The Effect of Complexity and Value on Architecture Planning in Agile Software Development

Michael Waterman, James Noble, and George Allan

Victoria University of Wellington, New Zealand
{Michael.Waterman,kjx,George.Allan}@ecs.vuw.ac.nz

Abstract. A key feature of agile software development is its prioritisation of responding to changing requirements over planning ahead. If an agile development team spends too much time planning and designing architecture then responding to change will be extremely costly, while not doing enough architectural design puts the project at risk of failure. Striking the balance depends heavily on the context of the system being built, the environment and the development teams. This Grounded Theory research into how much architecture agile teams design up-front has identified system complexity as an important factor in determining how much planning a team does up-front, while system size, although related to complexity, has a much less direct impact. Furthermore, when determining how much design to do up-front, value to the customer can be a more important factor than overall development cost. Understanding these factors can help agile teams to determine how much up-front planning is appropriate for the systems they develop.

Keywords: Software architecture, agile software development, Grounded Theory.

1 Introduction

A software architecture represents the high-level structure and behaviour of a software system [1] and can be difficult to change once development has started [2]. Architecture is about planning ahead – getting the design of the system right and avoiding costly refactoring during development. On the other hand, one of the key features of agile software development is the ability to respond to changing requirements in preference to planning ahead [3]. There is therefore a tension between up-front architecture design and agile methods. Many agile teams deal with this tension through just enough up-front design to allow development to begin [4]. How much just enough is depends on the context, where context is made up of technical factors, environmental factors and the team itself.

This paper presents results from ongoing research that examines the relationships between complexity and size, value and cost, and the effects that they have on how much architectural planning teams do up-front. Understanding

H. Baumeister and B. Weber (Eds.): XP 2013, LNBIP 149, pp. 238–252, 2013.
© Springer-Verlag Berlin Heidelberg 2013

these factors can help agile teams to determine how much up-front planning is appropriate for the systems they develop.

Following this introduction, section 2 discusses the problem of architecture planning in agile development, section 3 describes the research methodology used (Grounded Theory), section 4 presents the findings of this research and section 5 discusses the results in context of the literature. Section 6 discusses the limitations of the research, and finally section 7 concludes the paper.

2 Background

2.1 The Tension between Agile and Architecture

There are many definitions of software architecture. Kruchten defined software architecture as "the set of significant decisions about the high level structure and the behaviour of the system" [1]; Booch extended this by noting 'significant' can be measured by the cost of change [2]. We can therefore summarise architecture as comprising the planning and design decisions that are made up-front and are difficult to change once development has started. Examples of architectural decisions are the choice of technology stack (including the development frameworks), architectural styles or patterns and the system's high level components.

Delivering *value* to the customer and other stakeholders lies at the heart of being agile; many of the twelve principles of the agile manifesto directly relate to delivering value earlier and faster [3]. Scrum and XP maximise value by prioritising tasks according to business priorities [5,6], and Lean places high importance on value streams and eliminating waste [7,8].

Agile methods focus on value through delivering software frequently, responding to changing and evolving requirements in preference to planning ahead, delivering quality, and simplicity [3]. Behind its ability to respond to changing and evolving requirements is the principle of 'the simplest thing that will work', or YAGNI – 'you ain't gonna need it': any additional work, such as developing features that *might* be required, will be wasted if those features never actually make it into the final product [6].

Architecture design is often seen as contrary to the philosophy of YAGNI, delivering little immediate value to the customer [9]. Agile developers therefore often avoid or minimise architectural planning [10], with the architecture being either neglected entirely or only implicitly defined. Too little architecture may lead to an *accidental architecture* [11] – one that has not been carefully thought through – and may lead to gradual failure of the project, while on the other hand too much architecture planning will at best delay the start of development, and at worst lead to expensive architectural rework if the requirements change significantly.

The agile principle of YAGNI is therefore in tension with architectural planning.

2.2 The Subjectivity of Up-Front Architecture Decision-Making

Agile methodology instructions generally advise developers to deal with the tension between agile and architecture by designing just enough architecture to start development, with the rest being completed during development as required [4,12,13]. How much is just enough depends heavily on context, with context depending on environmental factors such as the organisation and the domain, as well as specific factors such as project size, criticality, business model, architecture stability, team distribution, governance, rate of change and the age of system [9]. More than this however; context also includes social influences [14], such as the background and experience of the architects. Booch and Fairbanks noted that a particular system may have more than a single correct architecture [15,16], and two architects are likely to produce different architectures for the same problem with the same boundaries [14]. Taylor described architecture as being as much about 'soft' (subjective) factors as it is about objective design [17].

2.3 The Effect of Size on Up-Front Planning

Boehm undertook a study [18] using the COCOMO II model [19] in which he demonstrated a relationship between the level of up-front architectural effort and the overall development effort (and hence cost). This study showed that up-front architectural effort is a compromise between the amount of time spent planning up-front and the amount of time spent on rework caused by doing too little, with a 'sweet spot' at the overall minimum cost. 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 planning. This difference is due to the diseconomies of scale of software development [18].

Figure 1 shows that an increase in size from 100 KSLOC (thousand equivalent source lines of code) to 10,000 KSLOC increases the up-front planning sweet spot from around 20 per cent of the total effort to around 40 per cent of the total effort.

2.4 The Research Gap

There has been very little empirical research on the relationship between software architecture and agile development to date [20]. Breivold et al. performed a survey of the literature and concluded that studies have been small, diverging, and in some cases, performed in an artificial setting [20]. Dybå and Dingsøyr also noted the need for more knowledge of software development in general, particularly through empirical studies [21]. This lack of research does not mean that it is not an important issue: at the XP2010 conference, how much architectural effort was rated as the second-equal most burning question facing agile practitioners [22].

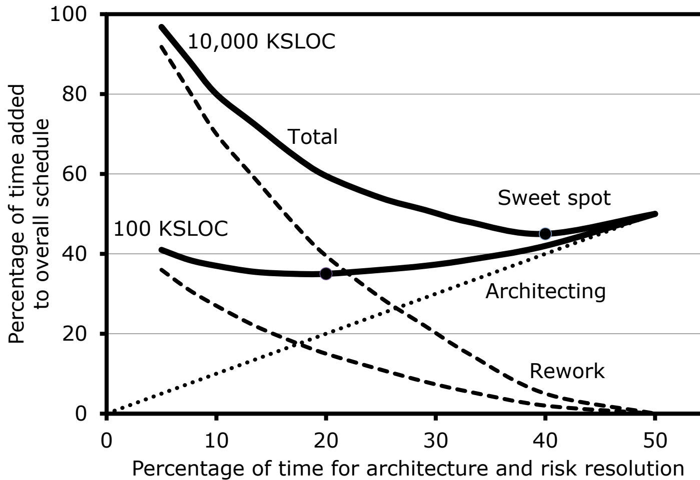

Fig. 1. The effect of system size on the up-front architecture sweet spot, from Boehm [18]

This paper presents results from ongoing research that helps address this gap by investigating the relationships between complexity and size, value and cost, and how they affect how much architectural planning teams do up-front. These results can be used to guide agile development teams when making decisions on how much up-front architecture design and planning is appropriate for systems they develop.

3 Research Method

This research into up-front architecture uses the qualitative Grounded Theory methodology [23]. Qualitative methods such as Grounded Theory are used to investigate people, interactions and processes. As noted above, architecture is very dependent on the architects themselves and the development teams. Qualitative research is generally *inductive* – it develops theory from the research, unlike *deductive* research which aims to prove (or disprove) a hypothesis or hypotheses. Because of the scarcity of literature on the relationship between architecture and agile methods [20], an inductive methodology that will develop a new hypothesis is more suitable for this research. We selected Grounded Theory because it is a systematic and rigorous method [24] that allows researchers to develop a *substantive* theory that explains the processes observed in a range of cases [25].

3.1 Data Collection

In this research, we collected data primarily through face-to-face semi-structured interviews with agile practitioners who design or use architecture, or who are

otherwise architecture stakeholders. Participants were typically architects, developers, project leaders/managers and customers, and are all involved with business-type applications. We collected additional data in the form of documentation and discussions by email and telephone to seek further information or clarifications on earlier interviews.

3.2 Data Analysis

The first step of data analysis, *open coding*, can begin as soon as the first data is obtained. In open coding, phenomena in the data are methodically identified and labelled using a code that summarises the meaning of the the data [26].

As open coding progresses, emerging codes are compared with earlier codes; codes with related themes are aggregated into higher levels of abstraction called *concepts*. This process, called *constant comparison* [27], continues at the concept level, with similar concepts being aggregated into a third level of abstraction called *categories*. Categories are the highest conceptual elements of Grounded Theory analysis; a Grounded Theory research project may have hundreds of different codes but will typically have no more than four or five categories [28]. The relationships between the categories are analysed and focused using *selective coding*; a dominant category emerges as the *core category*, which becomes central to the emerging theory. Throughout the analysis process, *memos* – free form notes ranging anywhere in size from a sentence to several pages – are written to record thoughts and ideas about developing relationships between codes, concepts and categories, and to aid the development of the theory [29].

Grounded Theory uses iteration to ensure a wide coverage of the factors that may affect the emerging theory [26]: later data collection is dependent on the results of earlier analysis. Data collection and analysis continue until *saturation* is reached, which occurs when no new insights are learned, and all variations and negative cases can be explained [30].

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

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

We coded this as 'having regulatory changes'.

Similarly another participant commented on the pharmaceutical regulations that affected his company's product:

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

We also coded this as 'having regulatory changes'.

Codes that had similar themes to this example included 'having changes in usage patterns or system load', 'requirements evolving' and 'understanding of requirements changing'. We combined these similar codes into a concept called 'having unstable requirements'. Figure 2 shows the relationship between the underlying codes and 'having unstable requirements'.

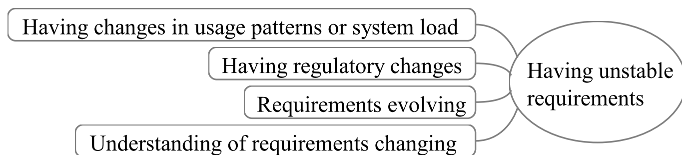

Fig. 2. An example of a concept emerging from its codes

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

We asked participants to select a project that they had been involved with to discuss during the interview. Types of projects varied hugely, from green fields to system redevelopment, from standalone systems to multi-team enterprise systems, and from start-up service providers and ongoing mass market product development to bespoke business systems. Systems varied from highly critical systems such as air traffic control and health record management, to business critical systems such as banking and retail, through to largely non-critical administration and entertainment broadcast systems. We also obtained documentation where possible to corroborate the interview data.

To maintain confidentiality, the participants are referred to using codes P1 to P32. A summary of participants and their projects are listed in table 1.

4 Findings

This paper presents findings on the effects that complexity and size have on up-front effort, and on using value rather than cost to determine how much effort a team should put into architectural planning. An earlier paper [29] captured the effects of architectural frameworks and templates, and the architects' experience, on the amount of up-front effort required in architectural design.

We used the Grounded Theory category "complexity over size" to form the basis for part of these results. This category consists of the concepts "indicators of complexity", "up-front effort affected by complexity of system" and "up-front effort affected by size of system" (figure 3a). The concept "indicators of complexity" in turn emerged from the codes "complexity leading to multiple frameworks", "frameworks reducing architectural complexity", "legacy systems

Table 1. Participant summary

	Role	Organisation type	Domain	Agile methods	Team size or no. of teams	Duration	System description
P1	Developer	Government agency	Health	Single developer	1 team member	6 months	Web-based, .NET
P2	Dev./ architect	Start-up	E-commerce	Scrum	3 team members	Ongoing	.NET, cloud-based
P3	Dev. manager	Vendor	Human resources	Scrum	3 teams	Ongoing	Web-based, .NET,
P4	Director of architecture	Government dept.	Digital archiving	Scrum	5 developers	Ongoing	Java, rich client, suite of standalone tools
P5	Coach/dev. manager	Start-up	Entertainment	Scrum/ kanban	Various	N/A	Various
P6	Man. Dir./ lead dev.	Vendor	Telecoms	Informal, iterative	1–3 developers	Ongoing	Suite of standalone applications
P7	BA	Telecoms operator	Telecoms	Scrum	12 team members	1 year+	Suite of web-based services
P8	Lead developer	Government dept.	Digital archiving	Scrum	4–14 team members	1 year+	Ruby on Rails, Java back-end
P9	Developer	Financial services	Telecoms	Bespoke	2–24 team members	3 years	Web-based system
P10	Coach	Multinat. hardware vendor	Transport	Scrum/XP	500–800 developers	Several years	Large distributed web-based system
P11	Architect	Government	Government services	Scrum	8 team members	Several years	Web-based services, .NET
P12	Senior developer	Service provider	Financial services	Scrum	6–7 developers	7 months	.NET, suite of web-based applications
P13	Architect	Government	Health	Scrum	12 team members	4 years	Monolithic .NET app
P14	Architect	Government	Animal health	Scrum	6–8 team members	18 months	.NET, large GIS component
P15	Customer	Start-up service provider	Retail (electricity)	Scrum	7 developers	Ongoing (3 years)	Ruby On Rails
P16	CEO/chief engineer	Start-up	Retail (health)	XP	5 team members	5 months	Ruby On Rails
P17	Manager/ coach	Government	Statistics	Scrum	6 dev + admin	2–3 years	Web-based, PHP using DAO pattern
P18	Dev. manager	Multinat. hardware vendor	Health	Scrum	15 team members	Ongoing (>2 years)	Web-based, Java platform
P19	Dev. manager	Start-up service provider	Retail (travel)	Lean	4 developers	Ongoing (<1 year)	PHP/Symfony, Javascript/Backbone
P20	Coach and trainer	Independent consultant	N/A	Scrum	N/A	N/A	N/A
P21	Manager/ coach	Service provider	Retail (publishing)	Scrum	3 teams; 40 total	Several years	.NET, Websphere Commerce, SAP, others.
P22	Senior manager	Service provider	Contact management/marketing	Scrum/XP	More than 40 total	N/A	.NET
P23	Senior manager	Vendor	Pharmaceutical	Own methods	3 teams	Ongoing	Various web based, client/server
P24	Customer	Start-up service provider	Retail (electricity)	Scrum	7 developers	Ongoing (3 years)	Ruby On Rails web applications
P25	Team lead	Service provider	Banking	Scrum	1 team	Ongoing	.NET, single tier web
P26	Team lead	Government	Water management	Scrum	8 team members	1 year	.NET, web based, 7 tier
P27	CEO/coach	Start-up service provider	Retail (electricity)	Scrum	7 developers	Ongoing (3 years)	Ruby On Rails
P28	Technical lead	Service provider	Broadcasting	Scrum	42 team members	N/A	Python with Django, CMSs for multiple websites
P29	Dev. manager	Banking	Banking	Kanban	20 team members	Ongoing	Web based, AJAX, interface to mainframe
P30	Consulting architect	Service provider	Telecoms	Scrum	7 team members	2 years+	Python with Django and Twisted, NoSQL
P31	Enterprise architect	Government	Transport	Bespoke	7 team members	13 week pilot	Web services, SOA using .NET/WCF
P32	Software dev. director	Vendor	Government	FDD, kanban	N/A	N/A	N/A

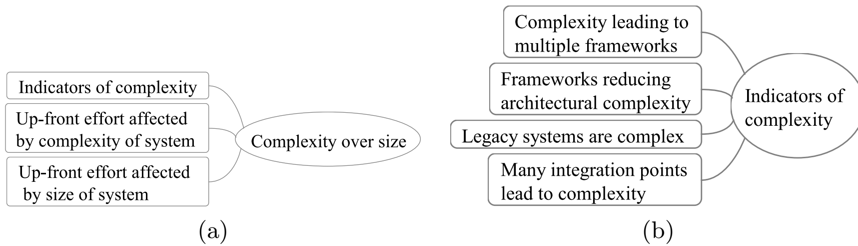

Fig. 3. (a) the category "complexity over size" emerging from its concepts; (b) the concept "indicators of complexity" emerging from its codes.

are complex" and "many integration points lead to complexity" (figure 3b). The other concepts likewise emerged from their respective codes; they are not listed here for the sake of brevity.

4.1 The Effect of Complexity on Up-Front Architecture Effort

Participants reported that the complexity of a system is an important determinant of how much architecture planning a team does up-front. System complexity is caused by demanding requirements and quality attributes, and results in design decisions that are intertwined and have multiple dependencies [31]. Complexity may extend or break the limits of the development frameworks being used, and therefore increases the decision-making effort required to select appropriate frameworks and tools, and to design a suitable architecture. A less complex system will require less effort and a less sophisticated design.

For example, a participant in this research described how complexity affects up-front architectural effort in his work:

> *"Typically, [the length of the start-up phase] depends on the complexity." [...] "The project that I am working on now is not a very complex architecture. For that I don't think we need a visual modelling tool – just a simple whiteboard or flipchart with coloured Post-Its can work." (P21, manager/agile coach)*

Participants noted that complexity can often be indicated by the need to use bespoke components and libraries, by the need to use multiple technologies, having many integration points, and by having to work with legacy systems. These are explored below.

Bespoke Components: Modern vendor frameworks such as .NET, Hibernate and Ruby on Rails provide standard solutions to problems, reducing the up-front effort required to build a system, and enabling architectural changes to be made with a lot less effort [29]. Frameworks are used by development teams to greatly reduce the amount of up-front architecture and development effort required, particularly in business-type applications, with participants commenting that they did not need to make as many architectural decisions when using modern frameworks:

> *"You choose the proper plug-ins and then you get the functionality that you are looking for." (P16, CEO)*

What used to be considered architectural decisions ten years ago are now sometimes considered design decisions, or even simply configuration decisions:

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

Frameworks, however, cannot always provide a complete solution. There are frequently parts of systems that cannot be implemented using components or libraries from the frameworks and have to be designed and developed from scratch. These may be because the problem is unique or because the framework components do not meet non-functional requirements such as performance. Non-framework components increase complexity and result in extra up-front effort as teams 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 bespoke replacements:

> *"There were a number of architectural things that were developed in-house. [...] We wrote our own data binding framework for instance. [...] We did a bit of prototyping, we built the data binding framework that we came up as a result of all of those factors. We had a bit of a go with what Microsoft had off the shelf previously, found it painful and limiting, and felt that it confirmed our decision to go our own way with data binding." (P13, architect)*

Multiple Technologies: Like the need for bespoke components, a system with complex requirements may not be able to be implemented entirely using a single vendor framework, and instead may require multiple frameworks to implement the required features and functionality. Not only does selecting these frameworks require extra up-front 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)*

Legacy Systems: Legacy systems are older systems that were created using outdated techniques and technology [32], and are no longer being 'engineered' but rather are simply patched as requirements change [33] without consideration of the technical debt being incurred [9]. These patches add to the system's complexity [34]:

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

Good engineering practices such as simplicity, modularity and high cohesion are eroded, and continuing to develop, or even interfacing with, these entropic legacy

systems 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. 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.

> *"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)*

4.2 The Effect of Size on Up-Front Architecture Effort

The size of a system is frequently considered by the literature as a factor in determining how much up-front architectural effort is required [9,18] (section 2.3). Contrarily, the participants in this research reported that size is not as important as complexity:

> *"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)*

Size may be measured explicitly by using a metric such as lines of code or number of components, or implicitly using a metric such as the project's budget or development time required. Size typically has some correlation with complexity: a small system is usually not very complex, and a large system has the potential for a high level of complexity. The relationship is not linear however; sometimes there may only be a small correlation.

A system, independent of size, may not have any of the sources of complexity described above – bespoke components, integration, multiple technologies and legacy systems – and hence will have a low level of complexity, and will require less up-front architectural effort:

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

Specifically, a large system that can be implemented entirely using the components and libraries of a framework with an acceptable level of risk is often not very complex, and will require less up-front effort than a similar sized complex system. For example, P27's team was building a large system that had complex requirements and complex functionality, but the team was able to decouple this complexity from the architecture through implementing the system entirely within the boundaries of Ruby on Rails. They were therefore able to build the system with very little up-front planning:

"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 up-front planning if it is complex:

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

The use of frameworks to avoid complexity allowed some participants (such as P27) to completely avoid up-front planning, allowing them to increase their agility and respond to change and deliver early value much more effectively.

4.3 Using Value and Cost Minimisation to Determine How Much Up-Front Architectural Effort

Section 2.3 above described Boehm's analysis which uses the minimum effort (and hence cost) to determine the sweet spot of architectural effort. While "cost is always a concern" (P10) in agile development, in some situations agile teams are more concerned about delivering business value to their customer – at the expense of cost.

For many businesses, particularly those building a new commercial mass market product or service, the value of software is the economic value that it adds to the business, and is measured by participants in this research in terms of cash flow or net present value, with early value being provided by early adopters of the service. When faced with a decision of either doing more architectural planning up-front (and delaying the release) or minimising architectural planning (and releasing as early as possible with less functionality), the teams consider which option will provide the most value to the business:

"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)

An early release may be in the form of a Minimum Viable Product [35], which is a marketing experiment – a release with limited functionality designed to determine which features are desirable, rather than a fully functional version of the software.

Focusing on business value and minimising the up-front effort may lead to the need to re-design the architecture later and increase the overall cost. By this stage the customer is in a better position to pay for that rework:

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

and

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

Agile teams must therefore consider value, and not just cost, as a measure for determining the level of up-front architectural planning the team does.

5 Discussion

This study of agile practitioners has found that the complexity of a system is an important factor in determining how much up-front architecture planning is required. Indicators of complexity include bespoke components, multiple technologies, integration with other systems, and dealing with legacy systems. On the other hand, system size by itself is not a good determinant of up-front effort, a result that is at odds with Boehm's analysis [18], described above in section 2.3. That analysis presented a clear relationship between the up-front architectural effort sweet spot and system size (see figure 1), due to the diseconomies of scale of software development. Boehm's analysis is based on data derived from the COCOMO II cost model, a model released in 1996 that calculates the cost of development of software systems using a complex regression algorithm and historical parameters, calibrated with the experiences obtained from a set of 161 software projects [18].

There may be a number of reasons for the difference between Boehm's result and these findings.

Boehm's analysis did not distinguish between complexity and size. The data that COCOMO II is based on is likely to be from projects from the mid 1990s or earlier. COCOMO II therefore predates modern frameworks that developers currently use to reduce complexity and up-front effort. When considering systems that do not use modern frameworks, there may be a good correlation between size and complexity which allows size to be used as a proxy for complexity.

The importance of the effect of early delivery on value was noted by Boehm in the context of software economics [36]: "The primary value realized may not be in cost avoidance but rather in reduced time to market." COCOMO II, a cost model, does not consider the benefit gained from early delivery of value, simply using cost to determine the sweet spot. Poort and van Vliet similarly proposed a method of determining the level of up-front architecture which is based on

prioritising risk and minimising cost [37]. They claimed that stakeholder value is implicit in the presence of the solution's goals and business requirements. However this assumption is not appropriate for agile development, because firstly it assumes that the solution's goals and business requirements do not change after development has started, and secondly it assumes that there is no value in delivering functionality to the end user before development of the entire system has been completed.

Boehm and Turner presented an earlier comparative model similar to figure 1 which had risk exposure (rather than effort) as the dependent (y-axis) variable [38]. Risk exposure included the business risk of delay caused by spending too much time on architectural planning, and is therefore more appropriate for agile development. Higher levels of risk exposure caused by higher levels of up-front planning would cause the sweet spot to move towards the less-planning end of the scale. Poort and van Vliet did not consider business risk [37].

Abrahamsson, Babar and Kruchten [9] listed a number of factors that they suggested can affect the level of up-front planning, including the rate of change of requirements, governance, team distribution, stability of the architecture and the business model. These factors are not discussed in this paper.

6 Limitations

A substantive Grounded Theory is only applicable to the domain being studied [30], and therefore cannot be assumed to be applicable to other contexts, or in general. The result is therefore, to some extent, dependent on the participants selected for the research. For example, these results cannot be applied to embedded software because we did not include any participants who develop embedded software systems.

7 Conclusion

This paper considers the relationships between complexity and size, value and cost, and how they affect how much up-front planning agile teams do.

Previous analysis undertaken by Boehm using the COCOMO II model presents a clear relationship between size and up-front architecture planning. However, results from this research show that the relationship between system complexity and up-front planning is more important than the relationship between size and up-front planning. Complexity, caused by demanding requirements and quality attributes, greatly increases the up-front planning required, and may be indicated by the need to build bespoke components, which are required where the framework does not provide the functionality needed or cannot meet non-functional requirements such as performance, by the need for multiple frameworks, by the need for many integration points with other systems, and by the need to work with legacy systems. While complexity is closely related to size, size in itself does not always directly affect the amount of up-front planning, particularly if the system has a low level of complexity.

The need for architecture planning is in tension with agile's need to respond to changing requirements: too much planning results not only in unnecessary effort but also wasted effort if requirements change, while too little planning leads to more effort to address architectural problems that arise. Therefore the minimum overall cost is sometimes used to determine how much up-front planning a team should do. However, in agile development, the need to provide early value to the customer may override the need to minimise overall cost, if early value will lead to an improved cash flow for the customer. To provide early value, the team may do less up-front planning, potentially with more architectural rework later when the customer's cash flow is more able to support that architectural effort.

Agile teams must consider complexity and value when determining how much architectural design to do up-front. Further results from this research will explore other factors that influence how much up-front design is required.

References

1. Kruchten, P.: The Rational Unified process – an Introduction. Addison Wesley (1998)
2. Booch, G.: Architectural organizational patterns. IEEE Software 25(03), 18–19 (2008)
3. Beck, K., et al.: Agile manifesto (2001), http://agilemanifesto.org/
4. Ambler, S.W.: Agile architecture: Strategies for scaling agile development, http://www.agilemodeling.com/essays/agileArchitecture.html
5. Deemer, P., Benefield, G., Larman, C., Vodde, B.: The scrum primer (2010), http://assets.scrumtraininginstitute.com/downloads/1/scrumprimer121.pdf
6. Beck, K.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley Professional (2005)
7. Poppendieck, M., Poppendieck, T.: Lean Software Development: An Agile Toolkit. Addison-Wesley Professional (2003)
8. Coplien, J.O., Bjørnvig, G.: Lean Architecture for Agile Software Development. John Wiley and Sons, Ltd. (2010)
9. Abrahamsson, P., Babar, M.A., Kruchten, P.: Agility and architecture: Can they coexist? IEEE Software 27(02) (2010)
10. Kruchten, P.: Agility and architecture: an oxymoron? In: SAC 21 Workshop: Software Architecture Challenges in the 21st Century (2009)
11. Booch, G.: The accidental architecture. IEEE Software 23(03), 9–11 (2006)
12. Booch, G.: An architectural oxymoron. IEEE Software 27(05), 96 (2010)
13. Avram, A.: 10 suggestions for the architect of an agile team (September 2010), http://www.infoq.com/news/2010/09/Tips-Architect-Agile-Team
14. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. SEI Series in Software Engineering. Addison-Wesley (2003)
15. Booch, G.: The irrelevance of architecture. IEEE Software 24(03), 10–11 (2007)
16. Fairbanks, G.: Just Enough Software Architecture: A Risk Driven Approach. Marshall and Brainerd (2010)
17. Taylor, P.R.: The Situated Software Architect. PhD thesis, Monash University (December 2007)
18. Boehm, B.: Architecting: How much and when? In: Oram, A., Wilson, G. (eds.) Making Software. O'Reilly (2011)

19. Boehm, B.W., Clark, Horowitz, Brown, Reifer, Chulani, Madachy, R., Steece, B.: Software Cost Estimation with COCOMO II with CD-Rom, 1st edn. Prentice Hall PTR, Upper Saddle River (2000)
20. Breivold, H.P., Sundmark, D., Wallin, P., Larson, S.: What does research say about agile and architecture? In: Fifth International Conference on Software Engineering Advances (2010)
21. Dybå, T., Dingsøyr, T.: What Do We Know about Agile Software Development? IEEE Software 26(05), 6–9 (2009)
22. Freudenberg, S., Sharp, H.: The top 10 burning research questions from practitioners. IEEE Software 27(05), 8–9 (2010)
23. Glaser, B.G., Strauss, A.L.: The Discovery of Grounded Theory: Strategies for Qualitative Research. Aldine de Gruyter (1967)
24. Allan, G.: The legitimacy of Grounded Theory. In: European Conference on Research Methods (keynote address) (July 2006)
25. Strauss, A., Corbin, J.: Grounded theory methodology. In: Denzin, N.K., Lincoln, Y.S. (eds.) Handbook of Qualitative Research. Sage Publications, Inc. (1994)
26. Allan, G.: A critique of using grounded theory as a research method. Electronic Journal of Business Research Methods 2 (July 2003)
27. Bryman, A.: Social Research Methods, 3rd edn. Oxford University Press (2008)
28. Glaser, B.G.: The grounded theory perspective III: Theoretical coding. Sociology Press (2005)
29. Waterman, M., Noble, J., Allan, G.: How much architecture? Reducing the up-front effort. In: Agile India 2012, pp. 56–59 (February 2012)
30. Charmaz, K.: Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis. SAGE Publications Ltd. (2006)
31. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: WICSA 2005, pp. 109–120 (2005)
32. Bennett, K.: Legacy systems: Coping with stress. IEEE Software 12(01), 19–23 (1995)
33. McGovern, L.: What is legacy code? (2008), http://www.flickspin.com/en/software_development/what_is_legacy_code
34. Lehman, M.: Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 68, 1060–1076 (1980)
35. Ries, E.: The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses. Crown Publishing Group (2011)
36. Boehm, B., Sullivan, K.: Software economics: a roadmap. In: Proceedings of the Conference on The Future of Software Engineering, ICSE 2000, pp. 321–343. ACM, New York (2000)
37. Poort, E.R., van Vliet, H.: Architecting as a risk- and cost management discipline. In: WICSA 2011, pp. 2–11 (2011)
38. Boehm, B.: Get ready for agile methods, with care. IEEE Computer 35(01), 64–69 (2002)