# Combining Behaviour-Driven Development with Scrum for Software Development in the Education Domain

Pedro Lopes de Souza[1], Antonio Francisco do Prado[1], Wanderley Lopes de Souza[1],
Sissi Marilia dos Santos Forghieri Pereira[2] and Luís Ferreira Pires[3]

[1]*Department of Computing, Federal University of São Carlos, PO Box 676, 13565-905, São Carlos, São Paulo, Brazil*

[2]*Department of Medicine, Federal University of São Carlos, PO Box 676, 13565-905, São Carlos, São Paulo, Brazil*

[3]*Centre for Telematics & Information Technology, University of Twente, PO Box 217, 7500AE, Enschede, The Netherlands*

Keywords: Software Engineering, Methodologies for System Development, BDD, Scrum, ICT in Education, PBL.

Abstract: Most of the Brazilian universities employ teaching-learning methodologies based on classic frontal lectures. The Medicine Programme of the Federal University of São Carlos (UFSCar) is an exception, since it employs active learning methodologies. The Educational and Academic Management System for Courses Based on Active Learning Methodologies (EAMS-CBALM) was built and it is currently used to support this programme, and has been made available for other programmes as well. This system was developed using Scrum, but during its development project it was often necessary to reconsider system behaviour scenarios, and consequently the product backlog items, mainly due to poor communication between the Product Owner (PO) and the development team. This paper discusses a case study in which Behaviour-Driven Development (BDD) has been used in combination with Scrum to redesign some EAMS-CBALM components. The paper demonstrates that the communication between the PO and the development team can be improved by using BDD as a communication platform to unambiguously define system requirements and automatically generate test suites.

## 1 INTRODUCTION

Until recently, most universities still employed traditional teaching-learning methodologies based on classic frontal lectures (Weltman, 2007). The Medicine Programme of the Federal University of São Carlos (UFSCar) has broken this trend, by introducing active learning methodologies in its curriculum. This programme was established in 2006, and follows a socio-constructivist educational approach, has a competency-oriented pedagogical program, and employs active learning methodologies, such as Problem-Based Learning (PBL) (Rhem, 1998), and Practice-Based Learning (Carlisle et al., 2009).

In order to provide computational support to these active learning methodologies, the Ubiquitous Computing Group (UCG) of UFSCar coordinated the development of the Educational and Academic Management System for Courses Based on Active Learning Methodologies (EAMS-CBALM) (Santos et al., 2016). EAMS-CBALM was developed using Scrum (Schwaber and Sutherland, 2016), which prescribes, amongst others, *sprint reviews*, which are meetings involving the development team and *Product Owner* (*PO*) to define the *product backlog* and evaluate the sprint results. In this project, it was often necessary to redefine some system behaviour scenarios, and consequently the product backlog items, due to misunderstanding of the stories reported by the PO.

Behaviour-Driven Development (BDD) is an approach to software development that prescribes the definition of usage scenarios (behaviour specifications) upfront, as a way to better understand what the software is supposed to do (its 'behaviour') (North, 2006). This paper discusses a case study in which we combined BDD with Scrum in order to address the problems mentioned before, in which we redesigned some EAMS-CBALM components. This paper demonstrates that the communication between the PO and the development team can be improved quite a lot by applying BDD in combination with Scrum.

This paper is further structured as follows: Section 2 introduces the UFSCar Medicine Programme, Section 3 describes the original EAMS-CBALM architecture, Section 4 presents the case study, Section 5 discusses the benefits and drawbacks of our results, Section 6 presents some related work and Section 7

Souza, P., Prado, A., Souza, W., Pereira, S. and Pires, L.
Combining Behaviour-Driven Development with Scrum for Software Development in the Education Domain.
DOI: 10.5220/0006336804490458
In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017) - Volume 2*, pages 449-458
ISBN: 978-989-758-248-6

gives our conclusions.

## 2 UFSCar MEDICINE PROGRAMME

The UFSCar Medicine Programme curriculum is based on educational activities organised in three educational units, as shown in Figure 1: Education Unit of Simulation of Professional Practice (EUSPP), Education Unit of Professional Practice (EUPP), and Education Unit of Elective Activities (EUEA) (UFSCar, 2007). Figure 1 also shows that simulation practice is gradually replaced by professional practice throughout the programme.

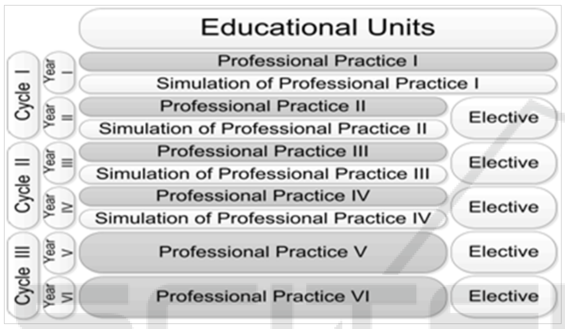

Figure 1: UFSCar Medicine Programme curriculum.

The educational activities of the UFSCar Medicine Programme have learning triggers, which are problems that simulate or portray the daily activities to be performed by the students. These triggers activate the constructivist spiral (Tempski, 2014), which encourages students to reflect and stimulate them to develop their capacities. For each learning trigger, the students traverse the constructivist spiral, starting by identifying the problem, formulating explanations, preparing learning questions, looking for new information, building new meanings, and finally evaluating the process.

## 3 EAMS-CBALM

The Educational and Academic Management System for Courses Based on Active Learning Methodologies (EAMS-CBALM) (Santos et al., 2016) was developed by the UCG/UFSCar, in partnership with the DMed/UFSCar, the Teaching and Research Institute (TRI) of the Sírio-Libanês Hospital (SLH), and the TokenLab software house. EAMS-CBALM was developed using Scrum (Schwaber and Sutherland, 2016).

### 3.1 Development Process

In Scrum, the units of work are divided in *sprints*, which can last between a week and a month, period in which the development team creates an increment of the product to be delivered to the user. The development team must have a *Product Owner* (*PO*), who is designated by the client to follow the development process, which in a nutshell consists of: sprint planning; product delivery at the end of each sprint; and final product delivery, properly deployed and tested.

Scrum emphasises the interactions between the users and the development team, mainly when the requirements are established and selected in user stories, in order to obtain a fast delivery and a satisfactory quality of the product. Most of the clarifications and details regarding the product under development are obtained during these interactions, allowing product adaptations to be done quickly, thus avoiding bottlenecks and delays and reducing the likelihood of unsatisfactory results.

Throughout the one-year development of the EAMS-CBALM, weekly meetings were held at TokenLab in São Carlos-SP (Brazil) for the planning and analysis of the different phases of this project, involving the UCG/UFSCar coordinator, the coordinator of the TokenLab development team, two POs designated by TRI/SLH and DMed/UFSCar (the product clients), respectively, and a PhD student. Monthly sprint review meetings were held at TRI/SLH in São Paulo-SP (Brazil) to present the products resulting from the sprints, involving the participants of the TokenLab meetings and TRI/SLH members directly or indirectly related to the project. In addition, the PhD student participated in courses offered by the DMed/UFSCar and the TRI/SLH to observe the teaching-learning process of these courses.

During the TokenLab meetings, the POs reported user stories informally in Portuguese, describing the activities to be performed by the EAMS-CBALM users. From these user stories, system requirements were captured and specified also informally in Portuguese. Using these requirement specifications, the development team defined system behaviour scenarios and implemented the screen pages for these scenarios, which were presented and discussed at the next meeting. A total of 109 hours have been spent with these meetings, in which the CBALM teaching-learning process was scrutinised, resulting in the definition of the functional and non-functional system requirements, and the overall system architecture.

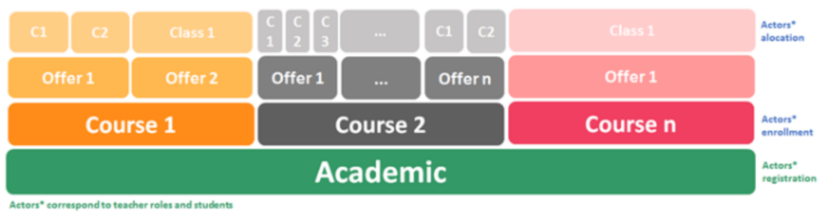

Figure 2: EAMS-CBALM academic module.

## 3.2 System Architecture

EAMS-CBALM has two main functional modules: academic and pedagogical. The academic module, shown in Figure 2, allows its users to define the programme structure, the CBALM community, and the curriculum. The programme structure is used for creating courses, where each course can be offered several times, and each course offer can have several classes (like, e.g., 'the class of 2016'). The CBALM community consists of students and teachers. A teacher can play the following roles: facilitator in curricular activities, supervisor, preceptor, consultant, appraiser, author of simulated practices and of educational units, manager of educational units, working groups or educational resources. Registered students can be enrolled in courses, and then assigned to groups. The curriculum is used to create a curriculum structure with at least two and at most five hierarchical levels. For example, the UFSCar Medicine Programme curriculum, which is shown in Figure 1, has five levels: cycles, years, educational units, curricular activities and educational actions.

The pedagogical module, shown in Figure 3, reflects the curricular structure over each class of each course previously recorded in the academic module. The educational actions are programmed in the pedagogical module, giving rise to the educational environment in which students and teachers perform and record their activities. The planning of an educational action starts with the meeting preparation, where each meeting corresponds to a step of the constructivist spiral, and ends with the evaluation preparation.

EAMS-CBLAM allows students to discuss through forums, exchange text documents, images, videos, and audios fragments, create individual or collaborative documents, store and retrieve document versions, and share the knowledge produced by other students and teachers. All these resources contribute to the student capabilities development leading to knowledge production.

Another EAMS-CBALM feature regards data visualisation, since all educational content produced during the teaching-learning process is provided by means of a curricular trajectory. This content ranges from the teaching contribution to the individual and collaborative knowledge produced by the students.

The curricular trajectory can be viewed in a timeline or be associated with an educational unit, allowing students and teachers to visualise all developed practices in the most adequate way at each moment. In addition, the produced knowledge is indexed by keywords, facilitating the search of any content created throughout the teaching-learning process.

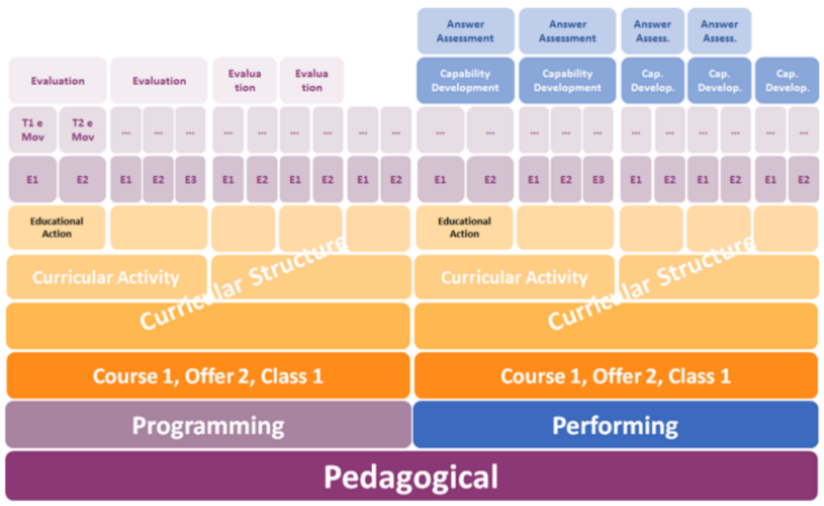

Figure 3: EAMS-CBALM pedagogical module.

## 4 CASE STUDY

During the EAMS-CBALM development the non-functional requirements were defined according to 5 qualitative design criteria: Performance, Interoperability, Portability, Responsiveness and Security. The functional requirements of this system were organized into 19 modules: Authentication, General Settings, Access Profiles, Users, Course, User Registration in Class, Calendar, Programming, Groups, Pedagogical Planning, Evaluation Management, Trajectory, Repository of Group Documents, Repository of Individual Documents, Forums, Meetings, Frequency, Evaluations, and Performance Record.

The Evaluation Management module generated the most controversies between POs and developers, since the CBALM student evaluation is different and more complex than the traditional one. This module is composed of 14 functional requirements, and although the scenarios of Evaluation Instrument Register requirement have been redone several times, its final implementation did not fully satisfy the POs. Therefore, we selected this module in this case study, by focusing on the CBALM student evaluation process. Furthermore, one of the POs that was involved in the EAMS-CBALM development participated in this case study. By keeping regular meetings with this PO, we developed new source code for the Evaluation Instrument Register requirement by combining BDD with Scrum. This new source code was presented to the PO and compared with the implementation originally produced by TokenLab.

### 4.1 Behaviour-Driven Development

Test Driven Development (TDD) (Beck, 2002) is a software evolutionary development methodology, based on short development cycles, in which automated tests are described previously to the functional code. Acceptance Test Driven Development (ATDD) (Koskela, 2008) extends TDD by using acceptance tests to represent stakeholders requirements.

Behaviour-Driven Development (BDD) is a software agile development methodology, originally proposed by Dan North (North, 2006) and considered an evolution of TDD and ATDD, whose fundamental principle is: “stakeholders and developers should refer to the same system in the same way”. For achieving this goal, an ubiquitous language is required that is understandable by all those involved in system development and that enables executable granular specifications of the system’s behaviour and testing (Diepenbeck and Drechsler, 2015).

Six main characteristics of BDD were identified in (Solis and Wang, 2011): ubiquitous language; iterative decomposition process; user story and scenario templates; automated acceptance testing with mapping rules; readable behaviour-oriented specification code; and behaviour-driven at different phases. Using these characteristics, (Solis and Wang, 2011) also analyses seven of the main BDD tools: NBehave (NBehave, 2014) and JBehave (JBehave, 2015); MSpec (MSpec, 2016) and RSpec (RSpec, 2016); StoryQ (StoryQ, 2010); Cucumber (Cucumber, 2016); and SpecFlow (SpecFlow, 2016).

BDD is an evolving methodology that does not have a clear definition and unanimous understanding, and the existing support tools focus mainly on the implementation phase, providing limited support to the requirements gathering, analysis, and design phases of software life cycle. Starting from (Solis and Wang, 2011), we did a systematic review of BDD-related work, and an analysis of the current BDD toolkits. Figure 4 shows the BDD process depicted using the Structured Analysis and Design Technique (SADT) diagrammatic notation.

The purpose of this case study has been to experiment with the combination of BDD and Scrum, aiming at exploiting the benefits of this combination. During a Scrum sprint planning meeting involving POs and the development team, requirements are listed according to their priority and added as user stories to the product backlog. The development team then decides which stories are tackled during the sprint, and creates a sprint backlog with the tasks to be performed in the sprint. In order to avoid rework, a clear understanding of these requirements and the corresponding functional behaviour is necessary. According to (Chauhan, 2016) BDD can be used for this purpose if it takes a central role in some Scrum artefacts and rituals: the product backlog and the sprint backlog; the daily scrum meeting; and the sprint meeting. In addition, according to (Malik, 2013) BDD can also be applied in backlog refinement. In the sequel, we discuss how we combined BDD and Scrum in our case study by discussing each of the stages of the BDD process in Figure 4.

### 4.2 Ubiquitous Language

A ubiquitous language is essential in BDD. It must have a structure derived from a business domain model, offer a terminology that is understandable to both clients and developers, be used in all system development phases (Evans, 2004). Therefore, a ubiquitous language had to be created for the communication between the POs and developers.

Most of the ubiquitous language terminology

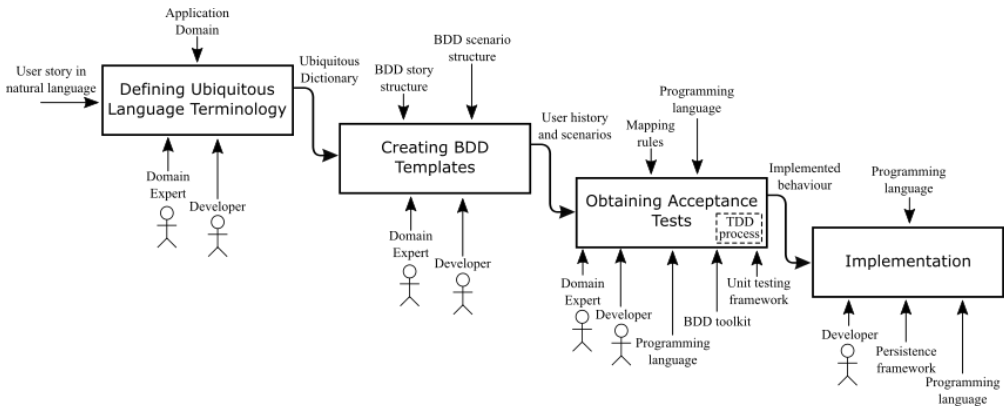

Figure 4: BDD process in SADT.

should be defined in the analysis phase, but new terms can be added at any time and at any phase. In the design and implementation phases, this terminology is used to name classes and methods, making the code clearer and better readable. BDD itself has a simple pre-defined ubiquitous language for the analysis phase, which is domain-independent. This language is used to formulate the user stories and scenarios.

In the first meetings with the PO we defined the terminology employed in the CBALM student evaluation process. Evaluations are performed by all involved in CBALM activities, expressing their perceptions, indicating the relevant aspects, and aspects that need to be improved, reworked or replaced. Two evaluation types are supported, namely formative and summative evaluations, and the possible results to be received/given are ‘satisfactory’, ‘needs improvement’ and ‘unsatisfactory’. The student who does not reach a satisfactory result must undertake an improvement plan proposed by the teacher, and then be re-evaluated (UFSCar, 2007). Evaluations are consolidated by applying a set of instruments, which are described in the following story reported by the PO:

“In order for the course coordinator to carry out a student evaluation, six instrument types have to be previously registered. This requirement is needed because the evaluation form heading changes according to the employed instrument. When registering an instrument, the system must keep the following information: name, acronym and the relationship between who responds to the evaluation, who evaluates and who is evaluated. This last information is crucial because in conjunction with the curricular activity it defines which form type is applied when registering an evaluation. The six instrument types are:

(a) Performance Assessment of the Teaching-Learning Process (acronym: PATLP) - It is formative. Teacher evaluates student. The Respondent is the teacher. A PATLP has 3 steps: teacher evaluation, classmate evaluation, and improvement plan if necessary. The PATLP also allows a Self-Evaluation (SE). SE skills are essential for the success of every professional in maintaining professional competence according to the teaching-learning process.

(b) Reflective Portfolio (RP) - It is formative and summative. The monitoring of each student’s portfolio by the teacher is part of the formative assessments. The summative evaluation refers to its preparation and presentation according to pre-established delivery dates that must be recorded into the system.

(c) Written Examination (WE) - It is summative. Each WE is composed of questions created by the teacher, answered by the student, and then assigned by the teacher. Each question is individually analyzed in order to determine the student progress. All question must have a ‘satisfactory’ result for the student to pass the WE. Questions that failed to get a ‘satisfactory’ result are considered as progress deficit and will be worked out in the next WE.

(d) Progress Test (PT): It is formative and summative. A PT consists of multiple choice questions. The teacher monitors the performance of each student’s performance in a PT as part of the formative assessments. A PT is also summative because the students presence at a PT gives them a ‘satisfactory’ result.

(e) Objective and Structured Evaluation of Professional Performance (OSEPP) - It is summative. Instead of questions like WE, the students act in clinical cases and are evaluated by the teacher. The OSEPP evaluation is similar to the WE evaluation.

(f) Problems Based Exercise (PBE) - It is formative. A PBE assesses the student's individual ability to study and identify health needs, formulate patient problems, and propose a healthcare plan for a particular context and problem situation."

Based on this user story, we defined the main ubiquitous language terminology for the CBALM student evaluation process. Table 1 summarises the terminology that has been employed in the development of the Evaluation Instrument Register, showing the relationships between the involved actors.

Table 1: Ubiquitous language terminology for Evaluation Instrument Register.

| Instrument type | Respondent | Appraiser | Appraisee |
|---|---|---|---|
| PATLP | Teacher | Teacher | Student |
| PATLP (SE) | Student | Student | Student |
| RP / PT<br>WE / PBE<br>OSEPP | Student | Teacher | Student |

## 4.3 BDD Templates

The BDD analysis phase starts by identifying the most expected system behaviours, which are derived from the business outcomes to be produced by the system. Based on these business outcomes, feature sets are defined, where each feature set can contain subsets, and indicates what should be accomplished to achieve a specific business outcome. When combining BDD with Scrum we prescribe that the POs and the development team should agree on the feature set, and ideally define it together.

Features are expressed through user stories, and describe the interactions among a user and the system. A user story must elucidate the users role in this story, the feature desired by the user, and the benefit gained by the user if the system provides the desired feature. Due to different contexts, a user story may have different versions that will lead to different story instances, called scenarios, which should describe specific contexts and outcomes of this user story. When using Scrum, this should be provided by the POs.

A scenario describes how the system that implements a given feature should behave when it is in a main context with possible additional contexts that could be represented by additional conditions. When an event happens, e.g., a system entry, the scenarios result can be one or more actions that change the state of the system or produces some system output.

This initial analysis may be sufficient for a first implementation, even if important system behaviours are yet undisclosed. However, as new behaviours are revealed, the whole process illustrated in Figure 6 can be performed again, thus characterising the "iterative decomposition process" (Solis and Wang, 2011).

BDD user stories and scenarios follow the predefined templates described in (North, 2006) by employing a simple ubiquitous language. However, BDD tools generally do not exactly follow these models. For example, JBehave supports a slightly different user story template than the one proposed in (North, 2006), but the same scenario template. In addition, JBehave supports most of the BDD characteristics presented in (Solis and Wang, 2011), it is well-accepted and widely used in the BDD community, it is open source software, and is frequently updated. Those are the main reasons that have led us to choose JBehave in this case study.

The JBehave user story template is as follows

```
Narrative: [story title]
In order to [benefit]
As a [role]
I want to [feature]
```

where **Narrative:**, optionally followed by a *story title*, describes an activity to be performed by a user; **In order to** describes the *benefit* obtained by the user once the activity is performed; **As a** defines the *role* played by the user on that story; and **I want to** defines the *feature* provided by the system that allows the user to perform the activity.

The JBehave user story for our case study is the following:

```
Narrative: Evaluation Instrument
Register
In order to create the student
evaluation form header
As a course coordinator
I want to previously register all
instrument types into the system
```

The JBehave scenario template is as follows

```
Scenario: [scenario title]
Given [main context]
And [additional contexts]
When [specific event]
Then [main outcome]
And [additional outcomes]
```

where **Scenario:**, followed by a *scenario title*, describes how the system that implements a given feature should behave; **Given** defines the system *main context* that could be represented by a system state; **And** is an optional clause that defines *additional contexts* that could be represented by additional conditions; **When** defines a *specific event* that could be some system input; **Then** defines the *main outcome*, which could be an action for changing the system state; **And** is an optional clause that defines *additional outcomes* that could be other actions, for example, to produce system outputs. This template is similar to an Extended Finite State Machine (EFSM) model (El-Fakih et al., 2016).

There are six scenarios in our case study, one for each instrument type. The JBehave scenario for the PATLP instrument is the following:

```
Scenario: Register the PATLP fields
into the system
Given the name Performance Assessment
of the Teaching-Learning Process
And the acronym is PATLP
And the respondent and appraisee is a
student and the appraiser is a
teacher
When the user select register
Then the system register the
instrument
And the PATLP form header for student
evaluation is created
And the Evaluation Instrument list is
updated
```

## 4.4 Acceptance Tests

A BDD acceptance test is an executable specification of the system behaviour, which verifies the interactions or behaviours of objects rather than their states. The produced scenarios are translated into tests that guide the implementation. A scenario is composed of several steps, where each step is an abstraction that represents the three elements of a scenario: context, event and action. The meaning of these elements is: in a particular case of a user history or context *C*, when the event *X* happens, the system response should be *Z*. Each step is mapped to one test method, and the scenario passes only if all its steps pass. Each step follows the TDD's process: red, green, and refactoring to make it pass.

Since in BDD all scenarios must be executed automatically, the acceptance criteria must also be imported and analysed automatically. The classes that implement the scenarios read their specifications, which are written in the ubiquitous language, and execute them. Therefore, the mapping between scenarios and testing code needs to be explicitly defined.

There are six acceptance tests in our case study, one for each scenario. The most relevant excerpts of the JBehave acceptance test code for the PATLP scenario are shown below

```
public class InstrumentRegisterSteps
                            extends Steps{
   ... // setting variables and methods to
       // support the scenarios
   @Given("the name is Performance Assessment
          of the Teaching-Learning Process")
   public void setInstrumentName(String
                           instrumentName){
  ... //set instrument name }
   @Given("the acronym is PATLP")
   public void setInstrumentAcronym(String
                                 acronym){
      ... //set instrument acrynom }
   @Given("the respondend and appraisee is
           the student and the appraiser
                          is the teacher")
   public void setRelationship(Boolean
                 sameRespondentAndAppraisee){
      ... //set relationship }
   @When("the user select register")
   public void addInstrument() {
      ... //perform action }
   @Then("the system register the instrument")
   public void saveInstrument() {
      ... //data persist }
   @Then("the PATLP form header for student
     evaluation is created")
   public void instrumentHeader() {
      ... //creating and persisting headers }
   @Then("the Evaluation Instrument list
                              is updated")
   public void instrumentListUpdate() {
     ...//Evaluation instrument list updated }
... // next scenario }
```

JBehave uses the *@Given*, *@When* and *@Then* annotations to relate scenario specification clauses to Java methods, and the Java class that implements these methods should extend the *Steps* class. JBehave allows the scenario to be executed as a JUnit test (JUnit, 2016). The link between JBehave's executor framework and the textual scenarios is provided by the *Embeddable* class definitions. This class extends class *JUnitStory* and its name can be mapped to the textual story filename.

## 4.5 Implementation

In BDD the code must be readable, contain the specification, describe the behaviour of the objects and be part of the system documentation. The classes and methods names must be sentences, and the names of the methods must indicate their functionality. Mapping rules assist in the production of readable behaviour-oriented code and ensures that classes and methods names are the same as user story titles and scenarios, respectively. In addition, these names make use of the ubiquitous domain-specific language defined in the project. A behaviour-oriented code of the JBehave implementation for the Evaluation Instrument Register requirement can be seen below.

```
public class EvaluationInstrumentRegister{
// variables
   public void setInstrumentName(String
                         instrumentName){
```

```
   //set instrument name }
public void setInstrumentAcronym(String
                              acronym){
   //set instrument acrynom }
public void setRelationship(Boolean
                         relationship){
   //set relationship }
public void addInstrument() {
   //perform action }
public void saveInstrument() {
   //data persist }
public void instrumentHeader() {
   //creating and persisting headers }
public void instrumentListUpdate() {
     //Evaluation instruments list updated
   } ... }
```

Since the EAMS-CBALM was implemented according to the Model-View-Controller (MVC) pattern, splitting the Model layer into a Business layer and an Integration and Persistence layer, in our case study we followed this same pattern and employed a similar architecture to be able to compare both codes. These layers were implemented with the Eclipse Communication framework services, and PostgreSQL was used for persistence.

## 5 DISCUSSION

This case study exposed many benefits of combining the BDD process stages with Scrum. The use of a ubiquitous language derived from the Education domain, and the definition of a CBALM terminology understandable to both PO and developers avoided misinterpretations of user stories, since they provided a common language and vocabulary for better understanding user requirements. The use of the BDD user story template for describing the Evaluation Instrument Register requirement of the EAMS-CBALM, and the BDD scenario template for describing the scenarios related to the instrument types of the CBALM student evaluation process allowed to naturally transform this requirement into software functional behaviour. Furthermore, JBehave enabled the automatic generation of the BDD acceptance tests for the instrument types of the CBALM student evaluation process, and allowed to obtain clearly written and easily understandable tests to developers, which could also be inspected by the PO, improving their communication. Finally, JBehave facilitated the BDD implementation of the Evaluation Instrument Register requirement, allowing us to produce a readable behaviour-oriented code to developers, which could also be understood by the PO, improving their communication.

This case study also confirmed that the BDD process stages are beneficial when employed in the some Scrum artefacts and rituals (Chauhan, 2016). In the *product backlog* and *sprint backlog*, we used the CBALM ubiquitous language and terminology, the BDD user story for the Evaluation Instrument Register requirement, and the BDD scenarios for the instrument types of the CBALM student evaluation process, which provided a common BDD platform to transform this requirement into functional behaviour in the product backlog, and to transform this functional behaviour into a implementable specification. In the *daily scrum meetings*, this common BDD platform improved the understanding of each activity involved in the Evaluation Instrument Register development, avoiding confusion and bridging the knowledge gap among the developers, and resulting in a final implementation of this requirement that fully satisfied the PO. Finally, in *sprint meetings*, since all developers and PO are committed to define the Evaluation Instrument Register requirement, the produced BDD documents became the reference to discuss the behaviour that is required by the users of the Evaluation Instrument Register product, improving in this way the traceability of what was required, why it was needed, and who has requested it.

All these benefits and improvements to Scrum were possible mainly due to the use of the ubiquitous domain-specific language and BDD templates. Although the developers needed to learn another language and to deal with more documentation, the use of this common platform during all BDD stages allowed the PO to follow them in a better way, improving quite a lot the communication with developers.

In this case study, we performed a qualitative comparison of Scrum and BDD in combination with Scrum with respect to the communication between the PO and the development team, since it is quite hard to organise a formal experiment to compare this communication in a quantitative way. Nevertheless, we have the following evidence that our hypothesis that communication has been improved holds: it took 03 meetings of around 03 hours each between the PO and the Tokenlab developers for delivering the Evaluation Instrument Register product, and it also took 03 meetings but of only around 01h20min each between the same PO and the developers of this case study for delivering the same product. This shows that the development team could be more productive, thus it was less disturbed by miscommunication overhead. However, we need to take into account that rebuilding the same component has favoured the Scrum/BDD combination, but the positive results obtained in this case study have motivated the developers to apply BDD to all the requirements of the EAMS-CBALM Evaluation Management module.

## 6 RELATED WORK

In (Lubke and Van Lessen, 2016) a platform is presented for integrating systems responsible for land registration in Switzerland. The goal is to reduce process execution time between systems and also the communication time between POs. In order to model executable integration processes between various systems and to develop test cases to validate these processes, the authors used BDD and Business Process Model Notation (BPMN) as the ubiquitous language for the test case models (scenarios) construction. The BPMN was chosen because this language is known by the POs. By combining BDD with BPMN scenarios, the authors found improvements in communication between developers, users and investors, which contributed significantly to the more agile development of the platform.

In (Oruç and Ovatman, 2016) a tool is proposed to facilitate the creation of scenarios for web services test. The tool uses BDD in conjunction with the ubiquitous language Gherkin (Wynne and Hellesoy, 2012) for dynamically generating test scripts. These scripts are run in JMeter, a test tool for analysing and measuring the performance of web applications (JMeter, 2016). The authors claim to achieve two benefits with this tool: because Gherkin is a domain-specific language, it allows any domain expert to create and run web services tests even without software knowledge; and developers do not need to write unit tests manually since JMeter automates testing.

In (Silva et al., 2016) an approach based on BDD is proposed to support the automated assessment of artefacts along the development process of interactive systems. A formal ontology model is defined for describing concepts used by platforms, models and artefacts that compose the design of interactive systems, allowing in this way a wide description of User Interface (UI) elements and its behaviours to support testing activities. In addition, the approach proposes improvements in how teams must write requirements for testing purposes. Once described in the ontology, the behaviours can be reused freely to write new scenarios in natural language, providing test automation in BDD and decreasing manual coding.

In (Soeken et al., 2012) a methodology is presented to assist developers carrying out the BDD steps. It proposes a design flow where the developer engages in a dialog with a computer program in an interactive way. This dialog contains the user history, and this program processes each spoken sentence and generates the step definitions and code blocks (classes, attributes and methods) of each scenario. Some natural language processing tools are explored and a case study illustrates the application. Rather than going manually through the established BDD steps, this methodology suggests some scenario skeletons for tests refinement and implementation.

The main difference between our work and the ones above is that in addition to applying BDD in software development our work also exposes the benefits of using BDD in combination with Scrum. Moreover, our case study was developed for the Education domain, more specifically for courses based on active learning methodologies, and to the best of our knowledge the combination of BDD and Scrum in the development of software systems for the Education domain has not been addressed before in the literature.

## 7 CONCLUSIONS

In order to accomplish this work, first we did a systematic review on BDD and its toolkits, and properly defined the BDD development process. Then, we conducted a case study, focusing on the Evaluation Instrument Register requirement of the EAMS-CBALM Evaluation Management module. By keeping regular meetings with the same PO that was involved on the EAMS-CBALM development, we generated a new source code for this requirement following the BDD process. This new source code was presented to the PO and compared with the corresponding one produced by the TokenLab software house.

This case study showed that the BDD process improved the communication between the PO and the developers with respect to the Scrum method employed in the EAMS-CBALM development. That was mainly due to the use of a ubiquitous language for the education domain together with the BDD scenarios and acceptance tests, which allowed the PO to follow and properly communicate with the developers throughout the development process.

Furthermore, this case study also showed that BDD needs more support for the requirements gathering, analysis, and design phases of software life cycle. As future work, we intend to develop formal support for the ubiquitous language terminology, for example, by means of ontologies (Guizzardi, 2005), in order to eliminate the ambiguities intrinsic to natural languages as a BDD ubiquitous language. We also intend to define a formal model based on EFSM (El-Fakih et al., 2016) in order to improve the BDD iterative decomposition process. Finally, we intend to evaluate this BDD formal support by applying it to system development for the Education domain.

## REFERENCES

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Carlisle, C., Calman, L., and Ibbotson, T. (2009). Practice-based learning: The role of practice education facilitators in supporting mentors. *Nurse Education Today*, 29(7):715–721.

Chauhan, R. (2016). Fusion of Agile-Scrum to BDD. http://docplayer.net/28500425-Fusion-of-agile-scrum-to-bdd.html. Accessed: 2016-12-19.

Cucumber (2016). Cucumber simple, human collaboration. https://cucumber.io. Accessed: 2016-12-19.

Diepenbeck, M. and Drechsler, R. (2015). *Behavior Driven Development for Tests and Verification*. Springer Fachmedien Wiesbaden, Wiesbaden.

El-Fakih, K., Yevtushenko, N., Bozga, M., and Bensalem, S. (2016). Distinguishing extended finite state machine configurations using predicate abstraction. *J. Software Eng. R and D*, 4:1.

Evans, E. (2004). *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

Guizzardi, G. (2005). *Ontological foundations for structural conceptual models*. PhD thesis, University of Twente, Enschede.

JBehave (2015). JBehave - a BDD framework. http://jbehave.org. Accessed: 2016-12-19.

JMeter (2016). JMeter graphical server performance testing tool. http://jmeter.apache.org. Accessed: 2016-12-19.

JUnit (2016). JUnit - a framework to write repeatable tests. http://junit.org/junit4. Accessed: 2016-12-19.

Koskela, L. (2008). *Test Driven: Practical TDD and Acceptance TDD for Java Developers*. Manning Pubs Co Series. Manning.

Lubke, D. and Van Lessen, T. (2016). Modeling test cases in bpmn for behavior-driven development. *IEEE Software*, 33(5):15–21.

Malik, Z. (2013). Value of Behavior-Driven Development for Backlog Refinement in Scrum. https://www.scrumalliance.org/community/articles/2013/march/value-of-behavior-driven-development-for-backlog-r. Accessed: 2016-12-19.

MSpec (2016). MSpec - Context/Specification framework. https://github.com/machine/machine.specifications. Accessed: 2016-12-19.

NBehave (2014). NBehave - a BDD framework. https://github.com/nbehave/nbehave. Accessed: 2016-12-19.

North, D. (2006). Introducing BDD. https://dannorth.net/introducing-bdd. Accessed: 2016-12-19.

Oruç, A. F. and Ovatman, T. (2016). Testing of web services using behavior-driven development. In *CLOSER 2016 - Proceedings of the 6th International Conference on Cloud Computing and Services Science, Volume 2, Rome, Italy, April 23-25, 2016.*, pages 85–92.

Rhem, J. (1998). Problem-based learning: an introduction. www1.udel.edu/pbl/deu-june2006/supplemental/NTLF-PBL-introduction.pdf. Accessed: 2016-12-19.

RSpec (2016). RSpec: behaviour driven development for ruby. http://rspec.info/. Accessed: 2016-12-19.

Santos, H. d. F., de Souza, W. L., do Prado, A. F., and Pereira, S. M. d. S. F. (2016). Augmented reality approach for knowledge visualization and production (arakvp) in educational and academic management system for courses based on active learning methodologies (eams–cbalm). In Latifi, S., editor, *Information Technology: New Generations: 13th International Conference on Information Technology*, pages 1113–1123, Cham. Springer International Publishing.

Schwaber, K. and Sutherland, J. (2016). The definitive guide to scrum: The rules of the game. http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf. Accessed: 2016-12-19.

Silva, T., Hak, J.-L., and Winckler, M. (2016). Testing prototypes and final user interfaces through an ontological perspective for behavior-driven development. *Lecture Notes in Computer Science*, 9856:86–107.

Soeken, M., Wille, R., and Drechsler, R. (2012). Assisted behavior driven development using natural language processing. *Lecture Notes in Computer Science*, 7304:269–287.

Solis, C. and Wang, X. (2011). A study of the characteristics of behaviour driven development. In *Proceedings of the 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, SEAA '11, pages 383–387. IEEE Computer Society.

SpecFlow (2016). SpecFlow - Cucumber for .NET. http://specflow.org. Accessed: 2016-12-19.

StoryQ (2010). StoryQ - a bdd framework for .net 3.5. https://storyq.codeplex.com/. Accessed: 2016-12-19.

Tempski, P. (2014). "Caderno do Curso Educação na Saúde para Preceptores do SUS". Teaching and Research Institute of the Sírio-Libanês Hospital (in Portuguese).

UFSCar (2007). Curso de Medicina - CCBS Projeto Político Pedagógico Medicina UFSCar. http://www.prograd.ufscar.br/cursos/cursos-oferecidos-1/medicina. Accessed: 2016-12-19.

Weltman, D. (2007). *A Comparison of Traditional and Active Learning Methods: An Empirical Investigation Utilizing a Linear Mixed Model*. University of Texas at Arlington.

Wynne, M. and Hellesoy, A. (2012). *The cucumber book : behaviour-driven development for testers and developers*. The pragmatic programmers. Dallas, Tex. Pragmatic Bookshelf.