A Taxonomy of Dependencies in Agile Software Development

Diane E. Strode
Sid L. Huff
School of Information Management
Victoria University of Wellington
Wellington, New Zealand
Email: diane.strode@alumni.unimelb.edu.au
Email: sid.huff@vuw.ac.nz

Abstract

Dependencies in a software project can contribute to unsatisfactory progress if they constrain or block the flow of work. Various studies highlight the importance of dependencies in the organisation of work; however dependencies in agile software development projects have not previously been a research focus. Drawing on three case studies of agile software projects, and the IS literature, this paper develops an initial taxonomy of agile software project dependencies. Three distinct categories of dependency are found: task, resource, and knowledge dependencies. This paper contributes to theory by providing a taxonomy of dependency types occurring in the area of agile software development. Practitioners can use this taxonomy as sensitising device to ensure they consider dependencies they might face that could hinder their projects, enabling them to take appropriate and timely mitigating action.

Keywords

Agile software development, Dependency analysis, Dependency Taxonomy, Software project dependencies.

INTRODUCTION

Dependencies in a software project can contribute to unsatisfactory progress if they constrain or block the flow of work. Unsatisfactory progress can, in turn, contribute to late delivery of software, or even project failure. In this paper we propose that constraints in software projects can be attributed to dependencies that are not managed or organised appropriately. Furthermore, we define dependency as a situation that occurs when the progress of one action relies upon the timely output of a previous action or the presence of a specific thing. Various ISD (information systems development) reference literatures highlight the importance of dependencies, including organisation studies, IS project management, and software engineering. However, dependencies occurring in co-located agile software development projects have not previously been studied.

Agile software development came to prominence in the late 1990s and is now well-accepted (West and Grant 2010). Agile software development is achieved by adopting one or more agile methods such as Scrum or Extreme Programming (Dyba and Dingsoyr 2008). These methods differ significantly from traditional approaches to software development because they involve short iterations of development, close customer and team interaction to achieve shared understanding, working software as a focus of communication, and an acceptance of change within the development project, as opposed to attempting to control and reduce change (Beck et al. 2001). Another notable characteristic of this class of system development methodologies is their focus on work practices such as pair programming, daily stand-up meetings, and using story wallboards, rather than modelling methods or documentation.

Software development projects typically evolve in an environment of multiple dependencies (Grinter 1996; Wagstrom and Herbsleb 2006). It is therefore likely that projects using an agile approach also involve dependencies. The same types of dependency may occur in agile and non-agile projects, or they may differ in nature and relative importance. In this study the focus is on understanding the nature of dependencies in typical agile software development projects. Therefore, this paper addresses the research question: *What dependencies occur in agile software development projects?*

The paper is organised as follows. Literature on dependencies in work practice is briefly reviewed to determine how dependencies are defined, and why previous research does not adequately explain dependencies in agile projects. The multi-case study research method used to address the research question is described, including the rationale for selecting cases, and the data collection and analysis procedures. Findings are presented in the form of a taxonomy supported with evidence from the cases and the empirical research literature. The paper discusses the contributions of this taxonomy to ISD theory, and its practical utility. Then the paper concludes and discusses how the taxonomy can be used in future work.

THE CONCEPT OF DEPENDENCY

Dependencies are considered important in various domains relevant to software development. Spanning these domains is an interdisciplinary theory of coordination developed by Malone and Crowston (1994). Their Coordination Theory focuses on dependency as a fundamental element in coordination. Malone and Crowston's Coordination Theory is based on the tenet that "*coordination is the managing of dependencies between activities*" (Malone and Crowston 1994, p. 90). Later, Malone et al. (1999) proposed that a dependency belongs to one of three fundamental types: fit, flow, or sharing. In this conceptualisation, resources and activities interact to form dependencies. A fit dependency occurs when multiple activities produce a single resource. A flow dependency occurs when one activity produces a resource used by another activity, and a sharing dependency occurs when two or more activities use a single resource.

Pooled, sequential, reciprocal, and team interdependencies have been a focus in organisation studies. As interdependency can be conceived of as two distinct dependencies; in organisation studies the two terms are treated as equivalent. Staudenmeyer (1997) synthesised the research on interdependencies in this field and found that most theorists adopt Thompson's (1967) basic definitions. Thompson conceived four types of interdependency related to workflow: pooled, sequential, reciprocal, and team. In pooled workflow, work enters a work unit and actors perform work activities independently; work does not flow between them. Sequential workflow occurs when work enters a unit, passes between actors in a single direction and then passes out of the work unit. In the case of reciprocal workflow, work enters the work unit, and passes back and forth between actors. In team workflow, work enters the unit and actors diagnose, problem-solve, and collaborate as a group, working concurrently to deal with the work.

IS project management, and project management in general, focuses on dependencies between tasks, using this information to reduce project timeframes and/or costs. In planning a project, tasks and their dependencies are identified so that tasks can be carried out in a sequence that minimises project delays. Furthermore, tasks that are independent of other tasks can be carried out simultaneously. Traditional project planning methods such as PERT and Critical Path Modelling use this idea (Kerzner 2003).

Software engineering is concerned with the development of large software systems. Consequently, studies of dependencies in this domain involve large-scale projects, many of which are distributed in nature. Dependencies in software development projects, and coordination mechanisms for managing them, were the focus of Grinter's (1996) work. She noted that "*developers must manage a cadre of dependencies simultaneously if they are to build any working systems at all*" (Grinter 1996, p. 50). In her study of three software development organisations, she found the following dependencies: vendor, customer, lifecycle, "big picture", testing, parallel development, change, expertise, integration, historical, configuration management, and interface. Dependencies in globally distributed software development were investigated by Espinosa, Slaughter, Kraut, & Herbsleb (2007) within a single European telecommunications firm. They found technical, process, and temporal dependencies were the main types of dependency.

Each of these research domains acknowledges the importance of dependencies in organisational and project work arrangements, but each views dependency from a different perspective. Coordination theory is a general analytical framework and does not discuss dependencies in any particular domain. Dependency categories from organisation theory were not considered appropriate for our study because they focus on routine work, as opposed to non-routine, time-bounded project work. Although the team workflow arrangement appears to coincide with how work is conducted in agile projects, this conceptualisation does not provide for detailed insight into the agile project context. IS project management focuses on instances of tasks rather than generic dependencies occurring in projects; therefore it offers no means of exploring or explaining dependencies in agile software development projects. Finally, in software engineering the focus is on large, distributed projects. Dependencies in these contexts are likely to be of a different nature to those of small and co-located projects, which are the environments where agile methods are normally adopted. For these reasons - the importance of dependencies in managing and organising work, and the lack of a suitable framework in the existing literature to explain dependency in agile software development projects - we chose to carry out an in-depth exploratory study of dependencies in this context.

RESEARCH DESIGN

A multi-case study research approach was selected to explore the dependencies in ongoing software development projects. This approach is a well-accepted way to investigate phenomena in natural system development contexts where events cannot be controlled and where it is important to capture the detail in a situation (Eisenhardt and Graebner 2007; Pare 2004; Yin 2003). To ensure accepted standards for validity and reliability, we followed guidelines developed by Dubé and Paré (2003) for carrying out rigorous exploratory positivist case study research in the field of information systems.

Note that the findings reported in this paper form part of a larger study reported in Strode, Huff, Hope and Link (2012).

Case selection followed a replication strategy, a tactic recommended in case study design (Miles and Huberman 1994; Yin 2003) which involves selecting cases that are similar, and therefore likely to provide similar results (literal replications), or selecting cases that are dissimilar and therefore likely to provide contrasting results for predictable reasons (theoretical replications). The study consisted of three literal replications. Each project was a typical case selected because it was expected to show normal or average characteristics (Eisenhardt and Graebner 2007; Pare 2004). Case sites used the agile method Scrum, or Scrum with some additional practices from Extreme Programming, with a team size between 2 and 10. Ten is the maximum team size recommended for effective use of these methods (Beck 2000; Schwaber and Beedle 2002). All projects were required to have co-located teams. For practical reasons all projects were located in Wellington, New Zealand. No greenfield projects were located (i.e. a project without constraints imposed by prior work), therefore each project involved the replacement or extensive upgrading of an existing computerised system. Eisenhardt and Graebner (2007) recommend selecting cases that vary in some dimensions because this supports the development of theoretical concepts accounting for a wider range of contexts. Therefore projects were selected from different types of organisation using different types of development contract. Table 1 provides details concerning the three selected projects, which were code-named Land, Storm, and Silver.

Table 1. Organisation, project, and data collection summary

Case name	Land	Storm	Silver
Organisation type	Government	Commercial service provider	Commercial software development firm
Organisation size	2000 in NZ	200 in Australasia, Asia and Europe	20 in NZ
Project purpose	To improve the organisations interactions with the public	To migrate a critical legacy system to a modern technology platform	To provide a replacement reporting system for an external client
Contractual basis	In-house development	Independent contractors working on the client site	Development for external client
Development methodology	Scrum	Scrum and XP	Scrum
Team size	6	10	5
Interviews	2	5	4
Roles of interviewees	1 Project manager 1 Software developer	1 Project manager 2 Software developers 1 Tester 1 Domain expert	1 Development manager 1 Scrum coach 2 Software developers

The primary data collection method was semi-structured interview. An interview schedule appropriate for collecting data on dependencies in software development projects was developed based on suggestions by Crowston (1991) for identifying coordination mechanisms and dependencies, McChesney and Gallagher (2004) for software project profiles, and Spradley (1979) for interviewing in natural situations. In each project, the first interview was conducted with a senior team member who could provide details on the project, its purpose, background, history, and issues. Following this interview, and depending on team size, up to four other project team members in a variety of roles were interviewed for 1 to 1½ hours (see Table 1 for roles).

Data analysis was a two-step process. The first step was to prepare a full description of each case using a pre-defined framework. This description included details of the organisation, the project, the technologies, the team, the development method, and any problems in the project. The description was sent to one project participant for verification, and any factual errors found in the description were then corrected. The second step was within-case analysis. A general inductive coding approach following the guidelines of Miles and Huberman (1994) and Thomas (2006) was used. Heuristics provided by Crowston and Osborne (2003) for analysing dependencies and coordination mechanisms in a situation were followed:

- Dependency-focused analysis. Identify dependencies, and then search for coordination mechanisms. In other words, look for dependencies and then ask which activities manage those dependencies. Failure to find such activities might suggest potentially problematic unmanaged dependencies.

- Activity-focused analysis. Identify coordination mechanisms, and then search for dependencies. In other words, identify activities in the process that appear to be coordination activities, and then ask what dependencies those activities manage. This approach asks directly whether all observed coordination activities are necessary. (Crowston and Osborn 2003, p. 352)

Each case was analysed with dependency-focused and activity-focused analysis. Once instances of dependencies were identified they were grouped into similar types and given a unique code name (e.g., Expertise). In performing analyses, codes identified in the first case were used as starter codes for the second case, and codes from the second case acted as starter codes for the third case. When new types of dependency emerged during analyses they were added to the list of codes. In a final step, dependency types were grouped into higher-level categories based on their commonalities. Thus there were three levels of analysis: instances of dependency, types of dependency, and high-level dependencies.

DEPENDENCY TAXONOMY

We present our findings in the form of a taxonomy of dependencies. In organisation studies, taxonomies are "*classification systems that categorize phenomena into mutually exclusive and exhaustive sets with a series of discrete decision rules*" (Doty and Glick 1994, p. 232). In IS, Gregor (2006) classifies taxonomies as analytic theories defining 'what is' in a bounded situation. Such theories describe, analyse and summarise "*salient attributes of phenomena and relationships among phenomena. The relationships specified are classificatory, compositional, or associative, not explicitly causal* (Gregor 2006, p. 623).

The following sections of this paper develop the dependency taxonomy. Each dependency type is defined along with supporting evidence from the cases. When dependencies could conceivably be categorised into more than one dependency type, then decision rules are developed to aid in classification. These rules are described at the end of this section. Note that Williams (2010) provides a useful resource describing the details of the agile practices occurring in the examples below.

Requirement Dependencies

A requirement dependency is a situation where domain knowledge or a requirement is not known and must be located or identified, and this affects progress. Requirements are a critical input to software development projects as they define the basic needs the software under development should address. In agile software development projects, requirements are acquired, elaborated, and implemented in small increments as development progresses. This is usually achieved by interaction between the team and suitable knowledgeable stakeholders (i.e., end-users, clients, customers, or their proxies). Requirement dependencies were found in each case. That is, each project presented a situation where a requirement was not known, and this lack of knowledge affected or had the potential to affect, project progress. This situation occurred when information from suitable stakeholders was not readily available either during iteration planning, or during development. For example, Silver had problems communicating with their client, and the project team was not always able to get required requirements information in a timely manner (pertinent sections of quotes are underlined): *"It sometimes changed our priorities…so we have sometimes, once or twice, taken stories out of the sprint because we <u>couldn't get the information</u> and sometimes it was relatively high … priority stories, that we could not implement in a subsequent sprint…because they did not come back to us."* [Silver, Development Manager]. In the most complex project, Storm, involving the replacement of a critical and complex mainframe-based system, it was decided to integrate a domain specialist into the team. This specialist was located with the team in the same room and so that he could be readily consulted on requirements: "*We have a person on our team, who is one of the gurus, and he sits with us, and <u>we annoy him constantly to get stuff</u> [i.e. information on requirements] like that"*. [Storm, Developer]. Prior research by Crowston and Kammerer (1998) into problems in the requirements development process in large-scale software projects has also identified requirement dependencies.

Expertise Dependencies

Historically, expertise has been recognised an important factor influencing the success of software projects (Brooks 1995; Curtis et al. 1988; Faraj and Sproull 2000), it is also acknowledged in agile software development (e.g. (Beck 2000; Cockburn 2002). Expertise dependencies were found in all of the case projects. An expertise dependency is defined as a situation where technical or task information is known only by a particular person or group, and its absence affects project progress. Expertise dependencies tended to show *how* expertise was acquired, and this was taken as a reflection of the importance of gaining and sharing expertise among the team, as without such expertise the project might not have progressed so smoothly.

In Silver, for example, the project team as a whole developed expertise about how the software system worked by testing the current system version just prior to its presentation to their client: "*...we had…a day and a half* *when development would stop…, and everyone would test the thing to destruction effectively, and the knowledge of how it worked at that point would be picked up*." [Silver, Development Manager]. In Storm, expertise was required for many parts of this complex project. The project team were all in a single project room, and were able to acquire their needed expertise by asking anyone in the room, or ensuring expertise was shared by calling an informal meeting: *"…we can just yell out over our shoulder and grab someone, and also if they come to a complex design decision then they should bring all of the team to the table so that everyone hears the design decision that is made."* [Storm, Developer]. In Land, a small project consisting of a project team of three developers and three business people, an understanding of who had what expertise was gained at weekly sprint planning meetings: *"… in terms of who to ask within the business specific questions,…it was in the weekly meetings, I figured out that Brian was doing this, Mary was doing this, and the other Mary had a different role entirely."* [Land, Developer]. In Land, there were traditional role divisions in the team (analyst, developer, designer) with very little shared expertise. Team members tended to pass tasks over to those with the needed expertise: *"Sam waited. He would do the first line of testing. So he would always do a initial user acceptance testing to verify that everything was OK. Graeme would wait for me to complete the work so he could apply his styling."* [Land, Project Manager].

Faraj and Sproull (2000) identified dimensions of expertise in software development: technical expertise (knowledge about a specialised technical area), design expertise (knowledge about software design principles and architecture), and domain expertise (knowledge about the application domain area and client operations). However, in our conceptualisation of expertise, domain knowledge is considered a form of requirements knowledge rather than expertise.

Task Allocation Dependencies

A task allocation dependency occurs when who is doing what, and when, is not known, and this affects project progress. In traditional projects, tasks are allocated to developers by someone not directly involved in development, such as a project manager, whereas in agile software development project team members allocate tasks to themselves. For example: *"As each day comes, you are meant to go grab the next highest priority one [task] off the board, but generally we aren't quite that strict about it, we all know, since we are in a team of four people or five people, what each person is going to be best tackling. And you just go 'yes' I will do that, and no one argues."* [Storm, Developer]. In agile projects using Scrum, stories (high-level requirements) are developed in a sprint planning session. When a story is due for development it is decomposed into individual tasks. Stories and tasks under development in a sprint (or iteration) are displayed on a publically visible wallboard. When a project team member selects a task to work on they attach their avatar or name to the task. This means that all project team members can readily see who is doing what and approximately when.

An example from Storm indicates that knowing what other people are doing can be useful: *"…just if you look at how you would do it with an online system it is like, you can't tell what other people are working on as easily as looking up on a wall."* [Storm, Developer]. A display of task allocations is useful information because each individual might at times need to know the relationship of their task to other tasks. Other tasks might precede, follow, or be concurrent with their task. This enables them to monitor progress, and better manage their work, as well as helping others complete their tasks. For example: "*So a developer, if he wants his story to be Done,… if they see a couple of test tasks hanging on there for a good couple of days, they may just say '[Sam] do you want me to do these? You look pretty busy there,' and they'll go off and do them."* [Storm, Developer].

Teamwork theory provides an explanation for this task allocation dependency. Based on a literature review, Salas, et al. (2005) proposed five components contributing to effective teams. Two are pertinent here: performance monitoring and backup behaviour. Mutual performance monitoring is the ability to keep track of fellow team members' work while performing your own. Monitoring leads to backup behaviour: a team member willingly providing resources or task-related effort to another when a workload distribution problem is recognised. Evidence for the task allocation dependency was found in each project.

Historical Dependencies

A historical dependency occurs when knowledge about past decisions is needed, and if such knowledge is not readily available, project progress is affected. Grinter (1996) defined historical dependencies in terms of the need to mine organisational memory or old code versions for previous decisions, when studying large-scale software development projects. These dependencies tend to occur when existing poorly documented systems need replacement, which was exactly the situation in Storm. In that project, an existing complex system was to be completely redeveloped. The old system was largely undocumented, and had been extensively enhanced over 15 years. The project team needed to know what decisions had been made in the past, and why, so they could decide which algorithms in the old system needed to be reproduced and which parts could be omitted from the new system. The project team addressed this problem in part, by bringing domain experts who had developed the original system, into the team. They also sent their own team members to reside temporarily with other project teams who had expertise in the functions of the old system. This comment reflects the situation: *"…we have a good ear of [Walter] who … used to be one of the Delphi programmers, and also I think he was heavily involved in the VMS for a while, so he's got a great history,…he understands why a lot of stuff is there.."* [Storm, Developer].

Activity Dependencies

We define an activity dependency as a situation wherein an activity cannot proceed until another activity is complete, and this affects project progress. Activity dependencies are central to both organisation studies, Coordination Theory, and project management planning, as discussed previously in the literature review on dependencies. (Note that the concepts of task and activity are not clearly distinguished in the literature. We take the view that a task is an activity that has a goal). Activity dependencies were found in all projects. The Storm project team made efforts to reduce dependencies during breakdown sessions: *"… we are getting much better at doing, in our weekly breakdowns or bi-weekly breakdowns, is prioritising things so there is minimal blockage happening for everyone*." [Storm, Tester]. In Silver, dependencies sometimes appeared unexpectedly during development: "*…it became a discovery process. If ... we were ignorant of the possibility of these dependencies, until we found that somebody would be working on a story at,…the top of the board that affected or was required by, a story below it...occasionally we started off trying to work on the second story and then found out 'hold on, I can't complete this because it requires something that you are working on'*." [Silver, Developer].

Business Process Dependencies

A business process dependency occurs when an existing business process causes activities to be carried out in a certain order, and this affects project progress. System integration is acknowledged as a non-trivial aspect of IS (Hasselbring 2000). Fitting new systems with existing systems can involve integrating technical systems and business processes. It is sometimes difficult to tease out the boundary between the technical system and the business process when an existing business process is embodied within an IT system. We identified business process dependencies only in project Land. In this project, it was the existing business process embodied within an existing technical system that the project team needed to accommodate in their development decisions. In Land, a portion of the existing organisational web site was replaced and additional functions incorporated to improve a service to the public. The new system needed to integrate seamlessly with the existing system, and this constrained the development to follow a particular sequence. The developer explained: *"…are translated onto what you did online. <u>The flow the user was taken through</u>. So there was a certain dependency inherent in the process, ... you were choosing something to apply for, then you had to provide some details, then you had to pay. If you break it down like that…<u>that naturally ordered the [development] tasks</u>."* [Land, Developer].

Entity dependencies

An entity dependency occurs when a resource (person, place, or thing) is not available, and this affects project progress. Entities are things, objects, or artefacts. In an IS project, entities include physical things such as people, servers, and documents. When Crowston and Osborn (2003) explored the idea of dependencies proposed in Coordination Theory they argued that all dependencies involved relationships between resources and activities. Such dependencies occur in most projects because they rely on the timely presence of certain things. In Land, Storm, and Silver, waiting for the IT operations group was a common issue: *"…the main people we annoy with that are the operations group, …, the people who look after the servers; and we suddenly go 'oh by the way in two weeks we want to release this new thing to you' and they go 'well I'm not sure I can be ready in two weeks' and you go" [rolls his eyes]*. [Storm, Project Manager]; ***"…****essentially system administration. I would have to occasionally wait on them to do something for me. Like I needed something installed on a server."* [Land, Developer]. IS project management recommends that projects be well-resourced to minimise problems with acquiring needed resources, in a timely manner (Kerzner 2003).

Technical Dependencies

A technical dependency results when a technical aspect of development (such as when one software component must interact with another software component) affects project progress. Software development is rife with technical dependencies, and a large body of research focuses on this problem (Cataldo et al. 2009). In project Land, no technical dependencies were noted, which we attribute to the simple nature of the project - only a single developer was needed to create the system programs. The other two projects both encountered technical dependencies. These projects, Storm and Silver, were more complex technically and involved more developers than Land. In Storm, the team foresaw a technical dependency between modules and addressed it by waiting until various modules were complete so they could be integrated and deployed together, even though they knew there was no apparent value to the end-user in this work: *"...we should be able to include a whole bunch of our modules and the [Vole] modules, and just make one application that you deploy onto the desktop, and doing that, helping ourselves by not having to deploy two applications, more than actually getting through work that gives the user [value], or give us more things migrated...."* [Storm, Tester]. In Silver, a developer noted how continuous integration and automated testing affected code interdependencies: "*…working in a team requires a different discipline to working alone, … because you are interacting with other peoples' code. So having these tests in place, which basically made sure that the behaviour, or the functionality of the code, doesn't change. So that if someone comes along and makes a modification, that they think in their world is quite normal but breaks the way that you were expecting to use that code, then there is a little pop up. So yes, continuous integration is a really useful tool*" [Silver, Developer].

Once dependency types were identified, they were further categorised based on their similarities. Requirement, expertise, historical, and task allocation were categorised as knowledge dependencies because they involve forms of knowledge needed by project participants before they are able to perform project tasks. Activity and business process dependencies were categorised as task dependencies as these dependencies involve the performance of actions by one group of actors before other project actors can begin their required work. Entity and technical dependencies are resource dependencies because both involve the necessary availability of people or things, including software components (technical things), which are a form of resource. Table 2 shows the dependencies found in the projects, and Figure 1 presents the complete dependency taxonomy. The next section describes the classification rules used in allocating dependency instances to dependency types.

Table 2. Dependencies identified in projects

	Dependency	Land	Storm	Silver
Knowledge	Requirement	✓	✓	✓
	Expertise	✓	✓	✓
	Task allocation	✓	✓	✓
	Historical	-	✓	✓
Task	Activity	✓	✓	✓
	Business process	✓	-	-
Resource	Entity	✓	✓	✓
	Technical	-	✓	✓

Classification Decision Rules

Classification decisions were necessary when coding data into categories, and we applied the following decision rules. Allocation of evidential data to the entity, expertise, or requirement dependency involved specialisation logic; a requirement dependency is a specialised form of expertise dependency; and an expertise dependency is a specialised form of entity dependency. For example, if a person has particular requirements knowledge then this could be coded as 1) an entity dependency because a person is involved, or 2) an expertise dependency because the person has particular expertise, or 3) a requirements dependency because the person has particular expertise about requirements. In this example, following these decision rules, the dependency would be coded as a requirements dependency even though the dependency involves both expertise and an entity. Further, an historical dependency is a specialised type of expertise dependency. Another decision rule concerns entity and technical dependencies. In our schema, a technical dependency is a specialised type of entity dependency.

DISCUSSION

Agile software development is fundamentally different from other approaches to software development. Yet, agile projects are influenced by the same environmental characteristics as non-agile projects. Consequently this taxonomy contains many concepts familiar from prior research such as expertise, technical, activity, and entity dependencies. Historical and business process dependencies identified in these agile projects would also be likely to occur in non-agile projects when replacement systems and high levels of integration with existing systems are project characteristics. While identifying these similarities, the taxonomy also draws attention to differences between agile and traditional approaches. In particular the need for a continuous supply of requirements into agile project sprints, which is not necessary in projects organised into traditional phases. Another difference is in the task allocation concept. Although this is seen as a key factor in generic teamwork research, participants in a project knowing who is working on what, is seldom a focus in traditional software projects where a central knowledgeable figure allocates tasks.

Taxonomy of Agile Software Project Dependencies

A dependency is created when the progress of one action relies upon the timely output of a previous action, or the presence of some specific thing. Dependencies lead to potential or actual constraints on projects. Potential constraints are those that are currently organised or managed well, causing no problems in the progression of a project. Actual constraints are bottlenecks or points in a project that stakeholders are aware of, but have no immediate means to circumvent.

Knowledge dependency

A knowledge dependency occurs when a form of information is required in order for a project to progress. There are four forms of knowledge dependency:
Requirement - a situation where domain knowledge or a requirement is not known and must be located or identified and this affects project progress
Expertise - a situation where technical or task information is known only by a particular person or group and this affects project progress
Task allocation - a situation where who is doing what, and when, is not known and this affects project progress
Historical - a situation where knowledge about past decisions is needed and this affects project

Task dependency

A task dependency occurs when a task must be completed before another task can proceed and this affects project progress. There are two forms of task dependency:
Activity - a situation where an activity cannot proceed until another activity is complete and this affects project progress
Business process - a situation where an existing business process causes activities to be carried out in a certain order and this affects project progress

Resource dependency

A resource dependency occurs when an object is required for a project to progress. There are two forms of resource dependency:
Entity - a situation where a resource (person, place or thing) is not available and this affects project progress
Technical - a situation where a technical aspect of development affects progress, such as when one software component must interact with another software component, and its presence or absence affects project progress

Figure 1: A taxonomy of dependencies in agile software development projects

The paper contributes to the IS literature by developing a taxonomy of dependencies in agile software development. We identify dependency types, and propose that, in addition to the traditional ISD project management focus on identifying and managing task and resource in projects, knowledge in the form of expertise, requirements, historical, and task allocation should also be included because it too had the potential to affect project progress in the agile projects in this study. The paper makes a contribution to ISD practice because this taxonomy provides a sensitising device for project stakeholders, a way for them to ensure they consider dependencies they might face that could hinder their projects, and thus enable them to devise appropriate actions to address those dependencies.

This study has limitations. Case study research is limited in its ability to generalize beyond the specific cases studied, and we make no claim that this taxonomy is applicable in all agile software development contexts, but only to those identified in this study. However, following the guidance of Seddon and Scheepers (2012) we have attempted to improve the applicability of our findings to settings beyond these cases by drawing on supporting evidence from extant literature. Further, Gregor (2006) notes that taxonomy should be complete and exhaustive and we cannot make this claim; therefore this taxonomy is currently tentative. Further cases would strengthen the validity of the findings, and we argue for further research to verify our taxonomy. Limitations particular to this study include the selection of cases. Different cases in different contexts might have led us to find different dependencies. Furthermore, we might not have captured all dependency types in the projects in the study because interviews might not draw out all dependencies when participants recall not only recent events, but events occurring many months prior to the interview.

CONCLUSION

This paper has presented a taxonomy of dependencies based on evidence from three typical cases of small, co-located, agile software development projects, and relevant literature. Although some concepts in this taxonomy are widely recognized in prior literature, this paper offers a different perspective on them, that of agile software development, which is not currently available in the ISD literature. Furthermore, this tentative taxonomy makes a beginning at defining key dependency types that have the potential to contribute to project delay, and ultimately to unsatisfactory project progress.

Future work includes verifying the taxonomy with additional evidence. In addition, investigating the relative importance of dependency types would be valuable, for some dependencies might be more critical in certain types of project, or at certain times during a project. Further research could assess the applicability of this taxonomy in contexts such as non-agile software development projects, large-scale, and distributed software projects. This would contribute to a better understanding of the types of dependency in software projects, providing a sound basis for selecting effective mechanisms for managing them.

Another area for future work is in improving our understanding of what and how agile practices address particular dependencies in a situation. We propose that many agile practices, such as iterations, regular and ad hoc meetings, and many others, are designed to effectively manage dependencies in software development. The taxonomy presented in this paper is a starting point for research into this aspect of agile software development.

REFERENCES

Beck, K. 2000. *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., . . . Thomas, D. 2001. "Manifesto for Agile Software Development. ." Retrieved 1 January 2012, from http://www.agilemanifesto.org

Brooks, F. 1995. *The Mythical Man Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley.

Cataldo, M., Mockus, A., Roberts, J., and Herbsleb, J.D. 2009. "Software Dependencies, Work Dependencies, and Their Impact on Failures," *IEEE Transactions on Software Engineering* (35:6), pp. 864-878.

Cockburn, A. 2002. *Agile Software Development*. Boston: Addison-Wesley.

Crowston, K. 1991. "Towards a Coordination Cookbook: Recipes for Multi-Agent Action," in: *Alfred P. Sloan School of Management*. Cambridge, Mass: Massachusetts Institute of Technology, p. 352.

Crowston, K., and Kammerer, E.E. 1998. "Coordination and Collective Mind in Software Requirements Development," *IBM Systems Journal* (37:2), 27 October 2008.

Crowston, K., and Osborn, C.S. 2003. "A Coordination Theory Approach to Process Description and Redesign," in *Organizing Business Knowledge: The MIT Process Handbook*, T.W. Malone, K. Crowston and G.A. Herman (eds.). Cambridge, Massachusetts: The MIT Press, pp. 335-370.

Curtis, B., Krasner, H., and Iscoe, N. 1988. "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM* (31:11), pp. 1268-1287.

Doty, D.H., and Glick, W.H. 1994. "Typologies as a Unique Form of Theory Building: Toward Improved Understanding and Modeling," *The Academy of Management Review* (19:2), pp. 230-251.

Dube, L., and Pare, G. 2003. "Rigor in Information Systems Positivist Case Research: Current Practice, Trends, and Recommendations," *MIS Quarterly* (27:4), December, pp. 597-635.

Dyba, T., and Dingsoyr, T. 2008. "Empirical Studies of Agile Software Development: A Systematic Review," *Information and Software Technology* (50:9-10), pp. 833-859.

Eisenhardt, K.M., and Graebner, M.E. 2007. "Theory Building from Cases: Opportunities and Challenges," *Academy of Management Journal* (50:1), pp. 25-32.

Espinosa, A.J., Slaughter, S.A., Kraut, R.E., and Herbsleb, J.D. 2007. "Team Knowledge and Coordination in Geographically Distributed Software Development," *Journal of Management Information Systems* (24:1), pp. 135-169.

Faraj, S., and Sproull, L. 2000. "Coordinating Expertise in Software Development Teams," *Management Science* (46:12), pp. 1554-1568.

Gregor, S. 2006. "The Nature of Theory in Information Systems," *MIS Quarterly* (30:3), pp. 611-642.

Grinter, R.E. 1996. "Understanding Dependencies: A Study of the Coordination Challenges in Software Development," in: *Information and Computer Science*. Irvine: University of California, p. 130.

Hasselbring, W. 2000. "Information System Integration," *Communications of the ACM* (43:6), pp. 33-38.

Kerzner, H. 2003. *Project Management: A Systems Approach to Planning, Scheduling, and Controlling*, (8 ed.). Hoboken, New Jersey: John Wiley & Sons.

Malone, T.W., and Crowston, K. 1994. "The Interdisciplinary Study of Coordination," *ACM Computing Surveys* (26:1), pp. 87-119.

Malone, T.W., Crowston, K., Lee, J., Pentland, B., Dellarocas, C., Wyner, G., . . . O'Donnell, E. 1999. "Tools for Inventing Organizations: Toward a Handbook of Organizational Processes," *Management Science* (45:3), pp. 425-443.

McChesney, I.R., and Gallagher, S. 2004. "Communication and Coordination Practices in Software Engineering Projects," *Information and Software Technology* (46:7), pp. 473-489.

Miles, M.B., and Huberman, A.M. 1994. *Qualitative Data Analysis*, (2 ed.). Thousand Oaks: Sage.

Pare, G. 2004. "Investigating Information Systems with Positivist Case Study Research," *Communications of the Association for Information Systems* (13:1), pp. 233-264.

Salas, E., Sims, D.E., and Burke, C.S. 2005. "Is There a 'Big Five" in Teamwork?," *Small Group Research* (36:5), pp. 555-599.

Schwaber, K., and Beedle, M. 2002. *Agile Software Development with Scrum*. Upper Saddle River, New Jersey: Prentice Hall.

Seddon, P.B., and Scheepers, R. 2012. "Towards the Improved Treatment of Generalization of Knowledge Claims in Is Research: Drawing General Conclusions from Samples," *European Journal of Information Systems* (21), pp. 6-21.

Spradley, J.P. 1979. *The Ethnographic Interview*. New York: Holt, Rinehart and Winston.

Staudenmayer, N. 1997. "Interdependency: Conceptual, Practical, and Empirical Issues." Massachusetts Institute of Technology.

Strode, D.E., Huff, S.L., Hope, B., & Link, S. 2012. "Coordination in co-located agile software development projects," *The Journal of Systems and Software* (85:6), pp. 1222-1238.

Thomas, D.R. 2006. "A General Inductive Approach for Analyzing Qualitative Evaluation Data," *American Journal of Evaluation* (27:2), pp. 237-246.

Thompson, J.D. 1967. *Organization in Action*. Chicago: McGraw-Hill.

Wagstrom, P., and Herbsleb, J.D. 2006. "Dependency Forecasting in the Distributed Agile Organization," *Communications of the ACM* (49:10), pp. 55-56.

West, D., and Grant, T. 2010. "Agile Development: Mainstream Adoption Has Changed Agility." Forrester Research Inc., pp. 1-20.

Williams, L. 2010. "Agile Software Development Methodologies and Practices," in *Advances in Computers,* M.V. Zelkowitz (ed.). Amsterdam: Elsevier, pp. 1-44.

Yin, R.K. 2003. *Case Study Research*, (3 ed.). Thousand Oaks: Sage.

ACKNOWLEDGEMENTS

We are grateful to the case study research participants and also to Professor Gordon Hunter, Siva Dorairaj, and Kevin Carillo for comments on early versions of this paper.

COPYRIGHT