Four Perspectives of the Usage of Behavior-Driven Development by Software Engineers

Author 1
Affiliation 1
University 1
Country 1
Email1

Author 2
Affiliation 2
University 2
Country2
Email2

Author 3
Affiliation3
University 3
Country3
Email3

ABSTRACT

Software development is a collaborative activity to reach a challenging goal: the construction of a software system. Agile development arose as a way to address some of the challenges in previous software development approaches. Despite its success, agile teams still face issues, in particular, sharing knowledge and establishing a shared common ground among the different software development roles. Behavior-Driven Development (BDD), a set of practices designed to help teams to better communicate and work together, was created to minimize these issues. In this paper, we describe an empirical study with 24 software engineers about BDD. We sought to develop a deeper understanding on what BDD is used for, including the roles involved with it, the tools used to support it, and the perceived benefits and challenges of adopting it. Our analysis revealed four perspectives of BDD, which are BDD as: collaboration, communication, requirements management, and testing. Our study reflects upon how the BDD scenarios, which are the foundation of the identified perspectives, facilitate communication and coordination among different software development roles and by doing so address one of the major issues in software development.

Author Keywords

Behavior-Driven Development; Agile Development; Empirical Study; Grounded Theory; Boundary Object.

ACM Classification Keywords

H.5.3. Group and Organization Interfaces: Computer-supported cooperative work.

INTRODUCTION

Software development is a collaborative endeavor that requires dynamic and adaptive processes for software teams to keep up-to-speed with industry demands and rapid market changes. Agile methods are an attempt to respond to this challenging scenario [27]. These methods were defined founded on the Agile Manifesto [5] and propose that small, collocated cross-functional teams work close together with business stakeholders in order to define valuable software to attend the customer needs.

A set of agile practices have emerged over the years aiming to implement the values and principles defined in the Agile Manifesto and minimize or ease major software development issues repeatedly reported in software engineering literature. For instance, Pair Programming consists in two programmers working collaboratively on the same development-related activity, e.g., designing an algorithm. In a pair programming session, one of the developers acts as driver and develops the code while the other acts as the navigator (or observer), responsible for reviewing and preventing errors in the code [3]. This practice has been the focus of many research studies: Chong and Hurlbutt [9] have investigated the effects of interruptions in the activities of both solo and pair programmers while other researchers have focused on the effect of pair programming in product quality [23] and development productivity [40]. Although agile methods have been widely adopted by software organizations, they still have limitations including lack of shared common ground and domain knowledge, and difficulties in sharing knowledge, which are mostly caused by the lack of customer involvement and vocabulary misalignment between business and technical members. Both challenges often cause documentation to be obsolete and impose unnecessary changes to the project scope.

Therefore, a new agile practice has emerged recently: Behavior-Driven Development [26]. Behavior-Driven Development (BDD) is a set of software engineering practices designed to help teams to deliver higher quality software faster. BDD provides a common language based on simple, structured sentences expressed in English (or in the native language of the stakeholders). The structure is imposed by a set of pre-defined clauses as highlighted in italics in the following example: *Given* my Current account has a balance of 100.00 *And* my Savings account has a balance of 200.00 *When* I transfer 50.00 from my Current account to my Saving account *Then* I should have 50.00 in my Current account *And* I should have 250.00 in my Savings account. This common language is expected to help teams to better document and communicate requirements, keep documentation up-to-date, and facilitate knowledge sharing and knowledge transferring during and after the development cycle [32].

It is noteworthy that these are the *expected* benefits of BDD, since they are described by one of its proponents ([26]). In

fact, to the best of our knowledge, there are no empirical studies on the usage of BDD. Despite that, there is a growing interest in this agile practice. For instance, there are a lot of active groups exchanging ideas about BDD and the tools used to support it on GoogleGroups. LinkedIn has more than 50 groups about BDD and its tools. InfoQ is an independent professional community that focuses on the dissemination of innovation and knowledge in software development. It has more than 600 posts since 2006 on the topic.

Given the increasing interest in BDD and the lack of studies about it, we conducted an empirical study seeking to gain a deeper understanding on what BDD is used for and on its benefits. We also aimed to explore which challenges teams face in adopting BDD and what is the role of tool support in such usage. Therefore, our guiding research question was *"What BDD is used for and what are the related characteristics?"*

By contributing to a better understanding of the usage of BDD, we hope to inform industry practitioners of the BDD intricacies as well as researchers that can further propose solutions to improve the usage of this promising practice. Furthermore, we illustrate how an agile software development practice helps software developers and customers to collaborate and effectively build software systems.

To achieve our goal, we interviewed 24 individuals working in industry who have practical experience using BDD. BDD is presented as a potential approach to mitigate issues such as requirements understanding, obsolete documentation, and collaboration problems. BDD was reported as a way to identify the behavior of a software system, to promote the communication among different professionals, to trace the scenarios of software usage, and to validate these scenarios. In sum, we found that BDD is seen from four distinct perspectives, namely: BDD as collaboration, BDD as communication, BDD as requirements management, and BDD as testing. More importantly, we observe that these perspectives are possible because the scenarios used in BDD are boundary objects, i.e. "objects which both inhabit several social worlds and satisfy the informational requirements of each of them...is a key process in developing and maintaining coherence intersecting social words" [36]. In other words, BDD scenarios allow different stakeholders (customers, software developers, quality assurance personnel, etc) to share information and successfully collaborate and coordinate their work.

BACKGROUND

Collaboration in Software Engineering

Software development has long been a domain of interest of CSCW researchers [28, 19]. As a matter of fact, since 1988 communication and coordination issues have been regarded as a major problem in large scale software development [11]. This is mainly given the established 'traditional' processes in which a role gathers information, transform it into knowledge by generating artifacts, and passes these ahead to the next role in the pipeline [2]. For instance, in the Waterfall model requirements analysts gather requirements from customer, write them down in the format of specifications, and hand them over to developers. Developers have to understand the specifications and translate them into source code, which is later shared with testers. These professionals have to understand the specifications and translate them into test scripts to verify the source code. Therefore, it is not surprise to find CSCW researchers interested in software development as a collaborative endeavor. Aspects such as communication and shared understanding are at the core of it.

Several papers have been focused on the collaboration around software development. For instance, Halverson et al. [20] and Biehl et al. [7] present tools to support the collaboration among software developers. Meanwhile, Draxler and colleagues [12] and [25] present empirical studies about software teams and their implications for collaborative software development. In particular, a large number of artifacts is produced in a software development process [11] and several of these artifacts are boundary objects. Boundary objects are:

> *(...) objects which are both plastic enough to adapt to local needs and constraints of the several parties employing them, yet robust enough to maintain a common identity across sites. They are weakly structured in common use, and become strongly structured in individual-site use. They may be abstract or concrete. They have different meanings in different social worlds but their structure is common enough to more than one world to make them recognizable, a means of translation. The creation and management of boundary objects is key in developing and maintaining coherence across intersecting social worlds. [36]*

Examples of boundary objects in collaborative software development include problem reports [28], software architectures [33], and even application programming interfaces (APIs) [35]. They all facilitate the communication and coordination in different software development activities.

In addition to collaboration issues, early software development methods have been criticized for being too focused on formal processes, slow to respond to changes in the initial requirements, and not focused on customer engagement [5]. This occurs because the entire software development process is based on the assumption that the requirements for the software system can be identified early in the process, and more importantly, they remain stable. These problems have led practitioners to propose a 'movement' to change this situation. This movement called itself agile.

Agile Software Development

Agile software development aims for effective and appropriate responses to (all kinds of) changes, encourages team structures and attitudes that make communication among team members and business people more effective and facile, emphasizes rapid delivery of operational (or functional) software, adopts the customer as a part of the development teams as a way to bring business and technical worlds together, and recognizes that planning is uncertain and needs to be flexible [27]. Although all the characteristics above have been around for many years, it is just with the Agile Manifesto that they gain strength and draw the attention of a broader community. The Agile Manifesto consists of the following values [5]:

- *Individuals and interactions over processes and tools*
- *Working software over comprehensive documentation*
- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan*

The Agile Manifesto's authors highlight that *"while there is value in the items on the right, we value the items on the left more"*. From this starting point, several agile methods have been defined and are largely used by industry (e.g., Scrum [29], XP [3]). Each of these methods are composed of a set of practices that implement in different ways the agile values and principles expressed in the Agile Manifesto.

CSCW and software engineering researchers have been interested in understanding agile software development. For instance, Sharp and colleagues [31] studied a specific example of an agile method and their associated work practices.Their results illustrate how collaboration takes place on an agile method through specific agile practices including stand-up meetings, the planning game, among others. Bhat and Nagappan discuss how test-driven development (TDD) increases the quality of the source code [6]. Strode et al [38] empirically investigate coordination in agile teams and propose a coordination theoretical model for such context. In their model, several agile practices act as coordination mechanisms, which together form a coordination strategy. Hoda, Noble and Marshall [21] investigate the importance of adequate customer involvement on agile projects. They identified that the among the main cause of lack of customer involvement are lack of commitment, ineffective customer representatives, and skepticism towards agile development. Consequences of the lack of support from customers mostly referred to problems in gathering and clarifying requirements, priotization issues, and pressure to commit to an unclear scope– which are well-known issues in traditional development.

One agile practice that has *not* been empirically studied is BDD, despite its growing market adoption [39]. BDD promotes the engagement of business and technical stakeholders aiming to define better software requirements and to facilitate communication among them. It was originally proposed by Dan North [26] as an easier way to practice TDD–an agile practice that uses unit tests to specify, design, and verify application code [4]. Later it evolved to a set of practices that altogether facilitate that developers shift their mindset from defining what method or function is being tested to what the class should do, making easier for them to focus on the underlying business requirements, i.e., on the system behavior itself. It also allows for business personnel to easily communicate their needs expressed in the format of examples of use described using day-to-day language, reducing communication gaps and knowledge transfer glitches. The ubiquitous language introduced by Eric Evans [14] was then used by North as a means to define requirements unambiguously and transformed these into automated acceptance tests. In practice, BDD allows expressing the acceptance criteria for user stories in the form of loosely structured examples known as 'scenarios', formally written based on an structured language referred as Gherkin [32]. The structured scenarios and the automation to the downstream artifacts in the development cycle associated with automated documentation generation are at the core of the BDD approach.

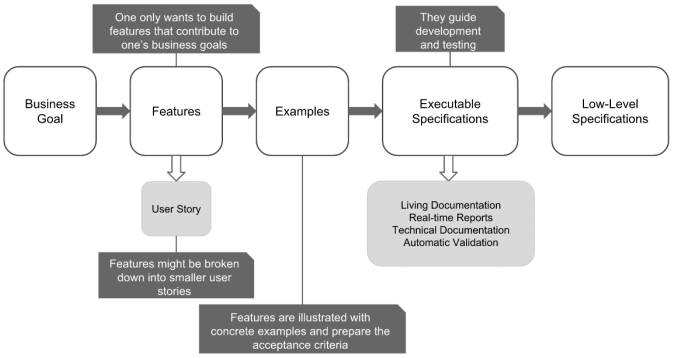

Figure 1: Main BDD activities and outcomes (Source: [32])

BDD in a Nutshell

BDD is a 'label' for a number of practices that altogether helps software teams to focus their efforts on identifying, understanding, and building features based on concrete examples of system behavior defined in collaboration with stakeholders in conversations about what adds value to their businesses. It encourages business analysts, software developers, and testers to collaborate more closely by enabling them to express requirements in a more testable way, in a form that both development team and business stakeholders can easily understand [32].

The elicited requirements (or user stories) are expressed in the format of *BDD scenarios*[1] written based on the Gherkin language. These stories and examples form the basis of the specifications that developers use to build the system, and the overall proposal encapsulates the transformation of examples (or acceptance criteria) into automated acceptance tests, also named 'executable specifications'. In other words, an executable specification is an automated test that illustrates and verifies how the application delivers a specific business requirement [32]. Given that these automated tests are expected to be part of the build process, allowing for automatic propagation of requirements changes to the development team, BDD also offers a mechanism to support up-to-date documentation. This feature is referred as 'living documentation' and is claimed as one of the main advantages of using the entire set of practices that compose BDD. Figure 1 illustrates the complete BDD 'cycle'.

This entire process needs tool support. There are several tools available in the market that can help turn examples into automated tests (or executable specifications). For instance, JBehave, Cucumber, RSpec, and Behat. These are named 'BDD frameworks'. However, there are also tools to support the definition of low level specifications, which express technical specifications describing how the application should behave, e.g. how it should respond to certain inputs or what it should

[1]We use the term 'BDD scenario' to stress the fact that we refer to scenarios written using the Gherkin format. In software engineering the term scenario is also used to express how a system is used applying other notations such as UML.

Feature: Buying a book using the bookstore card
In order to buy a book **As a** bookstore customer **I want to** pay for the book
Scenario 1: Paying with positive bookstore card balance
Given my Bookstore card has a positive balance of USD$ 300.00 And the Bookstore card gives a customer 15% of discount
When my purchase is of 100.00
Then I should pay USD$ 85.00 for the book And I should have USD$ 215.00 left on my Bookstore card
Scenario 2: Paying with insufficient credit on the bookstore card
Given my Bookstore card has a positive balance of UDS$ 50.00 And the Bookstore card gives a customer 15% of discount
When my purchase is of 100.00
Then I should receive an 'insufficient credit' error And I should have UDS$ 50.00 left on my Bookstore card

Table 1: Definition of an example using Gherkin

do in a given situation. These low-level specifications are derived from the (high-level) acceptance criteria and aim to help developers design and document the application code in the context of delivering the business requirement [32]. Examples of frameworks for writing low-level specifications are RSpec, NSpec, and Jasmine. Unit testing tools, JUnit and NUnit, are often also used in combination with these types of frameworks.

The Gherkin Language

A well-known format to write an example, used by most of the BDD frameworks, is recognized as 'the Gherkin language'. Gherkin-based scenarios (or BDD scenarios) are expressed in plain English (or any other language) but with a specific structure. Each scenario is made up of a number of steps, where each step starts with one of a small number of keywords (Given, When, Then, And, and But) as it is illustrated in Table 1.

The plain language and designed format are intended to make it easier for stakeholders to express themselves, while, at the same time being easy for software teams to automate them using software development tools. By using a BDD framework to write the scenarios using the Gherkin language and develop the downstream specifications, a software team can automate a large portion of its activities and ensure up-to-date and easy-to-access documentation to all its members.

Benefits and Challenges of Adopting BDD

Smart [32] lists a number of benefits an organization can expect when adopting BDD, as follows:

- Reduced waste, by focusing development effort on discovering and delivering features that add value to business and avoiding time wasted with those features that do not;
- Reduced costs, which is a direct consequence of the reduced waste;
- Easier and safer changes, by generating living documentation from the executable specifications using terms that stakeholders are familiar with; and
- Faster releases, by implementing automated tests and no longer requiring to carry out long manual testing sessions before each new release.

However, Smart [32] also points out a list of disadvantages and potential challenges of BDD, as follows:

- BDD requires high business engagement and collaboration since its core practices are based on conversations and feedback;
- BDD works best in an agile or iterative context given that it assumes that it is difficult, if not impossible, to define the requirements completely upfront, and that these will evolve as the team learn more about the project; and
- BDD does not work well in a silo, i.e., when organization structures still keep analysts, developers, and testers apart.

It is important to mention that those are *claimed* BDD advantages and challenges. To the best of our knowledge, there are no empirical studies that focus on the usage of BDD. The next section describes the methodology we adopted to conduct such a study.

RESEARCH METHOD

We conducted a qualitative empirical study with IT professionals who practice BDD in industry. More specifically, we conducted a Grounded Theory [37] study using semi-structured interviews for data collection [30] and coding techniques for data analysis [37]. Data collection and analysis were intertwined so that subjects were selected and invited to participate based on the results from previous interviews. After 24 interviews we realized that data collected was providing us with a somehow 'stable' content, therefore we finished the data collection process. We will detail all aspects of our methodology in the following sections.

Data Collection

We recruited 24 subjects through a combination of e-mails sent to our personal contacts, word of mouth (snowball sampling), and announcements on social media websites (e.g., LinkedIn) and local group discussion and mailing lists. We interviewed 8 of these professionals in person and 16 over Skype or Google Hangout given their physical location– distributed across Brazil, North America, and Europe.

The interviews lasted an average of 55 minutes and consisted of two main sections. The first section of the interview was designed to elicit deeper discussions about how BDD is used in practice, including the aspects of our interest, namely: what is the profile of BDD users, how and why they use it, which tools are used to support BDD adoption, and which benefits and challenges of using it are perceived by the practitioners. The second section focused on the respondents demographics and role in the organization, and on the company's characteristics. For example, we asked respondents to tell us about their previous working experience, current job description, and experience using agile methods and BDD. We also asked them whether they would know how to

ID	Role	Ag/BDD(Yrs)	Org	IT?
R1	Test Analyst	8/5	M	Y
R2	Software Engineer	7/4.5	LI	N
R3	Developer	-/4.5	LB	Y
R4	Developer	-/2	LI	Y
R5	Software Engineer	-/5	LB	N
R6	Developer	-/1	LB	Y
R7	Quality Analyst	6/1	LB	N
R8	Quality Coordinator	-/1	LI	Y
R9	Test Analyst	-/2	LB	N
R10	Developer	6/4.5	LI	Y
R11	Lead Business Analyst	7/5	LI	Y
R12	Test Analyst	2/1.5	M	Y
R13	Software Engineer	5/2	LB	Y
R14	Quality Analyst	4/3	LI	Y
R15	Software Architect	6/5	S	Y
R16	Lead Backend Engineer	5/3	M	Y
R17	Test Engineer	5/1	LI	Y
R18	Head of QA Lead	5/3	M	Y
R19	Developer	6/6	LB	N
R20	QA Engineer	5/5	M	Y
R21	Full Stack Engineer	6/4	S	Y
R22	QA Engineer	5/2	M	N
R23	System Analyst	1.5/1.5	LB	Y
R24	System Analyst	5.5/3.5	LI	Y

Table 2: Respondents' Demographics

BDD as	Total
Collaboration	09 = 4 Devs + 4 Testers + 1 Analyst
Communication	12 = 6 Testers + 5 Devs + 1 Analyst
Reqs Mngmt	13 = 5 Devs + 5 Testers + 3 Analysts
Testing	10 = 5 Testers + 4 Devs + 1 Analyst

Table 3: BDD Perspectives

describe their company's main business area (e.g., IT, education, health), number of employees, and main software development methodology (agile or 'traditional'). A summary of our respondents' demographics is presented on Table 2. The ID column indicates the respondent's identification; Role indicates the interviewee's role title; Ag/BDD reports the years of agile and BDD experience in industry, respectively; Org identifies the organization size (S: Small–from 20 to 99 employees, M: Medium–from 100 to 499 employees, LB: Large and Brazilian-based–more than 500 employees and of Brazilian capital, and LI: Large and International-based– more than 500 employees and a multinational enterprise); IT indicates whether the company's core business is IT (Y–Yes) or any other business (N–No). '-' indicates that a time was not filled out by the respondent.

Each identified perspectives is mapped on Table 3, where the Total column means the total number of respondents and their roles on each perspective. More details about the data collection and analysis are described in the following sections.

The interview script was individually validated [24] with three researchers who have industry experience and teach agile development and a practitioner of a multinational company who has been using BDD for over 3 years. Their feedback was considered and a final version of the interview script was prepared. This version was piloted with two practitioners selected by convenience: a test engineer and a software engineering with 3 and 2 years of experience in BDD, respectively. Both pilot interviews were transcribed and coded as a means to validate the script.

We asked subjects to provide consent for us to voice record the interviews and informed them that data collected would remain confidential and anonymous. Data collection and analysis took place interchangeably in an iterative manner following the principles of the Grounded Theory approach [37]. Subjects were selected and invited to participate based on the results from previous interviews, i.e., based on theoretical sampling [16]. For example, after 3 interviews with local people working in medium size companies, some of the usages of BDD puzzled us when contrasted to literature and insights from the pilot interviews; thus we looked for professionals from multinationals with experience in more structured environments. Given the iterative process, after a certain number of interviews we realized that data collected was providing us with somehow 'stable' content (a theoretical saturation [37, 10]). Therefore, our decision to remain with the 24 interviews despite the other 76 people who have volunteered to talk with us about their experiences.

Data Analysis

All interviews were transcribed, and transcriptions were prepared for analysis in the ATLAS.ti qualitative data analysis software [17]. Qualitative coding proceeded using Grounded Theory suggestions: open, axial and selective coding [37]. Immediately after an interview had been conducted or up to 24h later, a debriefing of the interview was prepared. The debriefing followed an script that aimed to capture the main observations and notes from the interviewer, and to promote her reflection upon what has been learned during the interview. The script was composed of 7 questions addressing the following: what were the key points/highlights, what finding was surprising, what was heard from the interviewee that was expected, were there any issues with conducting the interview, what would one ask if one could go back, what has worked well and should be repeated in the following interviews, and what should be changed. The other two researchers would review the interview debriefing up to 24h after it has been made available, and an e-mail-based discussion would take place until consensus on the discussed topics would been reached and a plan established for the next interview(s).

Given the qualitative nature of our study, data analysis is considered interpretative: one of the researchers analyzed each one of the interview script questions at a time and later consolidated the identified code for this question into a single code list for that question [37]. The set of coding lists was reviewed and discussed with the other researchers. During this review process, codes were collapsed or unified when

necessary and a new list generated and revisited. As the coding scheme was refined, we used a technique of comparison against available literature to identify the extent to which our emerging insights were in accordance or in contrast with current knowledge. The final code lists represent our findings– the four perspectives of BDD in practice along with their detailed characteristics, as presented next.

It is important to notice that a single informant might have commented on different metaphors, i.e., there is *not* a mapping from one informant to a unique metaphor. For instance, answers from informant R19 were mapped to two different metaphors: "BDD as collaboration" and "BDD as communication". Another informant, R11, mentioned aspects associated with "BDD as collaboration and "BDD as requirements management" metaphors. In other words, the informants answered our questions, but the metaphors are the result of our analytical work that noticed similarity among different answers and grouped them in themes.

MEANINGS OF THE USAGE OF BDD

During the interviews, we realized that our respondents had different views about BDD. This was confirmed upon our data analysis. For instance, business analysts use BDD centered on the requirements, i.e., as a means to identify, specify, and manage requirements. On the other hand, testers use BDD mainly to validate the developed system against the defined business needs.

We identified four views or perspectives of use about BDD from our dataset. Each represents a different meaning of how BDD is perceived by software professionals in practice. For each perspective, we identified the context of use, including the predominant role that mentioned it; the perceived benefits; the tools that provide support to its use; and the challenges associated to adopting BDD. The four identified perspectives are as follows:

- *BDD as collaboration: how BDD affects and promotes collaboration among team members.*
- *BDD as communication: how BDD impacts oral and written communication.*
- *BDD as requirements management: how BDD supports the identification, specification, and management of requirements.*
- *BDD as testing: how BDD supports the validation a system and of its expected behavior.*

BDD as collaboration

> *"BDD is a tool that helps [a team] to get rid of barriers/issues in agile development, for example, collaboration. In agile, we are expected to collaborate all the time, or, better..., we 'need' to collaborate as early as possible in the development cycle." (R21)*

The first identified perspective of BDD usage, "BDD as collaboration", is related to the collaboration that takes place among all members involved in a project that uses BDD– business stakeholders and software team. This view puts emphasis on closing the collaboration gap with stakeholders, which is well-known as one of the major challenges in software projects. The excerpt below illustrates this:

> *"Well, for me, it [BDD] promotes a development environment focused on the system behavior more than anything else. It allows one to establish a common ground and a foundation for the stakeholders to collaborate with one another, despite their roles. I have used it with QA's, who have a more detailed vision of the system, but also with senior managers and company directors, who see things from the top. I got there, read the stories, and quickly came up-to-date with what was going on, and managed to discuss the system with the stakeholders. They were well aware of what was going on too." (R22)*

This perspective was predominantly reported by respondents who were mainly focused in identifying the expected behavior of the system-to-be instead of worrying about technical specifications of the respective system. These respondents reported that their major concern was first to identify how the system should behave to later wonder about the technical solution and related definitions.

They also reported that by using BDD they managed to work closely together with business people, such as business analysts and product owners, and also with software/technical members, such as developers and testers. For instance, R17, a Test Engineer, argues:

> *"If the business analyst is involved in the definition of the [BDD] scenarios and system behavior, it is easier for the development team understand what she wants to say. In the end, she [the business analyst] has more knowledge than anyone else about what the system should do." (R17)*

R7, a Quality Analyst, reinforced this by saying:

> *"(...) it definitely promotes more collaboration. BDD kind of removes the gap between the quality analysts and the product owner, also between the quality analysts and the customer or the development teams [developers]." (R7)*

We also identified tools often used to support BDD when it is seen as a collaboration mechanism. These tools are: Slack, a tool to support communication and collaboration among those involved in the project (R22); paper and pencil, resources to drawn and explain one's examples to others (R22); and code repositories, a kind of tool to share source code and information about the project (e.g. R8, R9, R11, R21, R24), examples are: GitHub, GitLab, Mercurial, and Subversion. These tools focus on information sharing as a mean to support collaboration.

> *"Slack is a persistent tool. It is just there, notifying one whenever a new message arrives, allowing one to exchange information and experiences with members of the team, this is mainly important since we have remote people sitting in other offices." (R22)*

It is interesting to note that BDD as a collaboration mechanism was reported by respondents playing different roles–

from quality professionals (e.g., R22, R29) to developers (e.g., R10, R19). This collaboration also took place in different tasks–to define the BDD scenarios, to validate the requirements, as a means to close the gap with business people, etc. R11 highlights that geographically distributed projects, something very common in software development, can also benefit from BDD, facilitating the integration among the distributed team members.

The respondents also reported a few challenges related to this perspective. For instance, they mentioned that it is hard to maintain the team talking throughout the project (e.g., R4, R10), to make all members use BDD and reduce the resistance to its adoption (e.g., R12, R13, R8), to engage business people on its use (e.g., R7, R17, R13, R24), and, finally, to identify which scenarios are fit to be described using BDD (e.g., R7, R11, R22).

> *"I believe that BDD helps changing that attitude of 'this is responsibility of the developer, that of the tester, and so on'. It is 'everyone sitting together and investing time on discussing the scenarios'. I am tired of documents being updated here, sent there, that mix up of versions. With BDD, there is no more 'I do this and you do that', it is all part of one single unit. It is the team." (R10)*

> *"There are some members of our team that have not used BDD yet, so it is a bit hard to convince them to switch to it. If they only knew how easy it is to maintain things with it..." (R13)*

BDD as communication

The second perspective is "BDD as communication". Respondents believe BDD is an important mechanism to promote communication in a software project given that BDD uses an ubiquitous language that eases the establishment of common ground and facilitates vocabulary sharing. By not having to invest effort on establishing a communication protocol, the team can mostly focus on identifying, understanding, and defining the system behavior and its related features. The following quotes illustrate the important of the ubiquitous language in how BDD is used, and exemplify the identified perspective.

> *"The acceptance criteria are defined in terms of the Given, When, Then format and we do believe that this improves communication in the team." (R19)*

> *"It is a language that the product owner can easily understand. We can use it in the software team but the customer and an stakeholder can also use. I am a tester and use it. It is like a universal language that helps anyone to understand the requirements and the system specification." (R1)*

> *"The way we communicate has improved a lot with BDD. For instance, even when I have to report a bug, I do use this [language] format. The developers are already used to it, so it just makes it easier. We all use the same way of sharing information back and forth." (R12)*

These quotes reinforce that BDD uses a language that allows for any member, despite if from the software team or from the business stakeholders group, to clearly communicate with one another. It also supports the software team to learn about business terms and jargons, more easily acquiring knowledge to build the expected system and later validating it. Some respondents highlighted that the Gherkin language is mainly useful when discussing features with end users, who are used to speak their 'own' business language.

In addition, some respondents commented that they use the BDD scenarios defined using the Gherkin language as 'contracts' with the customers and as a means to negotiate and report what will be delivered in each release. It is also used to track changes to the agreed scope and easily find what is new.

From the above, it is clear that BDD supports communication and that its ubiquitous language offers a 'clean' way for those involved in the project establish a common ground and exchange information. The excerpts below add to the illustration of this metaphor.

> *"Communication with the customer is now much easier because when one tries to talk using technical terms with a customer the conversation does not flow." (R14)*

> *"It is important for the clients too. It is difficult for them to realize how a requirement is implemented in a system, and using BDD this issue is solved. It is not only the communication, the understanding of the system itself also improves." (R23)*

It is relevant to recall how agile methods reinforce the importance of communication. Therefore, we asked the respondents about which benefits come from using BDD and which from adopting agile methods, and they argued that communication improvements as a side effect of using BDD are much more clear and possible to measure than when considering agile methods as a whole. The example below corroborates this finding.

> *"We started using agile ceremonies [practices] first. Some time later we decided to start using BDD. It was when it became clear to us its benefits: it was the BDD that helped us to have a better communication and not the ceremonies as we expected in the beginning." (R10)*

A set of tools was also indicated to support this metaphor, which are: white board, a resource for drawing and supplementary use along with BDD frameworks to support (e.g., R2, R12); UI prototyping and mock-up tools, as a means to further the understanding of a feature (e.g., R4, R5, R9, R23); mental maps, a tool to organize ideas and words that represent the core concepts of the system and to map and relate BDD scenarios to one another (e.g., R1, R8); and teleconference tools were also cited as resourceful to facilitate team members reaching one another. Furthermore, text editors tools were also mentioned, such as Notepad (e.g., R9, R23), Subway (e.g., R13), GoogleDocs (e.g., R8, R13), and Microsoft Word (e.g., R1, R9). We note that the respondents use tools to support synchronous and asynchronous communication with both co-located and distributed members and for the purpose

of communicating the specifications defined using BDD or for solely reaching other members to share these specifications.

"We use white boards. About 70% of the company walls have white boards, so we just write on the boards or on cards and hang the cards on them [the boards]. We customized the cards so the user stories and the acceptance criteria would go together, facilitating the visualization of the definition of a BDD scenario." (R12)

"Mindmap, a mental map tool, is very useful to help me organize how I will define the tests based on the specifications [from BDD]." (R8)

BDD was reported as a communication mechanism mainly by those responsible for coding and testing the system (e.g., (R2, R6, R12, R17). As proposed by agile methods, some agile teams are cross-functional teams and have members performing several functions. For instance, there were instances of developers reporting that they are also responsible for testing, so, they use BDD to perform the associated tasks and this adoption facilitates the communication with others. R18 and R23, a tester and a business analyst, respectively, highlighted that in order to use BDD, the team has to stay in close touch but this effort pays off.

There are, however, difficulties related to this perspective. Respondents reported that it is not that simple to know what to include in a BDD scenario (e.g., R1, R12, R16) or which BDD scenarios are more important than others (e.g., R14, R23). They also mentioned that it is somehow challenging to have a novice professional writing high quality BDD scenarios (e.g., R4, R5, R11) or even that it is difficult to find detailed study material on the subject (e.g., R6, R18).

"Maybe our difficulty in early stages was to translate the specification steps into actions to be taken [source code]. It is easy to define the BDD scenario. The challenging question is 'how do I translate this now into code'?" (R12)

"When one starts is somehow difficult to know what to do first. Once one gets experience, it is easy to use the Gherkin language and related it to a requirement." (R16)

BDD as requirements management

"If one uses, for example, natural language to define a [BDD] scenario, the scenario is written in a similar form one writes a plain text and this [format] helps even when one has no technical background to realize what the technical notation behind it means. On top of that, there is this, let's say, this technical side of it, that maps the natural language to the execution of the scenario (...)." (R2)

The third perspective we have identified, "BDD as requirements management", is related to the management of requirements. The meanings given by the respondents to this perspective are as follows: it helps on the understanding of the requirements, allows to transform natural language into BDD scenarios, is the source for the estimation of the time, effort, and/or points required to develop the BDD scenarios (e.g., R2, R21). The following quote exemplifies how BDD supports the understanding of the requirements.

"Specification using [BDD] scenarios, acceptance criteria, etc help not only to clarify what the [system] scope is about without giving too much room for erroneous interpretations but also to guide the development process, the automation of testing cases, and the user stories approval." (R21)

The BDD scenarios, which are part of a requirements definition, can be associated to the source code allowing an effective traceability with downstream artifacts (e.g., specification, tests). This association allows for a transparent and sharp monitoring of the progress of the requirements implementation as pointed out next.

"As one can use it [a scenario] embedded in the source code and others can trace it down to [the] regression [tests], it ends up being easy for everyone to work this way. Before BDD, one would have to read the requirements, understand them, transform them into source code, generate the test cases, and it was all spread out here and there. Things would soon be forgotten... Now it is everything in a single place." (R18)

However, sometimes the scenarios are not used along side with the source code. For instance, some respondents reported that there are times that they use (paper) cards to write the scenarios following the structure of the Gherkin language instead of using a tool or text editor. As a consequence, the scenarios are not embedded within the same tool environment in which the code will be generated (e.g., R01, R21).

The tools cited as supporting resources for the use of BDD as a requirements management mechanism are as follows: Sublime and Textmate, text editors for the Gherkin language (R11); Excel, used to trace the scenarios (R2); Kanban board, as a mechanism to share and track the progress of the scenarios definitions when these are written in a 'card' format (e.g., R1, R21); Jira, a dashboard to share user stories, tasks, and acceptance criteria (e.g., R7, R20, R24). Jira is also used to control the progress of the development of the defined BDD scenarios (R18). Other dashboards were also mentioned as follows: Asana (R21), Confluence (e.g., R16, R22), Eiffel (R14), Gemini (R10), Kanbanery (R8), Mingle (R14), Pivotal (R13), Redmine (R23), Tablero (R14), and Trello (R4).

"[We use] Git to look at the BDD scenarios; on-line on the web if we want to." (R18)

"When working as a business person, I used to like to use Excel to write the BDD scenarios, to create tables to help with them and detail the acceptance criteria values for instance." (R2)

In addition to the above mentioned tools, BDD also creates what is called the 'living documentation', which maintains documentation up-to-date at all times. Respondents mentioned that they do take advantage of this documentation (e.g., R11, R14, R16) that is automatically created by BDD frame-

works. There are also respondents (e.g., R2, R10) using Pickles, an open source living documentation generator that works with BDD frameworks such as Cucumber and SpecFlow. In sum, we can see that tools support for this metaphor focus on providing ways to define the scenarios and generate downstream documentation, and to monitor the development of the scenarios, supporting management with a more broad view of the project scope.

Having BDD as a requirements management mechanism was mentioned by all profiles of respondents we interacted with: business analysts (e.g., R11, R24), developers (e.g., R16, R21), and quality professionals–testers, analysts, etc (e.g., R14, R18). They mainly emphasized how important it is to help them manage and ensure the requirements are aligned with the downstream artifacts.

Challenges were also discussed by the respondents. For instance, some respondents reported that it is too costly to explain to the customer what is the added value and how to train her on how to define BDD scenarios (e.g., R3, R14, R19, R24), coming to the point that some teams define them without the customer knowledge (e.g., R2). It is also costly to automate the scenarios in legacy projects (e.g., R16), to identify which can be reused (e.g., R13), and to change the scenarios directly in the source code when they are embedded to it without coordinating back with the business personnel (e.g., R11).

> *"A difficulty we have is to explain to the customer what is the added value behind the BDD scenarios definition. We can sit and define them by ourselves, but it is not effective. We do need to sit with the customer from the beginning." (R14)*

BDD as testing

> *"BDD is, mainly, the definition of the system behavior. This behavior has to be validated with the end user, this is her main responsibility: to know what is the focus of the system." (R8)*

This perspective, "BDD as testing", is perceived given the tool support and automation of scenarios into tests. These tests aim to validate and to ensure that the specified system behavior is aligned with the business goals and needs. But, the automation is not used all the time. Some respondents revealed that they use tests in BDD as a way to verify the expected behavior, instead of focusing on functional specifications or units as often seen in the traditional software development (pre-agile era). One respondent (R4) argued that when looking from the behavior point of view, tests allow for a better overview and understanding of the system than when considering its parts. The excerpt below expresses this.

> *"Instead of creating a unit test for testing a portion of the source code, I write test cases to check a certain behavior. I am now checking the behavior of rules." (R15)*

Although BDD proposes a cycle in which the scenarios give birth to executable specifications, including automated tests, there were instances in which a few respondents reported that scenarios are written by testers as a way to guide test case specifications (e.g., R1). In other words, this means that BDD is not always being used as a whole–from the scenarios definitions as a means to understand the system behavior to its transformation into tests, allowing for traceability and for the generation of living documentation.

Tool support was considered crucial in this perspective. The tools mentioned by the respondents are as follows: Appium (e.g., R12), Selenium UpDriver (e.g., R8), Selenium Grid (e.g., R8) and JMeter (e.g., R8) are used for test automation; and Team city (e.g., R2) and Jenkins (e.g. R12, R16, R20, R24) are used for the generation of testing progress report, including of those tests that automated.

Similarly to the "BDD as a requirements management" perspective, all three interviewed profiles–business analysts (e.g., R24), developers (e.g., R5 and R15), and quality professionals also mentioned this perspective (e.g., R20 and R22). For example, respondent R15 mentioned that his team of testers translated the system behavior into test cases and then validated them according to the BDD philosophy.

As on the previous perspectives, there were also difficulties reported here. For example, some respondents consider that it is difficult to know what tests should be automated since in some testing suites testing the large number of test cases gets too slow (e.g., R1, R2, R20,R21). There were those who reported that BDD scenarios are not well defined and thus testing gets impacted (e.g., R4, R15) and that the BDD framework choice might not have been proper for the project context and thus leading testing to be redone and not interconnected with specifications (e.g., R15).

> *"It gets challenging for the business analyst to write BDD scenarios that can be easily automated by developers and testers, and it is just like this." (R21)*

DISCUSSION

The research question this paper aims to answer is: *"What BDD is used for and what are the related characteristics?"*. Our results, presented in the previous section, illustrate that software developers use BDD to achieve four different goals: collaboration, communication, requirements management, and testing. Each one of these aspects is called a BDD perspective. Table 4 presents a brief summary of these four views.

It is important to mention that the perspectives we identified are interrelated, i.e., they are *not* independent of each other. Specifically, the *communication* perspective means that when adopting BDD, communication is simpler for the stakeholders within the software development company as well as for these stakeholders and their users. This happens because BDD scenarios have a structured format, the Gherkin format, that is unique to all stakeholders involved, therefore, these scenarios create a shared reference [22] about the customers requirements. A consequence of the communication being more effective is the increased collaboration among stakeholders, which is the first perspective we described: BDD as collaboration. This collaboration takes place in different stances, but, as our results in the previous section illustrate, it is especially important around a specific set of artifacts,

BDD metaphor	Example Quote	Context of use	Tools	Challenges (-) and Benefits (+)
BDD as collaboration	"BDD is a tool that helps [a team] to get rid of barriers/issues in agile development, for example, collaboration. In agile, we are expected to collaborate all the time, or, better..., we 'need' to collaborate as early as possible in the development cycle." (R21)	Used by those focused in identifying the expected behavior of the system-to-be instead of worrying about technical specifications of the respective system.	Slack Paper and pencil Code repositories	- To maintain the team talking throughout the project - To make all members use BDD and reduce the resistance to its adoption - To engage business people on its use - To identify which scenarios are fit to be defined using BDD + To define the scenarios + To validate the requirements + As a means to close the gap with business people + To facilitate integration among distributed team members
BDD as communication	"It is a language that the product owner can easily understand. We can use it in the software team but the customer and an stakeholder can also use. I am a tester and use it. It is like a universal language that helps anyone to understand the requirements and the system specification." (R1)	Used to align and discuss the system behavior through spoken, visual, and written communication.	White board UI prototyping and mock-up tools Mental maps Teleconference Text editors	- To know what to include in a scenario - To know which scenarios are more important than others - To have a novice professional writing high quality scenarios - To find detailed study material + To support clear communication + To facilitate the learning of business terms and jargons
BDD as requirements management	"Specification using scenarios, acceptance criteria, etc help not only to clarify what the [system] scope is about without giving too much room for erronecus interpretations but also to guide the development process, the automation of testing cases, and the user stories homologation." (R21)	Used to map, create, and follow-up their scenarios or their development.	Text editors Worksheet Dashboards Living documentation by BDD frameworks or Pickles	- To justify the high cost to use BDD - To automate the scenarios in legacy projects - To change the scenarios directly in the source code when they are embedded to it without coordinating back with the business personnel + To help on the understanding of the requirements + To estimate the effort and/or points required to develop the scenarios + To monitor the progress of requirements
BDD as testing	"Instead of creating a unit test for testing a portion of the source code, I write test cases to check a certain behavior. I am now checking the behavior of rules." (R15)	Used to validate the system behavior.	Automated testing tools Continuous integration software tool	- To know what tests should be automated to avoid the suite testing get slow - To use scenarios were not well defined will impact the tests + To validate and to ensure that the specified system behavior is aligned with the business goals and needs + To verify the expected behavior, instead of focusing on functional specifications or units + To allow for a better overview and understanding of the system + To way to guide test case specifications

Table 4: Summary of the four identified BDD perspectives

the BDD scenarios. These scenarios are associated with the third and fourth BDD views respectively, BDD as requirements management and BDD as testing. In other words, BDD scenarios facilitate the management of requirements, as well as the testing activities. In summary, BDD scenarios allow customers, management personnel (e.g., product owner), developers and quality assurance personnel (e.g., testers, quality analysts, etc) to effectively interact and ultimately develop software.

The fact that BDD scenarios allow effective communication among different software development roles needs further discussion, since as we mentioned before, this still is a major problem in agile and traditional software development [2]. The different software development roles include the customer and others associated with the different tasks required to develop software namely, requirements, implementation, and quality control. These roles are oftentimes associated with organizational boundaries, i.e., stakeholders with different roles belong to different organizational units. And, as previous CSCW research has shown [1], collaboration across organizational boundaries is more difficult. Carlile and Rebentisch [8] argue that one of the reasons for this difficulty is related to the *knowledge boundaries*, i.e., the boundaries that exist when there are different and specialized knowledge domains and the individuals of these domains need to interact and exchange information. They illustrate this problem by discussing how engineers from different groups (styling, engine and power-train, climate control, and safety) need to interact to design a new vehicle [8]. Furthermore, according to them, there are different types of boundaries. In a *syntactic* boundary, the differences and dependencies among actors are known and the knowledge is not so specialized. In this scenario, a common syntax or knowledge is sufficient to specify the differences and dependencies, and then the information can simply be transferred across the boundary. A *semantic* boundary occurs when different knowledge domains generate differences in interpretation, thus creating discrepancies of meanings. In this case, besides a common syntax, it is necessary to establish a common semantic in order to identify and translate these different interpretations and dependencies among the different knowledge domains. Finally, when the knowledge domains are so specialized that they generate different interests, building common knowledge to be shared among the groups becomes a political process. This is an example of a *pragmatic* boundary. In this case, both groups need to adjust and transform their current way of performing their activities and their knowledge in order to accommodate the knowledge from the other group and thus to collaborate in the boundary. In other words, to learn, one needs to transform her own knowledge [8].

Our results suggest that BDD scenarios work as *semantic boundaries*[2] between the different stakeholders involved in

[2]In addition to have a common syntax, based on the Gherkin format, BDD scenarios also have a semantics associated to them.

the development process. For instance, requirements engineers interact with the customers to construct the initial scenarios, which are then handed over to developers who will implement these scenarios, while the quality control (e.g., test engineers) will create test cases and other documents according to these same scenarios. It is important to mention that BDD scenarios are a *single* type of artifact in contrast to the *diverse* types of documentation from traditional software development, e.g., requirements documents, requirements specifications, etc. The fact that the same type of artifact is used helps team members reducing knowledge loss when transforming definitions from one phase (e.g., requirements) to another (e.g., coding). In short, BDD scenarios are artifacts that facilitate the communication and collaboration in software development, i.e., they are boundary objects [36] in collaborative software development. As mentioned in the Background section, there are other examples of boundary objects in software development, namely problem reports, software architectures, and APIs. However, these previous examples focus solely on the software development team, while BDD scenarios are interesting because they also involve *customers*, i.e., those whose software is being developed for. By doing so, BDD scenarios address not only the software requirements phase, which is regarded as one of the most important and problematics phases in software development [34], but also provide the added benefit of customer collaboration [13].

From a theoretical point of view, it is interesting to notice that new agile practices are being proposed by practitioners, i.e., agile methods are still evolving. Furthermore, our study illustrates how BDD was created to facilitate the communication among different stakeholders and, how it indeed "implements" boundary objects, i.e., BDD scenarios are artifacts that bridge different organizational worlds and are, at the same time, plastic and robust so that they can be successfully used by stakeholders from these different worlds. It would be interesting then to study other agile practices (e.g., planning game, continuous integration, stand-up meetings, retrospectives, etc) to find out whether and how they successfully, or not, facilitate collaboration. In other words, are there additional agile practices that "implement" boundary objects and therefore facilitate communication across organizational boundaries? Are there roles associated with additional agile practices that "implement" boundary managers similarly to what was reported by [15]? If not, as collaboration researchers, can we help software developers to successfully design such agile practices and associated artifacts? We believe these are important implications for CSCW research.

To sum up, according to our respondents, the adoption of BDD brings several advantages to software development teams including better communication and collaboration, better management of requirements, and easier testing activities. However, adopting BDD is not unproblematic. Our respondents reported some challenges that we described in the previous section. From a CSCW research point of view, the most important is the developers' difficulty in convincing customers and other software developers to adopt BDD. This seems to be related to the "classical" issue in CSCW about who does the (extra) work and who benefits from it? [18], i.e., customers and developers are concerned about having to do an extra-work (adopt BDD) without clearly seeing the possible benefits of this agile practice.

FINAL REMARKS

Software development has long been recognized as an example of collaborative work. More recently, agile methods, a somewhat new software development paradigm, have drawn attention of the scientific community because their values are very different from the traditional, process-oriented software development approaches [13]. Agile methods emphasize individuals and interactions, customer collaboration, working software and responsiveness to changes.

Agile methods are composed from different agile practices, i.e., ways of working. Agile method experts choose, and adapt the agile practices they will use in their projects and organizations. Somewhat recently, a new agile practice drawn the attention from software developers, the Behaviour-Driven Development, or simply BDD [26, 32]. BDD is a 'label' for a set of work practices to facilitate software development through the identification, understanding and creation of features based on scenarios. These scenarios are concrete examples of system behavior and of what adds value to their businesses. They are defined in collaboration with stakeholders.

There are several claimed benefits of BDD. However, since it is fairly new [32], there is a lack of empirical studies about it in the literature. In this paper we describe an empirical study of BDD conducted using qualitative methods. Specifically, 24 semi-structured interviews were conducted with practitioners that use, or recently have used BDD. Our results suggest that practitioners see BDD in four different ways: BDD as a means of collaboration among team members, including the customer; BDD as mechanism that facilitates communication; BDD as an approach to identify, specify, and manage requirements throughout the software development cycle; and finally, BDD as an approach that facilitate the validation of the software system with its expected behavior. Each one of these views has associated advantages, challenges, and associated tools. Seeing BDD as these four perspectives helps one to realize its full potential. Our study also illustrates *how BDD implements these perspectives*, i.e., BDD scenarios are boundary objects that facilitate the communication and coordination of software development activities.

It is important to keep in mind that this study is based on interview data, i.e., on the informants' report, which is limited. Therefore, it would important to conduct an empirical study of BDD usage through an ethnographic approach where a researcher could observe how the usage of BDD unfolds during the software development process. This would provide a rich understanding of how BDD is used "in the wild" allowing us to understand the limitation of current tools, and therefore, help tool developers to designer better tools. Another possibility is adopting a more quantitative approach, like a survey, to understand how BDD is being used in different organizations (with or without customers? for collaboration, communication, requirements management or testing?) around the

world. In fact, we plan to conduct such a large-scale survey with additional software developers to confirm the identified perspectives and, potentially, identify new ways in which BDD is seen by professional software developers.

REFERENCES

1. Mark S. Ackerman. 2000. The Intellectual Challenge of CSCW: The Gap Between Social Requirements and Technical Feasibility. *Human-Computer Interaction* 15, 2 (2000), 179–203. DOI: http://dx.doi.org/10.1207/S15327051HCI1523_5
2. Gojko Adzic. 2009. *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing*. Neuri Limited, London, UK. 258 pages.
3. Kent Beck. 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, USA. 224 pages.
4. Kent Beck. 2002. *Test-Driven Development: By Example*. Addison-Wesley, Boston, USA. 176 pages.
5. Kent Beck and *et al.* 2001. Manifesto for Agile Software Development. (2001). http://www.agilemanifesto.org/
6. Thirumalesh Bhat and Nachiappan Nagappan. 2006. Evaluating the Efficacy of Test-driven Development: Industrial Case Studies. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*. ACM, Rio de Janeiro, Brazil, 356–363. DOI: http://dx.doi.org/10.1145/1159733.1159787
7. Jacob T. Biehl, Mary Czerwinski, Greg Smith, and George G. Robertson. 2007. FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1313–1322. DOI: http://dx.doi.org/10.1145/1240624.1240823
8. Paul R. Carlile and Eric S. Rebentisch. 2003. Into the Black Box: The Knowledge Transformation Cycle. *Management Science* 49, 9 (2003), 1180–1195. DOI: http://dx.doi.org/10.1287/mnsc.49.9.1180.16564
9. Jan Chong and Rosanne Siino. 2006. Interruptions on Software Teams: A Comparison of Paired and Solo Programmers. In *Proceedings of the Conference on Computer Supported Cooperative Work*. ACM, Banff, Canada, 29–38. DOI: http://dx.doi.org/10.1145/1180875.1180882
10. J.W. Creswell. 2007. *Qualitative Inquiry and Research Design: Choosing Among Five Approaches*. SAGE Publications, Lincoln, USA. 472 pages.
11. Bill Curtis, Herb Krasner, and Neil Iscoe. 1988. A Field Study of the Software Design Process for Large Systems. *Commun. ACM* 31, 11 (Nov. 1988), 1268–1287. DOI: http://dx.doi.org/10.1145/50087.50089
12. Sebastian Draxler, Gunnar Stevens, Martin Stein, Alexander Boden, and David Randall. 2012. Supporting the Social Context of Technology Appropriation: On a Synthesis of Sharing Tools and Tool Knowledge. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 2835–2844. DOI: http://dx.doi.org/10.1145/2207676.2208687
13. Tore Dyba and Torgeir Dingsøyr. 2008. Empirical Studies of Agile Software Development: A Systematic Review. *Journal Information and Software Technology* 50, 9-10 (Aug. 2008), 833–859. DOI: http://dx.doi.org/10.1016/j.infsof.2008.01.006
14. E. Evans. 2003. *Domain-Driven Design: Tackling Complexity In the Heart of Software*. Addison-Wesley, Boston, MA, USA. 563 pages.
15. Mayara Figueiredo, Cleidson De Souza, Marcelo Zilio, Rafael Prikladnicki, and Jorge Audy. 2014. Knowledge Transfer, Translation and Transformation in the Work of Information Technology Architects. *Information and Software Technology* 56, 10 (2014), 1233–1252. DOI: http://dx.doi.org/10.1016/j.infsof.2014.04.001
16. B.G. Glaser and A.L. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company. 271 pages.
17. Scientific Software Development GmbH. 2002. ATLAS.ti: The Qualitative Data Analysis. (2002). http://atlasti.com/
18. Jonathan Grudin. 1988. Why CSCW Applications Fail: Problems in the Design and Evaluationof Organizational Interfaces. In *Proceedings of the 1988 ACM Conference on Computer-supported Cooperative Work*. ACM, New York, NY, USA, 85–93. DOI: http://dx.doi.org/10.1145/62266.62273
19. Jonathan Grudin. 1994. Computer-Supported Cooperative Work: History and Focus, Los Alamitos, USA. *Computer* 27, 5 (May 1994), 19–26. DOI: http://dx.doi.org/10.1109/2.291294
20. Christine A. Halverson, Jason B. Ellis, Catalina Danis, and Wendy A. Kellogg. 2006. Designing Task Visualizations to Support the Coordination of Work in Software Development. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*. ACM, New York, NY, USA, 39–48. DOI: http://dx.doi.org/10.1145/1180875.1180883
21. Rashina Hoda, James Noble, and Stuart Marshall. 2011. The Impact of Inadequate Customer Collaboration on Self-organizing Agile Teams. *Information and Software Technology* 53, 5 (2011), 521–534. DOI: http://dx.doi.org/10.1016/j.infsof.2010.10.009
22. L. Horn and G. Ward. 2004. *The Handbook of Pragmatics*. Blackwell Handbooks Ltd. 864 pages.

23. H. Hulkko and P. Abrahamsson. 2005. A multiple case study on the impact of pair programming on product quality. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 495–504. DOI: http://dx.doi.org/10.1109/ICSE.2005.1553595

24. Barbara Kitchenham and Shari Lawrence Pfleeger. 2002. Principles of Survey Research Part 4: Questionnaire Evaluation. *SIGSOFT Software Engineering Notes, New York, USA.* 27, 3 (May 2002), 20–23. DOI: http://dx.doi.org/10.1145/638574.638580

25. Stina Matthiesen, Pernille Bjørn, and Lise Møller Petersen. 2014. "Figure out How to Code with the Hands of Others": Recognizing Cultural Blind Spots in Global Software Development. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*. ACM, New York, NY, USA, 1107–1119. DOI: http://dx.doi.org/10.1145/2531602.2531612

26. Dan North. 2006. Introducing BDD. (2006). http://dannorth.net/introducing-bdd/

27. Roger Pressman. 2010. *Software Engineering: A Practitioner's Approach* (7 ed.). McGraw-Hill, Inc., New York, USA.

28. Kjeld Schmidt and Carla Simonee. 1996. Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. *Computer Supported Cooperative Work (CSCW)* 5, 2 (1996), 155–200. DOI: http://dx.doi.org/10.1007/BF00133655

29. Ken Schwaber and Jeff Sutherland. 2014. Scrum guide. (2014), 01–16.

30. Carolyn B. Seaman. 2008. *Guide to Advanced Empirical Software Engineering*. Springer London, London, Chapter Qualitative Methods, 35–62.

31. Helen Sharp and Hugh Robinson. 2004. An Ethnographic Study of XP Practice. *Empirical Software Engineering* 9, 4 (2004), 353–375. DOI: http://dx.doi.org/10.1023/B:EMSE.0000039884.79385.54

32. John Smart. 2014. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications, Shelter Island, NY. 384 pages.

33. Kari Smolander, Matti Rossi, and Sandeep Purao. 2008. Software architectures: Blueprint, Literature, Language or Decision? *European Journal of Information Systems* 17, 6 (2008), 575–588. DOI: http://dx.doi.org/10.1057/ejis.2008.48

34. Ian Sommerville. 2010. *Software Engineering* (9 ed.). Addison-Wesley, Harlow, England. 792 pages.

35. Cleidson R. Souza and David F. Redmiles. 2009. On The Roles of APIs in the Coordination of Collaborative Software Development. *Computer Supported Cooperative Work* 18, 5-6 (2009), 445–475. DOI: http://dx.doi.org/10.1007/s10606-009-9101-3

36. Susan Leigh Star and James R Griesemer. 1989. Institutional ecology,translations' and boundary objects: Amateurs and professionals in Berkeley's Museum of Vertebrate Zoology, 1907-39. *Social Studies of Science* 19, 3 (1989), 387–420. DOI: http://dx.doi.org/10.1177/030631289019003001

37. A.L. Strauss and J.M. Corbin. 1990. *Basics of qualitative research: grounded theory procedures and techniques*. Sage Publications. 272 pages.

38. Diane E. Strode, Sid L. Huff, Beverley Hope, and Sebastian Link. 2012. Coordination in Co-located Agile Software Development Projects. *Journal of Systems and Software* 85, 6 (2012), 1222–1238. DOI: http://dx.doi.org/10.1016/j.jss.2012.02.017

39. Versione. 2015. *State of Agile Survey*. Technical Report 9. VersiOne Agile Made Easier. http://info.versionone.com/state-of-agile-development-survey-ninth.html

40. Corrado Aaron Visaggio. 2005. Empirical Validation of Pair Programming. In *Proceedings of the International Conference on Software Engineering*. ACM, St. Louis, USA, 654–654. DOI: http://dx.doi.org/10.1145/1062455.1062588