This document is a submission to a journal. The article is published and available online.

The published article used APA 6th referencing but this version uses APA 7th referencing.

In the example below, I indicate in yellow some important things you should do when writing factual, research-based reports.

1. Abstracts
2. Paragraphs – topic, middle, end
3. Citations – two kinds, multi author, how to integrate
4. Integrating quotes into the text
5. References – different kinds – journal, book, conference paper
6. How proper citation and referencing (APA or any other kind of style) links research to prior research and acknowledges the work of others.

**Title**

A dependency taxonomy for agile software development projects

### Authors

Diane E. Strode

### Affiliation

Whitireia Polytechnic, New Zealand

### Address

Faculty of Business and Information Technology

Whitireia Polytechnic

Wi Neera Drive

Private Bag 50910  
Porirua City 5240  
New Zealand

### Email

[diane.strode@alumni.unimelb.edu.au](mailto:diane.strode@alumni.unimelb.edu.au)

### Alternative email

[diane.strode@whitireia.ac.nz](mailto:diane.strode@whitireia.ac.nz)

### Acknowledgements

The author acknowledges the research participants for informing this research with their experiences and the anonymous reviewers for improving the article with their insightful advice.

### Author Biography

Diane Strode is a Senior Lecturer in the Faculty of Business and Information Technology at Whitireia Polytechnic, New Zealand. Her teaching and research address behavioural aspects of information technology project success, in particular the influence of agile software development on projects and organisations. She has expertise in theory development related to project coordination. Her work has appeared in the *Journal of Systems and Software* and the *New Zealand Journal of Applied Computing and Information Technology*, in books such as *Modern Techniques for Successful IT Project Management*, and in conference proceedings including HICSS, PACIS, ACIS, and CRWIG. She reviews for the *Journal of Systems and Software, Information and Management, Information Systems Frontiers,* and *Journal of the Association of Information Systems.* Diane received her PhD in Information Systems from Victoria University of Wellington, New Zealand.

# A Dependency Taxonomy for Agile Software Development Projects

## Abstract

Agile software development is designed to achieve collaborative software development. A supporting pillar of collaboration is effective coordination, which is necessary to manage dependencies in projects. Understanding the dependencies arising in agile software development projects can help practitioners choose appropriate coordinative practices from the large number of practices provided by the various agile methods. To achieve this understanding, this article analyses dependencies in three typical cases of co-located agile software development and presents the dependencies as a taxonomy with decision rules for allocating dependencies into categories. Findings show that knowledge, process, and resource dependencies are present, with knowledge dependencies predominant. In addition, there are agile practices with a coordinative function that address multiple dependencies in these agile software development projects. These practices would be a good choice for coordinating a project and supporting collaboration in agile software projects.

## Keywords

Agile software development, agile methods, Scrum, dependency, selecting agile practices, coordination mechanisms.

An abstract has no citations!

An abstract summarises ALL the content of the article, which includes:

1. Topic of research
2. Problem the research addresses
3. Method for investigating the problem
4. Key findings
5. Contribution to theory and or practice

## Introduction

Agile methods are an effective class of systems development methodology that differs philosophically and practically from prior development approaches (Dyba & Dingsoyr, 2008; Iivari et al., 2004). Due to their continued popularity and the way they have evolved to fit different contexts, agile methods have fundamentally changed the way software development projects are organised (Stavru, 2014; Wang et al., 2012). Although originally designed for small co-located green-field projects, in the last decade agile methods have also been adapted for large projects, globally distributed projects, and legacy conversion projects (Ramesh et al., 2006; Schnitter & Mackert, 2011). When practitioners adopt one of the agile methods, such as Scrum, Extreme Programming, a hybrid method, or any assemblage of agile practices this is referred to as agile software development.

A paragraph should cover a single topic; usually starts with a topic sentence, has some sentences that support or explain the topic sentence, has a concluding sentence.

A known problem when adopting agile software development is deciding what agile practices to adopt (Meyer, 2014). When agile methods were first proposed, there was a belief that the practices from a particular method (e.g. pair programming, daily stand-up meetings, test-driven development, creating stories) needed to be adopted as a whole, otherwise the advantages of the agile method might be lost since the practices comprising a method interact with and support one another (Beck, 2000; Schwaber & Beedle, 2002). As agile method adoption increased, practitioners nevertheless, “mixed and matched” practices from different methods to suit project contingencies (Conboy & Fitzgerald, 2007; Fitzgerald et al., 2006; Wang et al., 2012) with little understanding of the utility of combining different practices in their projects.

Agile methods focus on collaborative teamwork and collaboration with customers to achieve early delivery of quality software (Cockburn, 2002). An agile software development project team, like any project team, collaborates effectively using mechanisms for coordinating their interdependent work. Coordination breakdowns are a major problem in software development (Curtis et al., 1988; Herbsleb, 2007; Kraut & Streeter, 1995). This problem was articulated by Nidumolu who asked, “How can software development projects be coordinated more effectively in the presence of uncertainty?”(Nidumolu, 1995, p. 213). Given that agile methods were designed to support development in changeable and uncertain environments, arguably they provide a solution to this coordination problem (Pries-Heje & Pries-Heje, 2011; Strode et al., 2012).

There is a wealth of research on coordination in organisation studies and software development at the organisation, project, and team level (Espinosa et al., 2004; Grinter, 1996; Hoegl et al., 2004; Kraut & Streeter, 1995; Marks et al., 2001; Staudenmayer, 1997; Wageman, 1995). Empirical research shows that effective coordination is necessary, although not sufficient, for software project success (Amrit & van Hillegersberg, 2008; Curtis et al., 1988; Faraj & Sproull, 2000; Kraut & Streeter, 1995; McChesney & Gallagher, 2004; Nidumolu, 1995), whereas conceptual research focuses on uncovering general principles of coordination. One central principle of coordination is that coordination mechanisms are necessary to address dependencies in a situation (Malone & Crowston, 1994). Dependencies are the focus in this article.

Dependencies are broadly defined by Crowston and Osborn (2003) who propose that a dependency occurs when the progress of one action relies upon the timely output of a previous action or on the presence of a specific thing, where a thing can be an artefact, a person, or a piece of information. When dependencies occur in a development project, they can be managed well, poorly, or not at all. When dependencies are well managed this suggests that appropriate coordination mechanisms are in place to support the smooth flow of interdependent work. When dependencies are poorly managed then coordination mechanisms might be inappropriate, inadequate, or absent. Furthermore, unmanaged dependencies can constrain or block progress leading to delay as people wait for resources, for the activities of others to be completed, or for necessary information. Dependencies are worthy of study because they are points in a project where bottlenecks, blockages on the flow of work, and waiting can occur. These problems potentially increase the likelihood of project delay, schedule overruns, and the need for task switching.

Another idea proposed by Crowston and Osborn (2003) about dependencies is that many alternative coordination mechanisms might be available to manage a given dependency. This idea was illustrated in the MIT Coordination Handbook (Malone et al., 2003) by showing how a business process can be coordinated using a variety of different coordination mechanisms. This present article uses this idea and proposes that many of the agile practices used in software development projects act as coordination mechanisms to address dependencies in a project.

Dependencies are ubiquitous when developing a software-intensive system and occur between people, groups, tasks, and artefacts, including the software components under construction (Curtis et al., 1988; Grinter, 1996; Wagstrom & Herbsleb, 2006). Research on dependencies has focused on dependencies in general (e.g., Malone & Crowston, 1994), dependencies in non-agile software development (e.g., Grinter, 1996), and dependencies in distributed software development (e.g., Espinosa et al., 2007), however, research addressing dependencies in the context of typical co-located agile software development is currently not available.

Understanding dependencies and understanding which dependencies are addressed by a particular agile method practice would help practitioners to select agile practices from the many that are available, and assemble them appropriately to achieve effective project coordination. Dependencies in agile software development projects potentially differ from traditional software development contexts. Firstly, in traditional development projects requirements are collected as a whole in a requirements gathering phase and written down in a detailed requirement specification. The requirements are then processed in a development phase (Pressman, 2010). In contrast, within agile projects requirements continuously emerge and are written down as brief stories and processed in short iterations of development as the project progresses. In agile projects, without a predefined requirement specification, the development process might be hindered when the needed requirement or its details are not obtained from the client or customer in a timely manner.

Secondly, in traditional software development expertise is located in different groups who have distinctly different roles such as clients, end-users, systems analysts, project managers, developers, and testers. Each role has its own distinct expertise with little sharing across roles. Each role performs prescribed work tasks during the various phases of planning, analysis, designing, coding, and testing with project work handed over from one phase to the next for the various groups to work on. For example, when the code is written by the software developers it is passed to a different group for testing. In agile software development roles are much less prescribed. One person usually takes the role of agile coach (called a Scrum master in the Scrum method) with another person acting as client representative embedded within the team. This person is constantly available to provide requirements, prioritise requirements, and discuss requirement details with the project team. (Schwaber & Beedle, 2002). The remainder of those working on the project share roles and expertise as much as possible forming a self-organising cross-functional project team. That is, all members are responsible for analysis, design, coding, testing, and user documentation (Hoda et al., 2010; Schwaber & Beedle, 2002). This situating of expertise across the project team might reduce the dependency issue in agile project teams compared with the traditional work arrangements because members of agile project teams acquire a greater shared knowledge of all aspects of the project. They might be less likely to need to draw on the expertise of a single project member or group with unique expertise.

Another key difference between traditional and agile projects is the way that tasks are allocated to project team members. In traditional projects, tasks are allocated by a project manager or team leader according to a project plan and the known expertise of the project team member. Whereas in agile projects, tasks are self-selected by project team members from a publically visible wallboard of stories and tasks. Potentially, all members work on all stories during the project. Dependencies between stories are addressed in sprint planning sessions so they are less likely to arise during the sprint. Task dependencies might occur but have less impact on the progress of agile projects because tasks tend to be very small units of work making task switching simpler and easier.

Due to the potential usefulness of better understanding dependencies in agile software development projects, which potentially differ from those of traditional projects, this article addresses the following questions.

1. What dependencies occur in a typical agile software development project?
2. Which agile software development practices act as coordination mechanisms to address the dependencies in a project?

To address these questions, evidence is drawn from three case studies of agile software development and the findings are presented in the form of a taxonomy of dependencies. The frequency with which the various types of dependency occur is assessed, with knowledge dependencies predominant. The taxonomy is then used to identify a set of agile practices that act as coordination mechanisms in the projects and leads to the finding that certain practices address multiple dependencies in a project. This group of practices can be considered effective for coordinating an agile software development project.

The article is organised as follows. First, the status of research on agile software development projects is discussed, followed by an overview of the concept of dependency in coordination theory and in domains relevant to agile software development. Taxonomy are discussed, including criteria for assessing the quality of taxonomy. Following this, the case study research method used in the study is described and findings are presented. A taxonomy is then developed which describes and defines each dependency type. The relative frequency of dependencies is analysed, taxonomy classification decision rules are defined, and the taxonomy is used to assess which agile practices address dependencies in the three agile software development projects. Next is a discussion of the findings which includes an assessment of the quality of the taxonomy. Theoretical and practical contributions are considered followed by the study limitations, and the article concludes.

## Agile Software Development

Agile methods are founded on management ideas from new product development (Takeuchi & Nonaka, 1986), theories of complex adaptive systems (Highsmith, 2000), observations on effective teamwork and communication, and improving product quality in software projects (Beck, 2000; Cockburn, 2002; DeMarco & Lister, 1987; Schwaber & Beedle, 2002). These methods came to prominence in the late 1990s and are now well-accepted world-wide with an estimated adoption level of 50% (Stavru, 2014). Over this period they have remained of considerable interest to researchers and practitioners because they have a strong impact on the development process and on the organisations where they are adopted. Not only has agile software development overturned many accepted guidelines for IT project management and software engineering (Boehm & Turner, 2004) but recognised issues with adopting agile software development in certain organisational cultures have yet to be resolved (Iivari & Iivari, 2011).

Agile software development differs significantly from traditional approaches to software development (Beck, 2000; Dyba & Dingsoyr, 2008). One notable area of difference is the way that project activities are organised. Practices for organising a project range from the adoption of sprints, daily stand up meetings, pair programming, frequent regular product demonstrations, continuous testing and integration, meetings for planning and reflection, and using story and task wallboards to display project status. For useful descriptions of individual agile practices mentioned in this article see Williams (2010) or Strode (2005).

The scope of research on agile software development is broad, occurring in information systems, IT project management, and software engineering. Chuang, Luor, and Lu (2014) summarise contributions in all of these domains from 2001 to 2012. Two issues impact this research. Firstly, there is little consensus on what constitutes agile software development (Conboy, 2009; Conboy & Fitzgerald, 2007; Wang et al., 2012). Organisations can adopt a single agile method but they also create hybrid agile methods such as Scrum with XP (Fitzgerald et al., 2006; VersionOne, 2014), and Scrum with Kanban (Wang et al., 2012). Furthermore, project teams can choose to assemble practices from a variety of methods as their project progresses. This variation in how agile software development is achieved happens because adoption is based on prior experience, training from external training organisations, self-teaching, or by hiring consultants to coach a team. There is little research-based guidance as to which practices to adopt and why, which practices work best together, and which practices achieve effective software project coordination. This article focuses on dependencies in agile software development projects with the aim of identifying those practices in agile software development projects that address dependencies and thereby provide effective coordination.

The second issue is that research on agile software development is largely a-theoretical. Dingosoyr, Nerur, and Moe (2012) noted that, in general, “*a majority of agile studies do not seem to be concerned about any theoretical underpinnings for the research exploration, which reinforces the generally popular perception that agile research tends to be a-theoretical*.” (p. 1217). This article directly addresses this latter concern by exploring the fundamental concept of dependency as it occurs in agile projects, and using the findings to build a basic theory in the form of a taxonomy (Gregor, 2006).

## Dependencies

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). This coordination theory focuses on dependency as a fundamental element in coordination. Malone and Crowston’s theory is based on the tenet that “*coordination is the managing of dependencies between activities*” (Malone & Crowston, 1994, p. 90). In addition, 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.

Interdependency is the focus in organisation studies. An interdependency can be conceived of as two distinct dependencies and in organisation studies the two terms are treated as equivalent. Staudenmayer (1997) synthesised the research on interdependencies in this domain 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 et al. (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 are applicable at the organisation level, rather than the project level. In addition, findings in this domain come from studies of routine work, as opposed to non-routine, time-bounded project work. Although the team workflow arrangement proposed by Thompson (1967) coincides with how work is conducted in agile projects, this conceptualisation does not provide for detailed insight into the agile project context and agile practices. 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 this context are likely to be of a different nature to those of small and co-located projects, where agile methods are considered most appropriate.

Dependency has not previously been a focus of research in agile software development whereas coordination, the flip side of dependency, is a focus of research in software development projects generally (Curtis et al., 1988; Faraj & Sproull, 2000; Kraut & Streeter, 1995; Nidumolu, 1995) and has also emerged in agile software development research. For example, Sharp and Robinson (2010) identify communication, collaboration, and coordination as three important functions of agile practices. Pries-Heje and Pries-Heje (2011) found that Scrum works because it supports communication and social integration, and provides effective mechanisms for project control and coordination. Although coordination has emerged as a factor provided by agile practices, this small body of research provides little insight into the objective of coordination; that is, dependencies are not considered in any of this research.

## Taxonomies in Information Systems

One way to organise knowledge about dependencies is to develop a taxonomy. Taxonomies systematically organise knowledge in a domain of interest, and are common in many fields of science. In biology, the Linnaean taxonomy divides the natural world into animal, vegetable, and mineral forms. In social science, Maslow’s taxonomy classifies different human needs, and Bloom’s taxonomy delineates learning domains. Taxonomies are also common in the computing fields. For example, Glass and Vessey (1995) surveyed the computing field from 1985 to 1993 and found 13 published taxonomies. Another survey by Nickerson, Varshney and Muntermann (2012) in the domains of information systems, computer science, and business identified 73 distinct taxonomies published prior to 2009.

Doty and Glick (1994) define taxonomies as “*classification systems that categorize phenomena into mutually exclusive and exhaustive sets with a series of discrete decision rules*.” (p. 232). In the information systems domain, Gregor (2006) classifies taxonomy 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).

Taxonomy are considered the simplest form of theory in most fields including information systems (Gregor, 2006), with some researchers arguing they are not theory at all (e.g. Doty & Glick, 1994; Weber, 2012). Theory or not, taxonomies are useful when little is known about a topic and categories, concepts, and constructs need to be identified. Taxonomies are also useful when much is known about a topic but that knowledge has not been organised in meaningful ways. Agile software development falls into this second group because there is a significant body of research available but, as noted earlier, much of it is a-theoretical. A theoretically-grounded explanation of how the approach and its myriad practices work together to achieve project success has just begun to crystallise in the research literature (e.g., Cao et al., 2013; Drury-Grogan & O'Dwyer, 2013; Goh et al., 2013; Stankovic et al., 2013; Strode et al., 2012).

Bailey (1994), writing for the social scientist, discusses how taxonomies organise knowledge into discrete categories, where each category has a unique name and a precise description. Categories (also called taxa) can be organised as a simple list, a hierarchy, a matrix, or in other ways. Taxonomies provide building blocks for theory development and testing, because their taxa can be considered concepts or constructs. Furthermore, a taxonomy can propose relationships between taxa, although this is not a requirement for a taxonomy.

A taxonomy can be developed inductively, deductively, or intuitively (Nickerson et al., 2012). Inductively constructed taxonomies analyse empirical data to determine categories. This can be achieved qualitatively, or quantitatively using methods such as cluster analysis (e.g. Sabherwal & Grover, 2010). Deductively constructed taxonomies are based on prior theory or conceptualisation, and intuitively constructed taxonomies are based on common sense or commonly agreed understandings, rather than prior research.

A taxonomy should be organised around a single meta-characteristic. This meta-characteristic serves as a basis for all of the categories occurring in the taxonomy, that is “*each category should be a logical consequence of this meta-characteristic*” (Nickerson et al., 2012, p. 8). Another aspect of taxonomy development is that, ideally, a taxonomy should meet some standard of quality. Nickerson et al. (2012) found no definitive quality criteria for taxonomies in a review of taxonomy literature in the social sciences. Based on their review and Bailey’s (1994) work on taxonomies, they proposed six subjective criteria for judging the quality of taxonomy for the field of information systems. A quality taxonomy must be:

**Concise** - A taxonomy should be concise or parsimonious, containing a limited number of dimensions, each dimension having a limited number of characteristics.

**Robust -** A taxonomy should be robust, containing enough categories to differentiate clearly the objects of interest. Categories must be mutually exclusive, disallowing overlap (Bailey, 1994).

**Comprehensive -** A taxonomy should be comprehensive. Collectively all the categories in the taxonomy should enable all known objects in the domain to be classified; that is, the taxonomy is collectively exhaustive (Bailey, 1994).

**Extendible** - A taxonomy should be extendible or dynamic, enabling the addition of new categories when new types of object occur.

**Explanatory -** A taxonomy should have categories set at a high enough level of abstraction to allow new-found objects to be readily classified. Furthermore, each category should be explained precisely enough so that the characteristics of objects belonging in that category are readily comprehended.

**Useful –** This taxonomy quality proposed byNickerson et al. (2012) can only occur over time; a taxonomy is useful if others use it.

These characteristics and quality criteria provide a way to assess whether a taxonomy makes a valid contribution to information systems knowledge. This article inductively constructs the dependency taxonomy by analysing empirical data to determine categories then assesses the quality of the taxonomy against these quality criteria.

## Research Design

A positivist multi-case study was used to explore the concept of dependency in agile software projects and provide a basis for taxonomy development. The case study approach is an appropriate way to investigate phenomena such as ongoing agile software development projects where events cannot be controlled and where it is important to capture contextual detail (Eisenhardt, 1989; Pare, 2004; Yin, 2003). Case selection followed a replication logic strategy, a tactic recommended in case study design (Eisenhardt, 1989; Miles & Huberman, 1994; Pare, 2004; Yin, 2003). This involves selecting cases that are similar and therefore likely to provide similar results. The guidance of Dubé and Paré (2003) on achieving rigor, and addressing validity and reliability in positivist case study research in information systems was followed. This study met each of the 34 quality criteria they found relevant to single researcher multi-case study research.

### **Case Selection**

The study consisted of three cases; each case was an agile software development project. This number of projects was deemed large enough to identify a pattern of dependencies across cases while providing detail rich enough to ensure dependencies could be thoroughly explored within a case (Eisenhardt, 1989). Each project was a typical case, selected because it was expected to show normal or average characteristics (Pare, 2004). The projects met pre-specified selection criteria regarding the development method. Projects were selected if they were using Scrum, Extreme Programming or a combination of the two, had a team size of 2 to 10 people, were co-located, that is, the team worked together in close proximity, and they used iterations (sprints in Scrum terminology) of 1-week to 1-month. Table 1 describes the cases selected from different organisations located in New Zealand. The cases, each one an independent software development project, were code-named Land, Storm, and Silver. Data from these cases has contributed to related research on coordination (Strode et al., 2012) and dependencies (Strode, 2013; Strode & Huff, 2012).

The three cases varied in two dimensions. Complexity was the first dimension. Storm was the most complex project because it had the largest team, and involved replacing a legacy system with multiple new systems that had to integrate seamlessly with one another and with other local and international systems. Silver was moderately complex with new technologies for the developers to learn while reproducing a straightforward reporting system. Land was the simplest system involving a single developer working with a business analyst, web designer, and three customer proxies, and the new system was developed with well-understood technologies.

The second dimension where the projects varied was in the degree of involvement of the customer (or knowledgeable end-user) with the team. Although Scrum and XP projects optimally embed a permanent customer representative in the team, in typical agile projects this does not occur. This has been reported in 10 cases of XP and Scrum by Cao, Ramesh and Baskerville (2007) and also by Martin, Biddle, and Noble (2004) in three studies of XP. In Storm, the end-user group was committed and concerned with the project but they were not consistently available for consultation. This group consisted of 80 people working on rotating shifts who all worked on another floor of the building. Land had three proxy end-users (the true end user was the New Zealand public), who were considered part of the team, but they were located on another floor of the building and were involved in other projects too, so they were not constantly available. Silver’s customer was in another organisation and direct communication with end-users was difficult to arrange. Requirements were provided by a designated manager from the client organisation.

### **Data Collection**

In each project, people in different roles such as project leader, developer, business analyst, domain expert, or tester were interviewed. The project leader tended to take on multiple roles as Table 1 shows. Interviews took 40 to 90 minutes following a semi-structured interview schedule. The aim was to interview at least half of the members of each team (i.e. 5 people for a typical team of 10) but due to availability of team members, this was not always possible. The tactic of interviewing people in a variety of roles in the project was used, whenever possible, to give a broad perspective on dependencies in the project rather than focusing on only one perspective (for example by only interviewing developers or only interviewing project leads). An interview schedule appropriate for collecting data on dependencies in software development projects was developed based on suggestions by Crowston (1991) for identifying dependencies and coordination mechanisms, McChesney and Gallagher (2004) for software project profiles, Grinter (1996) for dependencies in software development, and Spradley (1979) for interviewing in natural situations. Spradley, for instance, suggests avoiding ‘why’ questions to ensure that participants give a full and open range of responses. Therefore questions related to coordination and dependencies were not asked directly in the form of ‘why’ questions, e.g. “*Why do you have a wallboard to show tasks?”,* but more indirectly as ‘how’ questions such as “*How do you know what task to perform when the current task is complete?”* The interview schedule is provided in Appendix A.

Source data included interview transcripts, field notes taken during observation of work sites and when attending meetings, project documents, photographs of work sites, and questionnaire data. Team leaders completed a questionnaire providing background data about the organisation, the project, and the agile practices in use. In addition, each project team member completed a questionnaire about the agile practices in use. These questionnaires are available in Strode (2012). The completed questionnaires were used to ensure that facts about the project context were accurately recorded. They also ensured the selected project was using commonly accepted agile practices from Scrum and XP and therefore fit the case selection criteria. The questionnaires were not used to provide evidence of project dependencies. Table 1 summarises details about the organisation, the project, the project team, and the interviews carried out.

Table 1 Summary of organisation, project, and data collection information

|  |  |  |  |
| --- | --- | --- | --- |
| **Case** | **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 interactions with the public by fully automating a semi-manual system | To migrate a critical legacy system to a modern technology platform | To replace an existing 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 and XP |
| *Iteration duration* | 1-week | 2-week | 2-week initially, moving to 1-week |
| *Team size* | 6 | 10 | 5 |
| *Team composition* | All permanent staff | All contracted staff working at the client site | 4 permanent staff  1 contracted agile coach |
| *Interviews* | 2 | 5 | 4 (including one joint interview with technical lead and Scrum master) |
| *Roles of interviewees* | * Project manager / Senior systems analyst / Agile coach * Senior software developer | * Programme manager / Agile mentor and Scrum coach * Software developers (2) * Tester * Technical domain expert | * Technical lead * Agile coach / Scrum master (contracted expert) * Senior developer * Software developer (2) |

### **Data Analysis**

The first step in data analysis was to prepare a full description of each case using a framework as recommended by Eisenhardt (1989). This framework had sections describing the organisation, the project, the technologies, the team, the development method, and any problems in the project that emerged during the interviews. This description consolidated the data collected in each case. The description was sent to one project participant for verification, and any factual errors they found in the description were corrected.

The second step in data analysis was within-case analysis. To facilitate within-case analysis, interviews were transcribed and entered into the qualitative data analysis tool NVivo™. A general inductive coding technique as described by Miles and Huberman (1994) and Thomas (2006) was followed. Coding involved identifying data items that share a common meaning. A data item is a section of conversation from the interview transcript, or other data such as a photograph, a section of project documentation, or a section of text from field notes. Each data item was given a descriptive name, called a code, which was a label pertinent to the objective of the study. Each code was defined uniquely with respect to all other codes in the study. This approach was guided by heuristics developed by Crowston and Osborne (2003) for analysing dependencies and coordination mechanisms. Their two-phase procedure was as follows.

*Dependency-focused analysis* - identify dependencies, and then search for coordination mechanisms. In other words, look for dependencies and then determine 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 aims to determine whether all observed coordination activities are necessary (Crowston & Osborn, 2003, p. 352).

Each transcript was analysed using these heuristics. At least two passes were made over each transcript, once to identify dependencies and their associated coordination mechanism and again to identify coordination mechanisms and any associated dependencies. Once instances of a dependency were identified, they were grouped into categories based on their similarities and given a unique code (i.e., a simple descriptive name, such as Expertise) and a definition.

Data items provided by observation and field notes, photographs, and project documents were primarily used as supporting evidence for the codes developed from the interview transcripts. For example, when a participant discussed how the wall board was used, photographic evidence of a working wallboard displaying active stories and tasks was taken (whenever this was available and permission was granted) to support this discussion. Another example was when a participant discussed how informal story card breakdown into tasks was carried out. At a site visit, this story breakdown activity was observed and descriptive notes taken during the session by the researcher.

In performing analyses, codes identified in the first case were used as starter codes for analysing the data in the second case, codes from the second case acted as starter codes for the third case. When new codes emerged during analyses, they were added to the list of codes. Finally, all cases were re-checked to look for evidence for all emergent codes in each case. At this point, coordination mechanism-dependency pairs were identified and data was transferred to MS Excel to better visualise the pairs. In a final step, all dependency codes identified in the projects were grouped based on a distinct common property. This is discussed in the section Dependencies in the Agile Projects. Thus, the overall analysis involved three levels of abstraction: instances of dependency were found in the data, from this dependency categories were developed, and then these categories were grouped into higher-level categories based on a common property.

The cases provided evidence for a variety of dependencies which were then used to develop a taxonomy of dependencies. Selected evidence from each case is provided in Table 2, which shows one example from each case for each dependency. No evidence was found for the historical dependency in Land or the business process dependency in Storm or Silver. Table 2 shows dependences (underlined), and how photographs, observations, and project documents support the identified dependency. All names in the table are anonymised.

## The Taxonomy of Dependencies

Developing the taxonomy involved first identifying dependencies and their associated coordination mechanisms. To visualise dependencies and coordination mechanisms the findings in each case were arranged in a data display table, as Miles and Huberman (1994) recommend for the analysis of qualitative data. Figure 1 provides an example of the data display table for Storm, and includes three quotes from the interviews to illustrate typical evidence for a dependency-coordination mechanism pair. Coloured cells in the figure represent one or more quotes or other evidence (e.g. a photograph or note taken during an observation) identified as a coordination mechanism-dependency pair.

Although coordination is not the focus of this article, Figure 1 shows both dependencies and coordination mechanisms to illustrate how dependencies are addressed by coordination mechanisms. Coordination mechanisms include agile practices and other practices used in the projects to address a dependency. These mechanisms are arranged into categories. For example, Figure 1 shows, among others, coordination mechanisms for achieving synchronisation. A synchronisation activity is defined by Strode et al. (2012) as an activity involving all team members that brings them together at the same time and place for some pre-arranged purpose. Mechanisms for synchronising the project team include having a domain specialist work full-time with the team, self-assignment of a story (where a team member selects a task to work on from a publicly visible wallboard), and a breakdown session where the team sit together to divide a story into tasks. Because these mechanisms shared the common property of achieving synchronisation among the project team members they are named synchronisation activities. Such coordination mechanisms are not discussed further in this article.

The types of dependency identified in the data were named and defined. Evidence was found for requirement, expertise, historical, task allocation, activity, business process, entity, and technical dependencies. These dependency categories and their groupings were arranged as a taxonomy, which is shown in Figure 2. The following sections define and discuss each of these types of dependency in turn, followed by the higher-level categories named knowledge, process, and resource dependencies into which they were categorised.

#### Requirements dependency

Requirements are a critical input to software development projects as they define the basic functions and qualities the software under development should possess. In agile software development projects, requirements are acquired and elaborated in user story creation and task breakdown sessions involving the team and suitable knowledgeable stakeholders (i.e., end-users, clients, customers, or their proxies). Requirements are then developed in a short iteration of a week to a month. This cycle is repeated as new requirements emerge as the project progresses (Beck, 2000; Cao & Ramesh, 2008; Schwaber & Beedle, 2002; Sharp & Robinson, 2010).

Table 2 Evidence for the dependencies identified in projects

| **Dependency** | **Evidence** | **Explanation** | **Additional evidence** |
| --- | --- | --- | --- |
| **Knowledge dependency** | | | |
| Requirement | *“So, in our sprint planning meetings we would figure out what was needed for the next sprint and do some kind of design work on what the screen might look like, exactly what validation was required, those kinds of things.”* [Land, Senior developer] | The team sat together in the iteration planning meeting planning tasks to complete in the coming iteration and working together to formalise requirement details. Without this session, team members would not know the requirement details (dependency) | Project document showing a table of user stories (requirement details) to be addressed |
| *"We have a person on our team, who is one of the gurus, and he sits with us, and we annoy him constantly to get stuff like that.* [Storm, Developer] | There is a domain specialist located with the team in the same room who can be readily consulted on requirements (dependency) | Observation of ad hoc design discussion involving domain specialist |
| *"We knew from the Product Owner what they wanted, so we got together in this room, looked at the stories, discussed how we would implement, and discussed the acceptance test criteria, and went round [the table], and everyone had one story, and they would write down the acceptance test criteria that we agree on. And then make tasks for each of those stories." [Silver, Senior developer]* | At a story breakdown session, requirements from the Product Owner written on story cards where discussed among the team and acceptance tests were written for each story. Each story was broken down into tasks. In this way, team members learn about the requirements (dependency) and the tasks that make up each story | Photograph of functioning wallboard displaying stories and tasks |
| Expertise | *“… in terms of who to ask within the business specific questions, yeah, 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, Senior developer]* | The developer discovered the role and expertise (dependency) of the other team members at the weekly sprint planning meetings. Without this information, the developer would not know who to go to for information about existing system functions or technical issues. | None |
| *“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, Programme manager]* | The team members learn what the design decisions are (dependency) at an ad hoc team session around the table | Observation of work underway in team workplace |
| *"...we had effectively a day and a half when development would stop, testing would come in and everyone would test the thing to destruction effectively and the knowledge of how it worked at that point would be picked up." [Silver, Developer]* | The team members learn about the software system (dependency) by all working together on regression testing the system when they prepared for the product demonstration | None |
| Task allocation | *“Each day, so I guess I would coordinate with Sam, usually first thing in the morning we would kind of get together and decide, and have quick catch up about what I was working on, it could have been something I was working on from the previous day.” [Land, Senior developer]* | The ‘technical’ team, with desks side by side, would have a talk each morning about the current work. Without this session, the current work underway and who is working on particular tasks (dependency) would not be known to the other members of the technical team. | Observation of workplace layout |
| *"You learn that from the daily interaction with them and at the “stand ups”. What they’ve done … the previous day … and you soon learn that these guys do that, and those guys do that."* [Storm, Developer] | The team learns who is doing what and when (dependency) at the daily stand up session | Observation of daily stand-up meeting |
| *"So normally that would be the task wall you know, ‘where are the avatars?’ to quickly see what people are working on."* [Silver, Developer] | The avatars on tasks on the wallboard show the project team who is working on what task at present (dependency). | Photograph of functioning wallboard |
| Historical | No evidence for historical dependency in Land |  |  |
| *“A whole story is done according to … [the Done list]. … That is something we ... came up with recently because …you would think some stuff was done but it hadn’t been... people thinking that they had done something…and you know they had written all the stuff in one part of the system but they hadn’t actually done the integration with other things to see that it was…really working."* [Storm, Developer] | The done list is used to ensure that knowledge about past work (dependency) is signalled to the team. | Photograph of “Done” list |
| *"What do you put into Rally?"* [Researcher]. *"… all the user stories in the Product Backlog, all the details about the user stories, which would be a short description and the size, if we have it. It is also electronic backup of the task wall, because 6 months from now I will have forgotten velocity of Sprint 3."* [Silver, Scrum coach] | Rally™, acts as a repository for project information so that past decisions are available (dependency) |  |
| **Process Dependency** | | | |
| Activity | *“Sam waited. He would do the first line of testing. So he would always do an 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, Senior developer] | One team member transferred a completed software component (dependency) to another team member with different expertise to carry out additional tasks involving the component (e.g., user acceptance testing or interface design). | None |
| *"It’s true actually, and I think that that’s what 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] | The work of one team member is not blocked by the work of another (dependency) because the stories are prioritised in an appropriate order | None |
| *"Well, …, it became a discovery process… 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. So the next person, well 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] | The wallboard displaying stories and tasks acted as in indicator of potential activity timing conflicts (dependency) | Photograph of functioning wallboard |
| Business process | *"…are translated onto what you did online. The flow the user was taking through. So there was a certain dependency inherent in the process, so basically 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…that kind of naturally ordered the [development] tasks.”* [Land, Senior developer] | Development tasks were organised around the existing business process used to handle applications. This was because the output of one part of the process acted as input to the next part. This was managed by assigning each portion of functionality to an iteration in the same order that it occurred in the actual business process (dependency). |  |
| No evidence of business process dependencies in Storm |  |  |
| No evidence of business process dependencies in Silver |  |  |
| **Resource dependency** | | | |
| Entity | *“Project System Support, essentially system administration yeah. I would have to occasionally wait on them to do something for me. Like I needed something installed on a server, or I needed access to a server or something like that.”* [Land, Senior developer] | A server is not available (dependency) causing the developer to wait. | None |
| *"…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, Programme manager] | The resource, e.g. a server, is not available as and when needed (dependency). | None |
| *"They are really nice, and really helpful …and, you walk over and say ‘can you help me with this?’"* [Silver, Scrum coach] | The IT support team (dependency) in the next room were approached directly as and when needed | *Observation of work place layout* |
| Technical | *No evidence of technical dependencies identified in Land* |  |  |
| *"...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], give us more things migrated....There are story cards for it, but we keep trying to avoid them. And there were just jokes about that."* [Storm, Developer] | There was a technical dependency between a software module developed by the project team and modules developed by another external project team. This changed the order in which the project team processed user stories (and subsequently developed the code) because these software modules had to fit together at a later date (dependency) | None |
| *“…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 in fact breaks the way that you were expecting to use that user code, then there is a little pop up. So yes, continuous integration is a really useful tool or basically these automatic testing because that’s the primary thing it does is … automate the execution of the unit tests instead of going for a whole day and then deciding to run them."* [Silver, Developer] | The continuous integration and automated unit test system ensured that changes to one part of the code base had minimal impact on other parts. This indicates that technical dependencies within the code base (dependency) potentially affected the work of the development team. | None |

In the projects, when requirements failed to emerge in a timely manner this caused the project to falter unless the project team took action to cope with the situation. In Storm there was no single person to consult for story creation, prioritisation, or for detailed understanding of system algorithms because of the ever-changing nature of the large user group. The project team found it difficult to get prompt and consistent information about requirements and they found this was interfering with project progress. The team took action to identify four end-users that they nominated as beta testers and these people became the preferred group for consulting on requirements. In Silver, a similar situation occurred when their client did not provide timely information to the team on story prioritisation or design details on some user stories. To cope with this, the team chose to de-prioritise these blocked user stories and address lower priority stories to maintain workflow and keep to their iteration schedule. The development manager explained. *“It sometimes changed our priorities…so we have sometimes, once or twice, taken stories out of the sprint because we couldn’t get the information 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, Technical lead]. In Land, there were three proxy end-users who were considered part of the team. They were not committed full time to the project and therefore they were not consistently available for consultation. This caused the developer and designer on the team to have to wait at times for requirement details.

Based on evidence of this type, a requirement dependency is defined as a situation wherein domain knowledge (in the form of a requirement for a new system) is not known and must be located or identified, and this affects, or has the potential to affect, project progress.

Research into requirements management in agile software development also identifies this problem. This is the challenge of achieving customer access and participation, and the impact this has on the steady flow of requirements into the project. Ramesh et al. (2007) in exploring agile requirements engineering (RE) in Scrum and XP projects identified this as a challenge, and Martin et al. (2004) found the same issue in XP projects.

#### Expertise Dependency

Expertise is a recognised factor influencing the success of software projects (Brooks, 1995; Curtis et al., 1988; Faraj & Sproull, 2000), and its importance has also been acknowledged in agile software development (e.g. Beck, 2000; Cockburn, 2002). Expertise dependencies occurred in each agile project. For example, in project Land the developer noted that it was during discussions in the regular weekly sprint planning meetings where he identified the roles and expertise of other team members. He needed this information so he knew who to consult on different aspects of the system. In Storm, when two or more team members made a design decision then they would “*bring all of the team to the table so that everyone hears the design decision*” [Storm, Programme manager]. This ensured that needed expertise was shared within the team. In Silver, the team noted how they learned exactly how the software system worked as they worked together on regression testing to prepare for their product demonstration. In Silver, expertise was unevenly distributed among the project team members when the project began. Expertise therefore needed to be shared as the project progressed, and this situation led to a high number of expertise dependencies reported within the project. Based on observations of this type, an expertise dependency is defined as the situation wherein technical or task information is known only by a particular person or group and this affects, or has the potential to affect, project progress.

Three forms of expertise in software development have been identified by Faraj and Sproull (2000) including 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). Each of these forms of expertise needs to be shared in an agile project to reduce a dependency caused by one person having knowledge that other team members need to perform their work.

#### Historical Dependency

Historical dependencies were first identified in a study of large-scale software development projects by Grinter (1996), who defined a historical dependency as the need to mine organisational memory or old code versions for previous decisions. Historical dependencies were identified in two of the agile projects. In project Storm, an existing complex system was completely redeveloped. This old system was largely undocumented and had been extensively enhanced over the previous 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 out their own team members to reside temporarily with other project teams who were knowledgeable about 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].

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *“We have a person on our team, who is one of the gurus. He sits with us and we annoy him constantly to get stuff (requirements details) like that [Developer]*  A domain specialist is located in the room with the team (coordination mechanism) who can readily be consulted about requirements (requirement dependency) | | | **Dependency** | | | | | | | |
| Key   |  |  | | --- | --- | | Resource dependency |  | | Process dependency |  | | Knowledge dependency |  | | | | Knowledge | | | | Process | | Resource | |
| Expertise | Requirement | Task allocation | Historical | Activity | Business process | Entity | Technical |
| **Coordination Mechanisms** | Synchronisation activities | Domain specialist on team |  |  |  |  |  |  |  |  |
| Self-assignment of story |  |  |  |  |  |  |  |  |
| Breakdown session (iteration) |  |  |  |  |  |  |  |  |
| Daily stand-up |  |  |  |  |  |  |  |  |
| Cross-team talk |  |  |  |  |  |  |  |  |
| Pair program |  |  |  |  |  |  |  |  |
| Breakdown session (ad hoc) |  |  |  |  |  |  |  |  |
| Informal negotiation f2f |  |  |  |  |  |  |  |  |
| Continuous build |  |  |  |  |  |  |  |  |
| Synchronisation artefacts | Product backlog |  |  |  |  |  |  |  |  |
| Wallboard |  |  |  |  |  |  |  |  |
| User story |  |  |  |  |  |  |  |  |
| Task |  |  |  |  |  |  |  |  |
| Burn down chart |  |  |  |  |  |  |  |  |
| Done list |  |  |  |  |  |  |  |  |
| Whiteboard for discussing designs |  |  |  |  |  |  |  |  |
| Working software |  |  |  |  |  |  |  |  |
| Wiki for storing project information |  |  |  |  |  |  |  |  |
| Boundary spanning activity | Workshop to generate backlog |  |  |  |  |  |  |  |  |
| Nominated specialist end user |  |  |  |  |  |  |  |  |
| User story prioritisation |  |  |  |  |  |  |  |  |
| Software demo to user |  |  |  |  |  |  |  |  |
| Dedicated time for consult |  |  |  |  |  |  |  |  |
| Formal meeting |  |  |  |  |  |  |  |  |
| Informal negotiation f2f |  |  |  |  |  |  |  |  |
| Acquisition of specialist knowledge |  |  |  |  |  |  |  |  |
| Boundary spanning artefact | List of tests |  |  |  |  |  |  |  |  |
| Availability | Single priority team |  |  |  |  |  |  |  |  |
| Proximity | Team member co-location |  |  |  |  |  |  |  |  |
|  | Customer co-location |  |  |  |  |  |  |  |  |
| Substitutability | Redundant skill |  |  |  |  |  |  |  |  |
| Coordinator role | Tester |  |  |  |  |  |  |  |  |
|  | Project manager |  |  |  |  |  |  |  |  |
|

*"We’ve got the whiteboard, … we don’t do locks so you can get people working in the same code area, but generally you know we are all in the same [room, so] people are … aware of what other people are doing, people are … aware of what people are working on. So they just [think] I’m not going to go into that module because you are already there. [Tester] “So … where does this awareness come from?” [Researcher]*

*“The board and cross-team talk.” [Tester]*

The work of one team member is constrained by the work of another (activity dependency). The wallboard (coordination mechanism) coordinates this activity dependency.

*“…we can just yell out over our shoulder and grab someone, and … if they come to a complex design decision then they … bring all of the team to the table so that everyone hears the design decision that is made*” [Programme manager]

The team members learn about design decisions (expertise dependency) at an ad hoc team session (coordination mechanism)

Figure 1 Dependencies and coordination mechanisms identified in Storm

**Dependency**

When the progress of one action relies upon the timely output of a previous action, or the presence of some specific thing

**Resource**

When an object is required for a project to progress

**Process**

When a task must be completed before another task can proceed and this affects project progress

**Knowledge**

When a form of information is required for a project to progress

**Entity**

A resource (person, place, or thing) is not available and this affects, or has the potential to affect, project progress

**Business process**

An existing business process causes activities to be carried out in a certain order and this affects, or has the potential to affect, project progress

**Activity**

An activity cannot proceed until another activity is complete and this affects, or has the potential to affect, project progress

**Task allocation**

Who is doing what, and when, is not known and this affects, or has the potential to affect, project progress

**Historical**

Knowledge about past decisions is needed and this affects, or has the potential to affect, project progress

**Expertise**

Technical or task information is known only by a particular person or group and this affects, or has the potential to affect, project progress

**Requirement**

Domain knowledge or a requirement is not known and must be located or identified and this affects, or has the potential to affect, project progress

**Technical**

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, or has the potential to affect, project progress

Figure 2 Schematic of the dependency taxonomy

Similarly in Silver, knowledge about decisions made in the construction of the existing system needed to be understood by the project team so they could decide if they needed to recreate certain algorithms in the new system. Land had no historical dependencies because that project involved automating an existing manual system rather than replacing an existing system or parts of an existing system. Based on observations such as these, a historical dependency is defined as a situation wherein knowledge about past decisions is needed and this affects, or has the potential to affect, project progress.

#### Task Allocation Dependency

In agile projects using Scrum, stories are developed in a sprint planning session. When a story is due for development it is then decomposed into individual tasks. Stories and tasks under development in a sprint (or iteration) are displayed on a publicly 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. Seeing how tasks are allocated can provide useful information because each individual might at times need to know the relationship of their task to others’ tasks. Other tasks might precede, follow, or be concurrent with their task. Having ready access to this information enables each project team member to monitor progress and better manage his or her work, as well as helping others complete their tasks. Both Storm and Silver used a wallboard to display stories and tasks. In Land, this was achieved by storing stories and tasks in a shared document. The developer in Storm explained. “*So a developer, if he wants his story to be ‘Done’ … if they see a couple of test tasks [related to their current story] hanging on [the wallboard] 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]. What this illustrates is that, when the developers were aware of Sam’s task backlog they were able to take on some of this work so that their own progress was not affected. Similar task allocation dependencies were identified in each project. Based on observations of this nature, a task allocation dependency is defined as a situation wherein who is doing what, and when, is not known and this affects, or has the potential to affect, project progress

Teamwork theory explains this task allocation dependency. Salas, et al. (2005) proposed five components contributing to team effectiveness: team leadership, mutual performance modelling, backup behaviour, adaptability, and team orientation. 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. This monitoring leads to backup behaviour, which occurs when a team member willingly provides resources or task-related effort to another team member when a workload distribution problem is recognised.

#### Activity Dependency

Activity dependencies occurred in each of the projects. The Storm project team made efforts to reduce these dependencies during breakdown sessions: *“… [What] 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]. This response from a Silver team member describes an activity dependency: “*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*’.” Observations of this type led to a definition of activity dependency as a situation wherein an activity cannot proceed until another activity is complete, and this affects, or has the potential to affect, project progress.

There is no distinct body of literature focusing exclusively on activity dependencies in agile software development or software development in general. Coordination theory is founded on the idea that dependencies are formed by the interaction between activities and resources (Malone & Crowston, 1994) and the examples found in the projects quoted above illustrate this relationship.

#### Business Process Dependency

System integration is a challenging aspect of systems development (Gumm, 2006). Fitting together new systems with existing systems can involve integrating technical systems and business processes. When observing systems, it can be difficult to tease out the boundary between the technical system and the business process when the two are intertwined. Dependencies in this situation are not purely technical, but must also accommodate existing business processes.

A single agile project, Land, provided evidence for the business process dependency. In this project, there was an 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: *"… So there was a certain dependency inherent in the process, so basically you were choosing something to apply for, then you had to provide some details, then you had to pay.”*  [Land, Developer]. This observation led to a definition of business process dependency as a situation wherein an existing business process causes tasks to be carried out in a certain order, and this affects, or has the potential to affect, project progress.

#### Entity Dependency

There is general agreement that development and implementation projects should have senior management support and be well-resourced to minimise problems with acquiring needed resources in a timely manner (Yetton et al., 2000). Typical resources in an information systems development project are physical things such as people, servers, data, and documents, which can be broadly termed entities. Entity dependencies occurred in each project. Waiting for resources from the IT operations group was a common issue. One project manager explained it.  *"…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, Programme Manager]. Therefore, an entity dependency is defined as a situation wherein a resource (person, place, or thing) is not available and this affects, or has the potential to affect, project progress.

#### Technical Dependency

Software development is rife with technical dependencies and a large body of software engineering research focuses on this problem (Aspray et al., 2006; Cataldo et al., 2009; Herbsleb, 2007). Technical dependencies occurred in two of the agile projects. In project Land, no technical dependencies were noted due to the simple nature of the project, requiring only a single developer to create the bulk of the system programs. Storm and Silver were more complex technically and involved more developers than Land. In Silver, the project team recognised that code changes with unexpected consequences interfered with the smooth workflow on the project. They reported that continuous testing and integration reduced the negative impact of these events by providing prompt feedback to the team on the impact of code changes. 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. They explained that *"...[by waiting] 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 helps us by not having to deploy two applications, more than actually getting through work that gives the user [value]...”* [Storm, Tester]. In this situation, dependencies between code modules had the potential to affect project progress. If the team had decided to deploy the modules in sequence to the customer this would have involved more work than integrating and deploying the modules together because individually deploying dependent modules involves repeating preparation and deployment tasks for each module. Observations such as this led to the definition of a technical dependency as the situation that occurs when 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 or has the potential to affect, project progress.

#### Knowledge, Process, and Resource Dependencies

The dependency categories described in the previous sections form three higher-level categories based on common properties. These new categories act as an aid in categorising dependencies, which is discussed in the section on Classification Decision Rules, and depicted in Figure 2. The argument for forming knowledge, process, and resource dependencies is as follows.

Requirement, expertise, historical, and task allocation dependencies are categorised as knowledge dependencies because they occur when someone on the team needs information. For example, team members reported that they needed to know what stories to create, or the detail of an interface, or a process to be automated before they could continue their work. An expertise dependency occurs when someone on the team needs to know the expertise of another person to draw on that expertise to complete their task, this could involve locating information about some technical issue or information about how to complete a task. An historical dependency involves team members needing to know past decisions that had an impact on their project work. The task allocation dependency involves team members knowing who was doing what project tasks and when they were doing those tasks. Since each of these categories of dependency involves a need to know about aspects of the project these dependency categories were grouped into a category named knowledge dependency.

Activity and business process dependencies are categorised as process dependencies. An activity dependency occurs when one project task cannot proceed until another task is completed. This situation normally causes a project team member to either switch tasks to maintain their work flow or wait until someone else’s task is complete. A business process dependency occurs when an existing process in the organisation causes the work of the project to be completed in a particular sequence. Since both of these dependency categories involve task sequencing then they are grouped into a higher-level category named process dependency.

Entity and technical dependencies are forms of resource dependency. An entity dependency occurs when a resource (a person, a location, or any other concrete artefact) is not immediately available and causes project progress to halt or causes the project team to re-organise themselves to work around or accommodate the lack of a resource. A technical dependency occurs when a technical aspect of development affects progress. This dependency is concerned with the internal structure of the software. For example, if a bug in the software is identified and this causes a software module to be removed from the system for a period while repair work is done and this affects the progress of the project in some way then this is deemed a technical dependency. Entity and technical dependencies both involve the availability of resources so they form a higher-level resource dependency category.

### Frequency of Project Dependencies

Having identified dependencies it is also useful to know which dependencies occur most frequently in the projects. To calculate frequencies, the coloured cells in the data display tables (an example is shown in Figure 1) were counted and converted to percentages. The resulting summary of dependency data is shown in table 3.

Table 3 Comparison of frequency of dependency occurrences

|  |  |  |  |
| --- | --- | --- | --- |
| **Dependency** | **Land** | **Storm** | **Silver** |
| Requirement | 50 | 40 | 18 |
| Expertise | 9 | 15 | 23 |
| Historical | - | 4 | 20 |
| Task allocation | 23 | 11 | 14 |
| Activity | 5 | 19 | 16 |
| Business process | 5 | - | - |
| Entity | 9 | 2 | 2 |
| Technical | - | 9 | 7 |
| **Key**  All values are percentages indicating the number of coordination mechanism-dependency pairs identified in a project. For example, in project Land the 50% requirement dependency means that there were 11 dependency-coordination mechanism pairs from a total of 22 coordination mechanism-dependency pairs identified in that project overall. | | | |

The results from this analysis led to the following observations. Requirement dependencies were encountered in all projects and occurred when domain knowledge or a requirement was not known and had to be located or identified. Expertise dependencies were encountered on all projects and involved the need for technical or task information. Task allocation dependencies were encountered on all projects and involved project team members needing to know who is doing what and when on the project. Task allocation dependencies were more frequent in Land than in the other projects because the team was not fully co-located; some sat at adjacent desks while others were on another floor of the building. In addition, the project team did not use a wallboard to display stories and tasks but relied on other coordination mechanisms, such as weekly sprint planning meetings, to identify who was doing what and when. Historical dependencies were encountered in two of the projects where prior versions of the system under development were involved. Activity dependencies were encountered on all projects. These dependencies occurred when an activity could not proceed until another activity was complete. Business process dependencies were only encountered in Land, causing activities to be carried out in a certain order. Entity dependencies were encountered on all projects and typically involved acquiring servers from external business units such as the IT support team. All of the projects reported this as a problem that held up their work or caused them to re-organise work tasks. Technical dependencies were encountered in Storm and Silver when a technical aspect of development affected progress.

The next step in data analysis was to compare the frequency of knowledge, process, and resource dependencies. This involved summing all knowledge dependencies (i.e., requirements, expertise, task allocation, and historical dependencies), summing process dependencies (activity and business process dependencies), and summing resource dependencies (entity and technical dependencies). Table 4 shows this analysis, which highlights the predominance of knowledge dependencies; averaging 76% of dependencies across the projects. Process dependencies were the next most frequently occurring dependency at 15%, and 10% of dependencies were resource dependencies.

Table 4 Summary of dependencies

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Dependency** | **Land** | **Storm** | **Silver** | **Average** |
| Knowledge | 82 | 70 | 75 | 76 |
| Process | 9 | 19 | 16 | 15 |
| Resource | 9 | 11 | 9 | 10 |
| **Key**  All values are percentages indicating the relative number of coordination mechanism-dependency pairs identified in the project. For example, in project Land 82% of all dependencies were identified as knowledge dependencies. | | | | |

These findings indicate that in the three projects, lack of knowledge was reported to affect project progress more than any other dependency. More than half of the dependencies identified related to obtaining information about requirements, the structure and function of existing systems, technical and task information, and who is doing what and when within the project.

### Classification Decision Rules

A taxonomy should have explanatory power, which is achieved with decision rules for classifying newly found objects. In this taxonomy the object is a dependency. The three high-level categories of knowledge, process, and resource are the starting point for classifying any newly identified dependency. Classification begins by asking simple questions such as: is workflow blocked because there is something the team does not know? Is workflow blocked because the teams is waiting for a process to complete? Is workflow blocked because the team is waiting for a resource to become available? Once a dependency is categorised into one of the three high-level categories then it can be further categorised into one of the sub categories. For example, if the dependency was a knowledge dependency then the question becomes: is the team waiting for information about a requirement, a task or technical information, a past decision, or because some team members do not know what other team members are doing or when they are doing a task? Answering these questions enables a knowledge dependency to be further categorised into its appropriate sub-category. If a dependency cannot be categorised into any of the existing categories or sub-categories, then a new category needs to be inserted into the taxonomy.

For more complex situations, three decision rules were developed for allocating dependencies to categories. The first decision rule describes how to allocate a newly found dependency to the entity, expertise, or requirement category. In general, where the meaning of a category provided the greatest explanatory power or precision, the dependency was assigned to this category. For example, if a person knows a fact or piece of information about a requirement, this could be coded as, 1) an entity dependency because a person is involved, or 2) an expertise dependency because the person has a particular item of knowledge, or 3) a requirements dependency because the person has a particular item of knowledge about a requirement. Since category 3) describes the dependency most fully (i.e. has the greatest explanatory power), then this dependency would be categorised as a requirements dependency.

A second decision rule was devised for categorising a new dependency as historical. Although historical information is often embodied in a particular person and could potentially be categorised as an expertise dependency, the taxonomy differentiates between general expertise and knowledge concerning historical information. In the agile projects Storm and Silver, historical knowledge was sometimes known by a particular person, but historical knowledge was also embedded within the program code of a legacy system, or recorded in project artefacts such as a product backlog repository. Therefore, when there is a need to know about decisions made in the past that affect the project, no matter what the source, this type of dependency is best allocated to the historical category.

The third decision rule differentiates between entity and technical dependencies. Initially, found dependencies involving a dependency on a thing (i.e. person, artefact, or object) could all be categorised as entity dependencies. For some found dependencies, such as when a software component cannot be developed or integrated until another software component exists, or is in the correct form, it is more precise to categorise this as a technical dependency. The technical dependency category provides a more precise description and greater explanatory power in this situation.

### Using the Taxonomy to Assemble Agile Practices

To analyse which agile practices act as coordination mechanisms to address dependencies in the projects, the data in each of the three data display tables (an example is shown in Figure 1) was amalgamated. This amalgamated data is shown in Table 5. In this table, practices not normally associate with agile software development were removed (e.g. sending out a team member to reside for a period with another business unit to gain knowledge of that units’ requirements, which was a coordination mechanism for managing a requirements dependency found in Storm, but is not a typical agile practice). The number of dependencies addressed by each agile practice was counted. This indicated which agile practices are multi-purpose because they can address more than a single dependency. The count is shown in the final column of Table 5. To make the table easier to read, only counts for practices addressing three or more dependencies are shown.

Table 5 Dependencies and agile practices that act as coordination mechanisms in the three projects

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | **Dependency** | | | | | | | |  |
|  | | **Knowledge** | | | | **Process** | | **Resource** | |  |
| **Requirement** | **Expertise** | **Historical** | **Task allocation** | **Activity** | **Business process** | **Entity** | **Technical** | **Total (3 or more)** |
| **Coordination mechanisms** | Iteration zero planning session | ✓ |  |  |  |  |  |  |  |  |
| Iteration or sprint (1 or 2 week) | ✓ |  |  |  | ✓ | ✓ | ✓ |  | 4 |
| Iteration planning session | ✓ | ✓ |  | ✓ |  |  |  |  | 3 |
| Progress tracking with user stories |  |  |  | ✓ |  |  |  |  |  |
| Story point prioritising | ✓ |  |  |  |  |  |  |  |  |
| Software release | ✓ |  |  |  |  |  |  |  |  |
| Domain specialist on team | ✓ |  |  |  |  |  |  |  |  |
| Self-assignment of task | ✓ |  |  | ✓ |  |  |  |  |  |
| Breakdown session (iteration) | ✓ |  |  | ✓ | ✓ |  |  |  | 3 |
| Daily stand-up |  | ✓ |  | ✓ |  |  |  |  |  |
| Cross-team talk | ✓ | ✓ | ✓ |  | ✓ |  |  | ✓ | 5 |
| Pair program |  | ✓ |  |  | ✓ |  |  |  |  |
| Breakdown session (ad hoc) | ✓ |  |  |  |  |  |  |  |  |
| Continuous build and test |  |  |  |  |  |  |  | ✓ |  |
| Retrospective |  | ✓ |  |  |  |  |  |  |  |
| Preparation for product demo |  | ✓ | ✓ |  |  |  |  |  |  |
| Product backlog maintenance |  | ✓ |  |  |  |  |  |  |  |
| Acceptance test |  | ✓ |  |  |  |  |  |  |  |
| Product backlog | ✓ |  |  | ✓ | ✓ |  |  |  | 3 |
| Wallboard | ✓ |  |  | ✓ | ✓ |  | ✓ |  | 4 |
| User story | ✓ |  |  | ✓ | ✓ |  |  | ✓ | 4 |
| Task |  |  |  | ✓ | ✓ |  |  |  |  |
| Burn down chart |  |  |  | ✓ | ✓ |  |  |  |  |
| Done checklist |  | ✓ | ✓ |  |  |  |  | ✓ | 3 |
| Whiteboard for sketching ideas | ✓ |  |  |  |  |  |  | ✓ |  |
| Working software | ✓ |  | ✓ |  | ✓ |  |  |  | 3 |
| Wiki for storing project information | ✓ |  |  |  |  |  |  |  |  |
| Code standards |  |  | ✓ |  |  |  |  |  |  |
| Avatar on task |  |  |  | ✓ |  |  |  |  |  |
| Source code control |  |  | ✓ |  | ✓ |  |  |  |  |
| Unit test suite |  |  |  |  |  |  |  | ✓ |  |
| Ground rule |  | ✓ |  |  |  |  |  |  |  |
| Workshop to generate backlog | ✓ | ✓ |  |  |  |  |  |  |  |
| Nominated specialist end user | ✓ |  |  |  |  |  |  |  |  |
| User story prioritisation | ✓ |  |  |  |  |  |  |  |  |
| Software demo to user or client | ✓ |  |  |  |  |  |  |  |  |
| Informal negotiation (face to face) | ✓ |  | ✓ |  | ✓ |  | ✓ | ✓ | 5 |
| Single priority team | ✓ | ✓ | ✓ |  |  |  |  |  | 3 |
| Team member co-location | ✓ | ✓ | ✓ |  | ✓ |  |  |  | 4 |
| Customer co-location | ✓ |  |  |  |  |  |  |  |  |
| Redundant skills (cross-functional) |  | ✓ |  |  | ✓ |  |  |  |  |
| Total (83) | 24 | 14 | 9 | 11 | 14 | 1 | 3 | 7 |  |

Table 5 shows 83 pairs of dependency and coordination mechanism (each one an agile practice). Each of these pairs cannot be described in detail in this article due to space considerations. Therefore, to illustrate how a dependency is addressed by a coordination mechanism we describe how three historical dependencies were addressed by agile practices (see Figure 1 for three further examples of agile practices addressing requirement, expertise, and activity dependencies).

Firstly, Table 5 shows that the historical dependency was addressed by the agile practice of “preparation for product demo”. In project Silver, the working software was demonstrated to the customer at intervals and this required some preparation; the software was tested prior to the demonstration, and the team selected a team member to prepare for and carry out the demonstration. A participant reported that, during this preparation stage, the team often found out information from each other about historical decisions made when designing and developing the product.

Secondly, Table 5 shows that the historical dependency is addressed by the agile practice “working software”. In project Silver, the working software was used to investigate historical decisions made at an earlier stage of the project. This was found when one participant reported, *"...at times having to go and read the code to understand what it’s doing to find out what the decisions were*".

A third example in Table 5 shows that the agile practice “source code control” also addressed a historical dependency. This was reported by the Silver project team, who used a source code control system that allowed them to investigate historical information about when and why changes and had been made to the code base.

As Table 5 illustrates, certain agile practices were multi-purpose addressing five dependencies. These practices were cross-team talk (within the team) and informal face-to-face negotiation between the team and external parties. External parties included other business units or customers, clients, or end-users who were not engaged within the team. The sprint, the wallboard, the user story, and co-location of the team all addressed four dependencies. Agile practices addressing three dependencies were iteration planning sessions, story breakdown sessions at each iteration, a product backlog (stories yet to be addressed), a done checklist (a list of tasks that must be finalised before a story is considered to be complete), working software (a functioning tested code base) and a project team with a single priority (committed to only the current project). Therefore, for the projects in this study, these practices were the most effective at addressing dependencies and promoting a smooth workflow during the project. The 12 agile practices found to address three or more dependencies are as follows.

* Cross-team talk
* Informal face-to-face negotiation with external parties
* Sprints of 1 to 2 weeks
* A wallboard displaying current stories, tasks, and task assignment
* User stories for managing requirements
* A co-located team
* Iteration planning sessions
* Story breakdown sessions
* A product backlog
* A done checklist
* Working software at the end of each sprint
* A single priority team

A further 10 practices addressed two dependencies. They are self-assignment of task, daily stand up meeting, pair programming, preparing for the product demo, and using tasks, a burn down chart, a centrally located whiteboard for sketching ideas, source code control tools, and workshops to generate backlog. Having redundant skills among the team so people can readily take up other’s tasks when needed (e.g. a tester who can also act as a developer), also addresses two dependencies.

Another interesting finding, shown in Table 5, is that 24 different agile practices acted as coordination mechanisms to address the requirement dependency. This means that 24/41, or 58%, or over half of all the coordinative agile practices found in the projects managed a requirement dependency.

## Discussion

This article has explored the nature of dependencies addressed by agile practices in three typical cases of agile software development to address two research questions. The first question concerned what dependencies occur in agile projects. The cases provided evidence for eight types of dependency including requirements, expertise, historical, task allocation, activity, business process, entity, and technical dependencies. These dependencies were arranged as a dependency taxonomy with each of these dependency types categorised as either a knowledge, process, or resource dependency.

This dependency taxonomy meets some but not all of the quality criteria previously discussed. The most essential criterion is the presence of a meta-characteristic, which is the concept of dependency. The taxonomy is concise because it has eight distinct categories, which falls within the recommended seven plus or minus two categories proposed by Nickerson et al. (2012) in their discussion of taxonomy development. The higher-level category of the taxonomy consisting of resource, process, and knowledge dependencies is also simple and concise. The taxonomy is robust because it contains enough mutually exclusive categories to differentiate unambiguously the dependencies in three agile software development projects. The categories were found to be mutually exclusive, that is, a dependency cannot be allocated to more than a single category. Furthermore, simple decision rules are provided to assist in categorising newly found dependencies. This taxonomy is not comprehensive, however, since it is possible that newly found dependencies, not currently identified in the taxonomy, could arise in agile software projects. If future research fails to uncover additional types of dependency, this will strengthen the comprehensiveness of the taxonomy.

Extendibility is another quality criterion. This means that a taxonomy is dynamic, so that new categories can be added when new types of object occur. In its current form, there is no impediment to adding dependency categories to this taxonomy. In addition, a quality taxonomy is explanatory because it has categories set at a high enough level of abstraction to allow newly found objects to be readily classified. The two-level dependency taxonomy allows for the initial classification of a dependency as a knowledge, process, or resource dependency (i.e. involving knowing, doing, or an object or thing). Following this classification, a knowledge dependency can be further categorised. For example, does the knowledge involve knowing what to do (a requirement), knowing how to do an activity (expertise), knowing how or why things were done in the past (historical), or knowing who is doing what and when (task allocation). The taxonomy is also explanatory because each category is named and precisely described so that newly found dependencies in any agile software development project can be readily categorised. The final quality criterion is usefulness, however, this can only be shown if the taxonomy is used by others in the future.

Having identified dependencies it is also useful to understand which dependencies occurred most frequently in the projects. This information informs people when selecting an effective set of coordination mechanisms to address the most frequently occurring dependencies. Findings show that there is no single dependency type predominates. The aggregated category of knowledge dependencies, however, accounts for 76% of all dependencies across the projects. Process dependencies were the next most frequently occurring dependency at 15%, and 10% of dependencies were resource dependencies. This is an important finding because it indicates that addressing knowledge dependencies should have a significant impact on agile project coordination.

The second research question asked which agile practices address the dependencies in a project. The findings show that multiple agile practices in the three projects had a coordinative function and addressed one or more of the dependencies found in the dependency analysis. 12 multi-purpose agile practices were identified that address three, four, or five dependencies and these practices are potentially an effective minimal set for coordinating a project. A further 10 practices addressed two dependencies. The finding that 58% of coordinative agile practices address the requirements dependency reflects the strong emphasis on regularly acquiring, managing, processing, and verifying requirements in these agile projects.

The article contributes to theory by developing a taxonomy, which is a form of theory (Gregor, 2006). This taxonomy provides an integrated view of the types of dependency that occur in agile software development, a context where the concept has not previously been explored. The dependences identified in this study match some but not all of the dependencies found in prior research in other software development contexts. Grinter, in 1996, studied three software development organisations using traditional software development processes. In common with Grinter, this study identified expertise and historical dependencies in the software projects studied. In addition, her customer dependency is equivalent to the requirement dependency in this study (because the customer provides the requirements). The vendor dependency identified by Grinter is named a technical dependency in this study (for example, in project Storm, a vendor needed to adapt their product so it would integrate with the system under development, which was deemed a technical dependency in this study). Comparing the findings with the dependencies identified by Espinosa et al. (2007) in globally distributed software development, both their study and the current study found process dependencies and technical dependencies, but the current study did not find evidence for Espinosa et al.s’ temporal dependencies. Since the projects in this study had none of the time zone or geographical constraints that tend to create temporal dependencies this difference is understandable. This comparison with prior dependency research shows that although the current taxonomy includes dependencies previously identified in non-agile software development projects, studying agile projects has surfaced additional dependencies including task allocation (who is doing what and when in the project), business process dependencies, and entity dependencies.

This study contributes to practice because the taxonomy can be used by practitioners to understand what hinders, or has the potential to hinder, project progress in typical Scrum or XP agile projects because dependencies are points in a project where blockages and waiting can occur. Understanding these situations means that practitioners can design or select appropriate coordination mechanisms from the agile methods to address these dependencies before they seriously block progress. The list of 12 agile practices that address multiple dependencies would be a recommended starter set for providing coordination in an agile project with a profile similar to that of the projects selected in this study. When dependencies occur for which no appropriate practice from the agile methods toolbox is available then project teams must borrow practices from elsewhere, or devise new practices. For example, if knowledge about a system is scattered throughout an organisation, then identifying the appropriate source of requirements, expertise, and historical knowledge is an important task. Ensuring this knowledge is continuously available to the project team as the project progresses, perhaps by sending out project team members to learn from other groups, or bringing in groups with expertise to work with the team, might be useful tactics for mitigating and coordinating a potential knowledge dependency constraint. Also, when a particular dependency is identified then different practices for addressing the dependency could be trialled leading to an overall improvement in software project coordination. For example, using a daily stand up meeting to address the task allocation dependency could be substituted with the use of an electronic wallboard with real-time updates to address the task allocation dependency.

The limitations of this study include those common to qualitative case study research in general and those peculiar to this research study. Case study research is limited in its ability to generalize beyond the specific cases studied. Therefore, this taxonomy is applicable to the agile software development projects in this study. Choosing typical agile projects using practices common in many other scrum-based project does suggest that findings from the projects in this study might be applicable to projects with similar profiles. Furthermore, Seddon and Scheepers (2012) have explained that drawing on supporting evidence from extant literature can improve the applicability of findings to settings beyond the cases in a study, and supporting research has been provided for most of the dependency types. Gregor (2006) notes that taxonomy should be exhaustive, and, as noted previously, this claim cannot be made, and further research is needed to strengthen the validity of the taxonomy. Limitations particular to this study include the selection of cases. Choosing different cases in different contexts might have led to different dependencies being identified. Furthermore, within the selected cases all dependency types might not have been captured because data was collected over a short period whereas the overall project took many months to complete and participants might place less emphasis on events occurring many months prior to the interview. Conversely, because each project was studied during the final third of the project duration then project teams had plentiful experiences of dependencies and coordination to report. Another limitation is that reporting frequencies of identified dependencies in qualitative materials is highly subjective. If more interviews were carried within each case, or different cases were selected, this might have affected the frequency and distribution of identified dependencies.

## Conclusion

This article has explored dependencies in three agile software development projects and organised them into a taxonomy of dependencies. This taxonomy extends prior knowledge about dependencies found in non-agile software development project contexts. This taxonomy also provides basic knowledge about dependencies useful for deciding how to assemble practices from commonly used agile methods to achieve effective project coordination. The taxonomy comprises knowledge, process, and resource dependencies. Knowledge dependencies consist of requirements, task allocation, historical, and expertise dependencies; process dependencies include activity and business process dependencies; and resource dependencies include entity and technical dependencies. In addition, this article found that knowledge dependencies are predominant, which suggests that focusing on selecting agile practices that address the various types of knowledge dependency would benefit coordination in agile software development projects. Additional analysis showed that certain agile software development practices act as coordination mechanisms by addressing dependencies, and 12 common agile practices are able to address multiple project dependencies. This would be a good starter set of practices for practitioners to adopt to achieve effective agile software project coordination and support collaboration.

Future work includes investigating the relative importance of dependency types because 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 large-scale agile or distributed agile software development. This would contribute to a better understanding of dependencies in the various forms of agile software development project and provide a sound basis for choosing effective coordinative practices to better support a collaborative systems development environment.

## Appendix A

Interview schedule for individual project team member.

Notes:

* If participants choose to be interviewed in pairs or small groups that is acceptable.
* If questions arise in this schedule that participants have addressed in previous responses then omit the question.

1. Background questions
   * Name
   * Job title
   * Years of IT experience
   * Educational background
   * Describe your job (what is your role on this project) and its goals
2. What are your main work activities (3-5 activities) IN THIS PROJECT
   * For each activity (this will depend on specialisation/generalisation of work)
     + Main purpose of the activity
   * Depending if work is broken down into distinct activities or not…

* How is work assigned to you?
* How do you know when to start the work/activity?
* Who do you work with to complete the activity [for stakeholder identification]
  + - Who do you send communications to?
    - Who do you receive communications from?
* How do you share out/or delegate the work?
* How do you decide who to share out or delegate the work to?
* What resources (things or information) do you need to complete the activity?
* What technologies do you use to help you carry out the activity? (e.g., email, configuration management tools, wiki, project database, on-line project plan, on-line specification)
* What forms or documents do you need to perform the activity? Examples?
* What is the product or partial product of the activity? (documents, information, software)
  + - Who relies on the product of this activity?
* Do any of your work products need to be integrated or fit in with other peoples’ work or applications?
  + - Who waits for your work to be completed? [Ask for an example]
  + Individual activities:
* How do you know when the activity is complete?
* How do others know when the activity is complete?
* What things hinder this activity
  + - What do you wait for?
    - Negotiate for?
    - Bid for?
* What would happen if this activity was not carried out?
* What alternative ways could the outputs or goal of this activity be achieved?

1. Dependency prompts:
   * Who is your vendor(s) [for stakeholder identification]
   * Who is your customer(s) [for stakeholder identification]
   * What other business units do you interact with? [for stakeholder identification]
   * Lifecycle (only used when multiple teams are working on same product for different platforms) – how is this managed/organised?
   * Big picture - how do you achieve an overview of how all of the parts of the system fit together?
   * Testing – test versions - how do you manage/organise this? What types of testing do you do?
   * Parallel development – when two or more people work on the same code module – how do you manage/organise this?
   * Change – when changes are made how does this impact other code modules, documentation, testing?
   * Expertise
     + How do you all know what the product must do?
     + How do you know what the other people on the team are doing on a daily basis within the code?
     + How do you know what the other people on the team are doing on a daily basis within the documentation?
     + How do you know what the other people on the team are doing on a daily basis within the test bank?
     + How do you come to know what the skills and capabilities of the other team members are – how do you know who to ask about things?
   * Historical understanding – how do you find out about previous decisions made that impact
     + The code?
     + This project?
   * Integration – how do you integrate the code and other components?
     + Regularly, randomly, at the end of the project. To a schedule.
   * What is the main source of bugs in the system?

## References

Amrit, C., & van Hillegersberg, J. (2008). Detecting coordination problems in collaborative software development environments. *Information Systems Journal*, *25*(1), 57-70.

Aspray, W., Mayadas, F., & Vardi, M. Y. (Eds.). (2006). *Globalization and offshoring of software: A report of the ACM job migration task force*. Association for Computing Machinery. <http://www.acm.org/globalizationreport/pdf/fullfinal.pdf>.

Bailey, K. D. (1994). *Typologies and taxonomies: An introduction to classification techniques*. Sage.

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

Boehm, B., & Turner, R. (2004). *Balancing Agility and Discipline*. Addison-Wesley.

Brooks, F. (1995). *The mythical man month: Essays on software engineering*. Addison-Wesley.

Cao, L., Mohan, K., Ramesh, B., & Sarker, S. (2013). Adapting funding processes for agile IT projects: An empirical investigation. *European Journal of Information Systems*, *22*(1), 191-205.

Cao, L., & Ramesh, B. (2008). Agile requirements engineering practices: An empirical study. *IEEE Software*, *January/February*, 60-67.

Cataldo, M., Mockus, A., Roberts, J. A., & Herbsleb, J. D. (2009). Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on software engineering*, *35*(6), 864-878.

Chuang, S.-W., Luor, T., & Lu, H.-P. (2014). Assessment of institutions, scholars, and contributions on agile software development (2001-2012). *The Journal of Systems and Software*. <https://doi.org/http://dx.doi.org/10.1016/j.jss.2014.03.006>

Cockburn, A. (2002). *Agile Software Development*. Addison-Wesley.

Conboy, K. (2009). Agility from first principles: reconstructing the concept of agility in information systems development. *Information Systems Research*, *20*(3), 329-354.

Conboy, K., & Fitzgerald, B. (2007). The views of experts on the current state of agile method tailoring. In T. McMaster, D. Wastell, E. Ferneley, & J. DeGross (Eds.), *IFIP International Federation for Information Processing* (Vol. 235, pp. 217-234). Springer.

Crowston, K. (1991). *Towards a coordination cookbook: recipes for multi-agent action* Massachusetts Institute of Technology]. Cambridge, Mass. <http://hdl.handle.net/1721.1/49303>

Crowston, K., & Osborn, C. S. (2003). A coordination theory approach to process description and redesign. In T. W. Malone, K. Crowston, & G. A. Herman (Eds.), *Organizing Business Knowledge: The MIT Process Handbook* (pp. 335-370). The MIT Press.

Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM*, *31*(11), 1268-1287.

DeMarco, T., & Lister, T. (1987). *Peopleware: Productive projects and teams*. Dorset House.

Dingsoyr, T., Nerur, S., Balijepally, V., & Moe, N. (2012). A decade of agile methodologies: Towards explaining agile software development. *The Journal of Systems and Software*, *85*(6).

Doty, D. H., & Glick, W. H. (1994). Typologies as a unique form of theory building: Toward improved understanding and modeling. *The Academy of Management Review*, *19*(2), 230-251.

Drury-Grogan, M., & O'Dwyer, O. (2013). An investigation of the decision-making process in agile teams. *International Journal of Information Technology and Decision Making*, *12*(6), 1097-1120.

Dube, L., & Pare, G. (2003). Rigor in information systems positivist case research: Current practice, trends, and recommendations. *MIS Quarterly*, *27*(4), 597-635.

Dyba, T., & Dingsoyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and Software Technology*, *50*(9-10), 833-859.

Eisenhardt, K. M. (1989). Building theories from case study research. *The Academy of Management Review*, *14*(4), 532-550.

Espinosa, A. J., Lerch, F. J., & Kraut, R. E. (2004). Explicit versus implicit coordination mechanisms and task dependencies: One size does not fit all. In E. Salas & S. M. Fiore (Eds.), *Team Cognition: Understanding the Factors that Drive Process and Performance* (pp. 107-129). American Psychological Association.

Espinosa, A. J., Slaughter, S. A., Kraut, R. E., & Herbsleb, J. D. (2007). Team knowledge and coordination in geographically distributed software development. *Journal of Management Information Systems*, *24*(1), 135-169.

Faraj, S., & Sproull, L. (2000). Coordinating expertise in software development teams. *Management Science*, *46*(12), 1554-1568.

Fitzgerald, B., Hartnett, G., & Conboy, K. (2006). Customising agile methods to software practices at Intel Shannon. *European Journal of Information Systems*, *15*(2), 200-213.

Glass, R. L., & Vessey, I. (1995). Contemporary application domain taxonomies. *IEEE Software*, *12*(4), 63-76.

Goh, J. C., Pan, S. L., & Zuo, M. (2013). Developing the agile IS development practices in large-scale IT projects: The trust-mediated organizational controls and IT project team capabilities perspectives. *Journal of the Association for Information Systems*, *14*(12), 722-756.

Gregor, S. (2006). The nature of theory in information systems. *MIS Quarterly*, *30*(3), 611-642.

Grinter, R. E. (1996). *Understanding dependencies: A study of the coordination challenges in software development* [PhD Thesis, University of California]. Irvine.

Gumm, D. C. (2006). Distribution dimensions in software development projects: A taxonomy. *IEEE Software*, *September/October*, 45-51.

Herbsleb, J. D. (2007). Global Software Engineering: The future of socio-technical coordination. In *Proceedings of the International Conference on Software Engineering: Future of Software Engineering FOSE '07* (pp. 188-198). IEEE Xplore digitial library.

Highsmith, J. (2000). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing.

Hoda, R., Noble, J., & Marshall, S. (2010). Organizing self-organizing teams. In *Proceedings of the IEEE/ACM International Conference on Software Engineering, ICSE 2010, May 2-9, 2010* (pp. 285-294). ACM. <https://doi.org/10.1145/1806799.1806843>

Hoegl, M., Weinkauf, K., & Gemuenden, H. (2004). Interteam coordination, project commitment, and teamwork in multiteam R&D projects: a longitudinal study. *Organization Science*, *15*(1), 38-55.

Iivari, J., Hirschheim, R., & Klein, H. K. (2004). Towards a distinct body of knowledge for information systems experts: Coding ISD process knowledge in two IS journals. *Information Systems Journal*, *14*(4), 313-342.

Iivari, J., & Iivari, N. (2011). The relationship between organizational culture and the deployment of agile methods. *Information and Software Technology*, *53*(5), 509-520. <https://doi.org/10.1016/j.infsof.2010.10.008>

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

Kraut, R. E., & Streeter, L. A. (1995). Coordination in software development. *Communications of the ACM*, *38*(3), 69-81.

Malone, T. W., & Crowston, K. (1994). The interdisciplinary study of coordination. *ACM Computing Surveys*, *26*(1), 87-119.

Malone, T. W., Crowston, K., & Herman, G. A. (Eds.). (2003). *Organizing Business Knowledge: The MIT Process Handbook*. The MIT Press.

Malone, T. W., Crowston, K., Lee, J., Pentland, B., Dellarocas, C., Wyner, G., Qimby, J., Osborn, C., S, Bernstein, A., Herman, M. K., & O'Donnell, E. (1999). Tools for inventing organizations: toward a handbook of organizational processes. *Management Science*, *45*(3), 425-443.

Marks, M. A., Mathieu, J. E., & Zaccaro, S. J. (2001). A temporally based framework and taxonomy of team processes. *The Academy of Management Review*, *26*(8), 356-376.

Martin, A., Biddle, R., & Noble, J. (2004). The XP customer role in practice: Three studies. Agile Development Conference,

McChesney, I. R., & Gallagher, S. (2004). Communication and coordination practices in software engineering projects. *Information and Software Technology*, *46*(7), 473-489.

Meyer, B. (2014). *Agile! The good, the hype and the ugly*. Springer.

Miles, M. B., & Huberman, A. M. (1994). *Qualitative Data Analysis* (2 ed.). Sage.

Nickerson, R. C., Varshney, U., & Muntermann, J. (2012). A method for taxonomy development and its application in information systems. *European Journal of Information Systems*, *22*(3), 336-359.

Nidumolu, S. (1995). The effect of coordination and uncertainty on software project performance: Residual performance risk as an intervening variable. *Information Systems Research*, *6*(3), 191-219.

Pare, G. (2004). Investigating information systems with positivist case study research. *Communications of the Association for Information Systems*, *13*(1), 233-264.

Pressman, R. S. (2010). *Software engineering: A practitioner's approach* (7 ed.). McGraw-Hill Higher Education.

Pries-Heje, L., & Pries-Heje, J. (2011). Why Scrum works. . In *Proceedings of the Agile Conference 2011, Salt Lake City, UT* (pp. 20-28). IEEE Xplore digital library. <https://doi.org/10.1109/AGILE.2011.34>

Ramesh, B., Cao, L., & Baskerville, R. (2007). Agile requirements engineering practices and challenges: An empirical study. *Information Systems Journal*, *20*(5), 449-480. <https://doi.org/10.1111/j.1365-2575.2007.00259.x>

Ramesh, B., Cao, L., Mohan, K., & Xu, P. (2006). Can distributed software development be agile? *Communications of the ACM*, *49*(10).

Sabherwal, R., & Grover, V. (2010). A taxonomy of political processes in systems development. *Information Systems Journal*, *20*(5), 419-447.

Salas, E., Sims, D. E., & Burke, C. S. (2005). Is there a 'big five" in teamwork? *Small Group Research*, *36*(5), 555-599.

Schnitter, J., & Mackert, O. (2011). Large-scale agile software development at SAP AG. In L. A. Maciaszek & P. Loucopoulos (Eds.), *Evaluation of Novel Approaches to Software Engineering. Communications in Computer and Information Science* (Vol. 230, pp. 209-220). Springer.

Schwaber, K., & Beedle, M. (2002). *Agile software development with Scrum*. Prentice Hall.

Sharp, H., & Robinson, H. (2010). Three 'C's of agile practice: Collaboration, coordination and communication. In T. Dingsoyr, T. Dyba, & N. B. Moe (Eds.), *Agile software development: Current research and future directions*. Springer-Verlag.

Spradley, J. P. (1979). *The Ethnographic Interview*. Holt, Rinehart and Winston.

Stankovic, D., Nikolic, V., Djordjevic, M., & Cao, D.-B. (2013). A survey study of critical success factors in agile software projects in former Yugoslavia IT companies. *Journal of Systems and Software*, *86*(6), 1663-1678.

Staudenmayer, N. (1997). Interdependency: Conceptual, practical, and empirical issues. *Sloan working paper 3971*. Retrieved 1 June 2012, from <http://dspace.mit.edu/bitstream/handle/1721.1/2671/SWP-3971-38485337.pdf?sequence=1>

Stavru, S. (2014). A critical examination of recent industrial surveys on agile method usage. *The Journal of Systems and Software*, *94*, 87-97. <https://doi.org/http://dx.doi.org/10.1016/j.jss.2014.03.041>

Strode, D. E. (2005). *The agile methods: An analytical comparison of five agile methods and an investigation of their target environment.* [Master's Thesis, Massey University]. Palmerston North, New Zealand. <http://hdl.handle.net/10179/515>

Strode, D. E. (2012). *A theory of coordination in agile software development projects* [PhD Thesis, School of Information Management, Victoria University of Wellington, Wellington]. New Zealand. <http://hdl.handle.net/10063/2505>

Strode, D. E. (2013). Extending the dependency taxonomy of agile software development. In P. Antunes, M. A. Gerosa, A. Sylvester, J. Vassileva, & G. de Vreede (Eds.), *Proceedings of the 19th International Conference on Collaboration and Technology, CRWIG 2013, Wellington, New Zealand, 30 October to 1 November. LNCS* (Vol. 8224, pp. 274-289). Springer-Verlag. <http://link.springer.com/chapter/10.1007/978-3-642-41347-6_20>

Strode, D. E., & Huff, S. L. (2012). A taxonomy of dependencies in agile software development. In *Proceedings of the 23rd Australasian Conference on Information Systems, ACIS 2012, 3-5 December, Geelong, Australia* (pp. 1-10). <http://dro.deakin.edu.au/eserv/DU:30049080/strode-taxonomyofdependencies-2012.pdf>

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), 1222-1238. <https://doi.org/10.1016/j.jss.2012.02.017>

Takeuchi, H., & Nonaka, I. (1986). The new new product development game. *Harvard Business Review*, *64*(1), 137-146.

Thomas, D. R. (2006). A general inductive approach for analyzing qualitative evaluation data. *American Journal of Evaluation*, *27*(2), 237-246. <https://doi.org/10.1177/1098214005283748>

Thompson, J. D. (1967). *Organization in Action*. McGraw-Hill.

VersionOne. (2014). *8th Annual state of agile survey*. VersionOne Inc. Retrieved 3 March from <http://www.versionone.com/pdf/2013-state-of-agile-survey.pdf>

Wageman, R. (1995). Interdependence and group effectiveness. *Administrative Science Quarterly*, *40*(1), 145-180.

Wagstrom, P., & Herbsleb, J. D. (2006). Dependency forecasting in the distributed agile organization. *Communications of the ACM*, *49*(10), 55-56. <https://doi.org/10.1145/1164394.1164420>

Wang, X., Conboy, K., & Cawley, O. (2012). "Leagile" software development: An experience report analysis of the application of lean approaches in agile software development. *The Journal of Systems and Software*, *85*(6), 1287-1299.

Weber, R. (2012). Evaluating and developing theories in the information systems discipline. *Journal of the Association for Information Systems*, *13*(1), 1-30.

Williams, L. (2010). Agile software development methodologies and practices. In M. V. Zelkowitz (Ed.), *Advances in Computers* (Vol. 80, pp. 1-44). Elsevier. <https://doi.org/doi:10.1016/S0065-2458(10)80001-4>

Yetton, P., Martin, A., Sharma, R., & Johnston, K. (2000). A model of information systems development project performance. *Information Systems Journal*, *10*, 263-289.

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