, 'no, we need to talk every day.' Then we explain why it is like that, because you're not going to make a one-time decision on what we are going to do for the next six months." (P22, senior manager)*

and

> *"There is the expectation that, 'I've given you the requirements, I'm a busy person, go away and tell me when it's done.' And there was a lot of time involved in going back and saying, 'here it is for you to look at, please look at it.' It would often be a road block – you'd often be stuck waiting unworkable lengths of time sometimes." (P25, team lead)*

There are a number of reasons for customers not being able to match the development team's agility and being non-agile. These reasons are discussed in the following section.

6.5.2 Types of non-agile customer

Participants mentioned a number of reasons for customers not being agile, which typically resulted in the team having to do more up-front design than they would otherwise need to do or prefer. These reasons include being a process-bound customer, the need for approval of the architecture design, requiring a fixed-price contract, requiring approval from the Board, and being time-poor.

(a) The process-bound customer

If a non-agile organisation is not able or willing to empower the team or provide them with feedback, then that team will have difficulty being agile:

> *"Not only did a lot of key players not get it [agile], there was a push back to keep the status quo because that was what was familiar and comfortable and has always been done." (P25, team lead)*

For example, P18's organisation structure did not suit agile development: their developers and testers were in different teams with different managers, and so they were unable to implement common agile processes such as test-driven development or regression testing. Thus they were unable to manage changing requirements. P18 commented that they felt it might have been easier to fix the requirements in advance and use a more plan-driven approach:

> *"...so having all the requirements up-front would have helped." (P18, development manager)*

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)*

Other participants reported customers who would not relinquish control and allow the teams to talk directly to end users, and IT departments who were not able to deploy the system at the end of each iteration.

(b) The design benediction

Frequently a team will be building a system that is part of a larger system, and the customer may wish to approve or 'bless' the architectural design prior to development to ensure it is compatible with the architecture of the overall system or with company policy (for example, P17, P25, P29).

P7 described how their architecture has to be compatible with other teams' work:

> *"For those parts of the project which other projects have to work on to get working, we'll work this more waterfall-y way, so that these requirements and all that kind of stuff is by agreement between the different teams." (P7, business analyst)*

For work that was not part of a larger system, P7's team was able to bypass the coordinating programme management and talk directly to the customer:

> *"But parts of the project which are self-contained, where it's just us, we're allowed to work with the business [customer] direct and work in a more agile way and they can change their mind." (P7, business analyst)*

See section 8.1.3 for a discussion on design approval given from outside the team.

(c) Approval from the Board

A number of participants talked about non-agile customers who need to fix a budget and scope in advance so that they can get approval from the CFO or the Board for the work:

> *"A lot of customers [...] need to go to the Board with some answers [proposed list of features], and they need to go to the Board with some numbers [proposed cost], and they're not going to go to the Board until they've got both of those things." (P26, team lead)*

and

> *"Here's the shopping list of features, how much is it going to cost? OK, cool, two million dollars... I'll get two million dollars for that." (P14, architect)*

To calculate cost, teams must fix the scope and do more planning – the 'envisioning' phase – so that they are in a better position to understand the effort (and cost) required to deliver a relatively fixed set of features.

(d) The untrusting customer and the fixed price contract

The 'untrusting customer' is a traditional customer that does not have the trust that is so important to agile development, and who prefers to provide the team with a predefined tick-list of requirements, so that they can hold the team accountable if the requirements are not all met.

If the development team is an independent software vendor (ISV) then the customer will most likely provide them with a fixed price contract that requires a fixed scope and fixed delivery dates, perhaps with penalties if milestones are not met:

> *"[With agile] they feel they [the customers] don't have the bargaining chip of the requirements; 'you had agreed...,' because that's a great*

> *bargaining chip to say, 'this is out of scope, that's out of scope, we're going to go live with what you agreed four months ago because you agreed on that...' " (P7, business analyst)*

and

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

Similar situations also occur in government projects in which a high level of accountability is required:

> *"Our customer still does elaborate requirements documents and that's because they're obviously in a government context, they need to clearly state what they're going to be doing." (P32, software development director)*

Like when requiring approval from the board, teams must do more planning to understand the cost of delivering the list of features.

(e) The time-poor customer

Sometimes the customer simply cannot commit to ongoing time with the development team.

> *"There is the expectation that, 'I've given you the requirements, I'm a busy person, go away and tell me when it's done.' " (P25, team lead)*

Teams solved this problem by spending more time designing up-front:

> *"So we'd try to drive out as many assumptions as possible at the start. And that was our answer to [the client organisation] not giving us customers. So we probably did a lot more architectural there up-front, on several big projects." (P29, development manager)*

*

The environment in which an agile team operates affects how agile it can be, and therefore how much up-front design it does. A match between the team's culture and the customer's culture will greatly enhance the team's ability to be agile, while a less agile customer will force the team to do more planning and up-front design.

Figure 6.6 shows the relationships between F5 and other forces and strategies. Like F4, F5 is a success factor of RESPOND TO CHANGE (S1), and therefore also of EMERGENT ARCHITECTURE (S3). On the other hand, a non-agile environment may increase the team's need to use BIG DESIGN UP-FRONT (S4), which in turn has a negative impact on the team's ability to use S1.

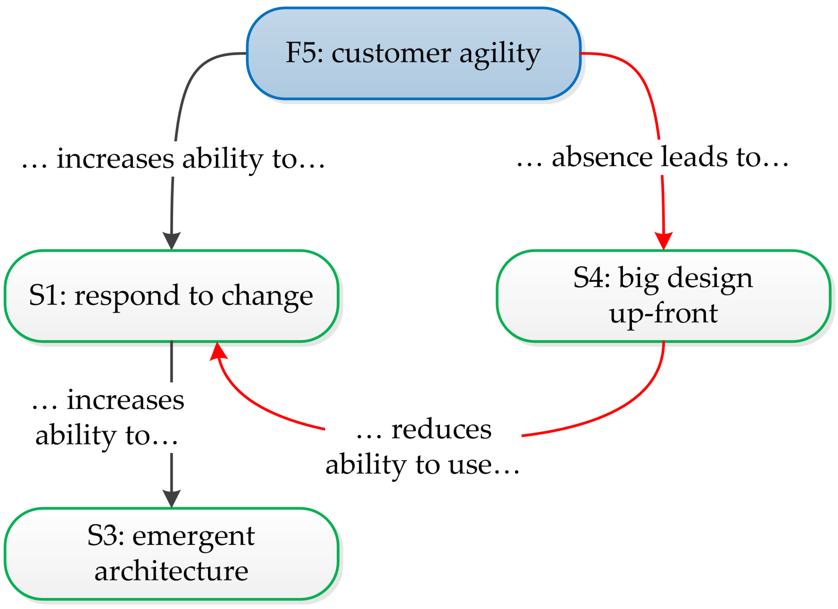

Figure 6.6: The relationships between CUSTOMER AGILITY (F5) and other forces and strategies

6.6 F6: Experience

DEFINITION OF F6: EXPERIENCE

Experience refers to the built-up knowledge and skill that the team has in architecture design and the technologies it is using. An experienced team is able to make more implicit decisions based on tacit knowledge, while an inexperienced team must make their decisions more explicit to ensure the decisions are appropriate.

F6 describes the impact that a software engineer's architecture and technical experience has on the time that an agile team spends planning. While an experienced software engineer can use tacit knowledge and make some decisions implicitly, an inexperienced software engineer relies more on explicit decisions that are written down and hence require more effort. (In contrast to F6, TEAM CULTURE (F4) includes agile development experience as a factor.)

Experienced architects have a 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 work. Hence they can make better and quicker decisions:

> *"I was drawing on the twelve years' experience and some mental models of what maintainable architectures would look like." (P13, architect)*

and

> *"Some of [our architectural decisions were] from experience... so things like single server versus dual server or multiple server, that sort of thing." (P14, architect)*

When the software engineer does not tacitly know the most suitable solution, experience helps them research and select the most appropriate solutions:

> *"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)*

and

> *"I think it's also important to have a degree of experience to know what you are reading is reputable and does this match with everything else that you already know." (P14, architect)*

Participants described experienced architects as being important for all development methods – not just agile:

> *"I think everybody has to be better, and they're not. That's the big and main problem in most big companies." (P8, lead developer)*

While important for all method types, experience is more important in agile development than in plan-driven methods because the tacit knowledge and the implicit decision-making ability support agile methods' reduced process and documentation [68], and hence reduce up-front effort:

> *"If you have a team of really, really experienced professionals then you can get away with a lot less [design up-front] because a lot of it is going to be implicit anyway." (P3)*

and

> *"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)*

and

> *"I don't think it [the architecture design] was particularly contentious, it became the obvious choice; I don't know how to explain that. [...] Most of us, I think we knew it was going to happen that way." (P12, senior developer)*

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

Knowledge of the technology helps the team to speed up design and use the technology's strengths (and avoid its 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)*

– although on the negative side, being too familiar with a technology can lead a software engineer to developing a too-narrow vision and an unwillingness or laziness to learn new technologies that may be more suitable.

If a team does not have the required experience, they can compensate through extra research:

> *"They also went out to find people who'd done similar systems, [someone] in Australia had done something not quite the same but similar processes, so we were able to go them and [ask], 'what's your architecture, can we have a look at it, why have you made these decisions,' which was very useful." (P8, team lead)*

– or they may bring in someone with suitable experience to join the team.

On the other hand, P2 ran into problems on which they blamed their unfamiliarity with the chosen architecture:

> *"Oh, you can always say it's because we didn't think about it enough, but I think the reality is we haven't done precisely this ever before, and not with this particular tool set, so it's unfamiliarity." (P7)*

Interestingly, one of the participants provided a contrary example where his lack of breadth of experience also simplified his decision making, with his options restricted to just one:

> *"With the prototype I chose ASP because I knew it, I could do VB-Script, I didn't know C#." (P1, developer)*

He then noted that if he knew C# he would have used it, implying the older technology (ASP/VBScript) was probably not the best solution.

*

Figure 6.7 shows the relationships between F6 and other forces and strategies. An experienced team is better able to use techniques such as keeping designs simple and using good design practices, which allow a team to design an architecture than can RESPOND TO CHANGE (S1). F6, with the other success factors for S1 – TEAM CULTURE (F4) and CUSTOMER AGILITY (F5) – and along with an absence of TECHNICAL RISK (F2), may allow a team to reduce their up-front effort to the point where the architecture is fully emergent, EMERGENT ARCHITECTURE (S3).

Conversely, a lack of experience may increase the team's need to do a BIG DESIGN UP-FRONT (S4), which in turn has a negative impact on the team's ability to use S1.

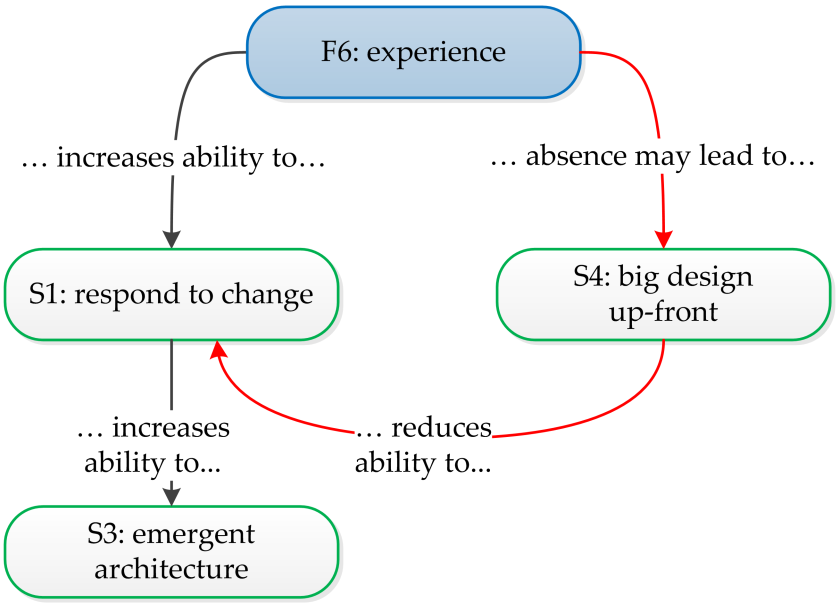

Figure 6.7: The relationships between EXPERIENCE (F6) and other forces and strategies

Chapter 7

The agile architecture strategies

"You've got to think about big things while you're doing small things, so that all the small things go in the right direction." (Alvin Toffler)

There are five agile architecture strategies that teams use to address the agile architecture forces (Chapter 6) when designing an agile architecture. Agile teams use the strategies to determine which architecture decisions need to be made up-front and which can be left emergent. The strategies are labelled S1 to S5:

- S1: RESPOND TO CHANGE
- S2: ADDRESS RISK
- S3: EMERGENT ARCHITECTURE
- S4: BIG DESIGN UP-FRONT
- S5: USE FRAMEWORKS AND TEMPLATE ARCHITECTURES

S1 increases the architecture's ability to manage and respond to change by increasing its modifiability (its ability to evolve) and its resilience to change. S1 generally decreases the up-front design effort that teams do. Conversely, S2 increases up-front design, and thus S1 and S2 are in tension with each other: teams have to balance the ability to manage change with mitigating risk.

S3 is an extreme case of S1 with no up-front design, while S4 may be considered an extreme example of S2 with no emergent design, although the forces they address are different. S3 and S4 are thus mutually exclusive. S5 can be used with all other strategies.

Figure 6.1 is repeated in Figure 7.1. The figure shows the relationships between the forces (Chapter 6) and the strategies. Each strategy and its relationships are described in this chapter.

7.1 S1: Respond to change

DEFINITION OF S1: RESPOND TO CHANGE

Responding to change is a strategy in which a team designs an evolving architecture that is modifiable and is tolerant of change.

Designing an architecture that can RESPOND TO CHANGE means making decisions so that the architecture can evolve and is tolerant of change. An architecture that evolves can be easily modified when requirements change; an architecture that is tolerant of change is less likely to need to be changed when requirements change.

An evolving architecture allows a team to ensure the architecture continuously represents the best solution to the problem as the team understands it as requirements evolve:

> *"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)*

and

> *"It's not a matter of architecture as a static representation to say, 'Ah! Us architects are phenomenal, we're like gods, we've foreseen everything!' Nah, it doesn't happen that way." (P10, coach)*

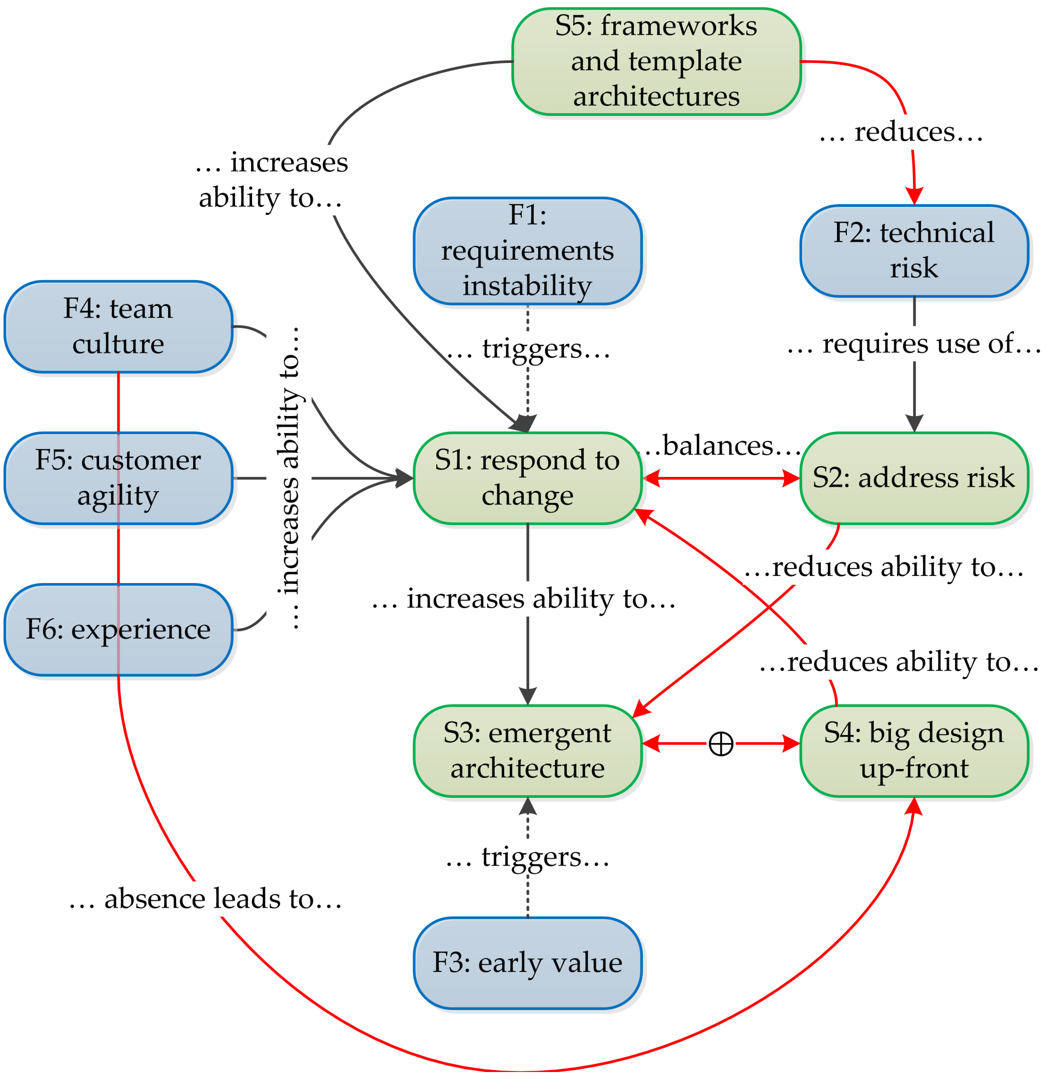

Figure 7.1: The relationships between the forces and strategies (Figure 6.1 repeated). The blue-coloured boxes represent forces, and the green-coloured boxes strategies. Arrows represent dependencies or causal relationships: a change in the independent force or strategy causes either a positive change (solid black line) or a negative change (solid red 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 ⊕ represents mutual exclusion.

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)*

and

> *"We constantly try to revisit these things, but periodically we've made fairly major decisions, and those decisions have always been based upon an evaluation of where we're at, what are our needs, what do we need to achieve going forward, and what's the best way to do that." (P36, development unit manager)*

Agile teams use S1 because of REQUIREMENTS INSTABILITY (F1). There are five tactics (methods used to implement the strategy) that teams can use to enable them to design an architecture that can respond to change.

Three tactics increase the modifiability of the architecture so that when requirements change or become known, the architecture can be easily updated. These three tactics are keeping the design simple, proving the architecture with code iteratively and following good design practices.

Two tactics increase the architecture's tolerance of uncertainty (its resilience to change), so that any changes to the requirements have less impact on the architecture. These tactics are delaying decisions and planning for options.

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 architecture effort.

Keeping the design simple, following good design practices and planning for options all affect the architecture itself: the use of these tactics may be determined a posteriori by inspecting the architecture. These three tactics are not dependent on an agile process. On the other hand, proving the architecture with code and delaying decisions are features of the agile process, and not the architecture itself.

These characteristics of the tactics are summarised in table 7.1. The tactics are described in the remainder of this section.

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

Table 7.1: A comparison of the RESPOND TO CHANGE tactics

7.1.1 Keep designs simple

Agile teams aim for simplicity in their designs: they only design for what is immediately required; no gold plating and no designing for what *might* be needed or for what can be deferred. Keeping designs simple avoids over-engineering: as well as causing waste during the initial design and development if requirements change ("it's a waste of time and effort" —P27, CEO and coach), an over-engineered design is harder to understand and reduces modifiability.

For example, P2 and P10 always designed with simplicity in mind, saying there is always a simple way of achieving results:

> *"So architecturally, we use the XP mantra: the simplest thing that would possibly do. So we've actually aggressively taken complex*

> *bits out and replaced them with not-quite-so-performing but easier-to-understand bits. It's just easier." (P2, developer/architect)*

and

> *"The one thing we've learned is there is always a better way. There is no excuse for complexity. There is always a simple way of achieving good results." (P10, coach)*

Using simplicity means a team *will* need to change its design later as requirements evolve and new functionality is implemented, but by the time they need to make the change they have a better understanding of what is required and are more likely to get it right than if they had predicted requirements and designed it earlier:

> *"The flipside of that [trying to build the simplest thing possible] is you will eventually find that is not enough, and you will eventually need to redo it. And you have a choice then – you simply know a lot more about what the business really needs at that stage, so it may be that you say, yes, now we do need to make this configurable, and I've seen that case emerge, or perhaps the original feature was the wrong one, and another one will supercede it. [...] There's a lot of effort gone down the tubes if you're trying to be predictive about where the thing will go." (P30, consulting architect)*

Simplicity and avoiding over-engineering reduce the cost of modifying the architecture, meaning it is less critical to get architecture decisions correct up-front. Teams can use techniques such as proving the design through code iteratively (section 7.1.2) to help keep the design simple and avoid predicting requirements and detailed up-front design.

7.1.2 Prove the architecture with code iteratively

Proving an architecture with code iteratively is a tactic used to help produce a simple design and reduce up-front effort, used when the team does not fully understand whether or not its initial design decisions will meet requirements. This tactic, possible because architecture design and development can take place simultaneously, proves the design by building it and testing it in real life rather than through analysis, and refines the design if it is not suitable.

The team starts designing what it believes is the simplest solution that *may* work – but not necessarily *will* work, because what will work is not known until implementation, testing or even until deployment. If the initial design proves inadequate the team will redesign the architecture – perhaps repeatedly – until it has a solution that is suitable.

For example, P2's original proposed authentication system turned out to authenticate too slowly in the system's operating environment; extra effort was required to implement a new authentication system that would greatly reduce the authentication time:

> *"Our hypothesis was, by changing the way we do our authentication we can reduce the number of round-trips and this will roughly halve our [call] duration. And we found we did. We changed to using an Oasis security standard, rather than using another standard because it meant that we could reduce the number of round trips for the authentication challenge process which in a high latency [environment like] New Zealand... we could reduce the call duration by 100 milliseconds. [...] We said to the customer, 'look, by using this Oasis standard and investing this time, we'll reduce by 20 per cent the duration of the calls.' And he said, 'yep, that's worth it.' " (P2, developer/architect)*

In some instances, the team may not initially fully understood the limitations of the technology being used:

> *"He [a developer] thought he could get [the password] from a credential vault, but it turns out you can only access that credential vault if you're coming in from a portlet, so we had to rewrite this to sit inside a portlet." (P7, business analyst)*

Proving the design iteratively is also useful when requirements such as user demand or system load cannot be understood in advance. Teams release an early version to test what the system load is and then refine the design in response to what the team has learned:

> *"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. [...] And you don't. You have to play it out in real life" (P10, agile coach)*

The alternative to proving the architecture with code is more up-front effort to accurately determine requirements and the architecture, which is less desirable in agile development:

> *"So additional analysis up-front I don't think would have helped at all. Probably would have hindered as well, because we wouldn't have been able to be so adaptive." (P17, manager/coach)*

– or over-engineering to ensure the design is able to cope with unexpected demand.

Proving a design iteratively can reduce the up-front effort of ADDRESS RISK (S2); proving a design with code are experiments (section 7.2.2) performed on production code.

7.1.3 Use good design practices

Good design practices that improve the architecture's modifiability are very important for agile architecture. While important for all development methods whether agile or not, they are more important in agile development because they make it easier for teams to make changes to the architecture and software as requirements evolve.

For example, separation of concerns refers to the division of a software system so that each part addresses one particular concern and has minimal effect on the behaviour of another part [43, 179]. Separation of concerns makes software systems easier to understand and change and therefore be more agile.

Encapsulated modules (modularity) are an implementation of separation of concerns, in which related concerns are grouped together into modules [179]. The interfaces between modules hide details of how the modules are designed. Change can be limited to the modules themselves, and not the interfaces between modules.

> *"Part of [making a design modifiable] obviously comes down to making your design as modular as you can... now this is a very old word, but it still makes heaps of sense which is that if you can break your design up into little pieces then you minimise the impact of a design revision completely killing everything. Because if it's modular with a bit of luck you'll only have to change one bit but the rest will be completely the same." (P6, managing director/lead developer)*

and

> *"They're going to move everybody off the Microsoft stuff for the front-facing sites and put them on Java-based Sun/Oracle [...] but you can set yourself up in a way that if you've architected it well enough and you've followed the good design principles [...] it's structured such that the risk is mitigated as much as it can be without knowing exactly what's going to happen down the road." (P25, team lead)*

In contrast are systems where modules perform many unrelated tasks and have many connections between them. These systems can result in the 'Big Ball of Mud' architecture [94], which is devoid of structure, or the 'Dirk Gently' architecture (after Adams (1987) [4]), in which "everything in the code is connected to everything else in mysterious ways" [171], and is difficult to change and update.

Good design practices extend to the class level. Some participants (for example, P4 and P10) mentioned using quality management tools such as Sonar and Structure 101 to calculate software metrics that can be used to check the dependencies between modules and classes, so they can be redesigned if necessary:

> *"Some of the tools we ran over it also identified things like high coupling, if you've got cross dependencies. One of the tools I used is a thing called Structure 101 which was essentially at the class level, package level, in Java, just analyse all the source code and identify dependencies between classes, and it would identify acyclic dependencies, so you could know, 'we're in a cyclic relationship here, what do we need to move to break the cyclic relationship?' " (P4, director of architecture)*

and

> *"Sonar [is] a code analysis tool that gives you all sorts of interesting metrics in terms of code complexity. It gives you an instant view of the code health, and you can see trends and coupling of classes. That way you can see – 'oh, that module there is being really naughty, look at the complexity spikes there showing up' – all of a sudden the architecture community can zoom in and say, 'Oh! What are you guys up to? How come your cyclomatic complexity is so high these days? What have you been doing these last couple of weeks?' " (P10, coach)*

Similarly, teams building larger systems (for example, P33–P36) used a service oriented architecture (SOA) to hide the implementation of services behind application programming interfaces (APIs):

> *"So the other reason for SOA is [it] allows us to effectively chunk things up [...] we can component-ise stuff and that helps us with changing requirements." (P33, product architecture team leader)*

One participant (P37) also noted the importance of the whole infrastructure – including the software system and its tools – being modular, so that even the tools can be changed without affecting the architecture, and vice versa: the architecture can be changed without affecting the choice of tools:

> *"It's modularity inside the system but also modularity between the system and all the infrastructure around the development tools, all the testing tools, and normally you're going to build your own testing infrastructure – specially at the system level because it's always different, and you have to be careful that they don't tie you into a particular choice of architecture." [...] "You can't consider the system as just the hardware and software components that you're going to deploy." (P37, consultant/freelance software developer/architect)*

Participant P29 noted that having a team that was smart enough to be able to do good modular design 'on the fly' (without explicit design) can reduce the amount of planning they need to do:

> *"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 or every change should only affect one encapsulated piece." (P29, development manager)*

Good design practices reduce the cost of architecture refactoring and hence increase the modifiability of the architecture.

7.1.4 Delay decision making

Delaying decisions reduces the impact of change by making decisions as late as possible. Delaying decisions reduces the up-front effort and reduces rework because decisions are made based on the most recent understanding of the requirements. For example:

> *"Yeah, so we were deciding, we were making concrete decisions as late as possible and only building what we absolutely needed to to deliver the feature that was being requested at that time, and trying*

> *to minimise the amount of rework we had to do for any particular features." (P25, team lead)*

and

> *"What you don't want to do is be committed to a development." (P6, managing director/lead developer)*

Delaying decisions helps prevent over-engineering caused by attempting to define too many requirements up-front, by only building what is needed as it is needed:

> *"But in general you don't want to plan for anything that is too far in advance, because you come back to the whole minimising waste... build that knowledge as and when you require it." (P3, development manager)*

P15 described how his team delayed decisions, incrementally improving their design and the system over time in response to requirements as they evolved:

> *"So there's a guy whose job becomes more and more mundane because it's manual. When we had a thousand customers it was fine; when we have fifty thousand customers, he starts to really struggle. So I go there and I go, 'so ok, I can speed this up for you, I can automate this a little bit, I can improve things for you there.' A week's work – bang, he's now happy for another year. And if you didn't do agile you couldn't do that." (P15, customer)*

Delaying decisions can benefit the team even if requirements are stable because it avoids knowledge being learned too far in advance and subsequently becoming lost or forgotten. For example, P29 delayed decisions so they did not miss out on opportunities to use new tools that came available:

> *If you design a year ahead, for all of that stuff that we have to do, then we betray the option of having new tools up our sleeves in a year's time when we actually get to the place to [build it]. This is effectively an opportunity cost of choosing to decide when you could do last responsible moment, which is one of the tenets of lean [development] and agile." (P29, development manager)*

7.1.5 Plan for options

Planning for options reduces the impact of change by making decisions that avoid closing off possible future requirements and reducing the need for future refactoring. To plan for options the team needs an understanding of the general direction development will take, without trying to predict requirements – a practice strongly discouraged in agile development. This tactic means that the architecture is not unnecessarily constrained and the team can avoid 'painting itself into a corner':

> *"And so having a notion of where you're going, having a notion of the sorts of things you could be expected to support makes it much more likely that what you're doing is going to be much more amenable to being extended." (P9, developer)*

and

> *"The star quality of a software architect is the magical ability to have invested some generality into the platform you build for your software, such that you appear to have magically foreseen what changes are going to be required." (P6, managing director/lead developer)*

Planning for options retains a flexible architecture which is able to cope with a range of possible future requirements for as long as possible during development and thus reduces expensive architectural rework:

> *"It's a real bummer when you design something clever and then find that it needs to be extended in a fashion that you didn't provide for, and you have to rewrite it from scratch. Refactoring is wonderful, except when your refactoring tool is `rm` [remove]." (P9, developer)*

Some teams determine the extremes the system could be required to operate at. P9 described this as 'bounding the design':

> *"You didn't complete the design – you bounded the design. The point being you only took the design as far as you needed to go to know where the solution was going to be. You don't have to have the whole solution, you just have to see to it that you know where the solution is and you don't close off access to it." (P9, developer)*

P2 and P10 described examples of how they planned for options:

> *"You know how you have 'cut along the dotted line'-type things? We had dotted lines where we could insert caches for additional performance." [...] "[And] we could take our database and 'shard' it by geography or by client. So we had all the bits in there but the customer wasn't willing to pay for it [just yet]. So that's how we've made it agile. Cut along the dotted lines." (P2, developer/architect)*

and

> *"So rather than attempting to be too forward looking, we'll add an indirection here because it'll be extensible." (P10, coach)*

Similarly, P26 built a system with more layers of abstraction than was necessary:

> *"In my opinion it was probably an architecture slightly too big for the requirement, but it didn't hamper us in that regard. It was an enterprise-scale architecture, and we had a New Zealand version of an enterprise scale app, not a truly global enterprise app." [...] "This*

> *was more of a seven-tier system where perhaps four would have done, if you like." [...] "There was often a lot of perception that these things may be needed; that architecture supported them in case you did need them." (P26, team lead)*

No participant admitted that planning for options increases the effort required – which would break the simplicity principle – although P26 said their system with extra layers of abstraction took new team members longer to learn.

Like delaying decisions, planning for options allows a team to reduce the up-front architecture effort required and design a more emergent architecture.

7.1.6 Not designing for change

P28 noted a problem caused by not using good design practices. He noted that their architecture did not have good modularity and appropriate abstraction layers, and so the system became very difficult to change. He gave an example of not abstracting the solution's messaging even though they knew the queue technology was going to change:

> *"But everybody knew that in six months we'd be changing our queue technology. So coupling directly to the libraries and not enabling an abstraction layer to the messaging was a bad business decision; you should have built an abstraction layer." (P28, technical lead)*

This poor design decision meant that the system was difficult to change later when the technology changed.

Part of the reason for these poor design practices was the teams underestimated the effort required to refactor and they did not understand the benefits that good design practices would provide:

> *"People at the time didn't see value in doing that [implementing the abstraction layer] because, oh, it's ok to stop and refactor your code when you need to!" (P28, technical lead)*

He also noted poor design practices can be caused by inexperienced developers:

> *"But if you have different qualities of programmers working with that, it's a real mess to refactor because you've got to stop to clean up the mess [first]." (P28, technical lead)*

The impact of experience is captured in EXPERIENCE (F6), in section 6.6.

*

Figure 7.2 shows the relationship between S1 and the forces and other strategies.

REQUIREMENTS INSTABILITY (F1) is a trigger for S1, while TEAM CULTURE (F4), CUSTOMER AGILITY (F5) and EXPERIENCE (F6) are all success factors: the more agile the team, the more the team is able to use S1's tactics, the more agile the architecture will be, and the more successful they will be at reducing up-front design.

These relationships mean that up-front design effort depends on a team's ability to design an architecture that can RESPOND TO CHANGE, and not how unstable the requirements are. An agile team usually does not know a priori which requirements are going to change; the team behaves as if any of the requirements can change. If the team does not know how unstable the requirements will be, and which requirements will change, it cannot determine how much up-front design to do.

For example, P27 (CEO and agile coach) could minimise his team's up-front design because they were a highly agile and experienced team with an agile customer (F4, F6 and F5, respectively) and not because the requirements were highly unstable, although the highly unstable requirements did motivate the team to become highly agile. A less agile team may attempt to reduce up-front design, but having less ability to RESPOND TO

CHANGE and evolve the architecture, over time the architecture is likely to no longer meet current requirements and become unsuitable.

S1 is also a prerequisite for EMERGENT ARCHITECTURE (S3): the more agile the architecture, the more successful the team will be designing an entirely emergent architecture. BIG DESIGN UP-FRONT (S4) decreases the team's ability to use S1, because while it may be able to design an architecture that is modifiable, it cannot delay decisions. USE FRAMEWORKS AND TEMPLATE ARCHITECTURES (S5) reduces the effort required to make changes and therefore increases the team's ability to use S1.

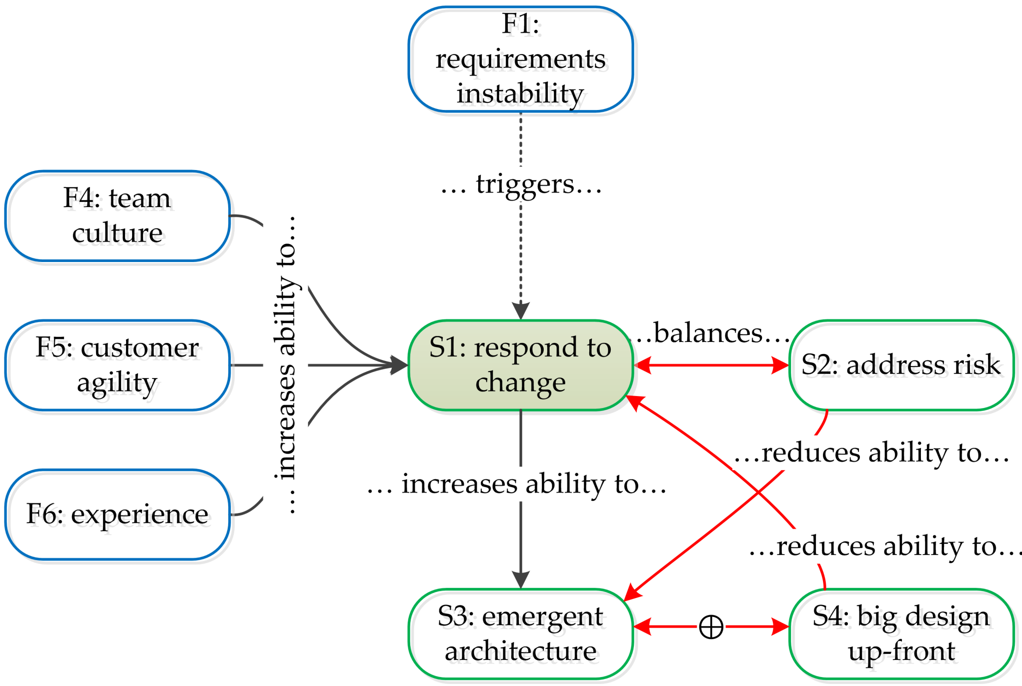

Figure 7.2: The relationships between RESPOND TO CHANGE (S1) and other forces and strategies

7.2 S2: Address risk

DEFINITION OF S2: ADDRESS RISK

Addressing risk is a strategy in which the team does sufficient architecture design to reduce technical risk to a satisfactory level.

Mitigating risk reduces its impact before it can become a problem, and is usually done up-front, particularly for risk relating to system-wide decisions (for example, risk in selecting the technology stack).

More TECHNICAL RISK (F2) means more up-front architecture design, which creates a tension between S2 and RESPOND TO CHANGE (S1): to reduce risk, a team must sacrifice some of the architecture's ability to respond to change. A team normally finds a balance by reducing risk to a level that is satisfactory to itself and to the customer, and delaying decisions using S1's tactics where risk is low.

A team designs the architecture in sufficient detail that the team 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, is trying to reduce the risk of the whole thing going off the rails and not achieving what you want it to achieve. [...] It's very much a risk-based process." (P36, development unit manager)*

and

> *"[Risk] was managed by making sure we'd done an appropriate amount of research up-front." (P25, team lead)*

The higher the impact of the risk, the more important it is to mitigate that risk early:

> *"It makes sense to focus on the risks in the first couple of weeks because [if] you make [a change with] a one per cent deviation or two*

> *per cent deviation, two months down the line it will be thirty per cent or forty per cent." (P16a, CEO/chief engineer)*

Indeed, teams prioritise architecture effort according to the level of risk, so that they can reduce risk as quickly as possible while the level of committed effort is still small:

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

and

> *"I think higher risk means higher importance." (P30, consulting architect)*

If a team does insufficient mitigation then risk of failure will be higher.

TECHNICAL RISK (F2) is increased in complex systems with demanding ASRs that result in a complex architecture, and include unique features that are not part of any existing framework or library. Demanding ASRs are hard to produce a design for and may require major trade-offs:

> *"A lot of [up-front architecture] is the design of critical aspects of the system, so things that are unique to the systems – the thing about the role-based security being correlated to your location in the country was quite a unique aspect to [our] system, so I'd probably consider the design of how that was going to work part of the [up-front] architecture of the system." (P14, architect)*

For example, P12, a senior developer, needed to prove that his team could meet their primary goal of being able to process a financial 'trade' in less than one second. This constraint was a key ASR for the system. The team's high-level design showed how the system could meet that requirement:

> *"Once we'd come up with the fact that one of our top priorities was to make a trade go through the system in sub-one second, then that became our top priority and we produced documents and architecture diagrams to show how that trade would flow through the individual systems at a higher level, broken down into lower levels." (P12, senior developer)*

7.2.1 Criticality and appetite for risk

The team's and the customer's appetite for risk – how much cost the team or customer is willing to bear when something goes wrong – is an important consideration. For critical projects with a high cost of failure, such as the possibility of losing lives or people being harmed, the tolerance for risk is low and more effort to reduce risk is required, while for non-critical projects such as simple websites, tolerance of risk is higher and so less effort is required. For example, P33's system had clinical and security risks that meant the design had to undergo additional scrutiny prior to development:

> *"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 is." (P33, product architecture team leader)*

– while P2's customer was willing to risk a financial loss under certain circumstances to get a cheaper system:

> *"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 low-ish odds that they'll redeem it twice." (P2, developer/architect)*

P2 often gave the customer the option of choosing their level of risk:

> *"I said [to the customer], are you willing to spend some money to reduce this risk? And he said, 'how can we do that?' And I said, 'well, we see quite a risk here in performance in this particular area. 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)*

While some participants reported they did not always know whether or not they could solve the problem when they started development, in these cases the team and its customer clearly determined the level of risk to be acceptable.

Addressing risk is balanced with responding to change (S1): a low risk system allows greater delay in decision-making, and hence allows a team to be more agile, while a high risk system requires more certainty, more up-front design and hence less agility.

7.2.2 Techniques for addressing risk

Teams use a combination of design techniques to find the most appropriate solution for the problem at that time to mitigate technical risk. Techniques include desk research, modeling, estimation, and experiments:

> *"We do investigations. We don't put any real constraints on how people go about those investigations, but typically they would be a combination of research and spike prototyping." [...] "We do formal modelling in some specific areas." (P36, development unit manager)*

(a) Research

If developers do not know the solution to a problem, they can do desk research to find solutions. Desk research can include searching reference resources (such as websites and books) and seeking help from the development community:

> *"I certainly spent a lot of time reading about how you'd apply different frameworks and ways that you could address some of the particular problems that we had, so there was a bunch of reading that I did." (P14, architect)*

and

> *"We follow the books. CQRS [command query responsibility segregation] and the WAG [Windows Azure Guidelines] and P and P – Patterns and Practices. Microsoft has a whole site dedicated to Patterns and Practices. Very few people read it. And there's also ALT.NET, the open source community's equivalent. And InfoQ has the same. So we just hoover up those." [...] "It means we are less likely to screw up." (P2, developer/architect)*

and

> *"The team also went out to find people who'd done similar systems. [Someone] in Australia had done something not quite the same but similar processes, so we were able to go them and [ask], 'what's your architecture, can we have a look at it, why have you made these decisions?', which was very useful. Unfortunately they had the budget and we didn't so we had to make some compromises." (P8, lead developer)*

(b) Modelling and analysis

Agile teams reported very little use of formal modelling, tools and analysis. Teams would sometimes model subsets of the system by creating data models or models of interactions:

> *"For the large implementations we have used modelling tools." (P21a, manager/coach) [...] "It's more the system level. How jPOS interacts with Websphere and how it interacts with SAP. So that level, very high level representation is what we do." (P21b, architect)*

and

> *"So what we did do though, when we built the services we took a contract-first approach, focused on the data model, driving it out of the canonical data model, schemas were driven off that." (P31, enterprise architect)*

Some software engineers favour running thought experiments to determine architectural solutions that best meet requirements:

> *"I have always credited either myself or the people I work with with enough intelligence to think sufficiently far through the process – I guess just based on having sufficient years of experience to be able to see the issues coming – to be able to pretty much figure out in advance – not always immediately but with enough thought – to do a thought experiment at the end of which you can probably already junk the other [architectural approaches] and actually only invest time in coding one of them." (P6, managing director/lead developer)*

(c) Experiments

Many software engineers believe that modelling and analysis is not sufficient for designing an architecture: the only way to prove that an architecture design really works is to prove the design by coding it in the form of a prototype, proof of concept or spike: "get your hands dirty and get your feet wet" (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] – there's only so much you can do as a thought experiment, and you can't afford to try and do everything as a thought experiment because the criticality of the system is such that you really need to know if you have enough headroom to support peak load." (P10, coach)*

and

> *"Regarding the architecture, basically proof of concepting the actual technology, so from our perspective that was probably the highest risk, using bleeding edge, leading edge technology, especially after the... not 'debacle,' but the previous generations of Microsoft Workflow not being quite so great. So in order to de-risk that we did the proof of concept, we also did initial performance testing, very roughly to get a feel for, is this engine actually going to deliver the kind of performance that we need?" (P3, development manager)*

and

> *"[Technical problems] would only become evident once you start digging into the code. These types of things I really don't think you could catch with a waterfall [development process]. You really have to do it to catch the problems. When you're doing waterfall you have to make this big fancy plan and you hit a roadblock somewhere, then you have to completely alter course. And then what do you do with the rest of this massive plan that you've made?" (P19, development manager)*

Some prefer to discard a proof of concept after it has provided the required answer:

> *"So prototypes should be throwaway. The whole idea in the [research and development] project is nothing should be dependent on it except an outcome. So no R&D programme will deliver a piece of technology, it'll just come up with an outcome saying we can do this and we should do it this way, or we can't do that." (P11b, architect)*

– while others preferred to build on the prototypes and migrate them to the production system:

> *"They evolved. We added more and more functionality." (P18, development manager)*

Spikes are time-boxed research and experiments [181]:

> *"Anything unknown has a spike at first." (P22, senior manager)*

and

> *"The whole notion of having architectural spikes is, 'whoops, we know what 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 counter-measure is at this time, and have it in place.' " (P10, coach)*

Spikes have clear boundaries of time spent and cost, and have clear goals to be achieved to avoid the research sidetracking development:

> *"We set short targets for each of the spikes, and say don't spend more than one or two days for one target. So if you're not able to get it, just look at alternatives or try to see a different way of implementing it. Or go to another target and come back to that previous target another time." (P21a, manager/coach)*

and

> *"[The prototyping was] quite time-boxed, quite focused. We go, 'we're going to spend half a day mocking up the database and loading some data into it.' So we learn a lot about the data and the nature of what it is." (P2, developer/architect)*

Proving the architecture with code iteratively, described in section 7.1.2, is a form of experiment in which the experiment is performed iteratively on production code, allowing decisions to be delayed.

(d) Walking skeleton

Some teams build a *thin slice* or *walking skeleton* [205] to prove their architecture meets requirements. A thin slice is an end-to-end implementation of the architecture with minimal functionality:

> *"One guy always called it an 'executable architecture.' And only once you've got that, so you've described the architecture in code, shown that it works to some extent, then you write up what worked. And that's way better because if you just write it up in theory then you may be making decisions that are impractical." (P7, business analyst)*

and

> *"I put a skeleton up and let everyone test it and argue about it, and it morphed into the form it needed to at each stage." (P30, consulting architect)*

Once proven, the skeleton is fleshed out as more functionality is added. Like experiments above, the skeleton is similar to proving the design with code iteratively (section 7.1.2), but may differ in that proving the design iteratively does not prove the architecture before implementation.

7.2.3 Not enough architecture

This section discusses examples of participants who did not design enough architecture up-front to sufficiently mitigate risk.

One participant who believed his teams did insufficient design was P28, a member of one team of many who were developing multiple web-based systems. He believed that the teams took the agile philosophy of minimising up-front design too far:

> *"This fear of doing up-front design has become just like a dogma." (P28, technical lead)*

Insufficient design meant that P28's teams made the wrong decision with the web development framework they were using, selecting one that was not designed to be used the way they were using it, which meant some functionality was difficult to build:

> *"Django was built for something that is not exactly what we are working on, so we have some clash with its design, what it was designed for, and what we need. [...] Sometimes we have to struggle with it to do things that are difficult to do with Django." (P28, technical lead)*

The teams therefore had to spend significant amounts of time forcing the technology to fit, leaving less time for developing functionality:

> *"Whenever you have a good software [technology] and you're programming, if everything's fine you just need to be concerned about the business. Whenever you stop caring about the business [because you have] to care about the software, you have a problem. So whenever we're developing a new feature in Django, a reasonable amount of time, we have to care about Django and not the business, so for me that's the biggest indicator that it wasn't the right choice." (P28, technical lead)*

In another example of not enough design, P21 selected their technology before completing performance testing, later found it was the wrong choice, and eventually had to replace the technology:

> *"Even after doing spikes [time-boxed experiments] we had still not done our performance benchmarking before we [made] our decision." [...] "I must confess that later on we found that we might have picked the wrong decision!" [...] "Maybe after one or two years, I came to replace [the original choice of] Hibernate with some other framework." (P21, manager/coach)*

P21 commented that if he were doing the project again today he would do the performance testing up-front to check the technology could actually do what was required.

*

Figure 7.3 shows the relationship between S2 and the forces and other strategies. More TECHNICAL RISK (F2) requires more risk mitigation. Because addressing risk is largely an up-front activity – particularly for system-wide risk – S2 is in tension with RESPOND TO CHANGE (S1) and is therefore unlikely to be used with an EMERGENT ARCHITECTURE (S3).

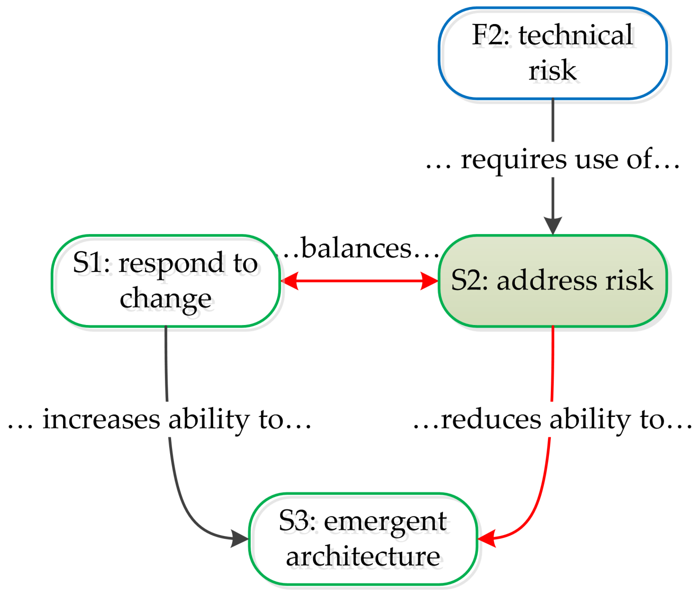

Figure 7.3: The relationships between ADDRESS RISK (S2) and other forces and strategies

7.3 S3: Emergent architecture

DEFINITION OF S3: EMERGENT ARCHITECTURE

An emergent architecture is an architecture in which the team makes only the bare minimum architecture decisions up-front, such as selecting the technology stack and the high-level architectural patterns. In some instances these decisions will be implicit or will have already been made, in which case the architecture is totally emergent.

S3 produces an emergent architecture in which no decisions are made up-front. S3 is similar to S1, but delays all architectural decisions apart from the bare minimum, such as selecting the system-wide technology stack and the top-level architectural styles and patterns, which affect the whole of the architecture and must be made before development starts. These up-front decisions are frequently implicit and require no effort, in which case S3 effectively produces a totally emergent design. (If these decisions are made explicitly, then the architecture is *nearly* emergent, but is still included in this strategy.)

The emergent architecture strategy is most successful when the team is very agile and can design an agile architecture that can RESPOND TO CHANGE (S1) and when there is no risk that needs mitigating up-front (ADDRESS RISK, S2).

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.

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)*

P27 only knew the requirements in detail two weeks out:

> *"The developers don't really know what's coming at a specific level for probably more than two weeks out. [...] We don't have a month's worth of work in there because it'll change!" (P27, CEO/coach)*

P16a and P16b also used an emergent architecture strategy:

> *"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. We try to go ahead, for once we've actually signed the contract with the customer, we get*

into this mode where we push something out to the customer so we can have a demo within a week. Within a week of having signed the contract! No later than that! So this is something that is a big part of our company." (P16a, CEO/chief engineer)

P16a, P16b and P27's systems were implemented entirely using standard framework libraries, and so had very little technical risk (see TECHNICAL RISK, F2; ADDRESS RISK, S2 and USE FRAMEWORKS AND TEMPLATE ARCHITECTURES, S5):

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

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 that require more up-front design to ADDRESS RISK (S2), precluding an emergent design.

7.3.1 Minimum up-front activities

The EMERGENT ARCHITECTURE has a minimum level of planning that a team must do before development can begin, including both architecture design activities and non-design activities.

Non-design up-front activities

Before a team can start development, it must carry out a number of planning activities in addition to the minimum architecture design, such as:

- bring the team together

- understand the business domain (and particularly understand the business problem or vision) and define or elicit any early requirements
- determine the development process
- select the development tools and environments such as the development suite, the testing framework and the continuous integration delivery set up
- release planning
- decide upon any programming guidelines.

For example,

> *"We went, ok, what's every task we can possibly think of that will put us in a position where the team could start coding and deliver features effectively, then we could mark a start. So that would be things like, getting environments up, getting everyone's PCs reset to whatever tooling we had, making decisions about any tooling that was going to change, making sure there was enough stories through our analysts to actually get on with the first iteration. All those kinds of things." (P29, development manager)*

and

> *"You should have your vision, your design should be ready, roughly, you should get your team together, get your resources together, get your backlog together [...] Make sure your environments are there, do a few – if you need any – preliminary sessions, training, get that in." (P11a, architect)*

and

> *"Sprint zero was figuring out what the actual brief was, bringing out the important parts of the brief, what parts were going to take priority, which parts we were going to focus on, figuring out whether it was possible, figuring out which parts of the system we had to talk*

> *to, whether we had the correct expertise in the team to do what we wanted to do, get to know the Scrum master...” (P12, senior developer)*

The team must also make some minimum architecture decisions up-front.

Minimum up-front architecture design

Before development starts, a team must make certain minimum architecture decisions. These decisions would normally relate to matters that have a system-wide impact and cannot be changed after development has started without discarding all or most of the work completed to date: the technology stack (the suite of technologies, products or frameworks that it will use) and the top-level architectural styles or patterns. For example:

> *“[We chose] the basic tech stack – PHP, Backbone and the databases. That was made up-front, and that took a lot of pulling back and forth between myself and [the team], just doing a lot of research to make sure that piece is solid.” (P19, development manager)*

and

> *“There were lots of decisions I guess. Are we going to use HTML5 and CSS1, are we going to put a unit testing framework and Backbone-type MVC into our Javascript so it’s now a first tier language in the game?” (P29, development manager)*

Some proportion of these decisions may be implicit due to the team’s experience with the technology and tacit knowledge, and therefore do not require any effort. These decisions may also be made externally, out of the control of the team, perhaps by the customer for strategic reasons or in line with other systems or projects or work streams. In these situations the decisions are constraints on the architecture. If these up-front decisions are all implicit, the architecture is totally emergent.

High-level requirements

While it is usually very difficult to define all requirements in detail up-front, agile software engineers need to know a certain level of detail so they can make the minimum high-level architectural decisions.

At the highest level, the team needs a description of the business problem or knows the business's vision or business plan, so that team members understand what they have been engaged to build and why they are building it. The business problem can be described in many ways; perhaps as conversations between the team and the customer and stakeholders, or as a project brief, a statement of work, or recorded in more detail in a tender document:

> *"The project sponsor gave us a brief – maybe a page and a half – maybe not even that much – and that essentially detailed the outcomes that they wanted." (P12, senior developer)*

For a team contracted to a one-off project the business problem is likely to be fixed for that project, while for ongoing development, such as a mass market product or service, it will be continually evolving as it looks forward over a certain period of time:

> *"The general direction that we're heading in over the next year, we pretty much know what that is, roughly. We're not going to have to do a complete about-turn. Things that are on the periphery may change." (P33, product architecture team leader)*

How far the team looks ahead – the *planning horizon* – will vary greatly depending on the organisation or business, and particularly the volatility of the environment or the market it is in; the planning horizon could be the next release or it could be several years out. For example, P33 (above) anticipated requirements a year out, while P15 looked ahead six months:

> *"So yesterday we had a senior management team meeting, and we discussed what it is that we want to deliver in the next six months.*

> *And there were six major items. That'll result in about four hundred changes probably." (P15, Head of Operations/customer)*

A longer planning horizon does not necessarily mean more up-front design; rather, it provides the team with information about what is probably coming so that it can plan for options, one of the RESPOND TO CHANGE tactics:

> *"We try to get [P15] to help set the scene to give us an idea of where we're generally headed, but the developers don't really know what's coming at a specific level for probably more than two weeks out." (P27, CEO/coach)*

The business problem must include the system-wide ASRs and constraints. In P12's case, there was a single system-wide ASR that related to performance:

> *"It was taking sometimes up to two minutes for a trade to flow through the whole [existing] system, and our brief was to get that to sub-one second" (P12, senior developer)*

In other cases, there may be many system-wide ASRs. These ASRs will guide up-front architecture design and will later be expanded to guide ongoing architectural design:

> *"[You'll] often think of doing a breadth-first coverage of the requirements to try and get the architecturally significant requirements, so that the broad architecture framework can be established. And then go into iteration one to drill down in detail of a particular area and get it implemented." (P7, business analyst)*

Without understanding the key ASRs the team may not know whether or not they can solve the business problem:

> *"We knew sufficient requirements to be confident that we could solve the problem [...] Quite often we go into it not confident we can solve the problem." (P2, developer/architect)*

If a team does not know if it can solve the business problem, then technical risk is high. This would normally require the team to ADDRESS RISK (S2), which precludes the use of S3, unless the team and customer accept that risk.

P17 described a problem caused by not understanding the business problem sufficiently: they did not consult all the stakeholders to ensure they understood all the system's constraints. His team had implemented their system using a DAO (data access object)-based architecture. When they came to release to production after nine months, the organisation's security department announced that they were not able to deploy it because it contravened their policy that required it to be based on an MVC architecture. Because of the high cost of changing to an MVC-based system at this late stage, the only solution was to purchase a separate security appliance to install between the application server and the outside world. This workaround required time to obtain the extra budget, procure and install. The error was caused by lack of communication with the project's stakeholders:

> *"More analysis up-front wouldn't have helped; more communication up-front would have." (P17, manager/coach)*

P17 noted however that if they had a quicker release cycle then they could have found the error earlier and could have avoided having to use the appliance workaround:

> *"It took more than eight weeks to get that Juniper device in" ... "if we could have first released in, let's say eight weeks, and we discovered [then], oh no, it's got to be MVC architecture – in hindsight it would have been better to refactor the architecture and throw away those eight weeks of work, and start again." (P17, manager/coach)*

He then talked about the point of no return with architecture decisions – where the amount of time to correct an incorrect decision is more than the time required to work around it:

> *"There is a point of no return with architecture. Somewhere between eight weeks [time to implement the workaround] and nine months [time to start again] is that point of no return! Where the cost of refactoring that core architecture is too great." (P17, manager/coach)*

This example clearly shows one of the benefits of early and regular deployment into production.

P28 also had to consider the option of replacing their wrong web development framework with working around the problem. They decided they could not replace the framework without starting development again from scratch:

> *"I don't think that [replacing Django] is possible without rewriting the whole project. Since we have so many projects built up on it... it's too late." (P28, technical lead)*

7.3.2 The minimum viable product

S3 is likely to be used when developing a minimum viable product, or MVP (see F3, EARLY VALUE). An MVP is a marketing strategy or experiment in which teams release versions of the software that contain marketing tests or experiments, so that the team can discover preferred potential features for development:

> *"We want to be able to put in the smallest, simplest, minimum viable experiment, prove an assumption, beef it up if we want to or pivot and follow it wherever it goes. That means our entire architecture is going to be emergent based on where we want to go." (P29, development manager)*

An example is the 'A/B test', in which the team develops a version (or versions) of its system that presents alternative options or features. The end users' responses to those alternatives guide further development:

> *"Every experiment we have to make a call whether it's an A/B test or whether it's a single thing that's going to create a feedback loop and we're going to refactor it or adjust it or... so there's many different testing models – sometimes we use A/B, sometimes we use canary and sometimes we use more of a big-bang, here's an MVP, let's see how people react to it, hopefully adjust it..." (P29, development manager)*

The aim of the MVP is to maximise learning by getting as much feedback about what end users want as early as possible, and is particularly useful in new and immature markets.

*

Figure 7.4 shows the relationship between S3 and other forces and strategies. The need for the customer to get early value from the product (F3, EARLY VALUE) motivates the use of S3, and being highly successful at responding to change (S1) is crucial. ADDRESS RISK (S2) increases the up-front effort, and is therefore reduces the team's ability to design an EMERGENT ARCHITECTURE. A system with a successful emergent architecture design will have low technical risk TECHNICAL RISK (F2) and low complexity (for example a simple business website), whereas a complex system with high risk is not likely to be able to suitable for S3.

7.4 S4: Big design up-front

DEFINITION OF S4: BIG DESIGN UP-FRONT

The big design up-front requires that the team acquires a full set of requirements and completes a full architecture design before development starts.

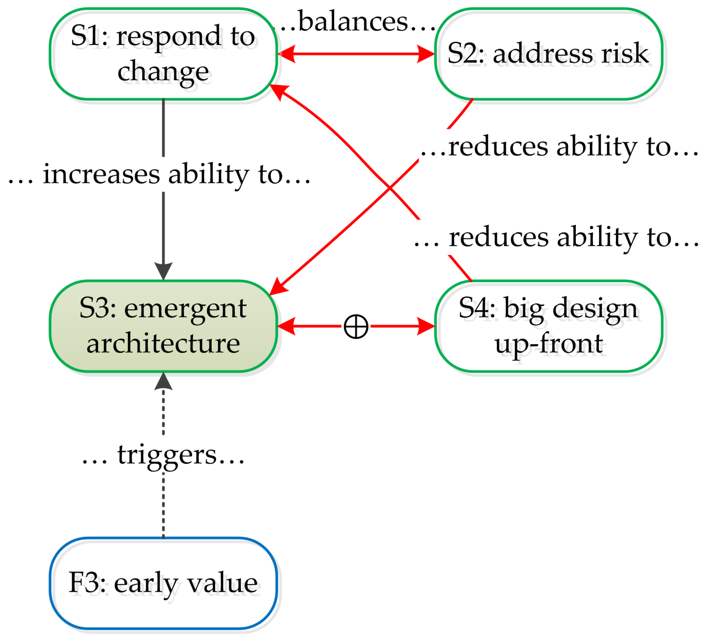

Figure 7.4: The relationships between EMERGENT ARCHITECTURE (S3) and other forces and strategies

Strategy S4 is a full up-front design (known as BDUF, or 'big design up-front') of the system. There are no emergent design decisions, although the architecture may evolve during development. While S4 may be considered the case of ADDRESS RISK (S2) taken to the extreme, in reality the use of S4 is driven primarily by the presence of non-agile customers (CUSTOMER AGILITY, F5) rather than by the presence of TECHNICAL RISK (F2).

S4 requires the customer to define a relatively detailed set of requirements up-front, sufficient for the team to complete its design:

> *"Our customer still does elaborate requirements documents and that's because they're obviously in a government context, they need to clearly state what they're going to be doing." (P32, software development director)*

A big design up-front is undesirable in agile development because it reduces the architecture's ability to RESPOND TO CHANGE (S1) by increasing the chance of over-engineering and increasing the time to the first opportunity for feedback. Despite this undesirability, an agile team may be compelled to use S4 because they are in a non-agile environment with a non-agile customer (CUSTOMER AGILITY, F5) who requires the team to spend more time on design up-front than they would if they had an agile customer. There are a number of reasons for a customer (whether internal or external) being non-agile, such as preferring a traditional development process, wanting to approve the design before implementation, needing to know the total cost of the development before it starts so they can get funding approval or for a competitive tender selection process, and not having the time to commit to ongoing interaction with the team.

The up-front design in S4 is sufficient to satisfy the customer's need for non-agility: either sufficient to prove that the team knows how to solve the problem before starting, or 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)*

and

> *"A lot of our key architectural decisions we're making as part of our sales process and RFP responses, and things like that – you're choosing your key platforms that you're going to run on as part of your responding to a RFP, so when it comes around to delivery, those major aspects of the project have already been determined." (P14, architect)*

S4 and the inability to delay decisions does not necessarily mean the team is not agile; they may still be able to evolve the architecture when requirements change. For example,

> *"We did have to come up with these tentative [cost] figures for the full programme [...] We did have to give them a confidence in what we could achieve in that year, and that was the tricky bit." [...] "At a whole end-to-end level, [requirements were understood] quite well. As in breadth, not depth:" [...] "[I mean] there was a lot of new stuff introduced as well – it wasn't just, 'there's your requirement, build that.' Because then there would scarcely be need for iteration, apart from a phasing perspective. There was legislation in progress and that in itself was quite changing – or had the potential to change – so [we were] delivering what we could when we could and phasing and moving things in and out." (P26, team lead)*

P32's company, who often needed to provide customers with cost estimates in advance, adopted a policy of only committing to deliver half of the required features that they estimated they could achieve within a period time, leaving the remainder of the features out of the equation to evolve in an agile manner:

> *"We've taken to committing to at least fifty per cent of the features we're [expecting]... if our current velocity says we should nail all of that based on the T-shirt sizing of the stories or features as we'd call them... we'd be nuts to go in there knowing that the planning cone of uncertainly and all that sort of stuff at that stage, so we do try to get fifty per cent in. [...] [We] commit to doing fifty per cent of the story size, because of all the uncertainty. [...] We don't contract to a fixed scope."*

While the team 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. Not being able to delay decisions 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 fall into this category, because their customers are often larger process-driven organisations who require more financial accountability:

"Big corporates such as IBM or HP always seem to need a lot of process." (P14, architect)

Larger ISVs therefore often struggle to become as agile as smaller organisations.

Some teams put up a traditional front, hiding their agility, so that to the customer they appear to be using a traditional process, but they use agile processes internally, such as using iterations and regular practices meetings:

"So if you've got a customer that's a government department you have to incorporate some sort of waterfall method in it; you have to! We call it 'wagile.' What we do is internally, internally for those sorts of projects, internally, we absolutely run it like this [agile]. From a reporting to the customer point of view, when we first sign up with them, we absolutely report it in waterfall." (P27, CEO/coach)

and

"So what we tend to do is put up a bit of a fake model over the top of ourselves to protect ourselves from the arbitrary bureaucracy." (P29, development manager)

A team may also find S4 useful when they have a non-agile team culture (TEAM CULTURE, F4), so that the team can make use of the explicit architectural guidance, and when the team has little architecture experience or technology experience (EXPERIENCE, F6), so that they can provide the team with a set of explicit architectural decisions.

*

Figure 7.5 shows the relationship between S4 and the forces and other strategies. A team typically uses S4 when one or more of S1's success factors are absent: an agile TEAM CULTURE (F4), CUSTOMER AGILITY (F5), or architecture or technical EXPERIENCE (F6). In these instances, BIG DESIGN UP-FRONT provides explicit team guidance to the team with a non-agile culture, provides comfort to a non-trusting, non-agile customer and provides explicit decisions for teams with little architecture or technology experience (EXPERIENCE, F6).

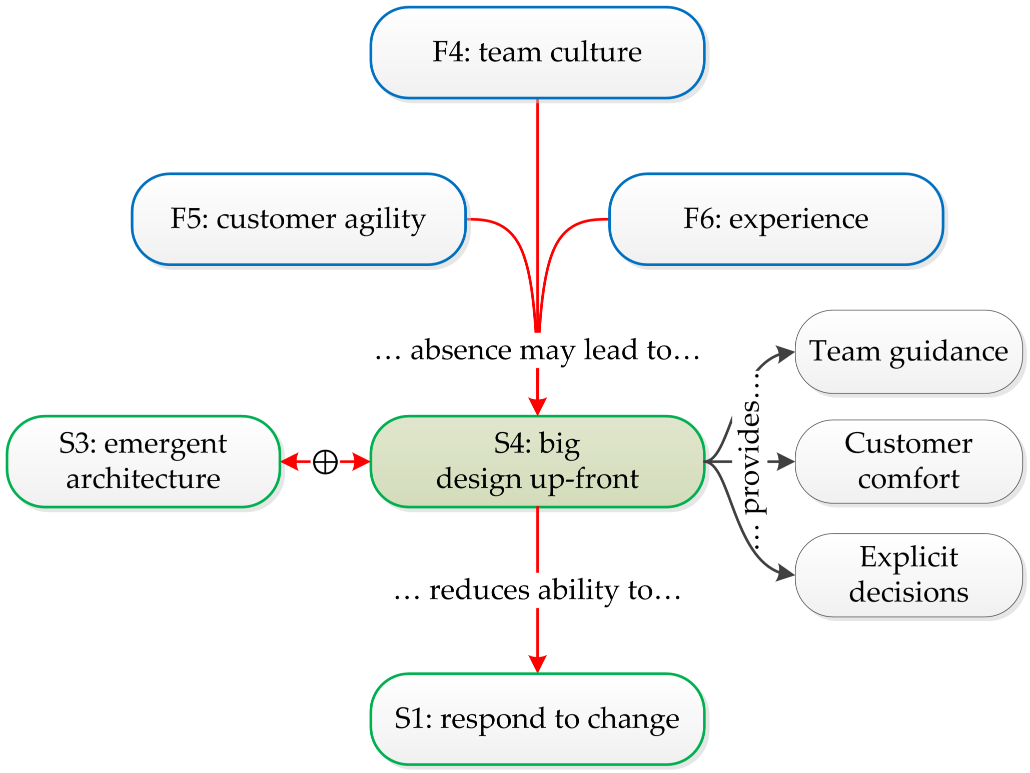

Figure 7.5: The relationships between BIG DESIGN UP-FRONT (S4) and other forces and strategies

7.5 S5: Use frameworks and template architectures

DEFINITION OF S5: USE FRAMEWORKS AND TEMPLATE ARCHITECTURES

Using frameworks and template architectures is a strategy in which teams use standard architectures such as those found in development frameworks, template or reference architectures, and off-the-shelf libraries and plug-ins. Standard architectures reduce the team's architecture design and rework effort, at the expense of additional constraints applied to the system.

Software frameworks such as the Java platform, .NET, Hibernate and Ruby on Rails are infrastructures that support the development of software systems through the use of predefined libraries, components, tools and abstractions. Frameworks typically include default or template architectures (or architectural patterns); many even constrain the system to a particular architecture or pattern. For example, Oracle and Microsoft recommend certain architectural patterns for different application types built using Java and .NET, respectively. Hibernate inserts an additional layer between the application and database and Ruby on Rails, like many website frameworks, imposes a model-view-controller (MVC) pattern.

Template and reference architectures are predefined architectures or architectural patterns, sometimes sourced from a particular framework vendor to be used with that framework. Software engineers can use these templates to solve specific problems. Examples include Microsoft's 'Model View ViewModel' architecture for use in .NET, and CQRS ('command query responsibility segregation'), used to keep data synchronised between different databases.

S5 can be used at the same time as any of the other strategies, S1–S4.

7.5.1 The benefit of frameworks and template architectures

The benefit of using frameworks and template architectures is they provide standard solutions to standard problems. Standard solutions mean software engineers do not need to make as many architecture decisions:

> *"A lot of vendors now have what they call candidate or template architectures, it's just going to be one of those: here it is, you run with it." (P4, director of architecture)*

and

> *"...those kinds of [architectural] things are there in [Ruby on] Rails, so you don't have to think, 'what should I do in this kind of situation,' because it's already there in Rails." (P16b, head of engineering)*

Thus frameworks greatly reduce the effort required to design a system and get it up and running:

> *"Getting a web app working in Rails, I don't think requires more than a day for a guy who has done some development before." (P16b, head of engineering)*

Architectural changes can also be made with a lot less effort when using frameworks [224]: what used to be considered architecture decisions in the past are now sometimes considered design (non-architecture) decisions, or even simply configuration 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:

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

A framework that provides a standard architecture – particularly those that follow the 'convention over configuration' paradigm – also greatly reduces the complexity of the architecture because many of the architectural decisions are embedded in the framework:

> *"You don't really need anyone to look at the database structure and design and SQL anymore." (P29, development manager)*

Frameworks provide less opportunity for the software engineer to make mistakes and fail:

> *"[Ruby on Rails] is a template. There's not much really to choose. People make a lot more mistakes when they're asked to make more choices. So why provide more options, why ask a programmer to make more choices than are necessary? That's my opinion from a business standpoint, it reduces the risk of us making bad decisions based on programmer bias." (P16a, CEO/chief engineer)*

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

7.5.2 When frameworks are not enough

While immensely important – and used almost ubiquitously throughout both the agile and plan-driven development worlds – 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 design is sufficiently complex or unique, there may be no suitable frameworks or existing libraries that a team can use to implement a design, either in part or as a whole. In these situations a team needs to design and build bespoke components or libraries and, for those bespoke components, miss out on the benefits that frameworks can provide.

Teams do extra design as they first identify the parts of the system that cannot be implemented using pre-built components, and then perform analysis and experiments to come up with satisfactory solutions:

> *"We had a lot of issues using Microsoft's off the shelf workflow product for this, and this was dragging on and on and it was looking really ugly and the customer looked horrified when they saw the prototype, and in the end we had no budget to do anything else in terms of investigating it, and I said, 'this is ridiculous. Sorry to do this, but I'm going to take a Saturday, a Saturday that I should be spending with the kids, and I'm going to sit here and write a prototype of how we could do our own workflow,' which is what I did and which is the path we ended up going down." [...] "In the workflow engine the fundamental principles remained in line with what we learned in line with the original prototype, but the level of complexity to make that work was beyond what we'd expected." (P13, architect)*

and

> *"We had enormous problems with printing, because we discovered part way through the project that Microsoft offers no supported way to print from .NET code from a server. It took me a month to even get them to give me a straight answer to that. They just don't! You can print from desktop apps, or you can serve a page to the browser and the user can print it. But if you want the server to talk directly to the printer there's actually no supported way of doing that with pure .NET code. So we have a hybrid that makes some non-native calls at one point. And that was staggering, I had not anticipated that would be a problem at all. The .NET framework, very rich framework, we never examined that as a possibility! So that was a big surprise. [...] If we'd known that server-side printing was not supported we probably would not have built a web app. It probably would have tipped us to the other decision, then built a desktop app." (P13, architect)*

The need for bespoke components increases the complexity of the architecture decisions that need to be made and technical risk – and hence increases the up-front effort required.

Alternatively, a solution may require multiple frameworks or technologies to implement the required features and functionality. Not only does selecting these frameworks require extra planning, but setting up automatic testing platforms, continuous integration delivery and other related set up activities become more difficult and require more effort:

> *"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)*

Extra effort is also required to ensure the technologies are able to communicate with each other efficiently.

∗

Figure 7.6 shows the relationship between S5 and the other forces and strategies. S5 can be used with any of the other strategies, S1–S4, and reduces architectural and development effort. However, most significantly, S5 increases the architecture's agility by increasing its ability to respond to change (RESPOND TO CHANGE, S1), and provides standard architecture solutions that reduce complexity and risk (TECHNICAL RISK, F2). S5 therefore allows teams to reduce their up-front effort and increase the agility of the architecture.

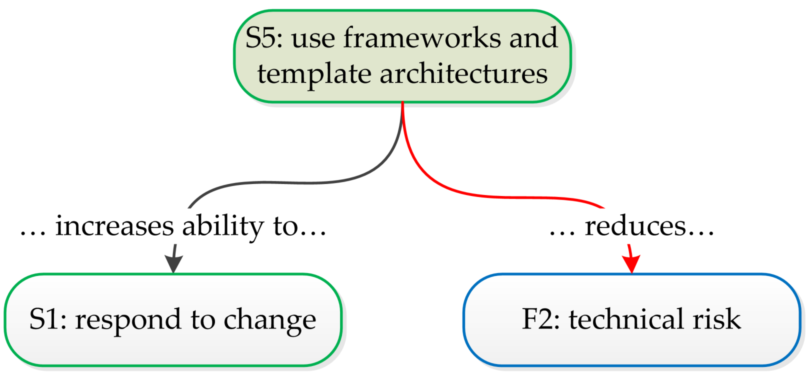

Figure 7.6: The relationships between USE FRAMEWORKS AND TEMPLATE ARCHITECTURES (S5) and other forces and strategies

Chapter 8

The agile architect

"Emergent structural design may be achieved by facilitating collaborative design, by delegating to individuals, or by using a mixture of both. The quality and coherence of the software architecture is largely dependent on how well this is done." (Paul Taylor [218])

In agile software development, the architecture is not the responsibility of a single designated architect (or team of architects). Architecture decisions are usually made collaboratively within the team. Collaborative decisions and a wide understanding of the architecture help improve the communication within the team, and help make the team more agile (F4, TEAM CULTURE). When the scope of the system the team is building extends beyond the team to other teams, additional input into architecture decision-making is required, by architects who have an understanding of the overall business problem being solved and the ASRs. If this wider architecture is designed collaboratively with the team then, like having an agile-friendly customer, the team's agility is improved (F5, CUSTOMER AGILITY).

Section 8.1 examines who makes the architecture design decisions for a team. In addition to making architecture decisions, the architect in an agile team may also perform other roles. Section 8.2 examines some of these roles.

8.1 The architecture decision makers

Responsibility for architecture decisions varies between teams. In all cases, it is important that the architects are active members of the team – usually the team's experienced developers:

> *"One of my fundamental beliefs is [...] architects should be active coders in the team who are just seniors and ideally that pool of seniors in your team act as a kind of proxy architecture committee." (P29, development manager)*

– rather than so-called *ivory tower* architects with one-way relationships with the development team:

> *"...someone who's supposedly got the title [of 'architect'] sitting in an ivory tower, and has never actually built that thing in the last ten years because they've been thinking high-level." (P29, development manager)*

Agile teams can use consensus-driven decision making, they can have decisions approved by a single, nominated team member, they can have decisions approved by non-team members, and, when appropriate, decisions may be made outside the team.

8.1.1 Whole team consensus

When the team is small and consists entirely of senior software engineers, decisions can be made democratically or by consensus, with the whole team planning the architecture together and coming to a team-wide agreement. This method of making architecture decisions was used by P2's team of three developers and P19's team of four developers. These teams made decisions using a mixture of group discussions and individual research or analysis:

> *"We had some discussions, the team, and we'd break and say, why don't you research this, why don't I research this, and why don't you research that, and then we'd do our independent research, come back and have this discussion again. And [negotiate] over what paradigm we're going to take or what technology we're going to adopt." (P19, Manager)*

While more effective in small teams, this method is rarely successful in medium or large teams because of the difficulty in a large number of software engineers reaching agreement. For example, P12's team consisted of seven senior developers who made architecture decisions by mutual agreement and found coming to a consensus very time consuming:

> *"There was no leader, so a decision was a democracy" [...] "All six or seven of us were involved with all architecture decisions. We often had planning meetings which went on for a long time, and every single person was involved, and we used a lot of whiteboards, we used a lot of sessions which involved one person being nominated to go out and draw it up [...] and then we'd come back together and go over all of it again." (P12, senior developer)*

Allocating responsibility for decisions to everyone has the risk that no agreement will be reached at all:

> *"Most of the time it worked." (P12, senior developer)*

In contrast, P13 was not able to share architecture decision-making responsibilities between even two architects:

> *"We actually had two architects, and in hindsight we had foolishly said to the boss, no, don't make one of us the lead, we'll co-lead it, and we had great difficulty with that." (P13, architect)*

His team changed partway through the project after one architect moved on, so that decisions were approved by a single team member (below), which they found much more successful.

P29, with a team of fourteen, had more success with team democracy:

> *"As issues that are going to affect the whole teams' standards or model or structure come up, which I see as the design and architecture decisions, then the rules are that there's a joint ownership of those, there shouldn't be a single [hat] making a rule, so everyone comes together to the table, nut it out as quickly as they can, decide if there's stories to go on the board about proof of concept, assign any research that's needed to someone – the team should assign that responsibility to someone – and then make a call and move on." (P29, development manager)*

A common factor among the teams making decisions by consensus are all the team members have similar levels of experience with the technology and all have a good domain knowledge and understanding of the business problem; there are no junior developers who do not have sufficient understanding to make the required architectural decisions.

The need to have an understanding of the business problem is consistent with the definition of architecture in section 2.3: those who make the architecture decisions are those who have a good understanding of the ASRs and how they fit into the context of the business. Consensus decision-making works best on single-team projects; there are no other teams they need to coordinate with.

8.1.2 Decision approval within team

Large teams can avoid the problem of having to reach a team-wide consensus by having a designated architect who makes the final decisions or has a 'casting vote'. The architect is effectively the architecture *owner* [14] or *coordinator* [79], and as such does not make architecture decisions on their own: they use their experience and knowledge of the business problem to confirm (or overrule) the decisions made by the team, or, in the case of lack of agreement, make the final decision:

> *"[The architect's role is] to make sure all the decisions are in line with the goal and fine tune to suit the thing [being built]." (P22, senior manager)*

and

> *"If there are long discussions that don't seem to get anywhere then somebody has to make a decision, so that's [where I step in] but apart from that it's very democratic." (P19, development manager)*

P30 summed up the role of the architect as a guidance and sanity checking role.

Having a single architecture owner within the team means the team does not need agreement from each team member. Certain members, such as team members inexperienced with the technology, can be excluded from the decision-making process if necessary.

While a team may largely comprise members with the title of 'developer' or 'software engineer,' the title of the team member who has responsibility for the architecture is typically 'architect' (e.g. P14), 'lead developer' (e.g. P8), 'senior developer' (e.g. P12) or 'technical lead' (e.g. P28). Despite the different titles, they carry out similar roles and thus the titles could be considered synonyms:

> *"My role is Java architect, and I'm leading a team of about ten people." (P21b, architect)*

The architecture owner's title may also be 'team leader' (e.g. P21c, P25, P26) or 'team manager' (P18, P19, P21a), perhaps contrary to the agile practice of teams being self-organising.

It is important to have someone with overall responsibility for the architecture when the project spans two or more teams, and it is not possible for the whole team to have an understanding of the overall business problem and architecture. For example, P8, lead developer (and team lead), discussed a problem that a two-team development project had which illustrates the importance of having an architecture owner role in larger projects. Prior to P8 joining the project the teams did not have an overall leader who understood how the different parts should fit together to address the business problem. He described the problems this caused:

> *"One of the problems was that the decisions weren't made at the beginning by somebody leading, they were kind of made by disparate groups [...]. By them not having someone there at the beginning who says 'this is my vision' – and I'm not saying somebody should be completely controlling but they would obviously take input from all the teams – essentially [they needed] one person coming out and saying 'ok, I've listened to what everybody's said and I can now see how the whole thing's going to be laid out' and by not doing that, it was something of a setback." (P8, lead developer)*

This lack of architecture owner meant each team designed their sub-system architecture in isolation, rather than designing for the overall system, which led to the overall system's performance requirements not being met and being hard to debug:

> *"...We noticed that the indexing was very slow. They [one of the teams] were doing work to speed it up [...] but they just hadn't noticed that they were looking in the wrong place because they didn't think they had any reason to look in the place that wasn't their responsibility. So it was only when I was just sitting there watching the queries come in, 'Why is it doing that? That's why it's taking so long.' All the optimisation is happening in a different place, but we could increase the speed ten-fold by making one tiny change in a place which previously they hadn't been looking at." (P8, lead developer)*

The need to understand the architecture across multiple teams becomes more important as the size and complexity of the system being built increases.

As the number of teams increases, it may become more appropriate to have decisions approved or made outside the team.

8.1.3 Decision approval outside team

In some cases, particularly within larger projects, a team makes the architecture decisions, but they must be approved by an architect or architects from outside the team before they can be implemented. Often the system being built by the development team is part of a much larger system, and the team does not have a detailed understanding of the architecture of the overall system and the business problem being solved. Therefore a single team cannot be sure that the architecture decisions it makes are appropriate in the context of the bigger system. In these instances, any architecture decisions that the team makes must be approved or reviewed by architects who are outside the team and are across the full context of the system being built, and understand how the team's architecture fits in with that system. These external architects may belong to the organisation with the overall responsibility for building the larger system or the customer itself.

External architects typically take the form of a dedicated architecture team (for example, P33–P36), an architecture review board (P23) or an architecture governance committee (P29). These groups have the responsibility for ensuring architectural consistency across the system – or across multiple systems, such as a product line.

Two participants from the banking industry, P25 and P29, both had their architecture decisions reviewed by company architects. The motivation behind these reviews was consistency across the company:

> *"The company architects also have a lot of power and they are the ones who essentially decide the overall architecture – not just of your system but also of all systems. So given the size of the company, they wanted to standardise as much as they possibly could. Our site didn't have a database, but if they did decide to give us one, it would have*

> *been a SQL Server database, because that's what we use for the .NET side of things." (P25, team lead)*

and

> *"We are assigned an architect from the central bank pool who monitors everything that is going across our team, and decides when we have to put a paper to that central [architectural governance] committee or not. We do our best every time to convince him that we don't have to, and he's pretty flexible and realistic on that." (P29, development manager)*

P23, an engineering manager working for a company developing software for the pharmaceutical research industry, was a team member who was also a member of an architecture review board that reviewed architecture decisions to ensure consistency across their entire suite of products:

> *"How we go forward for changing architecture is we have an architecture review board – we are the key members involved in a couple of [products]" [...] "We all sit together and review every week any changes that are happening in any of the products. And everyone will have their opinion and a lot of things get discussed." (P23, engineering manager)*

Participants P33–P36 worked for a company that develops a suite of healthcare management products customised for individual customers. Their organisation has an evolving high level reference architecture that is common to all products and which ensures consistency across these products. This reference architecture describes "the functional architecture, the deployment architecture and the component architecture" (P33, product architecture team leader). An architecture team approves each team's architecture designs, ensuring that they are in line with the reference architecture and that the high-risk ASRs such as privacy and security are met:

> *"Before [the development teams] start developing they have to have a number of sign-offs, and one of them is [...] a technical review, so that's basically members of my [architecture] team and a bunch of other senior technical people – they [the developers] have to tell us what they're going to do, we say, 'yep, that's all good,' or 'no, we're not happy with that, you need to change that because it doesn't line up with the reference architecture,' or, 'why are you doing that?' And either they might explain why that is, and we go, 'oh, ok, now you've explained it it makes sense.' [Or] we go, 'no, we're not quite happy with that'; some of them might have to go and revisit the technical design." (P33, product architecture team leader)*

P33 encouraged the teams to have complete designs before the review process, so that the review is little more than a rubber-stamping or approval exercise:

> *"We try to turn it very much into a rubber-stamping exercise, so we encourage the teams to get other people – members of my team and other senior technical people – to get involved in the actual design process when they're working through workshopping things and making those [architecture] decisions so that when we get to the tech review, it's not really a tech review, it's more, 'yep, that all make sense, away you go.' " (P33, product architecture team leader)*

The clinical and privacy requirements of this project are technically challenging, and have a high level of risk associated with them. The clinical risks are assessed in parallel to the architecture approval by an independent team to ensure the integrity of patient data will be maintained:

> *"I think from a doctor's point of view, the thing they fear the most is that they are making decisions about people's lives every day. And the information that we display on our screens helps them make those decisions." [...] "I think that's more serious than patient privacy.*

> *If someone finds out something about you, well, you're a bit embarrassed, but you're still going to live! But if someone doesn't receive news or receives the wrong news, we are actually talking about harm." (P34, development unit manager)*

Challenging requirements are an important contributor to architectural complexity – see F2 (TECHNICAL RISK). Addressing the risk associated with these requirements is an important strategy in up-front design – see S2 (ADDRESS RISK).

In other instances, it may be the customer who has the oversight of the overall architecture, and thus makes or approves the architectural decisions. For example, P7's team was one of many from different ISVs building software as part of the customer's enterprise system; no single team had a complete vision of the overall system, and thus the customer made the architecture decisions:

> *"The customer's got architects galore. Given that this involves other teams for these back-end systems and other teams for this and that, each team has their own component architects. Then 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." (P7, business analyst)*

While P7's team did not make the important architecture decisions, they did most of the architecture analysis and research behind those decisions, and put together recommendations for the customer:

> *"We developed a routine that in one iteration we would do a discovery task, which was normally find the high level architecturally significant requirements and produce an options paper on the architectural options for this." [...] "At the end of that [discovery phase] we aim to have it agreed with the client which architectural approach to take." (P7, business analyst)*

The options paper was a document that included the following information for each architecture decision to be made:

- the decision to be made
- the issue or problem
- assumptions
- motivation
- alternatives
- the recommended decision
- justification for decision
- implications.

In addition to decisions being approved outside the team, there were also instances of architecture designs being reviewed by third parties before development started to confirm the quality of the architecture designs. In one such case this was to give the customer reassurance of the ability of the development team:

> *"I think we were winning the customer's trust to a degree [...] so she wanted some independent QA of the project." (P14, architect)*

P14 noted that because of the need for this review, they had to do more design up-front than otherwise necessary – see F5 (CUSTOMER AGILITY). An independent QA assessment is most useful after the up-front design has been completed and before any development has been started, so that if any changes are required, no development effort is wasted. P14 did note that he had been subject to architecture reviews *after* the project had finished – at which point any changes are very difficult:

> *"How the hell are you going to make it better if it's already done?" (P14, architect)*

and

"There is a point of no return with architecture." (P17, manager/coach)

In another example, the team itself arranged an external review to give the team comfort that its designs were satisfactory:

> *"I had a design review with one of the programme managers [at Microsoft] earlier last week. It was good! On the topic of architecture it was a good design review because I was worried we didn't have enough architecture [...] So I sat down with this guy from the SQL Azure team, and said, 'look, this is our architecture, we're using this, this, this and this, our database model is tiny, you know, less than two dozen tables, that's what the customer wants!' [...] Basically then we went through the design and reviewed it. That was good." (P2, developer/architect)*

8.1.4 Decisions made outside team

In this situation, architecture decisions are made by architects who are not day-to-day members of the development team. Only one of the participants, P11b, was an architect who was not a coding member of a team. P11b was a member of the 'architecture team' that was responsible for the high-level architecture designs for a number of development teams, and collaborated closely with them, but was not a member of those teams. More detailed architecture designs were the responsibility of the teams themselves.

> *"Initially we'll have a high-level, very abstract... we work in our [architecture] team and we had an external review last year to say that that was a sound approach. But when we get into the detail of how we'll actually build it or what it's going to look like then we start working with the people with the expertise in that area." (P11b, architect)*

Agile teams generally view non-coding architects as undesirable because these architects are often disconnected from the system being built and therefore are not in the best position to design the architecture:

> *"If they [the architects] don't have to code with the team, then they generally don't have the pain of the team and they don't know what the latest frameworks are doing... if you've never tried to flesh it out then you're never going to know if it's going to work. And I think there's a significant problem with places that separate those roles [of developer and architect]." (P29, development manager)*

Many teams have constraints on their architectures which are imposed by the customer. These constraints may be high-level architecture decisions, such as the technology stack to be used. For example, P7's customer made platform decisions based on strategic partnerships:

> *"[The platform] was a [customer] decision. They've got a strategic partnership with IBM." (P7, business analyst)*

P8's customer made platform decisions when they split the development between different ISVs, each who specialised in different technologies:

> *"The customer decided the architecture when they went out and contracted all the parts." (P8, lead developer)*

P14's customer made platform decisions based on their own ability to support the technology:

> *[The platform] was probably almost determined already for me, before then! They [the customer] had already gone down the path of .NET, they'd built a bunch of applications in .NET already, so it would have been a bit of a silly decision to change completely, given that they had to support it. They were going to bring on support internally, they weren't going to have anyone contracted to do it, so they needed to have their own resources that could support all their applications." (P14, architect)*

8.1.5 Decisions made at the inappropriate level

It is important to have the architecture decisions made at the most appropriate level. A number of participants described problems caused by multi-team systems not having anyone with oversight of the common architecture.

For example, each of P28′s teams designed and built their own solutions. Similar websites had different architectures and became very difficult to maintain when changes had to be made across all websites:

> *"But if you're trying to deliver a [website media] gallery and it needs to have the same [construct across all websites] as you can see, you've got to abstract to a common set. So you don't duplicate. [...] You've got to do some architecture because twenty teams are developing their own sets of behaviour – and by behaviour I mean following the exact same [content] life cycle – content, title, subtitle, publishing date, issue date. They all have to be published, they all have to be scheduled. They all generate static files for them. [...] So with no design up-front [...] you have more than twenty [different] models following the same behaviour [and] you know that in six months you're going to have three hundred people working [on different versions of] the same basic model; not doing some architecture is insane!" (P28, technical lead)*

He believed the cause of this problem was the teams not doing enough architecture – they took the agile philosophy of minimising up-front design too far.

In another example, P8 described a team that made a wrong technology decision (prior to him joining the team), which led to a lot of refactoring to try to make it work. In the end the technology was dumped for a better solution:

> *"That [CMS solution] was probably the wrong decision" [...] "So we had to do lots and lots of work, refactoring work, in order to enable us*

to upgrade the underlying libraries. That was a huge problem." [...] "It didn't work in the end and I think a lot of that work was eventually abandoned, better solutions were found." (P8, lead developer)

This architecture design problem was caused (or exacerbated) by having two siloed teams lacking in experienced leadership that did not communicate their architecture needs with each other sufficiently.

Both of these examples could have been avoided if the teams had their architecture decisions approved by somebody with oversight of all teams' systems.

8.1.6 Summary

All participants in this research described architectural decisions as being either made by members of the development teams or in agreement with members of the teams. No architects made decisions in isolation of the team. Decisions are either made and approved by the teams themselves, are made by the teams and approved by external architects, or were researched by the teams themselves and the decisions made by the external architects. In the latter cases the external architects ensured the architectures of the teams' sub-systems met system-wide requirements or constraints that were perhaps not clearly visible to the teams themselves.

Having the whole team involved with the architecture design process ensures the whole team has an understanding of the architecture, its rationale and its importance. This understanding improves the team's ability to communicate and collaborate, and hence its team culture (F4, TEAM CULTURE) and its ability to use S1 (RESPOND TO CHANGE).

8.2 Other roles of the architect

Teams using agile methodologies prefer architects who are coding members of the team. As such, architects in agile teams often have other responsibilities or roles within the team, in addition to their architecture design responsibilities. These roles, typically informal, may include being the technical lead or 'über developer', knowing the 'big picture', creating a shared mindset for the team, creating (and enforcing) development guidelines, and driving the development methodology. These roles are described below.

8.2.1 Technical lead/'über developer'

The architect in an agile team is not usually a full-time architect: they are often a member of the team who also makes architecture decisions. Architects are likely to be experienced software engineers (section 8.1.2) who write code alongside the non-architecting team members:

> *"Architects should be active coders in the team who are just seniors." (P29, development manager)*

Because of their experience, architects often have advanced development skills; the architects use their experience and skills to solve the team's development problems and make the best decisions:

> *"I was wearing the hats of architect, BA, and also coder of tricky bits – being one more of the senior people on the team, that sort of stuff came my way." (P13, architect)*

Many of the architecture decisions made by external architects (for example, architecture teams), described in section 8.1.3, were made following recommendations by team members. In some cases the members of the teams making these architecture decisions are selected from the development teams themselves (e.g. P23, P33–P36).

8.2.2 Knowing the big picture

An architect who understands the big picture is able to relate the development to the business needs, and is better able to understand what the customer actually needs and why they need it – one of the important differences between agile and traditional development methods:

> *"Not only does he or she [the architect] need to be knowledgeable about the technical part of it but also the business part of it. I think that's a big change that agile brings in to the picture." (P21, manager/coach)*

and

> *"[When using agile methods] you're more, you know, working with the business more, you've got demos and generally it's more positive because they're getting more of what they want instead of waiting three months and getting a turkey." (P11a, development manager)*

In this regard, the architect has a lot in common with the business analyst (BA) role; some architects do indeed carry out the functions of a BA:

> *"I was architect, I was in effect lead BA, I seemed to be the person who had the unified view of what the customer was after in terms of functionality." (P13, architect)*

and

> *"[Team member's name] is the business analyst and plays a part lead in the architect role." (P21, manager/coach)*

and

> *"I think architects need to understand [engaging with the customer] – they need to seek support, lobbying, champion for the business what they can get out of doing this role, they've got to coach the business in what it's going to cost them, how much effort, what expectations they need to have, we can get really powerful outcomes that you need,*

> *rather than building all this stuff that you thought you needed back then but turns out to be something different." (P30, consulting architect)*

A number of participants involved with outsourced work described having architects provided by the customer in their team (for example, P21, P22 and P26), because of their better knowledge of the business problem.

8.2.3 Creating a shared mindset

This thesis defined architecture as the high level design decisions that the expert developers need to understand (section 2.3) – a shared mindset. It is the responsibility of the architect to create that shared mindset:

> *"The architect [is] the builder of a shared mindset, a shared understanding, and to me that's the architect's role." (P5, coach/development manager)*

– and to communicate decisions so everyone knows what they are building:

> *"The important thing about architecture is, is it communicatable?" (P5, coach/development manager)*

As well as communicating decisions as they are made, the architect brings new team members up to speed with the project:

> *"It [the architecture document] helped new people coming in to understand what the objectives were." (P30, consulting architect)*

and

> *"We're not necessarily writing this documentation for you, we're writing it for the person who's going to... pretend you're a new programmer who's just arrived and you've been pointed towards the project and Subversion and told to maintain it... what would you like?" [...] "Think of it [the Software Architecture Document] as an aid to the new person who's going to start." (P4, director of architecture)*

8.2.4 Creating and enforcing development guidelines

The role of the architect includes providing development guidelines to the team. Guidelines ensure that the team follows good design and coding practices, that they maintain consistency of coding styles, and that system quality is maintained.

For example, P5 talked about the architect making decisions on coding styles:

> *"When I think of architecture I expect there to be a definitive answer to the question: should you use C++-style exceptions or C-style error routine codes? [...] At a very fine grained level, you know I'm writing a function. Do I throw an exception or do I return an error code?" (P5, coach/development manager)*

P4 and P10 mentioned using tools to measure the quality of modules and classes:

> *"One of the tools I used [...] is a thing called Structure101 which essentially at the class level, package level, in Java, just analyse all the source code and identify dependencies between classes, and it would identify cyclic dependencies, so you could know, we're in a cyclic relationship here, what do we need to move to break the cyclic relationship?" [...] "Just ensure that they were following good approaches." (P4, director of architecture)*

and:

> *"Sonar gives you an instant view of the code health, and you can see trends and coupling of classes. That way you can see – 'oh, that module there is being really naughty, look at the complexity spikes there showing up' – all of a sudden the architecture community can zoom in and say, 'Oh! What are you guys up to? How come your cyclomatic complexity is so high these days?' " (P10, coach)*

8.2.5 Driving the development methodology

Team members who make the architectural decisions are also often in a position to drive the development process and methodology. They need to be aware of how the agile methodology they are using affects the architecture design, and they need to be aware of how the methodology is affected by the design:

> *"Architects need to help agile teams work well, [they] need to understand how agile processes work, and they need to understand what agile teams need over and above more traditional approaches." (P30, consulting architect)*

Because the architect is aware of the business goals and the best way for the software system to be built to satisfy those goals, they also need to be aware of the best processes to build that software:

> *"Architects are influencers in this [world] and they need to be actually promoting and understanding the real value that [for example] bringing functional testing right into the iterations offers, and understanding that deployment, automation of testing, is really a fundamental need and it takes a lot more work than people give it credit for, and it needs that extra support to make that happen. At least in the early stages." (P30, consulting architect)*

This role is similar to an agile coach or Scrum Master role [119].

8.2.6 Summary

Team members responsible for architecture decision making often perform many other roles in the team, usually informally. These roles include being the problem solver (über developer), knowing the big picture, creating a shared mindset with the rest of the team, creating and enforcing development guidelines, and driving the development methodology. These roles show that the architect is an important member of the development team and contributes to team culture (F4, TEAM CULTURE).

Chapter 9

Discussion

"If you only foresee risks in your project that can be handled by refactoring then you would not do any architecture design." (George Fairbanks [91])

This chapter discusses the theory of agile architecture in the context of related work. First, section 9.1 discusses the theory's presentation and its boundaries, and evaluates the theory. The remaining sections in the chapter discuss the theory in light of the literature. Section 9.2 discusses what the literature says about agile architecture, and its relationship with the theory of agile architecture. Section 9.3 discusses what the literature says about what affects the up-front architecture design, and compares it with the research findings about the forces in Chapter 6. The section includes discussions on context, size and complexity, the architectural effort 'sweet spot,' team culture and architectural experience. Section 9.4 discusses the *twin peaks* model of requirements and architecture. Section 9.5 discusses what the literature says about the role of the architect, and its relationship with the research findings. Finally, section 9.6 provides additional insights into the findings.

9.1 The theory of agile architecture in context

9.1.1 Presenting a grounded theory

A grounded theory is a formal explanation of events being researched [233]. It is not, as Adolph et al. wrote, "an inventory of concepts annotated with quotes from research participants" [8]. Suddaby (2006) wrote that presenting the research data (for example, quotes from interviews) is a sign that the researcher has not lifted the data to a conceptual level [214]. While the theory presented in this thesis is annotated with direct quotes, this is to highlight the traceability between the data and the theory, as recommended by Miles and Huberman (1994) [167] (see Table 9.5 below). These quotes do not form part of the theory, which could stand alone without the quotes.

9.1.2 Theory boundaries

A grounded theory can only be applied to the substantive area from which it was generated, and therefore the boundaries must be well defined [227]. The boundaries of this research were determined by the choice of participants and their contexts. The boundaries are:

- Agile software development – the theory only includes software developed using an agile development methodology, which welcomes or responds to change. Non-agile methods, such as plan-driven methods, or systems where the requirements do not change, are excluded from this theory.

- Software architecture – the theory only includes software architecture. There are many types of architecture within the IT industry – Brown lists seventeen [57]; examples include higher level architectures such as enterprise, business and IT architecture, and lower level architectures such as data, information and security architectures. This theory only includes software architecture (sometimes referred to as solution architecture), the architecture of software systems.

- Application software – the theory only includes application software running on general purpose computers or servers. Other types of software such as embedded software, compilers and operating systems are not included in this theory. Games are also not included.

  The application software may be mass market products or services with ongoing development, or specialised one-off systems. They may also be new (green field) systems or the redevelopment of existing systems.

The theory does not examine any particular techniques for making architecture decision or for generating architecture models (including defining requirements and analysing the architecture), beyond those that are specific to agile methodologies and which affect up-front design effort, such as spikes (section 7.2.2).

The theory does not attempt to determine how much architecture teams should design; rather, its purpose is to determine what affects how much architecture teams design (section 2.5). In other words, it answers the question "how do teams determine how much architecture?" rather than "how much architecture?"

9.1.3 Evaluating the theory

Traditional methods of evaluating research intended for quantitative, deductive research strategies are not suitable for qualitative strategies [81, 113]. Quantitative research evaluation requires ensuring the findings are well grounded in the data rather than verifying the reliability and accuracy of research procedures and tools and interpreting results [81].

There is still much discussion on how to evaluate qualitative research [80, 81, 113], and many possible approaches. One approach is to use criteria that are counterparts of quantitative research criteria. Lincoln and Guba (1986) is a frequently-cited reference that proposed the *trustworthiness* set of criteria that parallels the quantitative *rigour* set of criteria. These criteria are summarised alongside the concerns they address in Table 9.1.

Concern [81, 167]	**Rigour criteria for quantitative research [154]**	**Trustworthy criteria for qualitative research [154]**
Truth value: are the conclusions drawn from the research accurate?	internal validity	credibility
Applicability: are the conclusions applicable to other settings?	external validity	transferability
Consistency: is the methodology consistently applied?	reliability	dependability
Neutrality: are the conclusions free from unacknowledged bias?	objectivity	confirmability

Table 9.1: The quantitative *rigour* criteria and Lincoln and Guba's parallel *trustworthy* criteria for assessing qualitative research [154]

While convenient, Lincoln and Guba had reservations about their trustworthiness criteria, because they are derived from quantitative criteria. They also proposed an alternative set of criteria, the *authenticity* criteria, which are intended to be more suitable for qualitative strategies. The authenticity criteria include fairness, ontological authenticity, educative authenticity, catalytic authenticity and tactical authenticity [154]. These five criteria are not well defined, but, roughly, fairness refers to the researcher treating differences in the participants fairly, ontological authenticity refers to the participants having a better understanding of their situation after the research, educative authenticity refers to the participants having a better understanding of others' situations after the research, catalytic authentication means the research must facilitate and stimulate action, and tactical authenticity means the research must empower action [113, 154].

I have referred to other sources for suggestions on evaluating qualitative research to augment Lincoln and Guba. Frequently cited researchers include Miles and Huberman (1994) [167] and Creswell (2014) [81]; Charmaz (1986) [64] and Glaser (1978) [104] both published criteria specific to grounded theory. Several of these sources drew on the work of Lincoln and Guba. Miles and Huberman (1994) and Charmaz (2006) each had extra categories not included in Lincoln and Guba's set of four trustworthy criteria: these are *originality*, *resonance* and *usefulness* [64], and *application* [167]. Usefulness and application are similar to Lincoln and Guba's *authenticity* criteria.

These criteria are examined below, each summarised in a table that contains techniques that can be followed to help meet its respective criterion, or queries that can be answered to help check whether or not the criterion is followed. Miles and Huberman's criteria included fifty procedures or checks; for brevity, they are only included in the tables where they are the same as other sources, or where they are significantly different from the other sources and are pertinent. Each technique or query also contains my own assessment as to whether or not the theory presented in this thesis meets the criterion.

Credibility

The credibility criterion is used to determine if the conclusions drawn from the research are accurate: do the findings make sense? Do we have an 'accurate portrait' of the research subject [167]? Techniques for helping meet this criterion and queries to help determine whether the criterion has been met are listed in Table 9.2, along with my assessment of that criterion.

Technique	**Comment**
Prolonged engagement: "lengthy and intensive contact with the phenomena (or respondents) in the field to assess possible sources of distortion and especially to identify saliencies in the situation." [154]; also [81], **intimate familiarity** [64]	This research included 44 participants in 37 first interviews, at an average of 1 hour 10 minutes per interview (section 4.1). This is lengthy and intensive contact.
Persistent observation: "in-depth pursuit of those elements found to be especially salient through prolonged engagement." [154], **sufficient data** [64]	Yes: data collection continued until saturation (section 4.1.4).
Triangulation of data: "by use of different sources, methods, and, at times, different investigators." [154]; also [81, 167]	Data triangulation was achieved through interviews and documentation (section 4.1.3). I did not have the resources for methodology triangulation or additional investigators.
Peer debriefing: "exposing oneself to a disinterested professional peer to 'keep the inquirer honest,' assist in developing working hypotheses, develop and test the emerging design, and obtain catharsis." [154]; also [81]	Peer debriefing was limited to my academic supervisors. I did not have the resources for additional peer debriefing.
Negative case analysis: "the active search for negative instances relating to developing insights and adjusting the latter continuously until no further negative instances are found; assumes an assiduous search." [154]; also [81, 167]	Negative cases (problems and issues) were actively sought (see the interview questions in Appendix B).

Table 9.2: Techniques and queries for assessing the *credibility* criterion

Technique	**Comment**
Member checks: "the process of continuous, informal testing of information by soliciting reactions of respondents to the investigator's reconstruction of what he or she has been told or otherwise found out and to the constructions offered by other respondents or sources, and a terminal, formal testing of the final case report with a representative sample of stakeholders." [154]; also [81, 167]	Noting Glaser's advice on now asking participants to review the theory (see section 3.5.2), member checking was used with caution. Later interview questions were formulated specifically to elicit views on the developing theory.
Comparisons: "have you made systematic comparisons between observations and between categories?" [64]; also [167]	Yes: constant comparison was continued throughout the analysis (section 4.2).
Wide range: "do the categories cover a wide range of empirical observations?" [64]; also [167]	Yes: the wide range can be seen in the range of participants (section 4.1).
Logical links: "are there strong logical links between the gathered data and your argument and analysis?" [64]; also [167]	Yes: see the direct quotes in the findings chapters (chapters 5 to 8).
Evidence of claims: "has your research provided enough evidence for your claims to allow the reader to form an independent assessment – and *agree* with your claims?" [64]; also [167]	Yes: see the dependability and confirmability criteria (tables 9.4 and 9.5, respectively).
Clarify the bias: "clarify the bias the researcher brings to the study. This self-reflection creates an open and honest narrative that will resonate well with readers." [81]; also [167]	Sources of bias are presented in section 3.4.

Table 9.2, cont.

Glaser has four criteria for evaluating the rigour of a grounded theory: *fit*, *work*, *relevance* and *modifiability* [104]. These four criteria all concern *truth value* (Table 9.1), because they all judge whether or not the conclusions drawn from the research are accurate, and hence fit within the *credibility* criterion.

Fit means the categories of the theory must fit the data – the categories must not be forced to fit preconceived concepts. A grounded theory that is carefully induced should automatically meet the criteria of fit [105]. The theory presented in this thesis fits the data: it was not based on any preconceived concepts. For example, I did not deliberately look for the concept describing the design of an agile architecture (defined in Chapter 5) in the data, or any of the strategies for designing an agile architecture; I found them through constant comparison of the incidents and concepts.

Work means that the categories interpret and predict the major variations in the area being investigated. A grounded theory should also automatically meet the criteria of work [104]. Findings of this research have received positive feedback from participants and supervisors. I have presented two conference papers on the early findings of this research [224, 225], which were well received by conference attendees.

Relevance means the theory must be relevant to the area being investigated, and the core category indeed identifies the core issue [212]. A theory that meets the criteria of fit and work has also achieved relevance [105]. This theory is relevant because it identifies the core category, *architecting* (section 4.2.5), which covers the essential concerns of agile teams pertaining to architecture and architecture design.

A theory must be able to be *modified* as new data that presents new variations is obtained [105]: there should be no need for the existing theory to be discarded. Generating a grounded theory is an evolving process that plays out through the iterations of data collection and analysis; the concepts and categories continuously evolve as new relevant data is added. The theory presented in this thesis was continuously modified as it evolved, and it evolved continuously through the analysis: as previously noted, the core category did not emerge until very late in the analysis. The theory evolved from focusing solely on up-front architecture decision-making to wider theory of agile architecture that more completely described the architectural design processes and activities of agile teams. The theory can also be modified to account for any future data. Thus, this thesis presents a theory that is credible and meets Glaser's requirements for a rigorous theory.

Transferability

The transferability criterion is used to help a reader determine if the conclusions drawn from the research are applicable to other settings. Techniques for helping meet this criterion and queries to help determine whether the criterion has been met are listed in Table 9.3, along with my assessment of each technique or query applied to this research's theory. This table shows that the theory is transferable.

Dependability

The dependability criterion is used to help a reader determine if the process of the study is applied consistently and is reasonably stable [167]. Techniques for helping meet this criterion and queries to help determine whether the criterion has been met are listed in Table 9.4, along with my assessment of each technique or query applied to this research's theory. This table shows that the theory is dependable.

Confirmability

The confirmability criterion is used to ensure the study is free from unacknowledged bias: "do the conclusions depend on the subjects and the conditions of the inquiry rather than on the inquirer?" [167]. Queries to help determine whether the criterion has been met are listed in Table 9.5, along with my assessment of each query applied to this research's theory.

Technique	Comment
Thick descriptive data: “narrative developed about the context so that judgements about the degree of fit or similarity may be made by others who may wish to apply all or part of the findings elsewhere.” [154]; also [81, 167]	Multiple participants allow for thick description. The context is well defined by the boundaries of the research (section 9.1.2).
Descriptions: “Are the characteristics of the original sample of persons, settings, processes (etc) fully described enough to permit adequate comparisons with other samples?” [167]	The participants are summarised in section 4.1; the boundaries are described in section 9.1.2.
Threats to generalizability: “Does the report examine possible threats to generalizability?” [167]	The threats are described in section 3.5.2.
Sampling diversity: “Is the sampling theoretically diverse enough to encourage broader applicability?” [167]	Yes: the theory developed through this research is high-level (Chapter 5) with a very broad sample of participants (section 4.1).
Boundaries: “Does the researcher define the scope and the boundaries of reasonable generalization from the study?” [167]	Yes: the boundaries are described in section 9.1.2.

Table 9.3: Techniques and queries for assessing the *transferability* criterion

Technique	Comment
Audit trail, **External audit**: "...both the establishment of an audit trail and the carrying out of an audit by a competent external, disinterested auditor. That part of the audit that examines the process results in a dependability judgement, while that part concerned with the product (data and reconstructions) results in a confirmability judgement." [154]; also [81, 167]	The external audit was performed by my academic supervisors, as is appropriate for PhD research. The audit trail is included in the analysis chapter of this thesis (Chapter 4), and in the findings chapters, chapters 5 to 8, in the form of direct quotes from the data.
Transcripts: "check transcripts to make sure that they do not contain obvious mistakes made during transcription." [81]	All transcripts were personally checked after transcribing, and then sent to the participants for verification. (A few participants made corrections; one even corrected some spelling mistakes!)
Code drift: "make sure that there is not a drift in the definition of codes, a shift in the meaning of the codes during the process of coding. " [81]; also [167]	The constant comparison method meant there was a continuous shift in the meaning of codes (concepts). This shift was captured in the memos attached to the codes (section 4.2.3).
Team research and code cross-check: [81, 167]	Not applicable – I was the only researcher.
Research questions: "are the research questions clear, and are the features of the study design congruent with them?" [167]	The questions (listed in Appendix B) are high-level and general, characteristic of semi-structured interviews. The questions evolved over time as the research evolved.
Researcher's role: "is the researcher's role and status within the site explicitly described?" [167]	Yes: my role is described in section 3.4.

Table 9.4: Techniques and queries for assessing the *dependability* criterion

This table shows that the theory in this thesis meets the confirmability criterion.

Application and usefulness

The application or usefulness criterion (Miles and Huberman (1994) [167] and Charmaz (2006) [64], respectively) are used to determine if the research's findings make worthy contributions. Queries to help determine whether the criterion has been met are listed in Table 9.6, along with my response to the queries as applied to this research's theory. This thesis and the theory presented are useful, accessible and applicable.

Originality

The originality criterion used by Charmaz (2006) [64]) is used to assess the contributions that the theory makes. Queries to help determine whether the criterion has been met are listed in Table 9.7, along with my response to each query as applied to this research's theory. The theory presented in this thesis make significant contributions to theory and to practice.

Resonance

Table 9.8 presents Charmaz's resonance criterion [64], its techniques and queries, and my assessment of this research's theory.

Resonance is used to assess whether or not the findings engage and meaningfully affect the reader. Queries to help determine whether the criterion has been met are listed in Table 9.8, along with my response to each query as applied to this research's theory. The theory presented in this thesis has resonance.

*

Using these criteria, the theory of agile architecture is a trustworthy and good quality theory.

Technique	Comment
External audit: see Table 9.4 [154]; also [81, 167]	
Methodology description: "are the study's general methods and procedures described explicitly and in detail: do we feel that we have a complete picture, including 'backstage' information?" [167]	Yes: the grounded theory methodology is described in detail in section 3.3.
Audit trail: "can we follow the actual sequence of how data were collected, processed, condensed/transformed, and displayed for specific conclusion drawing?" [167]; see also Table 9.4.	Yes: the data collection and analysis procedures are described in Chapter 4.
Conclusion links: "are the conclusions explicitly linked with exhibits of condensed/displayed data?" [167]	Yes: the findings in chapters 6 to 8 contain direct quotes from the data to illustrate the theory and to provide links between data and findings.
Researcher role: "has the researcher been explicit and as self-aware as possible about personal assumptions, values and biases, affective states – and how they may have come into play during the study?	Yes: my role is described in section 3.4.
Competing hypotheses: "were competing hypotheses or rival conclusions really considered? At what point in the study? Do other rival conclusions seem plausible?" [167]	While competing hypotheses were not specifically considered, the core category and theory emerged very late in the analysis. When they did emerge, it was clear that they were the most plausible.
Retained study data: "are study data retained and available for reanalysis by others?" [167]	In accordance with this research's Human Ethics Committee research approval (Appendix A), the research data will be kept for two years. Due to privacy requirements, any identifying data are only available to myself and my academic supervisors.

Table 9.5: Techniques and queries for assessing the *confirmability* criterion

Technique	**Comment**
Usefulness: "does your analysis offer interpretations that people can use in their everyday worlds?" [64]	Yes: this thesis provides contribution to software engineering practice (section 10.2).
Generic processes: "do your analytic categories suggest any generic processes?" [64]	No, I believe the categories are specific to software architecture design in agile development.
"If so, have you examined these generic processes for tacit implications?" [64]	N/A.
Further research: "can the analysis spark further research in other substantive areas?" [64]	Yes: there is potential for significant future work (section 10.4).
Contribution: "how does your work contribute to knowledge? How does it contribute to making a better world?" [64]	This thesis provides contributions to theory (section 10.1) and implications for practice (section 10.2).
Accessibility: "are the findings intellectually and physically accessible to potential users?" [167]	Two conference papers have been published with early results from this research [224, 225]. This thesis will be publicly available at the Victoria University of Wellington library upon publishing. Other forms of publishing to make the theory more accessible to industry may be considered.
Working hypotheses: "do the findings stimulate 'working hypotheses' on the part of the reader as guidance for future action?" [167]	Yes: the findings are presented in the form of a set of strategies that are associated with different contexts, which readers can use to determine what is appropriate for their situation.
Level of knowledge: "what is the level of usable knowledge offered? It may range from consciousness-raising and the development of insight or self-understanding to broader considerations: a theory to guide action, or policy advice. Or it may be local and specific." [167]	The findings are in the form of a set of strategies that agile software engineers can use to determine how much architecture to design up-front (that is, guide their actions).

Table 9.6: Techniques and queries for assessing the *application* [167]/*usefulness* [64] criterion

Technique	Comment
Insight: “are your categories fresh? Do they offer new insights?” [64]	Yes: the research categories are both fresh and offer new insights. These are discussed in the remaining sections of this chapter.
New rendering: “does your analysis provide a new conceptual rendering of the data?” [64]	Yes: it presents the theory in the form of a set of strategies, which is novel (see Chapter 7).
Significance: “what is the social and theoretical significance of this work?” [64]	This theory of agile architecture is the only theory I am aware of to date that explains the relationship between architecture and agility, and helps fill a crucial knowledge gap in empirical software engineering research.
Contribution: “how does your grounded theory challenge, extend or refine current ideas, concepts and practices?” [64]	In addition to generating the only theory I am aware of on the relationship between architecture and agility, it also extends several other ideas such as the impact of size, complexity and minimising cost on up-front architecture design.

Table 9.7: Techniques and queries for assessing the *originality* criterion

Technique	**Comment**
Fullness: "do the categories portray the fullness of the studied experience?" [64]	Yes: the theory explains most of the concerns that were raised in the data.
Meanings: "have you revealed both liminal and unstable taken-for-granted meanings?" [64]	Yes: for example, the term 'agile architecture,' defined in this thesis, is an example of a term that is widely used but appears to have been given little meaningful thought.
Links: "have you drawn links between larger collectivities or institutions and individual lives, when the data so indicate?" [64]	Yes: the theory explains the impact of different types of companies, projects and the environment on agile architecture.
Participant acceptance: "does your grounded theory make sense to your participants or people who share their circumstances? Does your analysis offer them deeper insights about their lives and worlds?" [64]; also **Ring true**: "does the account 'ring true,' make sense, seem convincing or plausible, enable a 'vicarious presence' for the reader?" [167]	Yes: papers with early results from this research presented at conferences [224, 225] were well received by conference attendees and participants. See also comments on member checks in Table 9.2.

Table 9.8: Techniques and queries for assessing the *resonance* criterion

9.2 Agile architecture

The term *agile architecture* was defined at the start of Chapter 5 as an architecture that supports a team's agility by being easily modifiable and tolerant of change, and which has a more emergent design with a short planning period.

In contrast, the term agile architecture is most commonly used in the literature to refer to an evolving software architecture that is produced by the agile development process [114, 125, 156]. Kruchten (2013) preferred to refer to such as architecture as *agile architecting*, defining it as follows:

> "an agile way to define an architecture, using an iterative lifecycle, allowing the architectural design to tactically evolve gradually, as the problem and the constraints are better understood." [144]

Kruchten and a small number of other authors prefer to use the term agile architecture to describe an architecture that is designed to be modifiable [57, 131, 136], a definition that satisfies the definition of agility (section 2.2.3) and is similar to the definition used in this thesis. Kruchten defined agile architecture as:

> "A system or software architecture that is versatile, easy to evolve, to modify, flexible in a way, while still resilient to changes." [144]

Brown (2013) also defined agile architecture in this sense:

> "An architecture that can react to change within its environment, adapting to the ever changing requirements that people throw at it." [57]

Brown and Kruchten both emphasised that an agile architecture defined in this way is not necessarily the same as an architecture designed by agile architecting, with Brown highlighting the difficulty in producing an architecture that is modifiable:

> "Delivering software in an agile way doesn't guarantee that the resulting software architecture will be agile. In fact, in my experience, the opposite typically happens because teams are more focused on delivering functionality rather than looking after their architecture." [57]

The use of the term agile architecture to mean an architecture that is modifiable, as defined by Brown and Kruchten, does not specify the use of agile processes, and thus it is possible that it could be designed using a plan-driven methodology [144]. Indeed, maintainability (similar to modifiability but more about tweaks and less about evolution) is a common requirement for architecture [25, 50] whether agile or not. For example, maintainability is referred to in the Software Engineering Institute's description of architecture:

> "the architecture is the primary carrier of system qualities, such as performance, *modifiability*, and security, none of which can be achieved without a unifying architectural vision." [207] (emphasis added)

Good design practices, an important tactic for designing an architecture that can respond to change, are good design practices no matter what processes are used [25, 43, 179].

The term agile architecture is also frequently applied to enterprise architecture [16, 33, 170, 231] which is designed to cope with an organisation's dynamic structures and processes [16] and improve business agility [33]. Enterprise architecture is beyond the scope of this thesis.

Importantly, however, this thesis also recognises that the architecture of a software system developed using an agile methodology is not a static set of design decisions: an agile architecture evolves as the system is developed, and therefore also requires an agile architecting process. An agile architecture is therefore both an output of the architecting process (that is, a set of design decisions that can be described as being modifiable and tolerant of change) and is a part of the architecting process (that is, decisions are timed or made in such a way as to make the evolving design modifiable and tolerant of change). This means that an agile architecture is produced with less up-front design and more emergent decisions, and is able to respond to change by being modifiable and tolerant of change (Chapter 5).

The five tactics of S1 are used to improve the ability of the architecture to RESPOND TO CHANGE. The first three tactics increase modifiability: keeping the design simple, proving the design with code iteratively and using good design practices. The last two increase tolerance to change: delaying decision making and planning for options.

Three of the five tactics are characteristics of the architecture itself: keeping the design simple, using good design practices and planning for options are not exclusive to agile methodologies, and may be confirmed or discovered a posteriori by inspecting the architecture. The other two tactics, proving the design with code iteratively and delaying decisions, require an agile process (agile architecting). It may not be possible to determine their use by inspecting the architecture.

Keeping the design simple, proving the architecture with code and delaying decisions all reduce up-front effort, while following good design practices and planning for options may initially slightly increase the architecture effort, but have the benefit of lower architectural refactoring costs.

How the literature relates to each of S1's tactics is described below in sections 9.2.1 to 9.2.5. Section 9.2.6 then describes other ways for determining up-front design effort.

9.2.1 Keeping designs simple

Simplicity is an important part of agility; 'simplicity is essential' is one of the twelve principles listed in the agile manifesto (section 2.2.1). Simplicity means only designing for what is immediately required: no gold plating and no designing for what *might* be needed or for what can be deferred [27]. In XP, simplicity is referred to as *YAGNI* ('you ain't gonna need it') [96]. Simplicity reduces the effort required to make changes to the architecture, and therefore makes the architecture more easily adapted to change.

Developers are often encouraged to refactor existing designs into simpler designs when possible (for example, Beck [27]).

9.2.2 Proving the design with code iteratively

Proving that a design or model works by coding it is a core practice of Scott Ambler's *agile modelling* [18] practice-based methodology for modelling and documenting systems [13].

Proving that a design works with code acknowledges that models and theoretical abstractions may not work in practice, and ultimately can only be proven to work by coding the solution and testing it. Modelling is only one task of the iterative development process [18]; building and testing are an important part of the cycle.

This tactic is possible because architecture design and development can take place simultaneously in agile development (see also delaying decisions, described below).

9.2.3 Good design practices

An agile architecture is more easily modified than a non-agile architecture. Brown noted that an agile architecture should be designed using small, loosely coupled components or services that can be easily modified and tested in isolation, and even entirely replaced if necessary [57]. A poorly designed architecture is very difficult to refactor [91].

Kavis (2012) specified that for an architecture to be agile, it should have some or all of these qualities: open, configurable, modular, independent, elastic and abstract [131], all which increase its modifiability or tolerance of change.

Coplien and Bjørnvig (2010) suggested separating components according to their rates of change [79] and 'shearing layers' [51, 94].

Designing an architecture that it is modifiable may require more architectural and development effort than one that is not because (for example) creating extra APIs and levels of indirection requires more effort and discipline [57, 79].

9.2.4 Delaying decision making

Up-front design can be minimised by delaying architecture decisions where possible until after development begins (that is, by making more emergent decisions).

Many authors recommend delaying architecture decisions where possible, until just before the decision is required – the "last responsible moment" [1, 147, 184] or the "most appropriate time" [79]. Only the decisions that cannot be deferred are made up-front. Malan and Bredeyer (2002) described the minimum up-front design as the minimum required to achieve the most strategic architectural goals [158], while Coplien and Bjørnvig (2010) describe the minimum up-front design as the minimum that contributes to solving the long-term problem [79]. Brown (2013) used the term *just enough architecture* [57] – just enough to allow the team to understand the structure, create a shared vision and identify and mitigate the highest priority risks. Fairbanks (2010) focused on risk, with the minimum being just enough to reduce risk to satisfactory level [91]: "your effort should be commensurate with your risk of failure."

Minimising up-front design and delaying decisions is important to ensure the architecture is designed with only the most accurate and up-to-date information, and hence reducing waste. This is similar to Boehm's *cone of uncertainty* [42], in which uncertainty is reduced over time as decisions are made and the system evolves.

9.2.5 Plan for options

A feature of Madison's 'agile architecting' is it allows for options [156]: the team defines ranges and bounds – a bounded range of solutions that is narrowed down as more information becomes available, rather than specifying a solution too early. Distinct from delaying decisions, allowing for options means architectural direction can be defined earlier.

9.2.6 Up-front architecture design effort

Fairbanks (2010) proposed a risk-driven approach similar to ADDRESS RISK (S2), in which the architecture is designed in sufficient detail up-front to mitigate risk to a satisfactory level [91] – suggesting a balance between RESPOND TO CHANGE (S1) and ADDRESS RISK (S2).

He also listed four alternative ways to determine how much architecture to design up-front: *no design*, which includes some implicit design decisions, similar to an EMERGENT ARCHITECTURE (S3); *documentation package*, similar to the BIG DESIGN UP-FRONT (S4), in which the architecture is recorded in a document so that someone else can understand the architecture and recreate it; *yardsticks*, in which some fixed amount of time or proportion of total effort is spent architecting, perhaps based on past experience or determined by models such as Boehm's (section 9.3.3); and *ad hoc*, in which the architects make the decisions that they feel are the most appropriate at the time, and which is perhaps an informal risk-driven approach [91].

9.3 The impact of context on up-front effort

This section presents different perspectives from the literature on how context affects up-front architecture design and compares them with the findings of this thesis. It includes a general review of the forces (or factors) that affect up-front architecture design (section 9.3.1), then more specific comments on the relationship between size and complexity (section 9.3.2) and the architectural effort 'sweet spot' (section 9.3.3). It then discusses the impact of team culture (section 9.3.4), customer agility (section 9.3.5) and architecture and technical experience (section 9.3.6).

9.3.1 Context and the forces that comprise context

Abrahamsson, Ali Babar and Kruchten (2010) listed eight factors (or forces) that they suggested can affect the level of up-front architecture design [1]. These factors are size, stable architecture, business model, team distribution, rate of change, age of system, criticality and governance. These factors are shown in Figure 9.1, which repeats Figure 2.7.

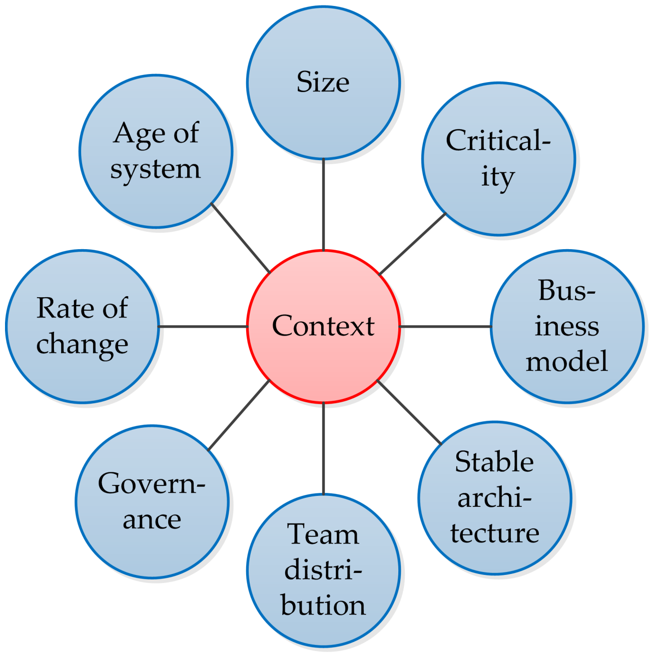

Figure 9.1: The factors that make up a project's context, after Abrahamsson, Ali Babar and Kruchten (2010) [1] and Kruchten (2013) [146] (Figure 2.7 repeated)

Size: Kruchten (2013) described the overall *size* of the system under development as the greatest factor affecting agility [146]. Size drives the size of the team, the number of teams, communication and coordination needs and the impact of change. This thesis has determined that size was not a direct driver of up-front architecture design; rather, complexity is – see TECHNICAL RISK (F2). The relationship between size and complexity and their impact upon up-front architecture design is discussed in more detail below in section 9.3.2. The impact of the number (and size) of teams is considered part of TEAM CULTURE (F4).

Stable architecture: The second factor is *stable architecture*. Many software systems do not require novel architectural solutions; a stable architecture can be defined at the start of development using existing tools [146]. The impact of this factor is included within the TECHNICAL RISK force (F2) and the USE FRAMEWORKS AND TEMPLATE ARCHITECTURES strategy (S5). Kruchten (2013) noted that 75 per cent of the time, the high level architecture choices – technology and platform – can be made implicitly [145]. Section 7.3.1 noted that these highest-level decisions are often made implicitly, particularly by teams using EMERGENT ARCHITECTURE (S3).

Business model: The *business model*, representing the money flow of the customer, can impact upon the up-front design effort. For example, is the system an internal system, a commercial product, or a component of a larger system [146]? Larger organisations often prefer to follow process, and generally prefer the development teams to spend more time on up-front design [122]. Similarly, many also have to get budgets approved in advance. ISVs (independent software vendors) often have to submit competitive tenders, which require more up-front design. Multi-team systems require more architectural oversight and coordination. The effects of these scenarios are all included within CUSTOMER AGILITY (F5). Some organisations – for example, government departments – are inherently risk averse, which compels more up-front design to ADDRESS RISK (S2). Finally, some organisations – particularly start-ups in a dynamic domain or industry – need cash flow as early as possible, and cannot afford the delay caused by any up-front design beyond the bare minimum (EARLY VALUE, F3).

Team distribution: The *team distribution* is closely linked to size of the project, as noted by Kruchten [146]. Team distribution affects the team's ability to communicate, and is thus included within TEAM CULTURE (F4).

Rate of change: *Rate of change* refers to the stability of the environment [146]. This research found that rate of change did not affect the amount of up-front design: teams are often prepared for any requirement to change. Indeed, changes are not known a priori. Rather, the team's ability to RESPOND TO CHANGE (S1) affects up-front design. S1 depends on a number of forces: REQUIREMENTS INSTABILITY (F1), TEAM CULTURE (F4), CUSTOMER AGILITY (F5), and architecture and technical EXPERIENCE (F6). F1 triggers the use of S1, while F4, F5 and F6 are success factors: teams that are more able to respond to change are able to do less up-front design. Agile teams can even use S1 when requirements are stable, although they may find the overhead of some agile practices which help reduce the feedback cycle make those practices unnecessary and even costly, and omitting those practices may therefore improve the team's efficiency.

Age of system: The *age of system* refers to teams working on legacy systems [146]. TECHNICAL RISK (F2) includes legacy systems as a factor that affects up-front design because of its effect on complexity.

Criticality: *Criticality* refers to the impact of failure of the system [146]. Criticality leads to higher technical risk exposure, and therefore is included as part of ADDRESS RISK (S2). A highly critical system has a low tolerance of risk, and is likely to have demanding ASRs, a complex architecture, and an associated high level of TECHNICAL RISK (F2). Complexity and low tolerance of risk lead to high risk exposure which requires more up-front design to mitigate the risk to a satisfactory level.

Governance: *Governance* refers to the management of the project and project team [146, 217]. Governance and its impact upon agility is reflected in TEAM CULTURE (F4) and CUSTOMER AGILITY (F5).

*

In addition to Abrahamsson et al.'s and Kruchten's eight factors that affect up-front architecture design, Boehm and Turner (2003) identified five factors that could be used to determine whether or not a team should use an agile development method or a plan-driven method [40]: *personnel*, *culture*, *dynamism* (of requirements), *team size* and *criticality*. While the authors did not discuss the impact of these factors on architecture design, they all map to forces presented in this thesis and hence can affect architecture design: personnel, culture and team size affect TEAM CULTURE (F4); dynamism affects REQUIREMENTS INSTABILITY (F1), and criticality affects risk exposure in the forms of F2 and S2 (as above).

Cockburn (2007) used *team size* and *criticality* to determine which of the Crystal family of methodologies teams should use [69], which vary according to how much process they involve. Smaller teams building less critical systems are suited to the more lightweight Crystal Clear, and larger teams or more critical systems require more process in the form of Crystal Yellow, Orange, Red and so on.

Kruchten (2004) used the term agile 'sweet spot' [139], which described the ideal context for an agile project. The ideal agile context would be a small group with a co-located team, would have high customer availability, would be a business application, a new development, have a RAD (rapid application development) programming environment and a short life cycle. In other words, a project in the sweet spot is more likely to have a team with an agile culture, have an agile customer and be building a system with low risk. Conversely, the agile 'bitter spot,' which makes agile development difficult, has large groups and distributed teams (which negatively impact upon the agile culture), no empowered customer representation (which negatively impacts the customer agility), long delivery cycle and inefficient programming environment (which impact upon the team's ability to RESPOND TO CHANGE, S1), and different development cultures (which also negatively impacts the agile culture). Projects at the 'bitter spot' can still be agile; however, their ability to reduce up-front architecture design and to respond to change will be reduced.

Boehm (2002) suggested that a rate of change of requirements of 1 per cent per month is the point at which it is better to use agile methods than to do a big up-front design [36]. This thesis's theory of agile architecture describes F1 as a trigger for S1 – that is, requirements instability is a trigger for designing an architecture that can respond to change. Boehm's value of 1 per cent per month value perhaps quantifies the threshold for triggering S1.

9.3.2 Size and complexity

The contexts identified by Kruchten [1, 146], Boehm and Turner [40], and Cockburn [69] all include size as a factor that affects up-front architecture design. Boehm (2011) also used size to find the 'sweet spot' of architecture effort (see section 9.3.3) [37]. Size as a factor is at odds with this thesis, which found that complexity, and not size, is a force (section 6.2.2).

Complexity refers to the interconnectivity and relationships between elements in a system [22]. In a complex system, a decision or action in one element will affect (to some degree) many or all other elements.

There are three facets, or attributes, that contribute to complexity: scale (the number of 'things' in the system), diversity (how many different 'things' there are in the system), and connectivity (how many relationships there are between the things) [143, 164]. These three facets each contribute to complexity; in particular, scale in itself is not a problem if the structure is regular, although it does emphasise problems caused by diversity and connectivity [164]. Thus a small system is usually not very complex, and a large system has the potential for a high level of complexity. Diversity increases the number of elements that have to be designed or analysed, and connectivity increases the relationships between the elements [164]. Kruchten described connectivity as the bottom line driver of complexity [143].

Abrahamsson et al. (2010) noted that the importance of architecture increases as the complexity of the system increases [1], despite being omitted from the list of factors from the same paper, shown in Figure 9.1. This importance is backed up by a survey of developers at the IBM Software Lab in Rome conducted by Falessi et al. [92]. When asked when developers should focus on architecture and given the options 'always', 'never', and 'when the system is complex,' 50 per cent of survey respondents selected complexity. A further question asked what the leading cause of complexity is, with the options of 'number of requirements or lines of code' (selected by 33% of respondents who selected complexity in the previous question), 'number of stakeholders' (29%), 'geographic distribution' (19%) and 'other' (19%). The high proportion of 'other' causes (19%) suggests that the causes of complexity in the list in the survey question are representative rather than exhaustive.

The symptoms of a complex architecture have been described as having decisions that are intertwined, cross-cutting and having multiple dependences [129] – that is, have demanding ASRs (TECHNICAL RISK, F2). Complexity is also increased when design rules and constraints are violated, and when obsolete design decisions are not removed – a phenomenon captured by Lehman's second law of program evolution: "as an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it" [151].

These problems all increase the cost of maintenance and therefore make it more difficult to RESPOND TO CHANGE (S1).

As discussed in the previous section, size is widely considered in the literature as affecting the up-front architecture design required. Size is, however, one facet of complexity, and hence authors are perhaps using size as a proxy for complexity.

The strategy ADDRESS RISK (S2) increases the up-front design required to mitigate risk caused by complex architectures. S2 notes that often the only way to fully understand an architectural solution is to build the system and see if it works. Similarly, many teams take the approach of proving an architecture design by building it (section 7.1.2). Gall (1978) wrote that complex systems are more likely to work if they evolve from simple systems that have been proven to work:

> "A complex system that works is invariably found to have evolved from a simple system that worked. The inverse proposition also appears to be true: a complex system designed from scratch never works and cannot be made to work." [102]

This is sometimes dubbed Gall's Law. Gall's Law emphasises the importance of the techniques such as prototypes and spikes (used in S2) and proving the design by coding (used in S1), and confirms that a big design up-front is often simply not sensible or even possible.

9.3.3 Early value and the architectural effort 'sweet spot'

S3 is an EMERGENT ARCHITECTURE, used by some organisations who need EARLY VALUE (F3) from their software. Summarising F3, early value is achieved by releasing a version of the product or service as early as possible, so that the organisation – typically a start-up in a dynamic industry – benefits from early adopters. An example of an early release is the minimum viable product (MVP) [192], a marketing experiment to determine the end users' biggest needs.

Boehm (2011) [37] undertook a study using the COCOMO II cost model [38] in which he demonstrated a relationship between the level of architectural effort and the overall development effort – and hence cost. Boehm's study showed that architecture effort is a compromise between the amount of time spent planning and the amount of time spent on rework (refactoring) caused by doing too little planning, with a 'sweet spot' at the overall minimum cost (Figure 9.2). This sweet spot has been referenced by others writing about architecture in agile development, such as Coplien and Bjørnvig (2010) [79] and Fairbanks (2010) [91]. Poort and van Vliet (2011) also used the sweet spot in their strategy for minimising architectural effort [182][1].

The location of this sweet spot is highly dependent on the context of the system; Boehm's study illustrated the impact of the size of the system, with a larger system requiring more time spent resolving architectural issues than a smaller system for any given level of up-front design effort. This difference is due to the diseconomies of scale of software development [37]. Building a two-module system costs more than twice that of a one-module system because of the additional effort required to implement communication between the two modules. For example, Figure 9.3 shows that an increase in size from 100 KSLOC (thousand equivalent source lines of code) to 10,000 KSLOC increases the up-front design sweet spot from around 20 per cent of the total effort to around 40 per cent of the total effort.

Figure 9.4 illustrates the effect of rapid change (volatility) on the sweet spot. Volatility of 50 per cent moves the sweet spot left (less time spent architecting up-front) because of the extra effort required to revisit decisions when requirements change, but also increases the overall architecture effort.

In a critical (high risk) project, the cost of rework is higher due to the higher cost of failure. Figure 9.5 shows the effect of a project with 50 per cent higher cost of rework on the sweet spot: the project has the same amount of architecture design effort, so the sweet spot moves to the right (requiring

[1]This sweet spot is not related to the sweet spot used by Kruchten to determine a project's suitability for agile, referred to in section 9.3.1.

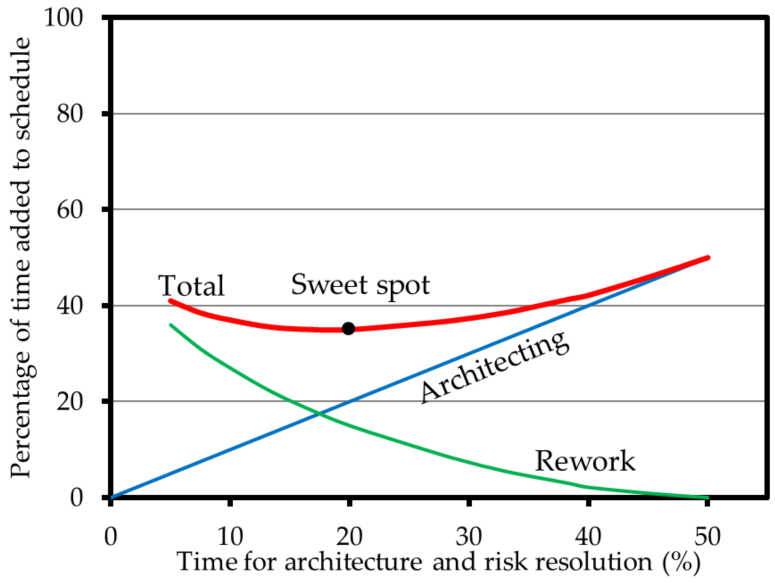

Figure 9.2: Locating the up-front architecture design sweet spot, after Boehm (2011) [37]

more effort), with an increase in the total because of the increased amount of rework. Too little architecture effort can also reduce the organisation's operational effectiveness and productivity [37].

It should be noted that the COCOMO II model and hence Boehm's study are not based on agile projects. They assume that all architectural effort is up-front, as per Boehm's definition of architecture: " 'Architecting' refers to [...] the overall set of concurrent front-end activities [...] that are key to creating and sustaining a successful building or software project" [37]. Boehm did comment in this study [37] and in earlier, similar studies [36, 39] that projects which require less up-front planning are more suited to agile, on the basis that agile software engineers prefer less up-front effort. Boehm did not consider in any detail, beyond comments made in passing, the tactics used to RESPOND TO CHANGE (S1) that are characteristic of agile projects or agile architectures, such as delaying decisions, planning for options and proving the design with code.

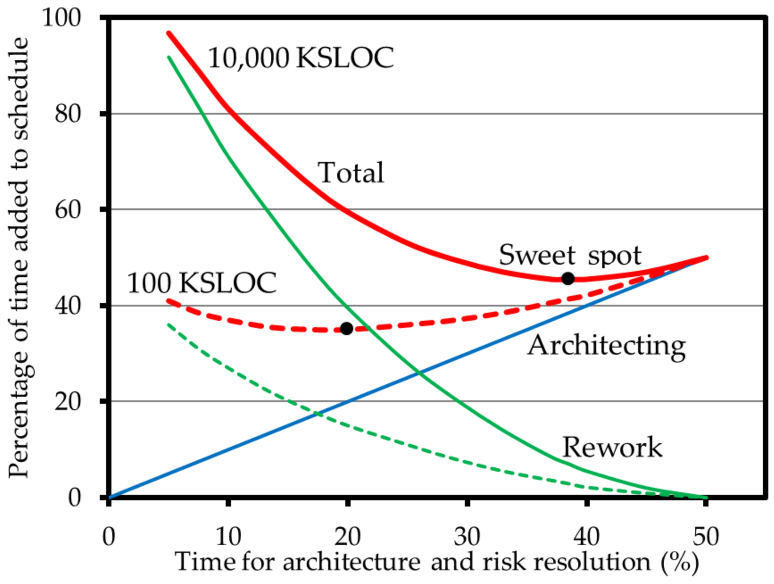

Figure 9.3: The effect of system size on the up-front architecture design sweet spot, after Boehm (2011) [37]

The theory of agile architecture presented in this thesis diverges from Boehm's study in four regards.

Firstly, this thesis's theory does not support Boehm's study's relationship between size and effort. TECHNICAL RISK (F2) notes that risk derived from complexity has an important impact on determining architecture effort. In particular, section 6.2.2 and section 9.3.2 discuss the relationship between size and complexity, noting that size is just one component of complexity. Section 9.3.2 suggests that many authors use system size as a proxy for complexity.

Secondly, in some situations teams do not aim to minimise the cost of architecting: maximising the delivery of value is often more important to the customer than purely minimising cost. If the customer requires EARLY VALUE (F3), and if the team designs an EMERGENT ARCHITECTURE (S3), the team will release software early at the expense of architecture rework later – that is, at higher cost. The teams fully understand that they will

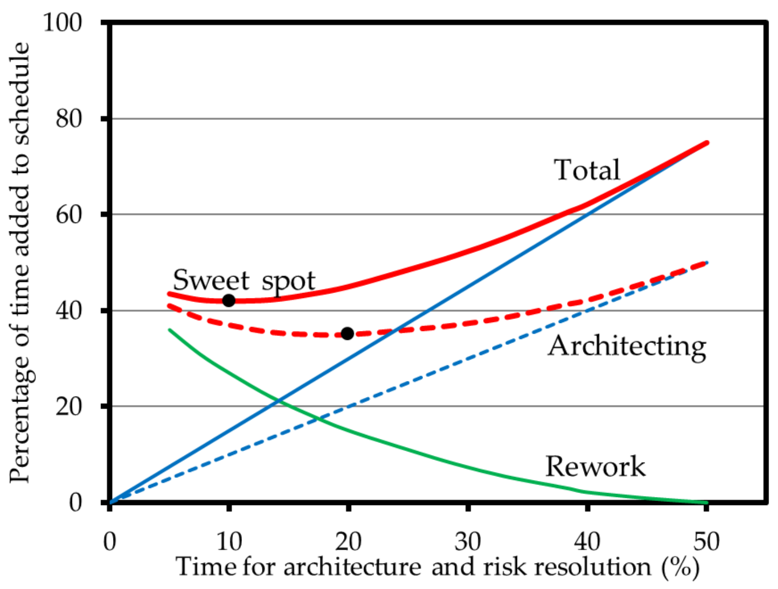

Figure 9.4: The effect of 50 per cent requirements volatility on the up-front architecture design sweet spot, after Boehm (2011) [37]

have to redo the design later when they come to design or build the longer term features. Therefore the team is aiming to the left of the sweet spot. F3 causing higher cost is discussed in section 6.3. For example, a team may deliberately choose flat file persistence instead of a database, or omit performance or security considerations, even though they know doing so means that in three or six months they will need those features and they will have to rework the architecture. However, balancing the higher cost is the benefit that comes from minimising over-engineering – teams can prove the design using code iteratively (section 7.1.2) and ensure that the system exactly meets its requirements without any over-engineering, so that the design can be simpler and the cost can be reduced. On the other hand, if the customer is not sympathetic to agile (CUSTOMER AGILITY, F5), the team may need to design BIG DESIGN UP-FRONT (S4), which is to the right of the sweet spot. While rework costs due to lack of architectural guidance may be low with S4, the requirements gathering and architectural

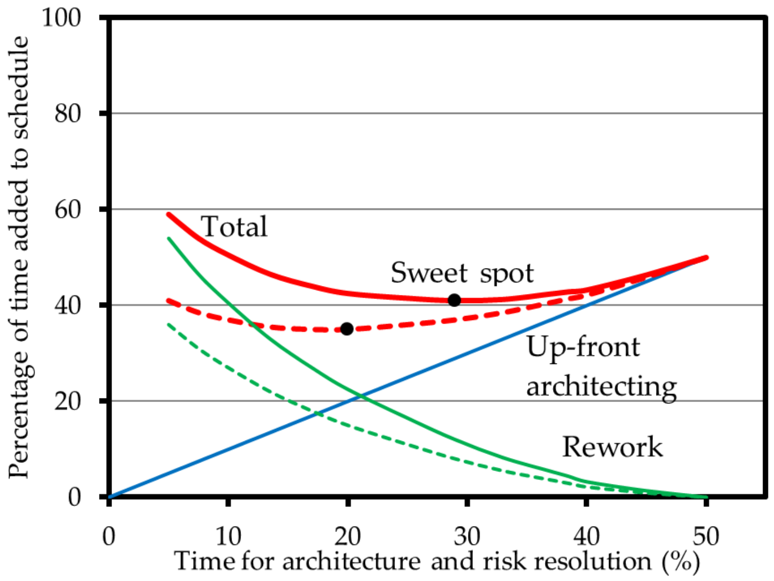

Figure 9.5: The effect of 50 per cent higher failure cost on the up-front architecture design sweet spot, after Boehm (2011) [37]

decision-making process is lengthy and costly.

Thirdly, while volatile requirements do increase architecture effort as decisions need to be changed, it is the team's ability to design an architecture that can RESPOND TO CHANGE (S1) that affects how much architecture a team designs up-front. Section 9.3.1 also discusses the impact of requirements volatility ('stable architecture'). S5, the USE FRAMEWORKS AND TEMPLATE ARCHITECTURES strategy, greatly reduces architecture effort, flattening the sweet spot curve.

Finally, rather than simply increasing the rework cost, criticality also increases architecture effort, as teams reduce risk to an acceptable level (S2, ADDRESS RISK). This effort may be up-front or after development has started (emergent), depending on the impact of the risk on the system, in line with S1. Increased design effort caused by increased risk may also impact the cost of rework, to mitigate the risk involved with that rework. While the overall architecture effort is higher, the impact upon the sweet spot is dependent on the relative impacts on architecture design and rework.

9.3.4 The importance of team culture on agility and up-front design

This section expands upon the effect that team culture, one of the constituent forces that make up the team's context, has on up-front design effort.

TEAM CULTURE (F4) – and in particular trust – is important for reducing the feedback cycle and hence increasing agility. The importance that team culture has on agility is well recognised in the literature [40, 72, 229], as is trust for teams in general [152] and agile software teams in particular [72, 85, 116].

With a short feedback cycle enabled by a trusting people-focused and collaborative culture – both with the team and the customer organisation – a team can rely less on documentation for communication and formal plans to guide development, whereas a team that does not have this culture has to rely more on documentation and plans. Coplien and Bjørnvig (2010) summarised the benefit of good communication on a change involving the architecture that took six months to make: "It isn't unreasonable to compress these six months down to one or two weeks if communication between team members is great" [79].

An agile team also requires a culture that is comfortable with or even thrives on uncertainty, in which the team gains comfort and empowerment from degrees of freedom, as opposed to plan-driven development which requires a team to gain comfort and empowerment from having its roles defined by policies and procedures [40]. A team that has a highly agile culture or attitude will feel more comfortable operating with the uncertainty that agile brings and the absence of up-front plans.

This dependency also works the other way: the 'ceremony' of architecture – the diagrams, the documentation and even the vocabulary – may suffocate collaboration [132], and actually hinder the end goal rather than help it.

Malan and Bredemeyer (2002) wrote that an organisation is less likely to be supportive of an over-architected solution: an over-architected solution will be harder for a team to understand, and each additional decision in the architecture's decision set dilutes the impact of the other decisions [158].

Hoda (2010) noted that it takes time for team members to gain the experience required to become truly self-organising and hence agile. In particular, it takes time for a team to properly learn self-evaluation and self-improvement, two qualities required to reach 'self-transcendence' and the 'peak of self-organisation' [119]. TEAM CULTURE (F4) also noted the importance of experience in agile development in acquiring an agile mindset.

9.3.5 The importance of customer agility

CUSTOMER AGILITY (F5) described the effect of the environment, and in particular the team's customer, on the team's agility.

While the agile organisation is about trust, leadership and collaboration [72, 87], a traditional process-driven organisation frequently has a *command and control* management style [22, 173], is non-learning [203] and prefers extensive planning and formal communication [72]. CUSTOMER AGILITY describes process-driven customers as often being large organisations; Hoda, Noble and Marshall (2010) had a similar finding [122].

Process-driven customers usually require fixed price contracts, in line with their need for planning. Fowler (2004) described fixed price contracts as a 'mirage' because they force the customer to focus on cost rather than value (see the discussion on value in section 2.2.3), and they force the developer to agree in advance to requirements they cannot understand – and hence cannot put a legitimate price on [98]. Thus, while in theory the contract allows the development team to be held accountable when things go wrong, in reality both parties may suffer [184].

9.3.6 The importance of experience on agility and up-front design

This section provides more discussion on the effect of architectural and technical experience, another constituent force of context that has an important impact on software development in general, and on agile development in particular.

Cockburn (2007) described competent and experienced people as a critical success factor in any methodology, whether agile or plan-driven [69]. Boehm (1984) believed it so important that he showed the effect of personnel and team capability in a graph on the cover of his book (Figure 9.6) [42].

Other authors have also written about the importance of experience on architecture design in general [25, 50, 174] (and inexperience [94]).

Boehm and Turner (2003) studied experience in the context of agile development. They categorised development ability using a scale of five levels of software method understanding [40], of which experience is a component. Ability ranges from the expert level – "able to revise a method, breaking its rules to fit an unprecedented new situation" down to the less able and agile-unfriendly "may have technical skills, but unable or unwilling to collaborate or follow shared methods." They believed that agile development requires a critical mass of experts throughout the project and it is risky to use the low-rated personnel, while plan-driven methods only require a critical mass of experts during project definition; fewer are required once the project has been established [40].

The benefit of experience is explained by Dreyfus and Dreyfus's five-stage learning model, which included the following levels: *novice*, *competence*, *proficiency*, *expertise* and *mastery* [86]. In this model, as a learner of an activity passes from proficiency to expertise, they must have gained sufficient experience that their decision-making ability changes from requiring *analytic thought* to being an *intuitive response* (Table 9.9). The expert

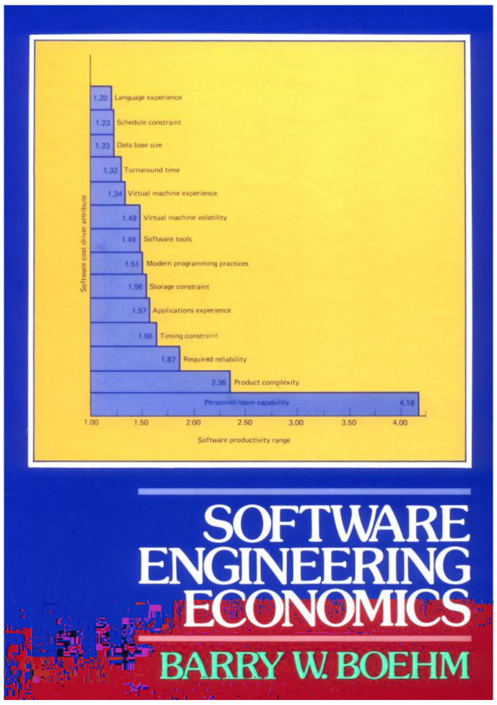

Figure 9.6: The cover of *Software Engineering Economics* [42], published in 1984, showing the importance of personnel/team capability (bottom bar) on software productivity, the x-axis. Product complexity is the second to bottom bar.

does not consciously apply rules, guidelines or maxims; instead, they have an intuitively appropriate response. In agile, this means decisions can be made more quickly with less need for analysis or explicit models.

Concerning the experience and the role of software architect, Taylor (2007) wrote: “the most experienced designers assemble the holistic context of the design engagement to allow design activity to be performed tacitly,” and added that this is why an expert designer can make complex software design look easy [218].

Mental function	**Skill level**				
	Novice	**Competent**	**Proficient**	**Expert**	**Master**
Recollection	Non-situational	Situational	Situational	Situational	Situational
Recognition	Decomposed	Decomposed	Holistic	Holistic	Holistic
Decision	Analytical	Analytical	Analytical	Intuitive	Intuitive
Awareness	Monitoring	Monitoring	Monitoring	Monitoring	Absorbed

Table 9.9: The Dreyfus and Dreyfus five-stage learning model [86]

9.4 The Twin Peaks of requirements and architecture

Defining requirements fully in advance of development is very difficult [53] – particularly in agile development methods, where change in welcomed and even encouraged [28]. Agile development methods therefore use current requirements in which detail is only defined close to the time of implementation [83], rather than current and future requirements [52, 73]. Requirements, like architecture, are therefore emergent [36, 130, 199].

Nuseibeh (2001) presented a model called *Twin Peaks* that represents the relationship between requirements and architecture [176]. The Twin Peaks model recognises that the relationship between requirements and architecture design is not simply a one way relationship with requirements driving the architecture (as in traditional plan-driven development [66]), but is a two way relationship developed iteratively, with not only the requirements driving the architecture, but also with the architecture affecting requirements, as shown in Figure 9.7. For example, architecture designs frequently impose constraints upon requirements according to what it is possible technically, and impose time and budget restrictions [103].

Twin Peaks progressively develops more detailed requirements and

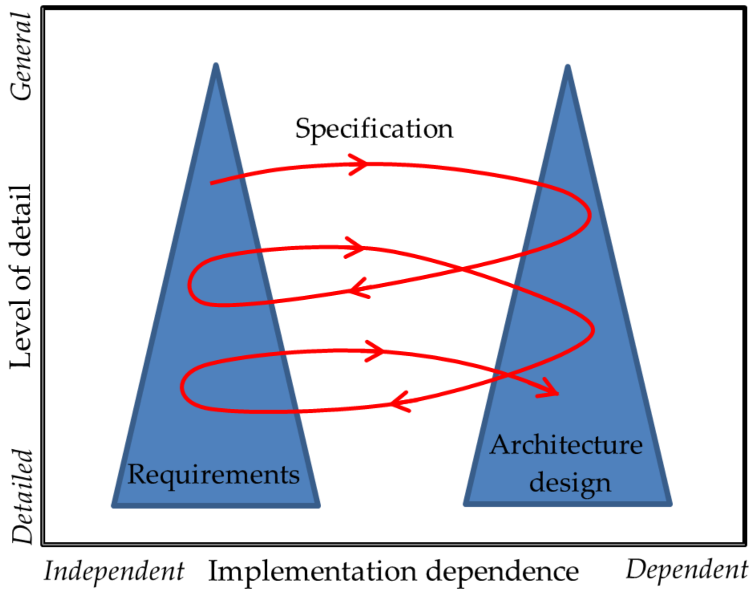

Figure 9.7: The Twin Peaks model of the relationship between requirements and architecture, after Nuseibeh (2001) [176]

architecture as the process proceeds [176], and therefore addresses three concerns earlier raised by Boehm (2000) [35]. The first concern is *IKIWISI* ("I'll know it when I see it"): the customer or users can only describe what they want after they have seen architectural models or prototypes. The second concern comes from the use of *COTS* (commercial off-the-shelf) software: the capabilities of COTS (perhaps in the form of frameworks – USE FRAMEWORKS AND TEMPLATE ARCHITECTURES, S5) constrain the system and hence its requirements. While it is always possible to build bespoke libraries or routines to meet requirements that a framework cannot, doing so increases the complexity of the solution (contributing to the team's need to ADDRESS RISK, F2), and therefore increases the architectural and development effort. This increased effort may increase the cost too much and lead the customer to abandoning that requirement: "it's not a requirement if you can't afford it" [35]. The third concern is *rapid change*: as is the case of a highly plan-driven development method, it is very difficult to respond to changing requirements if the requirements are fixed before any architecture design (and development) is started.

An EMERGENT ARCHITECTURE (S3) describes how agile developers define the highest level requirements (such as the business problem that needs to be solved) up-front, and then design the highest level architecture (such as the technology stack and high level architectural styles that affect the whole system and any system-wide risk concerns) up-front. This is consistent with Twin Peaks, because requirements are only determined or elicited as required. In addition, once the up-front phase is complete and development starts, teams continue to elicit requirements and design the architecture, which gives the developers, customers and users a better understanding of the system and its capabilities as it evolves. Demonstrations of the system itself, rather than just models of the architecture, give the customers opportunities to provide better feedback and give the developers proof that the technologies and architecture do indeed perform as expected.

There are variations on the Twin Peaks model. One variation is an extension for product lines [103], shown in Figure 9.8. This extension includes a third peak for the shared product line architecture, similar to P33–P36's reference architecture (see page 208). In this variation, the core requirements that are common to all products must be satisfied by the product line architecture and all products in the product line, and the variable requirements represent variation points in requirements which may or may not be implemented in a concrete product of the product line [103]. The model iterates over the core requirements and product line architecture, over the variable requirements and product line architecture, and both the core and variable requirements and the product architecture.

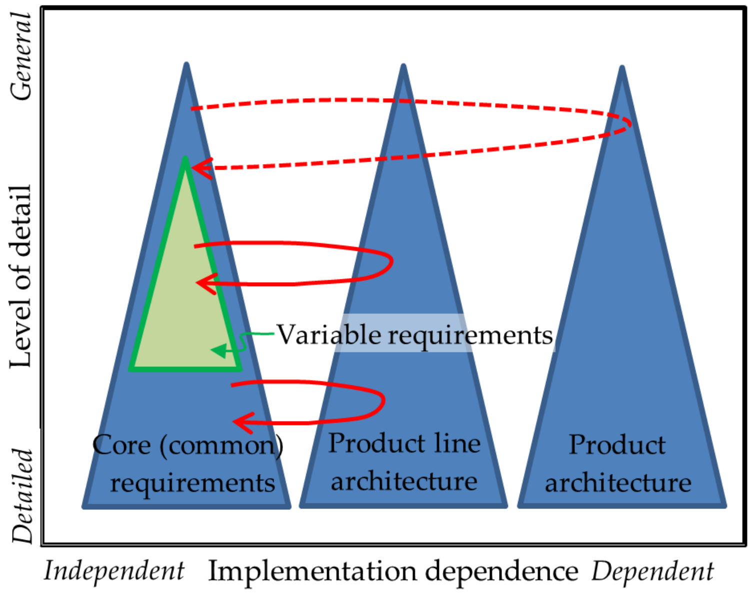

Figure 9.8: The Twin Peaks model in the context of software product line engineering, after Galster et al. (2013) [103].

9.5 The role of the architect

The agile literature universally recommends that anyone involved with architecture design is a *hands-on* architect who is closely involved with the development team, rather than an *ivory tower* architect who hands a completed architecture blueprint over to the development team to implement [1, 57, 79, 123, 142, 191].

In agile development it is less common to have a designated architecture role than in plan-driven development methods. Scrum teams are cross functional with all team members having responsibility for architecture [83], while Ambler's *agile modeling* [18] recommended that teams building larger systems have an *architecture owner* role, allocating the responsibility for coordinating the team's architectural design efforts to a technically experienced person on the team [14]. Ambler cited two problems of everyone being responsible for the architecture (that is, having decisions made by consensus): people do not always agree, and the method does not scale. These problems are discussed in section 8.1. Fairbanks (2010) recommended that all software developers, not just the architects, understand the system's architecture [91]. This is consistent with the definition of architecture in section 2.3, which states the architecture is the shared understanding that the expert developers have of the system.

Coplien and Bjørnvig (2010) avoided both using the role of 'architect' and the term 'architect' itself [79], because of the connotations that the architecture is solely their responsibility and that they are too far removed from coding. The contributions to the architecture by other roles are also important. Like Ambler, they instead urged an architecture *coordination* function. Similarly, Fowler (2004) wrote that design needs just one or two people to make an evolutionary architecture converge [96] – to "keep the design whole."

This thesis's theory of agile architecture includes the importance of having someone with overall responsibility for architecture in all but the smallest teams (section 8.1).

Buschmann (2012) described all architecture stakeholders as being contributors to the architecture, rather than simply receivers of the finished design [60].

Brown (2010) discussed the non-design roles of the agile architect. He divided the roles into two types: the *definition* of the software architecture and the *delivery* of the software architecture [56], which correspond to 'the architecture decision makers' (section 8.1) and 'other roles of the architect' (section 8.2), respectively. He listed the following architecture delivery roles, all of which have equivalents in this thesis: *ownership of the big picture* (section 8.2.2); *technical leadership*, which includes both creating a shared mindset (section 8.2.3) and creating and enforcing development guidelines (section 8.2.4); *coaching and mentoring*, which includes helping team members with their coding problems (section 8.2.1); *quality assurance*, which also includes creating and enforcing development guidelines; and *design, development and testing*, which nearly all architects in this research were involved with; even the architects that did not develop or test (such as P11b) were still closely engaged with the development team.

Kruchten (2009) also listed a number of roles that architects perform [142]. These include being the *visionary*, who considers the big picture of the system and guides its long term evolution (section 8.2.2); the *designer* which is the architecture decision making role; the *communicator*, which includes creating a shared mindset amongst the architecture stakeholders (section 8.2.3); the *troubleshooter*, which includes helping solve other developers' technical problems (section 8.2.1); the *herald*, which requires an understanding the big picture and how it relates to the business problem (section 8.2.2), and the *janitor*, who cleans up after the project manager and team.

Fowler (2003) described two types of architect: the *architectus oryzus* 'species' of architect [97], who collaborates closely with the team and is either a coding member of the team or otherwise has a very close relationship with the developers, and the *architectus reloadus*, who is more removed from the team: a 'maker and keeper' of the big decisions, because "a single mind is needed to ensure a system's conceptual integrity" [97]. Kruchten described both types of architect as being needed on large projects, at different stages of development [147].

Kruchten (2013) described three types of relationships, or 'boundaries,' across which the architect must communicate [145]: that of the business analyst (in which the architect deals with issues such as requirements and priorities), the project manager (in which the architect deals with management and with issues such as release plans, risks and costs) and the developers (in which the architect deals with the team and technology vendors, and with issues such as constraints and prototype evaluation). The architect must also translate terminology, abstraction level, volume and emphasis between the business analysts, project managers and developers.

These relationships are summarised in Figure 9.9.

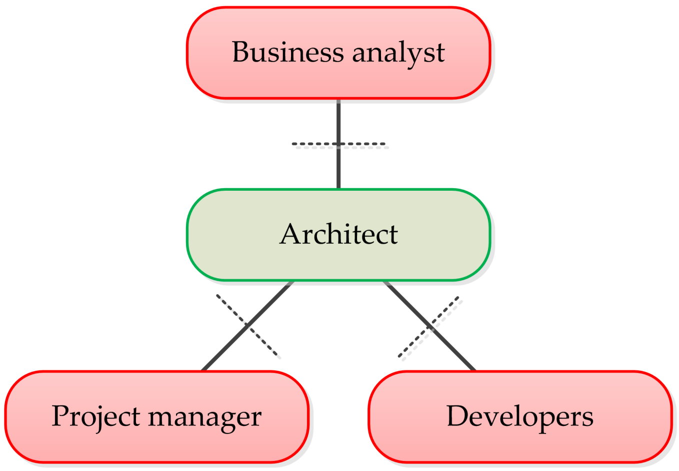

Figure 9.9: The three main communication boundaries for the architect, after Kruchten (2013) [145]

Similarly, Hoda (2010) identified a *translator* role in self-organising agile teams [121] which translates the language and jargon of the customer into something the team can understand, although this role was not specifically assigned to the architect.

In contrast with the hands-on agile architect, the so-called ivory tower architect, generally considered undesirable by agile teams, dictates architecture decisions without any knowledge or experience of the code being written (or the team writing it) [55, 97, 123]. The role of the ivory tower architect is consistent with the plan-driven methodologies that have entirely separate design and development activities [191]. Kruchten (2008) described the ivory tower architect as an anti-pattern [141].

9.6 Other insights

This section presents some miscellaneous insights.

9.6.1 Technical risk

Ambler (2013) identified five categories of risk in software development: business risk, technical risk, operational risk, process risk and organisational risk [17]. Fairbanks (2010) considered a narrower scope, with two categories: project management risk and engineering risk [91]. Technical (or engineering) risk (F2) is the risk of failure caused by problems within the technology suite, the architecture design, or the system itself [17, 91].

Demanding architecturally significant requirements (ASRs) lead to a small solution space [91], and therefore have more risk that needs to be mitigated with increased architecture design effort through the use of the ADDRESS RISK strategy (S2).

Daniel Dvorak, the lead for the NASA software architecture review board, described the concept of *driving requirements*, which "drive the architecture towards new territory," and lead to novel solutions [169] – that is, demanding ASRs. He commented that "we focus more attention on driving requirements because they entail more risk" – in other words, they ADDRESS RISK S2 because of TECHNICAL RISK (F2).

This theory of agile architecture only identified technical risk as affecting up-front architecture effort.

9.6.2 Agile undercover

Hoda, Noble and Marshall (2010) described the practice of an agile team hiding its agility from its customer by presenting the customer with a process-driven front [122]. The authors called this practice *agile undercover*. Teams use agile undercover for a number of reasons: customer scepticism about agile, team distribution, lack of time commitment, a large customer organisation and ineffective customer representation.

This theory of agile architecture confirms this practice. Teams using the BIG DESIGN UP-FRONT strategy (S4) sometimes put on an process-driven front to the customer because of the lack of CUSTOMER AGILITY (F5).

9.6.3 Frameworks and template architectures

Mirakhorli and Cleland-Huang (2013) noted the benefits of reference architectures: agile is "more effective when there are known architectural solutions, perhaps in the form of reference architectures" [169]. These reference architectures are the frameworks and template architectures referred to in the USE FRAMEWORKS AND TEMPLATE ARCHITECTURES strategy (S5).

Kruchten, Obbink and Stafford (2006) called frameworks *precooked* architectures [148] because a lot of the architecture is already defined within the framework. They make agile more effective because they simplify the design and reduce the architectural (and development) effort required [209, 225].

This simplified design and reduced effort increases the architecture's ability to RESPOND TO CHANGE (S1), reduces TECHNICAL RISK (F2), and hence reduces the need to ADDRESS RISK (S2).

Cervantes, Velasco-Elizondo and Kazman (2013) also commented on the use of frameworks, noting that the framework functionality may need to be extended if it does not provide the required functionality out of the box [63]. This increases risk and effort, addressed by S2.

9.6.4 'Projects' and continuous flow

Most software engineers describe the work they do as belonging to a *project*, which, traditionally, is defined as a well-defined block of work with a fixed delivery date [115, 185]. Because agile teams deliver working software at regular intervals, and the scope is not fixed, some prefer not to describe agile software development as projects. For example, Poppendieck and Cusumano (2012) wrote:

> "When software is delivered quickly, thinking about software development as a project is an inappropriate metaphor. It's much better to think of software as a flow system where soft-

> ware is designed, developed, and delivered in a steady flow of small changes.” [183]

Facetiously, Kelly (2014) likened this continuous flow to a waterfall [134].

The discussion on REQUIREMENTS INSTABILITY force (F1) noted that when developing using a continuous workflow, not only does defining a set of requirements up-front (and hence designing architecture in full up-front) become very difficult, even the notion of a set of requirements may not be valid. The architecture must have emergent decisions and must evolve.

(Note the term ‘project’ is also often used to loosely denote agile workflows, and is used in this way in this thesis.)

Chapter 10

Conclusion

"My goal is not following my design. My goal is to build the right application that meets my customer's needs. That's the purpose." (P22, senior manager)

This thesis has presented research in which a theory of agile architecture was developed to explain how agile software teams design a software architecture that is able to respond to change, and which explains how agile teams determine how much architecture to design up-front.

This chapter concludes the thesis. Section 10.1 lists this research's contributions to theory and section 10.2 describes the implications for practice. Section 10.3 discusses the limitations of the research and the theory, and section 10.4 discusses how the research could be extended in the future. Finally, section 10.5 concludes the chapter.

10.1 Contributions

This research makes four contributions to software engineering theory. The main contribution is a theory of agile architecture that describes how agile teams design an agile architecture and explains how teams determine how much up-front architecture they design. The other contributions are an explanation of the impact of requirements stability on up-front design effort, the impact of size and complexity on up-front design effort, and the impact of cost on up-front design effort.

10.1.1 A theory of agile architecture

The main contribution of this thesis is a theory of agile architecture.

An agile architecture is a software architecture that satisfies the definition of agility: an architecture that is designed so that it can respond to change by being modifiable and tolerant of change (see Chapter 5). An agile architecture is not a static set of design decisions; an agile architecture evolves as the system is developed. An agile architecture is therefore both an output of the architecting process (that is, a set of design decisions that can be described as being modifiable and tolerant of change) and is a part of the architecting process (that is, architecture decisions are timed or made in such a way as to make the evolving design modifiable and tolerant of change). An agile architecture has less up-front design and more emergent decisions than a plan-driven up-front architecture. Section 9.2 contains a more detailed discussion on agile architecture.

Designing an agile architecture

To increase an agile architecture's modifiability, teams use a number of tactics: teams ensure the design is simple, they prove the design with code iteratively and they use good design practices. To increase tolerance, teams delay decisions and plan for options.

Three of the tactics, delaying decisions, simplicity and proving the design with code, all decrease the up-front effort and make the architecture more emergent; the other two tactics, using good design practices and planning for options, may initially increase effort, but this effort is saved later through reduced refactoring when requirements change.

Three of the tactics affect the architecture itself: keeping the design simple, using good design practices and planning for options are all characteristics of the architecture; their use can be determined a posteriori by inspecting the architecture. The other two tactics affect the design process, but not necessarily the final design itself. These tactics are proving the design with code and delaying decisions. Proving the design with code helps keep a design simple by reducing over-engineering, and delaying decisions increases the tolerance to change by only making decisions at the last possible moment before the decision is required, and when uncertainty is lowest. Section 7.1 describes the tactics that teams use in more detail.

Determining how much architecture to design up-front

A team's ability to design an agile architecture is affected by the team's culture, the customer's agility and the team's experience in architecture design and the technology.

Reducing up-front effort increases a team's ability to respond to change; however, this reduction must be balanced with the need to mitigate technical risk (section 7.2): technical risk is reduced through additional up-front architecture design, to a level that the team and the customer are comfortable with. A system with a high level of uncertainty or a high cost of failure will require far more risk mitigation than a system that is well understood, has low risk and a low cost of failure. For example, a medical system used to record diagnostic data will require more up-front design than a business website used for marketing. Not sufficiently mitigating risk may mean requirements cannot be met, increasing later rework effort – and thereby reducing the team's ability to respond to change.

If a customer requires early value from the system being built – perhaps in the form of cash flow from early adopters or if they wish to carry out a minimum viable product (MVP) marketing experiment – and if the system is low risk, the team may be able to eliminate all up-front architecture design, and have an architecture that is totally emergent (section 7.3).

On the other hand, the team's customer may not support the team's agile practices: they may prefer traditional plan-driven processes, they may require a fixed-scope, fixed-price contract, or they may need to approve architecture decisions before development begins. In these circumstances the team may need to take the big design up-front (BDUF) approach, and perhaps put on an 'agile undercover' front for the customer (section 7.4). More design is also beneficial to teams that do not have an agile culture and are inexperienced.

Standard frameworks and template architectures greatly reduce technical risk for standard problems and hence greatly reduce the up-front design effort required. Frameworks make it easier to make changes when requirements change and make it less critical to get decisions right up-front (section 7.5).

A full discussion on how teams determine how much architecture they design up-front is provided in chapters 6 and 7.

The role of 'architect' varies between agile teams and the system being built. Architecture is designed collaboratively by the team; in small teams, this may be done by consensus. In a larger team where consensus is not possible, a number of team members may participate in the architecture design process, but a single team member normally has overall responsibility for architecture decisions. In larger systems that span multiple teams, decisions may be made by the team but approved outside the team – perhaps by external architects or by a group of representatives from the teams. This approval ensures the architecture is suitable for the overall system rather than each team individually. In some circumstances the architecture decisions are made outside the team: decisions may be made by architects who are not team members or by customers themselves, perhaps for strategic reasons. Decisions are usually made outside the team when the customer is not agile.

A full discussion on the role of the architect is provided in Chapter 8.

10.1.2 The impact of requirements instability on up-front design effort

This research has found that the amount of effort a team puts into up-front architecture design depends on the team's ability to design an agile architecture. The more agile the architecture is, the less up-front design the team needs to do, which in turns allows the team to respond to change sooner.

It is frequently stated in the literature that up-front design effort depends on the instability of the requirements: the less stable the requirements, the less up-front effort, and the more stable, the more effort (see section 9.3.1). The theory of agile architecture suggests this assumption is not correct. Requirements instability triggers a team to use agile methodologies and to design an agile architecture, but does not directly affect the amount of effort. In other words, when requirements are unstable, teams use agile methodologies. How much up-front design the team does depends on its ability to design an agile architecture and the presence of technical risk.

This is not to say that requirements instability and up-front effort are independent: a team in a highly changeable environment may be motivated to become more agile so that it can better respond to change and design a more agile architecture, while a team in a less changeable environment may not need to be as agile.

A full discussion on the impact of requirements instability is provided in section 6.1. Section 7.1 discusses how teams design an architecture that can respond to unstable requirements, and sections 9.3.1 and 9.3.3 provides a further discussion of requirements instability in the context of related work.

10.1.3 The impact of size and complexity on up-front design effort

This research has found that complexity is more important than size in determining how much architecture teams design up-front. Complexity increases risk which, in turn, requires more architecture design. On the other hand, size, frequently regarded in the literature as having a direct impact on up-front design effort, does not directly increase technical risk, and does not directly affect up-front architecture design.

Size, however, may have an indirect impact: size magnifies the effect of diversity and connectivity, which increases complexity and hence up-front effort. If there is very little diversity within the system and little connectivity with other systems, then size has very little impact on complexity and hence up-front effort.

Complexity is caused by demanding architecturally significant requirements (section 2.3.4), by many integration points with other systems, and by interaction with legacy systems (those that are no longer being actively engineered) (section 6.2).

A full discussion on the impact of size and complexity is provided in section 9.3.2.

10.1.4 Designing an architecture to minimise cost

This research has found that maximising the delivery of value is often more important to the customer than purely minimising cost, which means that in some situations, agile teams do not try to minimise the overall cost of development. If a team designs an emergent architecture to provide a customer with early value (through early cash flow), the team releases software early at the expense of architectural rework later and higher cost. The early release version is simple and is only capable of supporting the early adopters of the system. The teams fully understand that they will have to redesign the architecture later as the system grows, but this rework comes at a time when the customer has cash flow and is more able to pay for that work. Designing the large system from the start may avoid the rework and reduce the overall cost, but it will delay the initial release, leading to a delay in income for the customer. The customer has to fund more design and development work prior to release, and risks not being able to follow their business plan.

The impact of cost on up-front design is described in section 6.3, and in context of related work in section 9.3.3.

10.2 Implications for practice

The theory of agile architecture is intentionally presented in a form that can be applied to software engineering practice: it defines a set of tactics that teams can use to design an agile architecture, and a set of strategies they can use to determine how much architecture to design up-front.

Agile software teams may use these tactics, where applicable, to help them design an agile architecture which is able to respond to change and reduce their up-front design effort. Reducing up-front design helps reduce the time before the customer can provide the first feedback on the system being developed, and reduces rework if requirements change, and hence increases agility.

Teams must trade-off reducing up-front design effort with mitigating technical risk. Teams must understand their customer's and their own appetite for risk, and reduce risk to a satisfactory level through additional design. Teams should only have a totally emergent design if the risk of such a design is acceptable, and if the customer can benefit from an early release of the system, such as through early cash flow from early adopters or by being able to carry out minimum viable product (MVP) marketing experiments. The team needs to be aware that the overall cost of designing an emergent architecture that only addresses today's requirements may be higher than when designing a more future-proof architecture that requires more up-front effort.

Teams must also be aware that while development frameworks and template architectures can greatly reduce the up-front effort required, they are only of benefit where the predefined libraries, components and tools satisfy the system's ASRs, without being forced to fit. If the team needs to design its own libraries or components, then some of the benefits are lost. Risk may be increased and more up-front design may be needed.

This research also has implications for team managers and customers. Team managers and customers need to be comfortable with how much planning and design their teams do up-front, before the first demonstration of progress. The managers and customers need reassurance that the teams are not spending more time planning than they need to, which could lead them to being less responsive to change, and reassurance they are not spending too little time planning, which increases the risk of failure by not capturing all the quality attributes (section 2.4).

This research can be used by the group of agile practitioners who believe a dichotomy exists between architecture and agility, and that architecture is 'all or nothing' (section 2.4). The findings show that up-front architecture design is a trade-off between designing an architecture that can respond to change and addressing risk: the more risk or the lower the tolerance of risk, the more up-front effort is required (Chapter 7). Ignoring this trade-off may leave the system exposed to high rework costs or failure and, perhaps ironically, reduced agility.

This research is useful to non-agile architects: it highlights the differences between an architecture built by an agile team using an agile process and which is specifically designed to be responsive to change, and an architecture built up-front using a process-driven methodology which may be updated with small corrections, but is not generally designed specifically to be modifiable and tolerant of change.

10.3 Limitations

There are a number of limitations in this research.

The grounded theory methodology may limit the research. Grounded theory produces a substantive (empirical) theory that describes the participants under study and cannot necessarily be applied to other contexts. The theory of agile architecture is therefore only applicable within the research boundaries defined in section 9.1.2. While the forces and strategies identified in this theory may exist in other contexts, such as in embedded systems and enterprise systems, we cannot be sure that other forces and strategies do not exist, and that the relationships are the same.

The research design may also limit the research. I sought practitioners who were accessible to me, who were either local or were in places I was visiting. This limited the range of participants to those areas, which in turn may have restricted the practices, projects, experiences and practitioner characteristics included.

In section 4.1 I noted that I did not seek specific agile methodologies. Rather, I judged participants' agility based on whether or not their descriptions of the methodologies they used met the definition of agility offered in section 2.2.3. While many participants did describe how they responded to change and were clearly agile, others described specified particular methodologies and practices. As noted in section 2.2.3, following the practices of a particular methodology does not necessarily make a team agile. Indeed, some participants were clearly less agile, and this was identified in the interviews, and explained according to their context using appropriate forces (Chapter 6).

The data was biased towards data obtained from interviews (section 4.1.3). I obtained some written data in the form of software architecture documentation and models. I did not obtain data in other forms such as observations. Observations of architecture decisions would have needed to have been over an extended period of time to have been useful; this would have been difficult to set up, would have been very time consuming and may not have provided data that was as rich as the interview data. No participants were in the start-up phase of a project when I interviewed them.

While interview data was very rich, there may have been some limitations due to time constraints. Out of respect for participants' time, I tried to limit interviews to one hour. I failed almost universally to keep to this limit; the average interview length was one hour and ten minutes (section 4.1.2) and many early interviews were more than one and a half hours long. Even so, in many cases I did not finish my interview schedule and more time would have been useful to ensure all issues were covered. The impact of not finishing the interview schedule was not major: because of the semi-structured nature of the interviews, many participants answered later questions while discussing earlier questions; drawing interviews to a close was a judgement call based on further questioning yielding diminishing returns.

Another limitation may be in my ability as a researcher. I am a beginner grounded theorist; this research was my first grounded theory research project. It took me a lot of time to develop an understanding of grounded theory sufficient to undertake this project, and I took several backwards steps during analysis before progressing again. It took a long time for the categories and core category to emerge, which may (or may not) have been caused by my inexperience.

I am also new to the software industry. While this was an advantage overall because I could undertake this research from a neutral position (section 3.4), it also took some time for me to become fully comfortable with architecture terminology used by participants.

10.4 Future work

This research presents a number of opportunities for future work.

10.4.1 More in-depth understanding

The theory of agile architecture is a high-level theory that addresses a very broad objective (section 3.1.1). There is potential for further grounded theory research to continue to explore agile architecture in the same context with a more specific research objective. For example, further research could explore a selection of the forces, strategies or their relationships in more detail, linking them more tightly to specific scenarios, such as building ongoing mass market products and services versus one-off bespoke products, or green field development versus redevelopments.

Further work could examine the tactics used to design an agile architecture in more detail, or search for more tactics.

10.4.2 Different research approaches and methodologies

Further research into agile architecture could use a different qualitative approach to gain a more detailed understanding of certain aspects of agile architecture. For example, case study research could be used to study a small number of cases in more detail or action research could be used to investigate and solve a particular problem of a single participant.

A deductive quantitative approach, such as surveys or the analysis of architecture artefacts, could be used to test the theory of agile architecture. Quantitative research could, for example, be used to identify the relative significance of each of the forces identified in this research, and determine how widely each strategy is used.

An objective of this research was to address the question "how do agile teams determine how much architecture to design up-front?" (section 2.5); an interesting objective for further research would be to quantify this by investigating how much design teams actually do in different situations: "how much architecture do agile teams design up-front?"

10.4.3 Deriving a formal theory

This substantive grounded theory could be extended to form a *formal* grounded theory. A substantive grounded theory is limited to the domain in which it was derived (section 9.1.2); a formal grounded theory connects different areas of substantive research and is more general and more abstract [64].

A formal theory could, for example, be derived by expanding this substantive theory using comparative data from other areas, or could be compared with other new substantive theories from other areas [104].

10.5 Conclusion

This thesis commenced by introducing the need for balance between software architecture and agility. Architecture is the set of decisions that is difficult and expensive to change during development, and therefore is usually made up-front, before development begins. Agility is a software development team's ability to respond to change during development. Agile methods use a feedback loop in which the customer provides feedback to the team about changed or new requirements, and the development team responds to that change by adapting the system. The more agile the team, the shorter the feedback loop: the quicker the customer can provide feedback, and the quicker the team can respond to that feedback. Conversely, a less agile team has a longer feedback loop.

To increase agility, the planning time – including up-front architecture design – is reduced, so that the customer can start providing feedback earlier. This leads to a more emergent and implicit architecture. Reducing the architecture too far could lead to an accidental architecture which fails to meet the quality requirements, and requires more time to fix problems. A failed architecture reduces the team's ability to respond to change and hence reduces its agility.

To maximise agility, a team must find a suitable trade-off between a full up-front architecture design and a totally emergent design.

This thesis then introduced the research gap and the research problem. The trade-of between architecture and agility is not well understood. Various schemes of up-front architecture design offer little guidance as to how much architecture to design up-front and which decisions should be made up-front. There has been very little empirical research into the relationship between architecture and agility. This thesis aimed to help address this research gap with an objective to explore how much effort agile software teams put into architecture design, and discover how teams determine how much architecture to design up-front.

This research used a qualitative approach so that it could explore how architecture design decisions are affected by the experience, knowledge and judgements of the teams, how the teams work together and what they believe are the most appropriate decisions to make up-front. Grounded theory was chosen as the methodology so that a theory that explains how teams determine which decisions to make up-front could be generated. The research design included forty four participants in thirty seven first interviews and six second interviews. Additional data was also obtained in the form of documentation. Participants, who were all architects, senior developers or were otherwise involved with architecture, were from a number of different countries and represented a large spread of projects and domains.

The findings of the research formed a theory of agile architecture. The theory describes how teams design an agile architecture – an architecture that supports the team's agility by being able to respond to change. The theory also describes a set of strategies that teams use to determine how much architecture to design up-front.

This research makes four contributions. The main contribution is the theory of agile architecture. The second contribution describes the impact of requirements stability on up-front design effort. The third contribution explains the impact of size and complexity on up-front design effort, and the fourth contribution describes the impact of cost on up-front design effort.

Appendix A

Human ethics approval

The following documents were part of the approved human ethics application for conducting interviews for this research.

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

HUMAN ETHICS COMMITTEE
Application for Approval of Research Projects
Please write legibly or type if possible. **Applications must be signed by supervisor (for student projects) and Head of School**

Note: The Human Ethics Committee attempts to have all applications approved within three weeks but a longer period may be necessary if applications require substantial revision.

1 NATURE OF PROPOSED RESEARCH:

(a) ~~Staff Research~~/ Student Research (delete one)

(b) If Student Research Degree …**PhD**…………… Course Code …**SWEN690**…

(c) Project Title: **Reconciling Architecture and Agile: How Much Architecture?**

………………………………………………………………………………………………

2 INVESTIGATORS:

(a) Principal Investigator

Name …**Michael Waterman**…………………………………………………………………

e-mail address …**Michael.Waterman@ecs.vuw.ac.nz**……………………………………

School/Dept/Group …**Engineering and Computer Science** ……………………………

(b) Other Researchers	Name	Position
	………………………………………………..	……………………………………………………
	………………………………………………..	……………………………………………………

(c) Supervisor (in the case of student research projects)

Prof. James Noble, Professor of Computer Science ……………………………………
Dr George Allan, Senior Lecturer ……………………………………………………………

3 DURATION OF RESEARCH

(a) Proposed starting date for data collection …**1 August 2010** ……………………………
(Note: that NO part of the research requiring ethical approval may commence prior to approval being given)

(b) Proposed date of completion of project as a whole …**1 February 2013**……………………

4 **PROPOSED SOURCE/S OF FUNDING AND OTHER ETHICAL CONSIDERATIONS**

(a) Sources of funding for the project
Please indicate any ethical issues or conflicts of interest that may arise because of sources of funding e.g. restrictions on publication of results

N/A ……………………………………………………………………………………………………

………

(b) Is any professional code of ethics to be followed ~~Y □~~ **N** □

If yes, **name** ……………………………………………………………………………………………

(c) Is ethical approval required from any other body ~~Y □~~ **N** □

If yes, name and indicate when/if approval will be given

………

5 **DETAILS OF PROJECT**

Briefly Outline:

(a) The objectives of the project

This study is examining the up-front architecture design in agile software development projects. The investigation will use qualitative research methods (such as grounded theory) to observe how much up-front architecture design is used in industry and the outcomes (levels of success) of the projects.

(b) Method of data collection

A combination of qualitative methods such as grounded theory will be used for data collection. These include interviews and observations of agile software developers on their experiences of agile projects, participation in agile projects, and review of project documentation. Interviews will be audio recorded to reduce the risk of misinterpretation.

(c) The benefits and scientific value of the project

Projects using up-front design often cannot respond to changing requirements and thus do not meet the needs of the customer. Conversely projects using agile methods often suffer from the effects of too little up-front design, and thus risk ad-hoc architecture that is not robust enough to withstand the changes in design that is the characteristic of agile methods. This research will

determine how much up-front design should be included and will thus provide a method of reducing the risk of failure of software projects.

(d) Characteristics of the participants

Agile developers including architects, project leaders, project managers, project stakeholders, other corporate and IT staff.

(e) Method of recruitment

Potential participants will be recruited through existing industry contacts, industry user groups and email lists. I will discuss the purpose and scope of the research with potential participants and agree on suitable conditions under which they can participate.

(f) Payments that are to be made/expenses to be reimbursed to participants

None. It is not expected that participants will incur costs. ....

(g) Other assistance (e.g. meals, transport) that is to be given to participants

None. It is not expected that participants will incur costs. ....

(h) Any special hazards and/or inconvenience (including deception) that participants will encounter

None ....

....

(i) State whether consent is for (delete where not applicable):

(i) the collection of data
~~(ii) attribution of opinions or information~~
~~(iii) release of data to others~~
(iv) use for a conference report or a publication
~~(v) use for some particular purpose (specify)~~

……………………………………………………………………………………………...

……………………………………………………………………………………………...

Attach a copy of any questionnaire or interview schedule to the application

(j) How is informed consent to be obtained (see sections 4.1, 4.5(d) and 4.8(g) of the Human Ethics Policy)

(i) the research is strictly <u>anonymous</u>, an information sheet is supplied and informed consent is implied by voluntary participation in filling out a questionnaire for example (include a copy of the information sheet)

~~Y☐~~ N☐

(ii) the research is not anonymous but is confidential and informed consent will be obtained through a signed consent form (include a copy of the consent form and information sheet) Y☐ ~~N☐~~

(iii) the research is neither anonymous or confidential and informed consent will be obtained through a signed consent form (include a copy of the consent form and information sheet) ~~Y☐~~ N☐

(iv) informed consent will be obtained by some other method (please specify and provide details) ~~Y☐~~ N☐

……………………………………………………………………………………………...

……………………………………………………………………………………………...

With the exception of anonymous research as in (i), if it is proposed that written consent will not be obtained, please explain why

……………………………………………………………………………………………...

……………………………………………………………………………………………...

……………………………………………………………………………………………...

(k) If the research will not be conducted on a strictly anonymous basis state how issues of confidentiality of participants are to be ensured if this is intended. (See section 4..1(e) of the Human Ethics Policy). (e.g. who will listen to tapes, see questionnaires or have access to data). Please ensure that you distinguish clearly between anonymity and confidentiality. Indicate which of these are applicable.

(i) access to the research data will be restricted to the investigator

~~Y☐~~ N☐

(ii) access to the research data will be restricted to the investigator and their supervisor (student research) Y☐ ~~N☐~~

(iii) all opinions and data will be reported in aggregated form in such a way that individual persons or organisations are not identifiable ~~Y☐~~ N☐

(iv) Other (please specify)

Any individual opinions and data (such as names, companies and project descriptions) will be anonymised so they are not identifiable in any publications.

……………………………………………………………………………………………...

(l) Procedure for the storage of, access to and disposal of data, both during and at the conclusion of the research. (see section 4.12 of the Human Ethics Policy). Indicate which are applicable:

(i) all written material (questionnaires, interview notes, etc) will be kept in a locked file and access is restricted to the investigator **and supervisors** **Y** □ ~~**N**□~~

(ii) all electronic information will be kept in a password-protected file and access will be restricted to the investigator **and supervisors** **Y** □ ~~**N**□~~

(iii) all questionnaires, interview notes and similar materials will be destroyed:

(a) at the conclusion of the research **Y** □ **N** □

<u>**or**</u> (b) **two** years after the conclusion of the research **Y** □ ~~**N**□~~

(iv) any audio or video recordings will be returned to participants and/or electronically wiped **after two years** **Y** □ ~~**N**□~~

(v) other procedures (please specify):

………………………………………………………………………………………………...

………………………………………………………………………………………………...

If data and material are not to be destroyed please indicate why and the procedures envisaged for ongoing storage and security

………………………………………………………………………………………………...

………………………………………………………………………………………………...

………………………………………………………………………………………………...

(m)Feedback procedures (See section 7 of Appendix 1 of the Human Ethics Policy). You should indicate whether feedback will be provided to participants and in what form. If feedback will not be given, indicate the reasons why.

Participants will be given the opportunity to review interview transcriptions and will be offered publications that arise from this research.

(n)Reporting and publication of results. Please indicate which of the following are appropriate. The proposed form of publications should be indicated on the information sheet and/or consent form.

(i) publication in academic or professional journals **Y** □ ~~**N**□~~

(ii) dissemination at academic or professional conferences **Y** □ ~~**N**□~~

(iii) deposit of the research paper or thesis in the University Library (student research) **Y** □ ~~**N**□~~

(iv) other (please specify)

………………………………………………………………………………………………...

………………………………………………………………………………………………...

Signature of investigators as listed on page 1 **(including supervisors) and Head of School.**

NB: <u>All investigators and the Head of School must sign before an application is submitted for approval</u>

……………………………………………	Date………………………..
……………………………………………	Date………………………..
……………………………………………	Date………………………..

Head of School:

…………………………………………….	Date ………………………..

APPLICATIONS FOR HUMAN ETHICS APPROVAL

CHECKLIST

- Have you read the Human Ethics Policy?
- Is ethical approval required for your project?
- Have you established whether informed consent needs to be obtained for your project?
- In the case of student projects, have you consulted your supervisor about any human ethics implications of your research?
- Has your supervisor read and signed the application?

- Have you included an information sheet for participants which explains the nature and purpose of your research, the proposed use of the material collected, who will have access to it, whether the data will be kept confidential to you, how anonymity or confidentiality is to be guaranteed?

- Have you included a written consent form?
- If not, have you explained on the application form why you do not need to get written consent?
- Are you asking participants to give consent to:

    - collect data from them
    - attribute information to them
    - release that information to others
    - use the data for particular purposes

- Have you indicated clearly to participants on the information sheet or consent form how they will be able to get feedback on the research from you (e.g. they may tick a box on the consent form indicating that they would like to be sent a summary), and how the data will be stored or disposed of at the conclusion of the research?

- Have you included a copy of any questionnaire or interview checklist you propose using?

- Has your application been seen by the head of your school or department (or the person given responsibility to consider applications on behalf of the head (see section 4.5(b) of the Human Ethics Policy).

PLEASE FORWARD YOUR COMPLETED APPLICATION FORM TO THE SECRETARY, HUMAN ETHICS COMMITTEE OR, IN THE CASE OF APPLICATIONS FROM SCHOOLS OR DEPARTMENTS WITH AN APPROVED ETHICS SUB-COMMITTEE, TO THE CONVENER OF THAT SUB-COMMITTEE

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Reconciling Architecture and Agile: How Much Architecture? *Participant Information Sheet*

Michael Waterman

Researcher: Michael Waterman (Michael.Waterman@ecs.vuw.ac.nz, +64 4 463 5233 x8486)
Supervisors: Prof. James Noble (kjx@ecs.vuw.ac.nz; +64 4 463 6736)
Dr. George Allan (George.Allan@ecs.vuw.ac.nz, +64 4 463 6741)

This research is being conducted as a part of studies towards a PhD degree in the School of Engineering and Computer Science at Victoria University of Wellington, New Zealand.

Introduction

This study is examining architecture design in agile software development projects. I am interviewing agile practitioners involved in the high level design of software, in which we discuss agile projects that the practitioner has worked on.

Research results

The data gathered in interviews and documentation will form the basis for the research. It is expected that the research will be published in the form of a thesis (deposited in the University Library) and academic research papers.

If requested the participant will be provided with any research publications that arise from this research.

Treatment of data and confidentiality

All data obtained by each participant will be treated as confidential. In particular, the names, companies, customers and any commercial information will not be released or published.

- The participant will be given the opportunity to check the transcript of the interview and will have a period of one month in which to submit error corrections or to withdraw themselves or any data they have provided without having to give reasons and without penalty.
- Any results published will be anonymised so that it is not possible to identify the participant, his or her employer, the project or the project's customer.
- In keeping with the Privacy Act 1998, any data provided will only be used for the purpose stated, and will be destroyed (or returned) two years after the completion of the research.
- I am able to sign non-disclosure agreements if required.

This research has obtained ethics approval for research projects involving human participants, as required by the university.

If you have any questions or would like to receive further information about the research, please contact me or one of my supervisors.

Michael Waterman
School of Engineering and Computer Science
Victoria University of Wellington

Reconciling Architecture and Agile: How Much Architecture? *Interview Schedule*

Michael Waterman

26 May 2010

Researcher: Michael Waterman (Michael.Waterman@ecs.vuw.ac.nz, +64 4 463 5233 x8486)
Supervisors: Prof. James Noble (kjx@ecs.vuw.ac.nz; +64 4 463 6736)
Dr. George Allan (George.Allan@ecs.vuw.ac.nz, +64 4 463 6741)

This research is being conducted as a part of studies towards a PhD degree in the School of Engineering and Computer Science at Victoria University of Wellington, New Zealand.

General Information

This study is examining architecture design in agile software development projects. I am interviewing agile practitioners involved in the high level design of software, in which we discuss agile projects that the practitioner has worked on.

Interview schedule

The interview will be semi-structured. The schedule will include the following topics and types of questions:

1. Participant background – for example, role in the project team, and a brief overview of selected software development projects previously worked on and responsibilities.
2. A description of the project(s), including any factors that made the project unusual or difficult.
3. The development methodology used – for example, completely agile, partially agile, or not agile at all? What were the roles of the members of the development team?
4. The architecture of the project, including architecture design process, architecture design language, models and tools used.
5. Success factors, including whether or not the functional requirements and the non-functional requirements were met, and whether or not it was completed on time and on budget.
6. A discussion of any problems that arose.
7. Any other issues or comments.

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Reconciling Architecture and Agile: How Much Architecture? *Consent to Participate in Research*

Michael Waterman

Researcher: Michael Waterman (Michael.Waterman@ecs.vuw.ac.nz, +64 4 463 5233 x8486)
Supervisors: Prof. James Noble (kjx@ecs.vuw.ac.nz; +64 4 463 6736)
Dr. George Allan (George.Allan@ecs.vuw.ac.nz, +64 4 463 6741)

This research is being conducted as a part of studies towards a PhD degree in the School of Engineering and Computer Science at Victoria University of Wellington, New Zealand.

General Information

This study is examining architecture design in agile software development projects. I am interviewing agile practitioners involved in the high level design of software, in which we discuss agile projects that the practitioner has worked on.

Consent

This form allows us to record participants' consent to participate in this study. Consent is standard practice and is a requirement of the University.

The participant agrees that they:

- have been given and have understood an explanation of this research project,
- have had an opportunity to ask questions and have them answered to their satisfaction,
- understand that any information they provide will be kept confidential to the researcher and the supervisors, the published results will not use their name, and that no opinions will be attributed to them in any way that will identify them, their employer, their client(s) or the project(s) discussed,
- understand they will have the opportunity to check a transcript of the interview and have a period of one month in which to submit error corrections or to withdraw themselves or any data they have provided without having to give reasons and without penalty,
- understand that the data they provide will not be used for any other purpose or released to others without their written consent, and that all personal information will be destroyed two years after the completion of the research.

The participant consents to:

- being interviewed by the researcher,
- the interview being sound recorded to reduce the risk of misinterpretation.

Do you wish to receive copies (or notification) of publications that arise from this research:
Yes / No

Name:

Signature: Date:

If you have any questions or would like to receive further information about the research, please contact me or one of my supervisors.

Appendix B

Interview schedule

The following document contains two versions of the question schedule used in the interviews. The first schedule, on the following page, is the initial schedule used at the start of the data gathering process. The second schedule is a later version (version 7) used at the end of the data gathering process, and was developed in response to the changing focus of data gathering as the research progressed.

The differences are first version asks very general questions about the agile team, the agile project, the development methodology, the architecture and how it evolved, and any success factors or problems. The later version also asked many of these questions, but was looking more for information about specific themes, such as up-front architecture drivers: complexity and size, frameworks, agility and experience.

Reconciling Architecture and Agile: How Much Architecture? *Interview Schedule*

Michael Waterman

24 May 2010

Researcher:	Michael Waterman (Michael.Waterman@ecs.vuw.ac.nz, +64 4 463 5233 x8486)
Supervisors:	Prof. James Noble (kjx@ecs.vuw.ac.nz; +64 4 463 6736)
	Dr. George Allan (George.Allan@ecs.vuw.ac.nz, +64 4 463 6741)
Research topic:	Reconciling Architecture and Agile: How Much Architecture?

This research is being conducted as a part of studies towards a PhD degree in the School of Engineering and Computer Science at Victoria University of Wellington, New Zealand.

General Information

This study is examining architecture design in agile software development projects. I am interviewing agile practitioners involved in the high level design of software, in which we discuss agile projects that the practitioner has worked on.

Interview schedule

The interview will be semi-structured. The schedule will include the following topics and types of questions:

1. Participant background
    - Please describe your role in the project team, and how long you have performed that role.
    - Give a brief overview of selected software development projects you've previously worked on, and your responsibilities.
2. A description of the project(s)
    - Please give a brief overview of the project, including its domain, whether it was internal or external, its size (duration/team size/budget).
    - How well were the requirements understood in advance?
    - Did the requirements change during the project?
    - Were there any factors that made the project unusual or difficult, such as a distributed team or unfamiliarity with domain?
3. The development methodology used
    - Please describe the overall development methodology used. Was the project completely agile, partially agile, or not agile at all?
    - What were the roles of the members of the development team?
    - What agile experience do the members of the team have?
    - To what extent did the customer and other stakeholders buy-in to the methodology?
4. The architecture of the project

- What was the architecture?
- What architecture design process was used? How much of the architecture design was up-front and how much was developed during implementation?
- What architecture design language, models and tools were used?
- Were any architectural patterns used?
- Did the architecture change or evolve during development?
- Please tell me about any interactive and iterative methods that were used in the architecture design and coding processes.
- Describe any risk management or risk assessment undertaken.
- Did the architecture follow any standards (either in-house or external)?
- Was the architecture documented?

5. Success factors

   - Please describe the level of success of the project. Include (but don't limit to) the following success measures:
     - Describe to what extent the functional requirements were met and the customer was satisfied.
     - To what extent were the non-functional requirements met?
     - Was it completed on time and on budget?
   - Please note any success factors, such as staff experience or expertise, client interaction, tools used, etc

6. A discussion of any problems or issues, and to what they can be attributed

   - Please describe any problems relating to the design of the software that arose during its development. These may include going over budget, missing milestones, excessive refactoring or even needing to re-design the architecture.
   - Describe what you believe caused these problems.
   - Was there anything you or the team could have done to have prevent or mitigate the problem(s)?
   - Were any of the problems addressed – or not addressed – by risk management or risk assessment?
   - What would you do differently with the benefit of hindsight?

7. Are there any other issues you'd like to raise or comments you'd like to make?

   - For example, if the degree of agile methodology used reflects an expected level of up-front architecture design that is required.
   - Is there anyone else I can talk to?

Reconciling Architecture and Agile: How Much Architecture? *Interview Schedule*

Michael Waterman

24 May 2010; Updated 31 May 2013 (Version 7)

Researcher:	Michael Waterman (Michael.Waterman@ecs.vuw.ac.nz, +64 4 463 5233 x8486)
Supervisors:	Prof. James Noble (kjx@ecs.vuw.ac.nz; +64 4 463 6736)
	Dr. George Allan (George.Allan@ecs.vuw.ac.nz, +64 4 463 6741)
Research topic:	Reconciling Architecture and Agile: How Much Architecture?

This research is being conducted as a part of studies towards a PhD degree in the School of Engineering and Computer Science at Victoria University of Wellington, New Zealand.

General Information

This study is examining architecture design in agile software development projects. I am interviewing agile practitioners involved in the high level design of software, in which we discuss agile projects that the practitioner has worked on.

Interview schedule

The interview will be semi-structured. The schedule will include the following topics and types of questions:

1. Participant background
   - Please describe your role in the project team, and how long you have performed that role.
   - Give a **brief** overview of selected software development projects you've previously worked on, and your responsibilities.
2. A description of the project(s)
   - Please give a brief (1 minute!) overview of the project, including its domain, type of application, whether it was internal or external, its size (duration/team size/budget).
   - Were there any factors that made the project unusual or difficult, such as a distributed team or unfamiliarity with domain?
3. The development methodology used
   - Please describe the overall development methodology used and what agile practices were used. *(For example: timeboxing/sprint planning, backlogs, continuous integration, self-organising, interactive customer, daily stand-ups, retrospectives, sprint 0, TDD/unit testing, peer programming...)*
   - What were the roles of the members of the development team?
   - What agile experience do the members of the team have?
   - To what extent did the customer and other stakeholders buy-in to the methodology?

- How well were the requirements understood in advance? Did the requirements change during the project?

4. The architecture of the project

    - What was the architecture? (Please describe your architecture or its characteristics and the levels of architecture) (That is, **what *is* architecture?**)
    - How (and why) did you choose your architecture?

5. How would you describe the relationship between complexity, size and up-front architecture planning?

6. In your experience, what affect do bespoke components and libraries, multiple frameworks, legacy and integration have on complexity and up-front planning, and why?

7. In your experience, what affect does your level of agility have on up-front planning, and why?

8. In your experience, how does the type of product you're building affect the up-front planning? (e.g. redevelopment vs green fields; one-off vs ongoing; in-house vs external)

9. In your experience, is there anything else that affects up-front planning? (such as requirements stability, team size)

10. Who makes the architectural decisions?

11. What (else) does the person who makes the architectural decisions do?

12. How do you manage architectural tasks?(e.g. architectural task backlog)

13. Are there any other issues you'd like to raise or comments you'd like to make?

References

[1] ABRAHAMSSON, P., ALI BABAR, M., AND KRUCHTEN, P. Agility and architecture: Can they coexist? *IEEE Software 27*, 2 (Mar.–Apr. 2010), 16–22.

[2] ABRAHAMSSON, P., SALO, O., RONKAINEN, J., AND WARSTA, J. *Agile software development methods: review and analysis*. VTT Publications, 2002.

[3] ABRANTES, J., AND TRAVASSOS, G. Common agile practices in software processes. In *International Symposium on Empirical Software Engineering and Measurement (ESEM '11)* (Banff, Alberta, 2011), pp. 355–358.

[4] ADAMS, D. *Dirk Gently's holistic detective agency*. Heinemann, 1987.

[5] ADAMS, W. An agile cure for all ills? *IEEE Software 26*, 06 (Nov.–Dec. 2009).

[6] ADOLPH, S. What lessons can the agile community learn from a maverick fighter pilot? In *Agile Conference (AGILE '06)* (Minneapolis, Minnesota, 2006).

[7] ADOLPH, S., HALL, W., AND KRUCHTEN, P. Using grounded theory to study the experience of software development. *Journal of Empirical Software Engineering 16*, 4 (August 2011), 487–513.

[8] ADOLPH, S., HALL, W., AND KRUCHTEN, P. A methodological leg to stand on: lessons learned using grounded theory to study software development. In *Conference of the center for advanced studies on collaborative research: meeting of minds (CASCON '08)* (Richmond Hill, Ontario, 2008), pp. 166–178.

[9] ALLAN, G. A critique of using grounded theory as a research method. *Electronic Journal of Business Research Methods 2*, 1 (July 2003).

[10] ALLAN, G. The legitimacy of grounded theory (keynote address). In *European Conference on Research Methods* (Dublin, July 2006).

[11] ALVESSON, M., AND SKÖLDBERG, K. *Reflexive methodology: New vistas for qualitative research.* SAGE, 2009.

[12] AMBLER, S. W. Examining the agile cost of change curve. Retrieved 7 July 2014 from `http://www.agilemodeling.com/essays/costOfChange.htm`.

[13] AMBLER, S. W. Lessons in agility from internet-based development. *Software, IEEE 19*, 2 (Mar.–Apr. 2002), 66–73.

[14] AMBLER, S. W. Agile architecture: Strategies for scaling agile development, 2012. Retrieved 22 March 2014 from AmbySoft Inc: `http://www.agilemodeling.com/essays/agileArchitecture.htm`.

[15] AMBLER, S. W. The criteria for determining whether a team is agile, 2012. Retrieved 22 March 2014 from Ambysoft Inc: `http://www.agilemodeling.com/essays/agileCriteria.htm`.

[16] AMBLER, S. W. Agile enterprise architecture, 2013. Retrieved 22 March 2014 from AmbySoft Inc: `http://www.agiledata.org/essays/enterpriseArchitecture.html`.

[17] AMBLER, S. W. Agile risk management: A disciplined approach, Nov. 2013. [Blog entry.] Retrieved 22 March 2014 from: `http://disciplinedagiledelivery.wordpress.com/2013/11/28/agile-risk-management/`.

[18] AMBLER, S. W. Agile modeling, 2014. Retrieved 22 March 2014 from Ambysoft Inc: `http://www.agilemodeling.com/`.

[19] AMBLER, S. W. Agile models distilled: Potential artifacts for agile modeling, 2014. Retrieved 22 March 2014 from Ambysoft Inc: `http://www.agilemodeling.com/artifacts/`.

[20] AMELLER, D., AYALA, C., CABOT, J., AND FRANCH, X. Non-functional requirements in architectural decision making. *IEEE Software 30*, 2 (Mar.–Apr. 2013), 61–67.

[21] ANDERSON, D. J. *Kanban*. Blue Hole Press, 2010.

[22] ATKINSON, S. R., AND MOFFAT, J. The agile organization: From informal networks to complex effects and agility. Tech. rep., DTIC Document, 2005.

[23] BAKER, A., VAN DER HOEK, A., OSSHER, H., AND PETRE, M. Guest editors' introduction: Studying professional software design. *IEEE Software 29*, 1 (Jan.–Feb. 2012), 28–33.

[24] BALLARD, G. Positive vs negative iteration in design. In *Conference of the International Group for Lean Construction, (IGLC-6)* (Brighton, 2000), pp. 17–19.

[25] BASS, L., CLEMENTS, P., AND KAZMAN, R. *Software Architecture in Practice*, 2 ed. SEI Series in Software Engineering. Addison-Wesley, 2003.

[26] BECK, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

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

[28] BECK, K., ET AL. Agile manifesto, 2001. Retrieved 22 March 2014 from n.p.: `http://agilemanifesto.org/`.

[29] BEEDLE, M. Agile at 10 – a state of contradiction, May 2011. Retrieved 22 March 2014 from InfoQ: `http://www.infoq.com/articles/agile-10-contradiction`.

[30] BENNETT, K. Legacy systems: Coping with stress. *IEEE Software 12*, 1 (Jan.–Feb. 1995), 19–23.

[31] BLACK, S., BOCA, P. P., BOWEN, J. P., GORMAN, J., AND HINCHEY, M. Formal versus agile: Survival of the fittest. *IEEE Computer 42*, 9 (Sept. 2009), 37–45.

[32] BLAIR, S., WATT, R., AND CULL, T. Responsibility-Driven Architecture. *IEEE Software 27*, 2 (Mar.–Apr. 2010), 26–32.

[33] BLOOMBERG, J., AND SCHMELZER, R. *The Agile Architecture Revolution: How Cloud Computing, REST-Based SOA, and Mobile Computing are Changing Enterprise IT*. Wiley Online Library, 2013.

[34] BOEHM, B. *Software Engineering Economics*. Prentice-Hall Advances in Computing Science & Technology Series. Prentice-Hall, Inc., 1981.

[35] BOEHM, B. Requirements that handle IKIWISI, COTS, and rapid change. *IEEE Computer 33*, 7 (July 2000), 99–102.

[36] BOEHM, B. Get ready for agile methods, with care. *IEEE Computer 35*, 01 (Jan. 2002), 64–69.

[37] BOEHM, B. Architecting: How much and when? In *Making Software*, A. Oram and G. Wilson, Eds. O'Reilly, 2011.

[38] BOEHM, B., ABTS, C., BROWN, A. W., CHULANI, S., CLARK, B., HOROWITZ, E., MADACHY, R., RIEFER, D., AND STEECE, B. *Software Cost Estimation with COCOMO II with CD-Rom*, 1 ed. Prentice Hall PTR, 2000.

[39] BOEHM, B., LANE, J. A., KOOLMANOJWONG, S., AND TURNER, R. Architected agile solutions for software-reliant systems. In *Agile Software Development: Current Research and Future Directions*, T. Dingsøyr, T. Dybå, and N. B. Moe, Eds. Springer, 2010.

[40] BOEHM, B., AND TURNER, R. Using risk to balance agile and plan-driven methods. *IEEE Computer 36*, 6 (June 2003), 57–66.

[41] BOEHM, B., AND TURNER, R. Balancing agility and discipline: Evaluating and integrating agile and plan-driven methods. *International Conference on Software Engineering (ICSE '04)* (Edinburgh, 2004), 718–719.

[42] BOEHM, B. W. Software engineering economics. *Software Engineering, IEEE Transactions on*, 1 (1984), 4–21.

[43] BOOCH, G. *Object-Oriented Analysis and Design with Applications*, 2 ed. Benjamin/Cummings Pub. Co., 1994.

[44] BOOCH, G. The accidental architecture. *IEEE Software 23*, 3 (May–Jun. 2006), 9–11.

[45] BOOCH, G. The irrelevance of architecture. *IEEE Software 24*, 3 (May–Jun. 2007), 10–11.

[46] BOOCH, G. Architectural organizational patterns. *IEEE Software 25*, 3 (May–Jun. 2008), 18 –19.

[47] BOOCH, G. The defenestration of superfluous architecture accoutrements. *Software Architecture Challenges in the 21st Century (SAC 21 Workshop)* (June 2009).

[48] BOOCH, G. An architectural oxymoron. *IEEE Software 27*, 5 (Sep.–Oct. 2010), 96.

[49] BOOCH, G. The architect's journey. *IEEE Software 28*, 3 (May–June 2011), 10–11.

[50] BOSCH, J. *Design and Use of Software Architectures*. Pearson Education Limited, 2000.

[51] BRAND, S. *How buildings learn: what happens after they're built*. Penguin, 1995.

[52] BREIVOLD, H. P., SUNDMARK, D., WALLIN, P., AND LARSON, S. What does research say about agile and architecture? *The Fifth International Conference on Software Engineering Advances (ICSEA '10)* (Nice, 2010).

[53] BROOKS, F. No silver bullet: Essence and accidents of software engineering. *IEEE Computer 20*, 4 (Apr. 1987), 10–19.

[54] BROWN, S. Software architecture document guidelines. Retrieved 22 March 2014 from n.p.: `http://www.codingthearchitecture.com/pages/book/software-architecture-document-guidelines.html`.

[55] BROWN, S. The role of the architect (why software projects fail), 2009. [Video blog.] Retrieved 22 March 2014 from n.p.: `http://www.codingthearchitecture.com/2009/03/09/the_role_of_the_software_architect_video_part_1.html`.

[56] BROWN, S. Are you a software architect?, Feb. 2010. Retrieved 22 March 2014 from InfoQ: `http://www.infoq.com/articles/brown-are-you-a-software-architect`.

[57] BROWN, S. *Software Architecture for Developers*. LeanPub, 2013.

[58] BROY, M., AND REUSSNER, R. Architectural Concepts in Programming Languages. *IEEE Computer 43*, 10 (Oct. 2010), 88–91.

[59] BRYMAN, A. *Social Research Methods*, 3 ed. Oxford University Press, 2008.

[60] BUSCHMANN, F. A week in the life of an architect. *IEEE Software 29*, 3 (May–June 2012), 94–96.

[61] BUSCHMANN, F., AND HENNEY, K. Architecture and agility: Married, divorced, or just good friends? *IEEE Software 30*, 2 (Mar.–Apr. 2013), 80–82.

[62] CARNEIRO, C., AND BARAZI, R. A. Introducing the rails framework. *Beginning Rails 3* (2010), 1–12.

[63] CERVANTES, H., VELASCO-ELIZONDO, P., AND KAZMAN, R. A principled way to use frameworks in architecture design. *IEEE Software 30*, 2 (Mar.–Apr. 2013), 46–53.

[64] CHARMAZ, K. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. SAGE, 2006.

[65] CHEN, L., ALI BABAR, M., AND NUSEIBEH, B. Characterizing architecturally significant requirements. *IEEE Software 30*, 2 (Mar.–Apr. 2013), 38–45.

[66] CLELAND-HUANG, J., HANMER, R. S., SUPAKKUL, S., AND MIRAKHORLI, M. The twin peaks of requirements and architecture. *IEEE Software 30*, 2 (Mar.–Apr. 2013), 24–29.

[67] CMMI PRODUCT TEAM. CMMI for development version 1.3. Tech. rep., SEI-2010-TR-033, SEI Carnegie-Mellon University, 2010.

[68] COCKBURN, A. Agile software development joins the "would-be" crowd. *Cutter IT Journal 15*, 1 (2002), 6–12.

[69] COCKBURN, A. *Agile software development: the cooperative game*, 2 ed. Addison-Wesley, 2007.

[70] COCKBURN, A. Top ten ways to know you are not doing agile, 2008. Retrieved 22 March 2014 from n.p.: `http://alistair.cockburn.us/Top+ten+ways+to+know+you+are+not+doing+agile`.

[71] COCKBURN, A. Walking skeleton, 2014. Retrieved 22 March 2014 from n.p.: `http://alistair.cockburn.us/Walking+skeleton`.

[72] COCKBURN, A., AND HIGHSMITH, J. Agile software development: the people factor. *IEEE Computer 34*, 11 (Nov. 2001), 131–133.

[73] COHN, M. *User stories applied for agile software development*. Addison-Wesley Professional, 2005.

[74] CONBOY, K. Agility from first principles: Reconstructing the concept of agility in information systems development. *Information Systems Research 20*, 3 (Sept. 2009), 329–354.

[75] CONBOY, K., COYLE, S., WANG, X., AND PIKKARAINEN, M. People over process: key people challenges in agile development. *IEEE Software* (2010).

[76] CONBOY, K., AND FITZGERALD, B. Toward a conceptual framework of agile methods: a study of agility in different disciplines. In *ACM workshop on interdisciplinary software engineering research (WISER '04)* (Newport Beach, California, 2004), pp. 37–44.

[77] CONWAY, M. E. How do committees invent? *Datamation* (April 1968).

[78] COOLEY, C. Daily iterations: approaching code freeze and half the team is not agile. In *Agile Development Conference (ADC '03)* (Salt Lake City, Utah, June 2003), pp. 162–164.

[79] Coplien, J. O., and Bjørnvig, G. *Lean architecture for agile software development*. John Wiley and Sons, Ltd, 2010.

[80] Creswell, J. W. *Qualitative inquiry and research design: Choosing among five approaches*, 3 ed. SAGE, 2013.

[81] Creswell, J. W. *Research design: qualitative, quantitative, and mixed methods approaches*, 4 ed. SAGE, 2014.

[82] Cribb, J. *The accountability of voluntary organisations*. PhD thesis, Victoria University of Wellington, 2005.

[83] Deemer, P., Benefield, G., Larman, C., and Vodde, B. The Scrum primer, 2012. Retrieved 22 March 2014 from The Scrum Foundation: `http://www.scrumtraininginstitute.com/`.

[84] Dorairaj, S. *The Theory of One Team: Agile Software Development with Distributed Teams*. PhD thesis, Victoria University of Wellington, 2013.

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

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

[87] Dybå, T., and Dingsøyr, T. Empirical studies of agile software development: A systematic review. *Information and software technology 50*, 9 (2008), 833–859.

[88] Dybå, T., and Dingsøyr, T. What do we know about agile software development? *IEEE Software 26*, 5 (Sep.–Oct. 2009), 6–9.

[89] ELSSAMADISY, A. Interview: Joshua Kerievsky on system metaphor, Oct. 2009. Retrieved 22 March 2014 from InfoQ: `http://www.infoq.com/interviews/kerievsky-metaphor`.

[90] FABER, R. Architects as Service Providers. *IEEE Software 27*, 2 (Mar.–Apr. 2010), 33–40.

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

[92] FALESSI, D., CANTONE, G., SARCIA', S. A., CALAVARO, G., SUBIACO, P., AND D'AMORE, C. Peaceful Coexistence: Agile Developer Perspectives on Software Architecture. *IEEE Software 27*, 2 (Mar.–Apr. 2010), 23–25.

[93] FERREIRA, J., NOBLE, J., AND BIDDLE, R. Up-front interaction design in agile development. In *Agile Processes in Software Engineering and Extreme Programming (XP '07)* (Como, Italy, 2007), pp. 9–16.

[94] FOOTE, B., AND YODER, J. Big ball of mud. *Pattern languages of program design 4* (1997), 654–692.

[95] FOWLER, M. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[96] FOWLER, M. Is design dead? In *Extreme Programming Examined* (2001), G. Succi and M. Marchesi, Eds., Addison-Wesley Longman, pp. 3–17.

[97] FOWLER, M. Design – who needs an architect? *IEEE Software 20*, 5 (Sept.–Oct. 2003), 11–13.

[98] FOWLER, M. Fixed scope mirage, Sept. 2004. Retrieved 22 March 2014 from n.p.: `http://martinfowler.com/bliki/FixedScopeMirage.html`.

[99] FREUDENBERG, S., AND SHARP, H. The top 10 burning research questions from practitioners. *IEEE Software 27*, 5 (Sep.–Oct. 2010), 8–9.

[100] FURNISS, D., BLANDFORD, A., AND CURZON, P. Confessions from a grounded theory PhD: Experiences and lessons learnt. In *SIGCHI Conference on Human Factors in Computing Systems (CHI '11)* (New York, New York, 2011).

[101] GABRIEL, R. P. *Patterns of Software: Tales from the Software Community*. Oxford University Press, 1996.

[102] GALL, J. *Systemantics: How systems work and especially how they fail*. Pocket Books, 1978.

[103] GALSTER, M., MIRAKHORLI, M., CLELAND-HUANG, J., BURGE, J. E., FRANCH, X., ROSHANDEL, R., AND AVGERIOU, P. Views on software engineering from the twin peaks of requirements and architecture. *ACM SIGSOFT Software Engineering Notes 38*, 5 (2013), 40–42.

[104] GLASER, B. G. *Theoretical sensitivity: Advances in the methodology of grounded theory*. Sociology Press Mill Valley, 1978.

[105] GLASER, B. G. *Emergence vs forcing: Basics of grounded theory analysis*. Sociology Press, 1992.

[106] GLASER, B. G. *Doing grounded theory: Issues and discussions*, vol. 254. Sociology Press Mill Valley, 1998.

[107] GLASER, B. G. Conceptualization: On theory and theorizing using grounded theory. *International Journal of Qualitative Methods 1*, 2 (2002).

[108] GLASER, B. G. *The grounded theory perspective III: Theoretical coding*. Sociology Press, 2005.

[109] GLASER, B. G., AND HOLTON, J. Remodeling grounded theory. In *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*, vol. 5. 2004.

[110] GLASER, B. G., AND HOLTON, J. Basic social processes. *Grounded Theory Review 4*, 1 (2005), 1–27.

[111] GLASER, B. G., AND STRAUSS, A. L. *The discovery of Grounded Theory: strategies for qualitative research*. Aldine de Gruyter, 1967.

[112] GLASS, R. L. Agile versus traditional: Make love, not war! *Cutter IT Journal 14*, 12 (2001), 12–18.

[113] GUBA, E. G., LINCOLN, Y. S., ET AL. Competing paradigms in qualitative research. *Handbook of qualitative research 2* (1994), 163–194.

[114] HADAR, E., AND SILBERMAN, G. M. Agile architecture methodology: Long term strategy interleaved with short term tactics. In *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)* (Nashville, Tennessee, 2008), pp. 641–652.

[115] HEAGNEY, J. *Fundamentals of project management*. American Management Association, 2012.

[116] HIGHSMITH, J. A. *Adaptive software development: a collaborative approach to managing complex systems*. Dorset House, 2000.

[117] HIGHSMITH, J. A. *Agile software development ecosystems*, vol. 13. Addison-Wesley Professional, 2002.

[118] HIGHSMITH, J. A., AND COCKBURN, A. Agile software development: the business of innovation. *IEEE Computer 34*, 9 (Sept. 2001), 120–127.

[119] HODA, R. *Self-organizing agile teams: a grounded theory*. PhD thesis, Victoria University of Wellington, 2010.

[120] HODA, R., BABB, J., AND NORBJERG, J. Toward learning teams. *IEEE Software 30*, 4 (Jul.–Aug. 2013), 95–98.

[121] HODA, R., NOBLE, J., AND MARSHALL, S. Organizing self-organizing teams. In *International Conference on Software Engineering (ICSE '10)* (Cape Town, 2010), pp. 285–294.

[122] HODA, R., NOBLE, J., AND MARSHALL, S. Agile undercover: When customers don't collaborate. In *Agile Processes in Software Engineering and Extreme Programming (XP '10)* (Trondheim, Norway, 2010), Springer, pp. 73–87.

[123] HOLLANDER, T. The role of an architect in an agile team. In *Tech.Ed Australia* (Gold Coast, 2010). Retrieved 22 March 2014 from Tech.Ed: `http://channel9.msdn.com/Events/TechEd/Australia/2010/ARC204`.

[124] *IEEE Software 1*, 3 (Jul. 1984).

[125] ISHAM, M. Agile architecture is possible – you first have to believe! In *Agile Conference (AGILE '08)* (Toronto, 2008), pp. 484–489.

[126] ISO/IEC/IEEE. Recommended practice for architectural description of software-intensive systems, 2000. Retrieved 22 March 2014 from ISO/IEC/IEEE: `http://www.iso-architecture.org/ieee-1471/`.

[127] ISO/IEC/IEEE. Systems and software engineering – architecture description, 2011. Retrieved 22 March 2014 from ISO/IEC/IEEE: `http://www.iso-architecture.org/ieee-1471/`.

[128] JACOBS, D. *Accelerating process improvement using agile techniques.* CRC Press, 2005.

[129] JANSEN, A., AND BOSCH, J. Software architecture as a set of architectural design decisions. In *Working IEEE/IFIP Conference on Software*

Architecture (WICSA '05) (Pittsburgh, Pennsylvania, 2005), pp. 109–120.

[130] JEFFRIES, R. Essential XP: Card, conversation, confirmation, Aug. 2001. Retrieved 22 March 2014 from n.p.: `http://xprogramming.com/articles/expcardconversationconfirmation/`.

[131] KAVIS, M. Can you really be agile without an agile architecture?, May 2012. [Blog entry]. Retrieved 22 March 2014 from Kavis Technology Consulting: `http://www.kavistechnology.com/blog/can-you-really-be-agile-without-an-agile-architecture/`.

[132] KEELING, M. High ceremony software architecture kills collaboration, Feb. 2013. [Blog entry.] Retrieved 22 March 2014 from: `http://www.neverletdown.net/2013/02/high-ceremony-software-architecture-kills-collaboration.html`.

[133] KELLY, A. Agile demystified (v. 3), 2009. Retrieved 22 March 2014 from n.p.: `http://allankelly.net/static/writing/webonly/AgileDymistifiedV3.pdf`.

[134] KELLY, A. Waterfall 2.0: this time it's continuous flow, Jan. 2014. [Twitter Post]. Retrieved 22 March 2014 from Twitter: `https://twitter.com/allankellynet/status/425911045438603264`.

[135] KEMMIS, S., AND WILKINSON, M. Participatory action research and the study of practice. In *Action research in practice: Partnerships for social justice in education*. 1998, pp. 21–36.

[136] KNOERNSCHILD, K. Agile architecture, May 2010. [Blog entry.] Retrieved 22 March 2014 from: `http://techdistrict.kirkk.com/2009/05/06/agile-architecture/`.

[137] KRUCHTEN, P. The 4+1 view model of architecture. *IEEE Software 12*, 6 (Nov.–Dec. 1995), 42–50.

[138] KRUCHTEN, P. *The rational unified process: an introduction*, 3 ed. Addison-Wesley Professional, 2004.

[139] KRUCHTEN, P. Scaling down large projects to meet the agile sweet spot. *IBM developerWorks 13* (2004).

[140] KRUCHTEN, P. Voyage in the agile memeplex. *Queue 5*, 5 (Jul.–Aug. 2007), 38–44.

[141] KRUCHTEN, P. What do software architects really do? *Journal of Systems and Software 81*, 12 (2008), 2413–2416.

[142] KRUCHTEN, P. Agility and architecture: an oxymoron? *SAC 21 Workshop: Software Architecture Challenges in the 21st Century* (2009).

[143] KRUCHTEN, P. Complexity made simple. In *Proceedings of the Canadian Engineering Education Association* (2012).

[144] KRUCHTEN, P. Agile architecture, Dec. 2013. [Blog entry.] Retrieved 22 March 2014 from Kruchten Engineering Services Ltd: `http://philippe.kruchten.com/2013/12/11/agile-architecture/`.

[145] KRUCHTEN, P. Agility and architecture – a clash of two cultures? *Architecture* (2013).

[146] KRUCHTEN, P. Contextualizing agile software development. *Journal of Software: Evolution and Process 25*, 4 (2013), 351–361.

[147] KRUCHTEN, P. Software architecture and agile software development: a clash of two cultures? In *International Conference on Software Engineering (ICSE '10)* (Capetown, 2010), pp. 497–498.

[148] KRUCHTEN, P., OBBINK, H., AND STAFFORD, J. The past, present, and future for software architecture. *IEEE Software 23*, 2 (Mar.–Apr. 2006), 22–30.

[149] KVALE, S., AND BRINKMANN, S. *InterViews: learning the craft of qualitative research*. SAGE, 2009.

[150] LARMAN, C., AND VODDE, B. Lean primer, 2009. Retrieved 22 March 2014 from: `http://www.leanprimer.com/downloads/lean_primer.pdf`.

[151] LEHMAN, M. M. Programs, life cycles, and laws of software evolution. In *Proceedings of the IEEE* (Sept. 1980), vol. 68, pp. 1060–1076.

[152] LENCIONI, P. *The five dysfunctions of a team*. Wiley.com, 2006.

[153] LEVISON, M. What is sprint zero? Why was it introduced?, Sep. 2008. Retrieved 22 March 2014 from InfoQ: `http://www.infoq.com/news/2008/09/sprint_zero`.

[154] LINCOLN, Y. S., AND GUBA, E. G. But is it rigorous? Trustworthiness and authenticity in naturalistic evaluation. *New directions for program evaluation 1986*, 30 (1986), 73–84.

[155] LINDVALL, M., BASILI, V., BOEHM, B., COSTA, P., DANGLE, K., SHULL, F., TESORIERO, R., WILLIAMS, L., AND ZELKOWITZ, M. Empirical findings in agile methods. In *Extreme Programming and Agile Methods/Agile Universe 2002* (Chicago, Illinois, 2002), pp. 197–207.

[156] MADISON, J. Agile architecture interactions. *IEEE Software 27*, 2 (Mar.–Apr. 2010), 41–48.

[157] MAHONEY, M. S. Finding a history for software engineering. *Annals of the History of Computing, IEEE 26*, 1 (Jan.–Mar. 2004), 8–19.

[158] MALAN, R., AND BREDEMEYER, D. Less is more with minimalist architecture. *IT professional 4*, 5 (2002), 48–46.

[159] MAR, K., AND SCHWABER, K. Scrum with XP. *Informit.com* (2002).

[160] MARTIN, A. *A case study: exploring the role of customers on extreme programming projects*. PhD thesis, Victoria University of Wellington, Jan. 2003.

[161] MATTS, C., AND MAASSEN, O. "Real Options" Underlie Agile Practices, Jun. 2007. Retrieved 22 March 2014 from InfoQ: `http://www.infoq.com/articles/real-options-enhance-agility`.

[162] MCBREEN, P. Pretending to be agile, 2002. Retrieved 22 March 2014 from Informit: `http://www.informit.com/articles/article.asp?p=25913&seqNum=1`.

[163] MCBREEN, P., AND FOREWORD BY-BECK, K. *Questioning extreme programming*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[164] MCDERMID, J. Complexity: concept, causes and control. In *International Conference on Engineering of Complex Computer Systems (ICECCS '00)* (Tokyo, 2000), pp. 2–9.

[165] MCGOVERN, L. What is legacy code?, 2008. Retrieved 22 March 2014 from Flickspin Media Pty Ltd: `http://www.flickspin.com/en/software_development/what_is_legacy_code`.

[166] MCMAHON, P. E. Bridging agile and traditional development methods: A project management perspective. In *Systems and Software Technology Conference* (2004).

[167] MILES, M. B., AND HUBERMAN, A. M. *Qualitative data analysis: An expanded sourcebook*, 2 ed. SAGE, 1994.

[168] MINICHIELLO, V., ARONI, R., AND HAYS, T. *In-depth interviewing*, 3 ed. Pearson Education Australia, 2008.

[169] MIRAKHORLI, M., AND CLELAND-HUANG, J. Traversing the twin peaks. *IEEE Software 30*, 2 (2013), 30–36.

[170] MTHUPHA, B. *A framework for the development and measurement of agile enterprise architecture*. PhD thesis, Rhodes University, 2012.

[171] NATPRYCE. A new technical term: A 'Dirk Gently Architecture': everything in the code is connected to everything else in mysterious ways. Not good., Aug. 2012. [Twitter Post]. Retrieved 22 March 2014 from Twitter: `https://twitter.com/natpryce/status/238550198153015296`.

[172] NAUR, P., AND RANDELL, B., Eds. *Software Engineering: Report on a conference sponsored by the NATO Science Committee* (Oct. 1969), NATO.

[173] NERUR, S., MAHAPATRA, R., AND MANGALARAJ, G. Challenges of migrating to agile methodologies. *Communications of the ACM 48*, 5 (May 2005), 72–78.

[174] NORD, R., AND TOMAYKO, J. Software architecture-centric methods and agile development. *IEEE Software 23*, 2 (Mar.–Apr. 2006), 47–53.

[175] NORD, R. L., OZKAYA, I., AND SANGWAN, R. S. Making architecture visible to improve flow management in lean software development. *IEEE Software 29*, 5 (Sept.–Oct. 2012), 33–39.

[176] NUSEIBEH, B. Weaving together requirements and architectures. *IEEE Computer 34*, 3 (Mar. 2001), 115–119.

[177] OZKAYA, I., DIAZ-PACE, A., GURFINKEL, A., AND CHAKI, S. Using architecturally significant requirements for guiding system evolution. In *European Conference on Software Maintenance and Reengineering (CSMR '10)* (2010), pp. 127–136.

[178] PALMER, S. R., AND FELSING, M. *A practical guide to feature-driven development*. Pearson Education, 2001.

[179] PARNAS, D. L. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering 5*, 2 (Mar. 1979), 128 – 138.

[180] PARNAS, D. L., AND CLEMENTS, P. C. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering 12*, 2 (Feb. 1986), 251–257.

[181] PHILIPPUS, E. Architecture Spikes, Jul. 2009. Retrieved 22 March 2014 from Improvement BV: `http://www.improvement-services.nl/Articles/Architecture%20Spikes.pdf`.

[182] POORT, E. R., AND VAN VLIET, H. Architecting as a risk- and cost management discipline. In *Working IEEE/IFIP Conference on Software Architecture (WICSA '11)* (Boulder, Colorado, 2011), pp. 2–11.

[183] POPPENDIECK, M., AND CUSUMANO, M. A. Lean software development: A tutorial. *IEEE Software 29*, 5 (Sep.–Oct. 2012), 26–32.

[184] POPPENDIECK, M., AND POPPENDIECK, T. *Lean software development: an agile toolkit*. Addison-Wesley Professional, 2003.

[185] PROJECT MANAGEMENT INSTITUTE. *A guide to the Project Management Body of Knowledge (PMBOK Guide)*, 5 ed. Project Management Institute, 2013.

[186] PULEIO, M. How not to do agile testing. In *Agile Conference (AGILE '06)* (2006), IEEE.

[187] PUNCH, K. F. *Introduction to social research: Quantitative and qualitative approaches*. SAGE, 2005.

[188] QUMER, A., AND HENDERSON-SELLERS, B. Comparative evaluation of XP and Scrum using the 4D Analytical Tool (4-DAT). In *European and Mediterranean Conference on Information Systems* (2006).

[189] QUMER, A., AND HENDERSON-SELLERS, B. An evaluation of the degree of agility in six agile methods and its applicability for method engineering. *Information and Software Technology 50*, 4 (2008), 280 – 295.

[190] REEVES, J. W. What is software design? *C++ Journal* (1992). Retrieved 22 March 2014 from C++ Report: `http://www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm`.

[191] RENDELL, A. Descending from the architect's ivory tower. In *Agile Conference (AGILE '09).* (Chicago, Illinois, 2009), pp. 180–185.

[192] RIES, E. *The lean startup: how today's entrepreneurs use continuous innovation to create radically successful businesses*. Crown Publishing Group, 2011.

[193] ROBINSON, H., HALL, P., HOVENDEN, F., AND RACHEL, J. Post-modern software development. *The Computer Journal 41*, 6 (1998), 363–375.

[194] ROBSON, C. *Real world research: A resource for social scientists and practitioner-researchers*, vol. 2. Blackwell Oxford, 2002.

[195] ROYCE, W. W. Managing the development of large software systems. In *IEEE WESCON* (1970), vol. 26.

[196] RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. *The unified modeling language reference manual*, 2 ed. Pearson Higher Education, 2004.

[197] SAWYER, J. T., AND BRANN, D. M. How to build better models: applying agile techniques to simulation. In *Winter Simulation Conference (WSC '08)* (Solana Beach, California, 2008), pp. 655–662.

[198] SCHWABER, K. Scrum But replaced by Scrum And, Apr. 2012. [Blog entry.] Retrieved 22 March 2014 from: `http://kenschwaber.wordpress.com/2012/04/05/scrum-but-replaced-by-scrum-and/`.

[199] SCHWABER, K., AND BEEDLE, M. *Agile software development with Scrum*. Prentice-Hall, 2002.

[200] SCHWABER, K., AND SUTHERLAND, J. The Scrum Guide, Feb. 2010. Retrieved 22 March 2014 from n.p.: `http://www.scrumguides.org/`.

[201] SCRUM ALLIANCE. Scrum Alliance Core Scrum, 2012. Retrieved 22 March 2014 from Agile Atlas: `http://agileatlas.org/atlas/scrum`.

[202] SÉGUIN, N., TREMBLAY, G., AND BAGANE, H. Agile principles as software engineering principles: An analysis. In *Agile Processes in Software Engineering and Extreme Programming (XP '12)* (Malmö, Sweden, 2012), pp. 1–15.

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

[204] SHARP, H., AND ROBINSON, H. An ethnographic study of XP practice. *Empirical Software Engineering 9*, 4 (2004), 353–375.

[205] SHAW, M., AND GARLAN, D. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.

[206] SHEN, B., AND JU, D. On the measurement of agility in software process. In *Software process dynamics and agility*. Springer, 2007, pp. 25–36.

[207] SOFTWARE ENGINEERING INSTITUTE. Defining software architecture, n.d. Retrieved 22 March 2014 from SEI: `http://www.sei.cmu.edu/architecture/`.

[208] SOFTWARE ENGINEERING INSTITUTE. What is your definition of software architecture?, n.d. Retrieved 22 March 2014 from SEI: `http://www.sei.cmu.edu/architecture/start/glossary/definition-form.cfm`.

[209] SPINELLIS, D. Agility Drivers. *IEEE Software 28*, 4 (July–Aug. 2011), 96.

[210] STEPHENS, M. The case against extreme programming: A self-referential safety net, Aug. 2001. Retrieved 22 March 2014 from Software Reality: `http://www.softwarereality.com/lifecycle/xp/safety_net.jsp`.

[211] STERN, P. N. Eroding grounded theory. *Critical issues in qualitative inquiry* (1994), 212–223.

[212] STERN, P. N., AND PORR, C. *Essentials of accessible grounded theory*. Left Coast Press, 2011.

[213] STRAUSS, A. L., AND CORBIN, J. M. *Basics of qualitative research: techniques and procedures for developing grounded theory*, 2 ed. SAGE, 1998.

[214] SUDDABY, R. From the editors: What grounded theory is not. *Academy of management journal 49*, 4 (2006), 633–642.

[215] SUTHERLAND, J. Nokia test, 2009. Retrieved 22 March 2014 from n.p.: `http://jeffsutherland.com/nokiatest.pdf`.

[216] SUTHERLAND, J., DOWNEY, S., AND GRANVIK, B. Shock therapy: A bootstrap for hyper-productive Scrum. In *Agile Conference (AGILE '09)* (2009), pp. 69–73.

[217] TALBY, D., AND DUBINSKY, Y. Governance of an agile software project. In *ICSE Workshop on Software Development Governance (SDG '09)* (Vancouver, 2009), pp. 40–45.

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

[219] THOMAS, G., AND JAMES, D. Reinventing grounded theory: some questions about theory, ground and discovery. *British Educational Research Journal 32*, 6 (2006), 767–795.

[220] TOMAYKO, K., AND HERBSLEB, J. How useful is the metaphor component of agile methods? A preliminary study. Tech. rep., School of Computer Science Carnegie Mellon, Jun. 2003.

[221] TURK, D., FRANCE, R., AND RUMPE, B. Limitations of agile software processes. In *Extreme Programming and Agile Processes in Software Engineering (XP '02)* (Sardinia, 2002), pp. 43–46.

[222] URQUHART, C., LEHMANN, H., AND MYERS, M. D. Putting the 'theory' back into grounded theory: guidelines for grounded theory studies in information systems. *Information systems journal 20*, 4 (2010), 357–381.

[223] WALDO, J. On system design. *SIGPLAN Not. 41*, 10 (2006), 467–480.

[224] WATERMAN, M., NOBLE, J., AND ALLAN, G. How much architecture? Reducing the up-front effort. In *Agile India 2012* (Feb. 2012), pp. 56–59.

[225] WATERMAN, M., NOBLE, J., AND ALLAN, G. The effect of complexity and value on architecture planning in agile software development. In *Agile Processes in Software Engineering and Extreme Programming (XP '13)* (Vienna, 2013), pp. 238–252.

[226] WATERS, K. eXtreme programming versus Scrum, Apr. 2008. Retrieved 22 March 2014 from All About Agile: `http://www.allaboutagile.com/extreme-programming-versus-scrum/`.

[227] WEBER, R. Evaluating and developing theories in the information systems discipline. *Journal of the Association for Information Systems 13*, 1 (2012).

[228] WEYRAUCH, K. What are we arguing about? A framework for defining agile in our organization. In *Agile Conference (AGILE '06)* (Minneapolis, Minnesota, 2006), pp. 8–220.

[229] WHITWORTH, E., AND BIDDLE, R. The social nature of agile teams. In *Agile Conference (AGILE '07)* (Washington, D.C., 2007), pp. 26–36.

[230] WILLIAMS, L., AND COCKBURN, A. Agile software development: it's about feedback and change. *IEEE Computer 36*, 6 (Jun. 2003), 39–43.

[231] YAKYMA, A., AND LEFFINGWELL, D. The principles of agile architecture, 2013. Retrieved 22 March 2014 from Leffingwell LLC: `http://scaledagileframework.com/the-principles-of-agile-architecture/`.

[232] YIN, R. K. *Case study research: Design and methods*, 4 ed. Sage, 2009.

[233] ZIKMUND, W. G., BABIN, B. J., CARR, J. C., AND GRIFFIN, M. *Business research methods*, 8 ed. South-Western Cengage Learning, 2010.

Index