Information and Software Technology 141 (2022) 106717

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Supporting refactoring of BDD specifications—An empirical study

Mohsin Irshad [a,b,*], Jürgen Börstler [a], Kai Petersen [a]

[a] *Blekinge Institute of Technology, Karlskrona, Sweden*
[b] *Ericsson AB, Karlskrona, Sweden*

ARTICLE INFO

Keywords:
Refactoring
Normalized Compression Distance (NCD)
Normalized Compression Similarity (NCS)
Reuse
Similarity ratio (SR)
BDD
Behavior-driven development
Specifications
Testing

ABSTRACT

Context: Behavior-driven development (BDD) is a variant of test-driven development where specifications are described in a structured domain-specific natural language. Although refactoring is a crucial activity of BDD, little research is available on the topic.
Objective: To support practitioners in refactoring BDD specifications by (1) proposing semi-automated approaches to identify refactoring candidates; (2) defining refactoring techniques for BDD specifications; and (3) evaluating the proposed identification approaches in an industry context.
Method: Using Action Research, we have developed an approach for identifying refactoring candidates in BDD specifications based on two measures of similarity and applied the approach in two projects of a large software organization. The accuracy of the measures for identifying refactoring candidates was then evaluated against an approach based on machine learning and a manual approach based on practitioner perception.
Results: We proposed two measures of similarity to support the identification of refactoring candidates in a BDD specification base; (1) normalized compression similarity (NCS) and (2) similarity ratio (SR). A semi-automated approach based on NCS and SR was developed and applied to two industrial cases to identify refactoring candidates. Our results show that our approach can identify candidates for refactoring 6o times faster than a manual approach. Our results furthermore showed that our measures accurately identified refactoring candidates compared with a manual identification by software practitioners and outperformed an ML-based text classification approach. We also described four types of refactoring techniques applicable to BDD specifications; merging candidates, restructuring candidates, deleting duplicates, and renaming specification titles.
Conclusion: Our results show that NCS and SR can help practitioners in accurately identifying BDD specifications that are suitable candidates for refactoring, which also decreases the time for identifying refactoring candidates.

1. Introduction

Behavior-driven development (BDD) was initially proposed to facilitate the understanding of software requirements and link those requirements to test cases [1,2]. Nowadays, it is emerging as a separate software development process [3]. BDD overcomes the information loss between user stories and test cases by writing requirements in a structured natural language format that can be instrumented and used in automated testing. Furthermore, the information loss between the requirements and testing stages is reduced by providing a shared vocabulary for business analysts, developers, testers, and managers while discussing requirements. In BDD, the required functionality is defined in the form of scenarios [4]. A BDD scenario consists of (i) a specification (requirements) written using structured natural language and (ii) hooks to the test code for validating the specification [4].

Refactoring is a technique to improve software artifacts using small behavior preserving transformations [5]. Besides improved maintainability, software practitioners mention improved readability as the most critical refactoring benefit [6]. Researchers have demonstrated that refactoring can be performed manually by the practitioners, semi-automatically or automatically, with tool support [7,8]. Refactoring is introduced as a separate phase in BDD to improve the maintainability of the artifacts [9].

BDD specifications are executable, human-readable specifications that are frequently modified in agile development, resulting in lowering the maintainability of specifications [9]. Maintaining these specifications necessitates uncomplicated and automated (or semi-automated) methods to increase the system's reliability. Existing studies have identified that duplication exists in BDD specifications and it leads to low

* Corresponding author at: Blekinge Institute of Technology, Karlskrona, Sweden.
E-mail address: mohsin.irshad@bth.se (M. Irshad).

https://doi.org/10.1016/j.infsof.2021.106717
Received 22 December 2020; Received in revised form 10 August 2021; Accepted 11 August 2021
Available online 20 August 2021

maintainability of the resulting specification [10]. Duplication can be interpreted as missed opportunities for refactoring or reuse.

Researchers have suggested that refactoring is a fundamental part of BDD, and limited work is present on BDD specifications refactoring [11, 12]. Most of the existing refactoring approaches utilize IDE-based tools to analyze the code fragments ("hooks") that are associated with BDD specifications. Code duplication in the hooks are identified as maintainability issues. A limited number of approaches (such as [13]) take into account the BDD specifications' contents for detecting duplication. Furthermore, the existing literature neither provides specific techniques for refactoring BDD specifications nor identifying candidates for refactoring.

In this study, we propose measures to identify similar BDD specifications and a semi-automated approach to perform refactoring on these similar BDD specifications for improving the maintainability of specifications. This study is not aimed at other BDD aspects like development processes, test tools, or test management. This study is conducted in an industrial context based on BDD specifications from two products and software practitioners' feedback. The objectives of this study are:

- to provide a semi-automated approach to support refactoring of BDD specifications, thus improving maintainability;
- to evaluate the accuracy of the automated parts of the proposed approach in the industrial context (using practitioners and Machine learning based text classification).

The remainder of the paper is organized as follows. Section 2 provides backgrounds and related works on BDD and refactoring. Section 3 describes the research approach, research questions, and approaches for data collection and analysis. Section 4 answers the research questions and important findings are discussed in Section 5. Threats to validity and conclusions are highlighted in Sections 6 and 7, respectively.

2. Background and related work

This section briefly describes some background and related work on BDD and refactoring of BDD specifications.

2.1. Behavior-driven development

BDD starts with the identification of business requirements and describing them in a common textual language for the target domain [3]. These requirements reflect the expected behavior of the system and are then used as test cases to validate these requirements [14]. BDD helps in reducing problems with software requirements, like information loss and lack of common understanding [15] and thereby focusing on the actual goals, i.e. building the "right" software. According to Solís and Wang [3], the following six characteristics are commonly associated with BDD:

- BDD uses a ubiquitous language using terminology from the business domain.
- BDD supports an iterative decomposition process for transforming business goals into user stories and scenarios.
- BDD supports understandable feature descriptions in plain text by means of templates for user stories and scenarios.
- BDD supports automated acceptance testing with mapping rules to map scenarios to test code (hooks).
- BDD supports readable behavior-oriented specification code by means of executable human readable specifications.
- BDD is behavior-driven at different phases of software development (planning, analysis/test and implementation).

North proposed to describe requirements in the form of scenarios comprising three parts or steps [1]: (1) Pre-conditions are stated in *Given*, (2) *When* describes the role(s)/requirement(s) of the actor acting in the scenario, and (3) *Then* describes the action and/or outcome that is expected after completion. This format facilitates using the scenarios as test specifications for validating the requirements. The format of a BDD specification is described below:

Given the initial setting
When some event occurs
Then provide some outcomes

All three steps can have sub-steps joined together by "AND". Each of these steps and sub-steps is linked to a method, called hook, that can be written in any programming language and can therefore be implemented, refactored, or reused using approaches for code-based test cases [4,16]. Each time a BDD step or sub-step is invoked, it invokes the corresponding hook that performs the work associated with the BDD step or sub-step. An example BDD specification with sub-steps (a customer buying a mobile phone) is shown below:

Given the customer contacts agent at the shop
And he has a valid ID card
When he selects a mobile phone from stock
And customer pays the money
Then the customer gets the mobile phone

Fig. 1 describes a BDD process according to Borg and Kropp [9]. The process starts when a customer requests a feature from a product manager. In the next step, the product manager develops BDD scenarios (test specifications) in collaboration with relevant stakeholders (product managers, architects, development resources, etc.). In step 3, hooks are implemented for all steps and sub-steps of the scenarios. The hooks perform actions such as sending requests, receiving responses and asserting results. Initially all of these tests fail. In the fourth step, product code is developed to make all BDD specifications pass.

The BDD specifications are automated to run each time the code base is changed. The success or failure of this step determines how many features are working and therefore the progress of the development cycle. In step 5, refactoring is applied to the specification base and the product code to improve the product's maintainability and quality. In the last step, if all specifications are "green", the feature is released.

2.2. Related work: Refactoring of BDD specifications

Refactoring is an integral part of BDD [9,11]. Santos et al. [17] suggested that BDD should have the following phases "red, green, and refactor". Initially, all scenarios should fail. Once the product code is developed successfully, specifications would become "green". The last part is refactoring, where the specifications and corresponding hooks are improved in structure and complexity. Lai and Chu [18] claim that the refactoring phase of BDD facilitates a better and more understandable description of the behaviors that need to be tested. They suggested an approach to reduce the coupling of security requirements with the help of refactoring using a BDD-based process, successfully increasing a system's security. Bruschi et al. [11] incorporated refactoring of BDD scenarios as an integral part of their BDD process and claim that their BDD process increased collaboration and communication among quality engineers and business analysts. The study did not mention the details of refactoring techniques or refactoring processes of BDD specifications.

In a systematic literature review on BDD, Egbreghts [12] identified that refactoring is needed as a separate phase for BDD and suggested that better tools might improve the use of BDD in the industry. Borg and Kropp [9] claim that changes are introduced frequently in agile development, and therefore the specification base is changed very often. Modifying a specification base that has low maintainability is therefore challenging. To increase its maintainability, they introduced

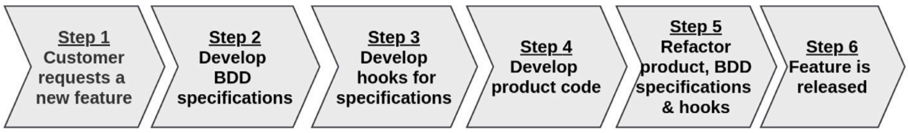
Step 1
Customer requests a new feature
Step 2
Develop BDD specifications
Step 3
Develop hooks for specifications
Step 4
Develop product code
Step 5
Refactor product, BDD specifications & hooks
Step 6
Feature is released

Fig. 1. A BDD process [9].

FIT tables[1] to support the refactoring of BDD specifications and implemented an Eclipse plugin to support it. They conceptually evaluated their automated refactoring approach using a hypothetical example. However, the approach has not been assessed in an industrial context. Bures et al. [20] proposed a method to identify repeated behaviors in BDD specifications and suggested that such specifications be refactored to reduce repetition. They introduced a tool, TestOptimizer, to identify reusable code fragments in BDD hooks, i.e., the approach was applied to the "glue code" joining the BDD specifications with the execution tool. They tested their approach using open-source projects and reduced a large specification base to a smaller one without reducing its test coverage. However, the approach was not directly applied to the BDD specifications but utilizes the similarity of test methods (hooks) to perform the refactoring.

In their study on micro-service architectures, Rahman and Gao [21] suggested that the high maintenance cost of BDD specifications is the primary reason of development teams for not using automated acceptance tests. They proposed a reuse-based method that allows the sharing of specifications among many micro-services to reduce refactoring time. However, their approach was based on the hooks (i.e. test code) and not applied to the BDD specifications. Sathawornwichit and Hosono [22] suggested that meta-data can be used to align changes required during refactoring in the BDD context, e.g., when the product code is changed, corresponding BDD specifications may need to be changed. The study does not describe a way to identify the candidates for refactoring, and the approach was not evaluated in the industry.

To sum up, refactoring is a crucial aspect of the BDD specification life cycle, but few studies discuss the refactoring of BDD specifications. The majority of the existing refactoring approaches are based on the refactoring or reuse of the hooks (test code), using meta-data of the system to trigger refactoring of specifications when the code is changed, or using an IDE-plugin for FIT tables. Few approaches are evaluated in an industrial context and only simple examples are used for a proof of concept evaluation.

In this paper, we propose a new approach for refactoring in BDD that works directly on BDD specifications and evaluate (the measures for identification of refactoring candidates) in two industrial cases from the telecommunications domain.

3. Research method

This section describes the context of our research, the research approach (Action Research) and the threats to validity.

3.1. Research context

The research was conducted inside a large scale software organization that is developing business support systems for telecommunication organizations. The organization uses hybrid agile development methodologies consisting of Scrum and XP practices to develop, deliver, and maintain software solutions. We studied two products of different functionality developed for different customers. In one product (case 1), the robot framework[2] is used as the BDD framework while the other product (case 2) uses Cucumber[3] as a specification framework.

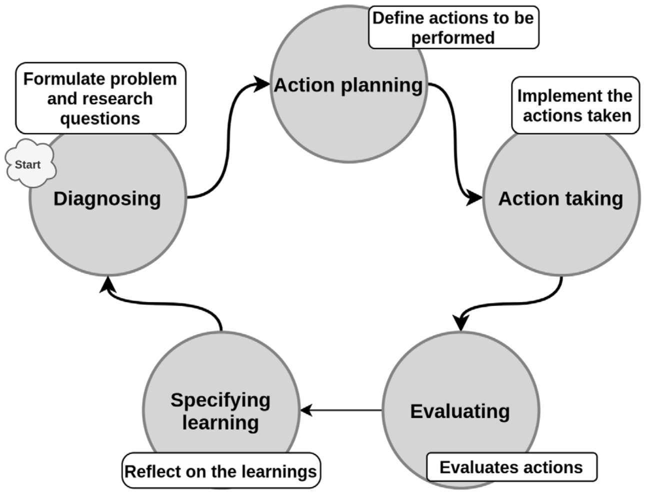

Fig. 2. Action research cycle.

It is important to note that the specifications in these systems describe end-to-end functionalities of large-scale systems. Each specification contained, on average, seven lines (7.1 in Case 1 and 6.8 in Case 2). The largest specifications in both cases contained thirteen lines. The smallest specifications in case 1 and case 2 contained three and four lines, respectively. The details of the cases are summarized in Table 1.

3.2. Research design and execution

The study was carried out according to Action Research. Runeson and Höst [23] characterized Action Research as a way to improve an existing phenomenon with flexible research designs where critical parameters of the study can be changed based on the needs of the study. Petersen et al. [24] suggested Action Research as a suitable method for transferring research results to industry. The existing literature uses Action Research to address problems such as improving software security, software process improvements, developing better UX design, etc. [25–27].

In this study, we have used Action Research to help a software organization improve the maintainability of BDD specifications. One of the authors is part of the software organization and actively participated in the implementation and evaluation of the proposed approaches. Susman and Evered [28] suggested five stages of Action Research (see Fig. 2), which were followed during this study. The stages and execution details are described in the following subsections.

3.3. Diagnosing

During this stage, the researcher(s) and the organization diagnose the underlying problems and agree upon a specific problem to solve. In this study, the problem was to improve the maintainability of BDD specifications.

During his work in the organization, the first author noted that practitioners struggle to maintain BDD specifications because of their textual nature and a lack of supporting tools and approaches comparable to those for maintaining test code.

[1] FIT = Framework for Integrated Testing, see for example [19].
[2] https://robotframework.org.
[3] https://cucumber.io/.

Table 1
Characteristics of the cases studied.

	Number of specifications	Average size of specifications	Test framework	SUT[a] type	No of teams
Case 1	72 Specs[b]	7.1 steps	Robot	Web GUI + Backend	3 teams of 24 members
Case 2	15 Specs[b]	6.8 steps	Cucumber	Backend	1 team of 6 members

[a]SUT = System Under Test.
[b]Specs = System Level Specifications (end-to-end functionality).

It was noted that several BDD specifications were similar to each other and that it would be desirable to reduce this similarity to minimize redundancy and decrease the specification base's size and thereby improve its maintainability. Refactoring was suggested since it might improve maintainability without impacting test coverage. The following research questions were formulated:

- **RQ 1:** How can the refactoring of BDD specifications be supported?
  RQ1 consists of two sub-questions related to (a) identifying candidates for refactoring (RQ 1.1) and (b) their actual refactoring (RQ 1.2):
  - **RQ 1.1:** How can similarity based measures help in finding refactoring candidates?
  - **RQ 1.2:** What types of refactoring techniques can be applied to BDD specifications?

  RQ1.1 describes the application (amidst essential steps) of the measures in identifying the refactoring candidates in the BDD test suite. RQ1.2 describes different refactoring techniques applicable to BDD specifications. Answering RQ1 helps us in developing a semi-automatic approach supporting the refactoring of BDD specifications.
- **RQ 2:** How accurately do the proposed approaches perform in identifying refactoring candidates?
  RQ2 illustrates how good similarity-based measures are in identifying refactoring candidates. To answer RQ2, we compare our proposed approaches with manual identification of refactoring candidates by industry practitioners and a natural language processing-based approach. This will help us in determining the accuracy and the performance of our proposed approaches.

3.4. Action planning

During this stage, actions are determined to address the identified problem. Previous studies have shown the effectiveness of refactoring for enhancing the maintainability of software development artifacts [9,21]. It was also agreed that refactoring should be applied with minimum human interaction.

The main goal of the actions was to refactor BDD specifications to decrease the specification base's size but without impacting its code hooks to yield better maintainability.

The maintainability of BDD specifications is cited as a problem in the existing literature (see Section 2). BDD specifications are based on natural language-based text and, therefore, are different from traditional code-based test cases. The code-based techniques to identify refactoring candidates utilize IDEs and code characteristics such as method names, method calls, input parameters, return types of methods, package names, and fields to identify refactoring candidates [29,30]. The refactoring techniques utilizing these characteristics are often limited to a programming language such as C or Java [29,31]. Unlike code artifacts, BDD specifications do not have characteristics such as return types, primitive types, and IDE support. The techniques used for code-based test cases are, therefore, not applicable to BDD specifications.

We supposed that maintainability could be improved by refactoring BDD specifications that are similar to each other, since duplication leads to low maintainability of BDD specifications [10]. Therefore, we looked into approaches for identifying similar BDD specifications. We proposed two new measures, Normalized Compression Similarity (NCS) and Similarity Ratio (SR), suitable for identifying similarity in BDD specifications. We applied these two measures using a systematic approach to our industrial cases and evaluated their accuracy in identifying refactoring candidates to improve maintainability.

NCS is defined in terms of an existing similarity measure presented in the literature called Normalized Compression Distance (NCD) [32]. SR is proposed by the authors.

3.5. Action taking

During this stage, the planned actions are executed. We selected two similarity measures (NCS and SR) and applied them using a systematic approach to all BDD specification pairs. We then selected the pairs with values under/over a specific threshold value for further evaluation and potential refactoring. The details are described in this section.

3.5.1. Normalized Compression Similarity (NCS)

Normalized Compression Similarity (NCS) is based on Normalized Compression Distance (NCD) to identify similar BDD specifications. NCD is used in a variety of ways to identify similar objects. Telles et al. [33] used NCD to detect similar text documents and demonstrated that NCD is better than manual methods to measure the similarity of text documents. Furthermore, it does not require any processing steps before its application. Rogstad et al. [34] used NCD for regression test selection of a database application. They used NCD to identify similar test cases and, based on the NCD value, decided which test cases should be selected for regression testing. Feldt et al. [35] used NCD to measure the diversity of test cases and proposed an NCD-based metric (TSDm) that can measure the diversity of a test suite. They conducted experiments on open source projects and suggested that TSDm can help selecting a specification set with higher structural code coverage and higher fault coverage. Since all studies above reported a positive experience of using NCD for text-based artifacts, we investigated the usage of NCD for determining similar BDD specifications.

NCD works by comparing the compressed sizes of two documents (BDD specifications in our case) with the compressed size of their concatenation. Since compression algorithms exploit repetitions, the compressed size of the concatenation of two similar BDD specifications will only be slightly larger than their compressed individual sizes. *NCD* can be defined by the following equation [32]:

$$NCD(s1, s2) = \frac{Z(s1s2) - min\{Z(s1), Z(s2)\}}{max\{Z(s1), Z(s2)\}}, \quad (1)$$

Based on NCD, we define Normalized Compression Similarity (NCS) as follows:

$$NCS(s1, s2) = 1 - NCD(s1, s2), \quad (2)$$

In Eq. (1), Z represents the compressor used for the calculation of NCD. $Z(s1)$ represents the compressed size of BDD specification $s1$, $Z(s2)$ represents the compressed size of BDD specification $s2$ and $Z(s1s2)$ represents the compressed size of the concatenation of $s1$ and $s2$. NCS values lie between 0 and 1, where 1 means that the BDD specifications are similar while 0 represents that they are entirely different. We selected zlib as a compressor for our study as it is suitable for short texts [36]. A discussion on the choice of a suitable compressor is provided in Section 5.

We did not use NCD directly because NCD represents the distance between objects, while NCS describes the similarity between objects. In the literature, researchers have also pointed out that these terms are interrelated but represent different ideas, i.e., distance measures represent a mathematical concept, and similarity depends on the context and domain of application [37,38]. Since similarity is high when distance is low, we defined NCS (as 1–NCD) to get consistent value ranges for both our similarity measures. Both (NCS and SR) express higher similarity when their values are closer to one.

Initially, we used NCS to identify similar specifications, but NCS did not support practitioners in identifying the specific parts of the specifications that could be refactored. We therefore introduced Similarity Ratio (SR) to support identifying the parts (lines) of a specification where refactoring can be applied.

3.5.2. Similarity Ratio (SR)

Similarity measures are commonly used in software engineering to solve problems such as identifying similarity, supporting reuse, and performing reverse engineering of software artifacts [39,40]. Often, these measures are customized to address different problems in various contexts. Girardi and Ibrahim [39] suggested a similarity-based approach to identify similar components using the natural language description of software artifacts. The natural language description (not the requirements) was used to classify the software components into different categories using lexical and semantic information. Later, the classifier is used to identify components suitable for reuse. The approach was evaluated using a simple example showing the working and parts of this similarity-based approach. Kwon and Su [40] proposed a metric for determining cohesion between modules based on the similarity of run-time properties (inputs, function calls, etc.). The approach considers the similarity of function calls and states of the objects during run-time to identify similar objects without using the source code of the objects. The approach was considered effective by using it to successfully perform the similarity analysis of malware.

To the best of our knowledge, similarity-based metrics have not been used to measure the similarity of BDD specifications.

In the case of BDD, a single line of a specification is the smallest reusable unit because a single line in the specification maps to a hook in the test code. The detection of similar lines is, therefore, an essential part of identifying the parts of a specification that can be refactored to reduce duplication.

The absolute number of similar lines of two BDD specifications can reflect their similarity but does not take into account their sizes. Therefore, we define similarity ratio as the ratio of lines of one specification that are similar to lines in the other specification. The higher the similarity ratio, the more similar are the BDD specifications and the higher the likelihood that they share parts that can be refactored.

Similarity ratio (SR) is defined as follows:

$$SR(s1, s2) = \frac{S(s1, s2)}{min\{N(s1), N(s2)\}} \tag{3}$$

Here, $s1$ and $s2$ are two BDD specifications. $N(s1)$ and $N(s2)$ is the number of lines in BDD specification $s1$ and $s2$, respectively. $S(s1, s2)$ is the number of lines of $s1$ that have the exact same text (i.e., similar) as $s2$. SR values lie between 0 and 1, where 0 means that the BDD specifications have no lines in common while 1 means that all lines of one specification appear in the other.

SR recognizes identical lines between specification pairs. Thereby, SR is sensitive to test data values, e.g., two specification lines that only differ with respect to test data are marked as different when calculating SR values.

3.5.3. Application of NCS and SR

The data collection phase involved examining the existing BDD specifications. NCS (using NCD) and SR were calculated for all pairs of BDD specifications for case 1 and case 2 using scripts, which are available online at [41]. The resulting values were stored in spreadsheets that also can be found online [41]. These spreadsheets were then analyzed to identify potential refactoring candidates.

The data were analyzed systematically using the following procedure. The calculated similarity values for NCS and SR associated with each specification pair were sorted in descending order. The data analysis was conducted using four steps:

1. The data was partitioned into four subsets, where the first subset consisted of 10% values that were close to one, the next subset consisted of the next 10% values, the third subset consisted of next 30% values. The fourth subset consisted of the remaining 50% of the values. For example, if 100 pair-wise values (in the range from 0 to 1) are produced, these values were sorted in descending order. Four sets are formed from these 100 values. The first two sets contain ten values each, remaining two sets contain 30 and 50 values e.g, Set 1: 10% highest values, Set 2: next 10% values, Set 3: next 30% values and Set 4: remaining 50% values.
2. Random pairs of BDD specifications from each of these four subsets were selected. The random selection was conducted using an online tool [42] to avoid bias in the selection of pairs and might impact the results of this study.
3. The texts and contents of the most similar specification pairs and the most dissimilar specification pairs were manually analyzed to see whether these specification pairs are indeed similar and dissimilar, respectively.

3.6. Evaluating

Our study evaluates the accuracy of the automated identification of refactoring candidates and does not evaluate the manual parts where refactoring techniques are applied over a BDD test suite. The evaluation was conducted in two ways:

- Comparison to manual evaluations of similarity by software practitioners.
- Comparison to automatic text classification based on machine-learning.

Comparison to manual evaluations of similarity by software practitioners: We involved experienced software practitioners during the evaluation. The practitioners had worked in the same domain for more than five years and have a working knowledge of the products related to the two cases. The comparison helped us understand the accuracy of measures concerning identifying refactoring candidates among BDD specifications. The background of the selected software practitioners is provided in Table 2.

This evaluation was conducted in three steps; first, the practitioners were briefed about the proposed measures and their working on the specification base. During this presentation, the practitioners asked questions to understand the working of the measures. In the second step, these practitioners were asked to answer a questionnaire for pair-wise comparison of various specification pairs. The questionnaire presented several specification pairs, and the practitioners were asked to answer (i) if the specification pairs are similar and (ii) if these specification pairs can be refactored. We found that the practitioners did not always agree. To overcome this issue, in the third step, the practitioners were asked to discuss and resolve disagreements, so that each specification pair was marked as either similar or dissimilar. This data (of assessment by each practitioner and final result) was collected using a spreadsheet, and an example of a questionnaire (and assessment data) is available online [41].

Table 2
Background of the software practitioners (P1–P5) participating in the evaluation.

	Working experience	Experience in product	Experience in BDD	Worked in	Role
P1	12 years	3 years	1 year	Case 2	Architect
P2	6 years	2 years	1 year	Case 1	Developer
P3	15 years	3 years	6 months	Case 1 & 2	Test developer
P4	12 years	3 years	1 year	Case 1 & 2	Test developer
P5	8 years	3 years	1 year	Case 1 & 2	Business analyst

Comparison to automatic text classification based on machine-learning: To evaluate the two proposed measures' accuracy, machine learning-based text-classification was performed on the specifications from Case 1 and Case 2 and compared to the similarity values obtained from NCS and SR. Previous research studies have also utilized text classification to assess the similarity between short text documents such as user reviews and software requirements [43,44].

We have used an ML-based text-classifier to classify the BDD specifications as similar. The most commonly used machine learning-based system types for text classification are Naive Bayes, Support Vector Machines, and Deep Learning-based algorithms [45]. We selected the Naive Bayes algorithm to give better results on small data sets requiring little computational resources [45].

For each specification in Table A.8 and in Table A.9, the remaining specifications (in specification suite) were used as the "training set", and the specification (of which we want to get a similar specification) is used as a "test set". Later, the text-classification approach suggested the closest matching specification in our industrial cases. The following steps (using our Python script available at [41]) were performed for this comparison i.e., working of the ML-based text-classification approach:

- Each specification file was read into memory using the script. The name of each specification was used as a category label for the specification. This labeling is required for the supervised learning algorithm so that each specification in the training set has a category label that helps in the classification of similar specifications.
- Features were extracted from all the specifications. During this step, the "bag-of-words" approach is used to convert text-based data into numerical feature vectors, understood by the machine learning model. The "bag-of-words" is commonly used to extract features from the text documents [46].
- A linear model was trained to perform the classification of specifications.
- The classification of each specification is conducted and suggesting the most similar specification.

The result of the above steps produced a pair-wise classification of each specification, e.g., specification A is similar to specification B. The results from the application of this approach are provided online [41]. The comparison results are presented in Table 3 (for Case 1) and in Table 4 (for Case 2).

3.7. Specifying learning

During this stage, it is described what could be learned from action taking and evaluating. A detailed evaluation of what we learned from the taken actions can be found in Section 4 and in Section 5.

4. Results

The Section 4.1 contains the details of an approach to identify refactoring candidates in a BDD specification suite and refactoring techniques applicable to BDD specifications. Later, the results from the evaluation of identification of refactoring candidates are described in Section 4.2.

4.1. Supporting refactoring of BDD specifications (RQ1)

Mens and Tourwé [47] have defined activities related to the refactoring in software artifacts. These activities are identifying where to apply the refactoring, applying the refactoring, validating that the refactoring preserves the behavior of the artifacts or products, and assessing the impact of the refactoring on the quality and consistency of the refactored software. Assessing the impact on the quality and consistency of the refactored software are specific to the software product that is refactored and are not applicable to BDD specifications. Alternatively, manual verification of the quality of the BDD feature suite can instead be performed by expert practitioners. Only the identification of refactoring candidates is supported as a semi-automated approach. The remaining activities of the refactoring process need to be performed manually by the software practitioners. The remaining three activities (identifying refactoring candidates, refactoring techniques, and validation of behavior preservation) are discussed in the subsections below.

4.1.1. Semi-automated approach to identify refactoring candidates (RQ1.1)

We devised the following four-step approach to identify the refactoring candidates using similarity of BDD specifications: Pre-processing, Measuring, Ranking, and Identifying refactoring candidates. These steps can provide guidelines on the identification of refactoring candidates of BDD specifications using similarity measures.

(1) Pre-processing: During this step, the data was prepared for the analysis. We ensured that each BDD specification was stored in a separate file with the name of the spec in the first line and one BDD step or sub-step per line in the following lines. All BDD-keywords were deleted to facilitate line-by line comparisons (for SR). This step produced 72 files for Case 1 and 15 files for Case 2.

(2) Measuring: In the next step, NCS and SR were calculated for all pairs of BDD specification files. Automated scripts, implementing NCS and SR, produced the pair-wise values. This yielded 2556 values for Case 1 and 105 for Case 2 (available at [41]).

(3) Ranking: In this step, the output from the previous step (the NCS and SR values) was analyzed and ranked to identify similar and dissimilar specifications. An NCS or SR value closer to 1 means that a specification pair is similar, while a value closer to 0 means that a pair is dissimilar. The similarity values of the pairs in the test suites were sorted in descending order. The similarity (i.e., NCS and SR values) of few specification pairs is present in Table 3 (for Case 1) and in Table 4 (for Case 2).

After that, threshold values for NCS and SR were selected to determine which specifications should be considered candidates for refactoring and which should not. In refactoring, threshold values are often used to identify when to perform a refactoring [30]. This threshold value can be associated with (i) code complexity, (ii) code coverage, (iii) artifact similarity (methods, classes, or documents), etc. We found only few studies defining threshold values for similarity-based refactoring [13,37,38,48,49]. Their values range from 0.30–0.73.

Tsantalis et al. [30] suggested a fundamental limitation with threshold values: that these are not general-purpose values. The threshold values for refactoring depend on the characteristics of each project. Xing and Stroulia [48] raised the following concerns when selecting a threshold value for refactoring:

- Higher threshold values lead to finding fewer cases of similarity.

Table 3
Case 1: Similarity values and assessment of "candidate for refactoring" for specification pairs based on NCS, SR and practitioners' views. **AC** = A candidate for refactoring.

Pair	NCS value > 0.50	SR value > 0.50	Practitioners' view	NCS on refactoring	SR on refactoring	Practitioners on refactoring
49–47	0.809	0.60	**Similar**	**AC**	**AC**	**AC**
38–35	0.811	0.667	**Similar**	**AC**	**AC**	**AC**
59–60	0.639	0.8	**Similar**	**AC**	**AC**	**AC**
8–5	0.615	0.583	**Similar**	**AC**	**AC**	**AC**
4–10	0.608	0.583	**Similar**	**AC**	**AC**	**AC**
17–23	0.563	0.583	**Similar**	**AC**	AC	**AC**
11–14	0.667	0.883	**Similar**	**AC**	**AC**	**AC**
26–7	0.571	0.778	**Similar**	**AC**	**AC**	**AC**
1–17	0.623	0.7	**Similar**	**AC**	**AC**	**AC**

Values for all evaluated specification pairs are present in Table A.8.

Table 4
Case 2: Similarity values and assessment of "candidate for refactoring" for specification pairs based on NCS, SR and practitioners' views. **AC** = A candidate for refactoring.

Pair	NCS value > 0.50	SR value > 0.50	Practitioners' view	NCS on refactoring	SR on refactoring	Practitioners on refactoring
13–12	0.745	0.833	**Similar**	**AC**	**AC**	**AC**
1–2	0.698	0.667	**Similar**	**AC**	**AC**	**AC**
9–8	0.687	0.50	**Similar**	**AC**	**AC**	**AC**
6–8	0.673	0.6	**Similar**	**AC**	**AC**	**AC**
5–2	0.643	0.6	**Similar**	**AC**	**AC**	**AC**
3–2	0.621	0.667	**Similar**	**AC**	**AC**	**AC**
6–7	0.608	0.6	**Similar**	**AC**	**AC**	**AC**
10–11	0.632	0.8	**Similar**	**AC**	**AC**	**AC**

Values for all evaluated specification pairs are present in Table A.9.

- Lower threshold values lead to more false positives.

As refactoring BDD specifications using similarity measures is a new area, we did not have any evidence for a suitable threshold value for identifying refactoring candidates. After discussion with the practitioners, thresholds of 0.50 for NCS and SR were considered as suitable, since pairs with similarity values $\geq$ 0.50 were considered suitable candidates for refactoring. Furthermore, 0.50 is in the middle of value ranges mentioned in the relevant literature [13,37,48–50].

In the future, when there are more changes in the BDD test-suite, the threshold values can be evaluated and optimized. However, this optimization of the threshold value is not addressed in the scope of this study.

(4) Identifying refactoring candidates: During this step, highly similar specifications are identified based on their similarity values. These similar specification pairs are assumed to be potential candidates for refactoring. A manual review of the potential refactoring candidates is required before finalizing the refactoring candidates. This review can be performed by experienced practitioners with considerable domain knowledge. Later, the refactoring techniques suitable for BDD specifications (see Section 4.1.2) are manually applied to the identified candidates. In Tables 3 and 4 the identified candidates for refactoring are marked with "AC". In our cases, most of the specification pairs identified as candidates for refactoring are the same for both NCS and SR; however, in few cases, there are different results, e.g., pair 8–10 pair in Case 2 in Table A.9. These differences were later analyzed to identify the reason behind this mismatch, see Section 5.

4.1.2. Refactoring techniques for BDD specifications (RQ1.2)

Refactoring requires domain knowledge [51] and should therefore be performed by software practitioners who have a good understanding of the software product. To support practitioners, we have provided techniques in Table 5 for BDD specifications (inspired by [47]) that can be used during the refactoring process. These techniques are exemplified using BDD specifications from an open-source system [52], see Table 6.

4.1.3. Validating that the refactoring preserved behavior

In this step of the refactoring process, one needs to ensure that the refactoring did not modify the BDD specification base's behavior. When refactoring code, this validation is conducted with the help of test cases [47]. In the case of BDD specifications, two strategies can be used to validate that a refactoring preserved the existing behavior of the specifications:

1. Comparison of pre-refactoring and post-refactoring log traces of the whole BDD suite. The specifications need to be re-run which involves calling the hooks. The output from the re-run can then be used to validate that the previous behavior is preserved after the refactoring.
2. Comparison of code-coverage information of the product.

4.2. Accuracy in identifying refactoring candidates (RQ2)

This section describes our evaluation of the accuracy of the similarity measures NCS and SR in identifying refactoring candidates. The assessment consisted of comparing results from the measures with results from the practitioners and a comparison of measures with a machine learning-based text classification approach. The subsections below describe the results of these evaluations in detail.

4.2.1. Comparison with software practitioners

The effectiveness of the semi-automated approach to identify refactoring candidates was evaluated with the help of experienced industry practitioners who had a good understanding of the rationale for each system-level specification. They were asked to manually assess the similarity of randomly selected specification pairs as described in Section 3. The results of the comparison of the practitioners' assessment and the two proposed measures (NCS and SR) are summarized in Table A.8 (for Case 1) and in Table A.9 (for Case 2).

Identification of refactoring candidates: A vital aspect of the proposed four-step semi-automated approach is that it uses similarity as a measure to identify refactoring candidates. The similarity is purely based on the content of the specifications; both approaches do not consider the context of the specifications. To identify the utility of similarity as a means to identify refactoring candidates, the practitioners were asked to classify each specification pair as either a refactoring candidate or not a refactoring candidate. The results from the practitioners' assessment show that our proposed semi-automated approach can be used to identify refactoring candidates among BDD specifications given suitable thresholds for the values of NCS and SR.

Table 5
Refactoring techniques for BDD specifications, inspired by Mens and Tourwé [47] & Suan [13].

Refactoring technique	When to use
Merging	If two specifications have common lines and few important dissimilarities, they can be merged to form one larger specification.
Re-structuring	If two specifications have common statements, the common statements are combined into a new statement. This new statement can be used in place of the common statements in the two original specifications (example provided in Table 6).
Deleting	If two specifications have the same functionality (e.g., different test data values testing same path/function in the code) or are duplicates of each other, one of them can be deleted.
Renaming	If two specifications have the same specification names but different functionality, a practitioner keeps both specifications by renaming one of the specifications.

Table 6
Exemplification of refactoring techniques from Table 5 on open-source project [52].

Merging		
Specification 1	Specification 2	Specification after refactoring
Scenario: **Account Creation** **Given** an account called "Cash" **Then** the list shows an account called "Cash"	Scenario: **Account Update** **Given** an account called "Credit" **When** I change the account name to "Debit" **Then** the list shows an account called "Debit"	**NCS = 0.614, SR = 0** Scenario: **Account Creation and Update** **Given** an account called "Cash" **When** the list shows an account called "Cash" **And** I change the account name to "Debit" **Then** the list shows an account called "Debit"
Re-structuring		
Scenario: **Payment Using Card** **Given** a transaction is initiated using Amazon Visa Card **And** bill payment is for Gas **And** 20 SEK are deducted **When** transaction is created **Then** money is deducted from Amazon Visa Card	Scenario: **Payment of Bills** **Given** a transaction is initiated using Amazon Visa Card **AND** bill payment is for Gas **And** 20 SEK are deducted **When** transaction is accepted **Then** gas payments are paid	**NCS = 0.680, SR = 0.50** Scenario: **Payment of Bills** **Given** transaction of payment **When** transaction is accepted **Then** gas payments are paid Scenario: **Payment Using Card** **Given** transaction of payment **When** transaction is created **Then** money is deducted from Amazon Visa Card **transaction of payment** **Given** a transaction is initiated using Amazon Visa Card **AND** bill payment is for Gas **AND** 20 SEK are deducted
Deleting		
Scenario: **Account Creation** **Given** an account called "Bank of Eng" **Then** the account list shows "Bank of Eng"	Scenario: **Account Creation** **Given** an account called "Bank of Sweden" **Then** the account list shows "Bank of Sweden"	**NCS = 0.752, SR = 0** **If two Scenario have different test-data for testing exactly the same path/function in the code, then one scenario is deleted.**
Renaming		
Scenario: **View Debit** **Given** Money exists in an account **Then** Verify debit is positive number	Scenario: **View Debit** **Given** Money does not exists in an account **Then** Verify debit is negative number	**NCS = 0.780, SR = 0** **Both specifications are kept, one of the two is renamed.**

Cost Savings with automated measures: On average, practitioners took 2 h for analyzing the similarity between the fifty-six randomly selected pairs of BDD specification in Tables A.8 and A.9. The automated scripts took less than 2 min to compute the NCS and SR values for all pairs of specifications and categorize each pair as "Similar" or "Not Similar", i.e. were about 60 times faster. On large specification bases, these automated measures can be a significant advantage compared with manual identification. The time saved by using automated identification of potential candidates for refactoring can result in considerable cost savings for an organization.

Lack of domain-knowledge: One of the drawbacks of our measures is that these measures work on the raw text of specifications and do not consider the domain-knowledge captured in the specifications during the identification of refactoring candidates. All statements may be the same in two BDD specifications, but the sequence in which the statements are executed covers the validation of two different requirements. Incorporating domain-knowledge may reduce the number of false positives. In our evaluation, this lack of domain-knowledge yielded some false positives, e.g., specification pair '8–10' in Case 2 was marked as a refactoring candidate due to its NCS-value of 0.642. However, practitioners assessed it as not similar and not a candidate for refactoring.

4.2.2. Comparison to automatic text classification based on machine-learning

Our analysis of the results from the ML-based approach revealed that many specification pairs that have been marked with high NCS and SR values are not marked as similar by the ML-based text classification approach. The results from the ML-based approach are shown in column "ML classifier" in Table A.8 and in Table A.9 showing a comparison between the ML-based approach, the usage of similarity measures and the practitioner's views.

Context-awareness: ML-based text classification approaches can capture the domain-specific vocabulary when the model is trained on the specifications. During the classification of a new specification, this knowledge of domain-specific vocabulary is utilized. This domain awareness should be an advantage over the NCS- and SR-based approaches that do not consider the specifications' context and semantics. However, our results in Table A.8 and in Table A.9 do not show any significant advantage of ML-based text classification approach over NCS or SR when used on specifications from our two industrial cases. It is important to note that there are many text-classification approaches present in the literature, and other approaches may (or may not) perform better than our selected approach.

Dependence on large data sets: ML-based approaches require large data-sets (i.e., large specification bases) to train a better model and

Table 7
Precision and recall values for NCS, SR & ML-classifier of the data present in Tables A.8 and A.9.

		Precision	Recall
Case 1	NCS	100%	100%
	SR	90%	90%
	ML-classifier	100%	30%
Case 2	NCS	88%	100%
	SR	100%	100%
	ML-classifier	100%	75%

classification system. Even though we selected a classification approach with low demands on the training data, the approach did not perform well enough. In comparison, NCS and SR successfully identified similar specifications (as confirmed by practitioners) regardless of the number of specifications.

4.2.3. *Precision and recall in evaluated candidates*

Precision and recall are two metrics for evaluating the accuracy of (binary) classifications [30]. In our context, precision reflects the fraction of actual candidates for refactoring among the potential candidates identified by some mechanism. Recall represents the fraction of actual candidates identified by some mechanism in relation to all existing actual candidates.

We used the manual classification conducted by software practitioners as the "truth" and evaluated how well NCS and SR performed against this baseline. For case 1 (see Table 7), NCS has a precision and recall of 100% while SR has precision and recall of 90%. For case 2 (see Table 7), NCS has a precision of 88% and recall of 100%, respectively, while SR has precision and recall of 100%. I.e. both similarity measures perform well on our small subsets of BDD specifications. Furthermore, we assessed the precision and recall values of the ML-classifier for our evaluated data-set. For case 1, ML-classifier has a precision of 100% and recall of 30%. For case 2, NCS has a precision of 100% and recall of 75%, respectively.

Please note that the software practitioners manually classified only a limited number of specification pairs due to time constraints. Our baseline sets were therefore quite small. For a more thorough evaluation of the accuracy of the classification by NCS and SR, practitioners would need to evaluate hundreds of test-case pairs, which was not possible in the present study.

5. Discussion

This section discusses important aspects of the measures and the refactoring approach introduced and evaluated in this study.

5.1. *Scalability of approach*

In large specification bases, it can be challenging to identify refactoring candidates manually. The two automated scripts (available at [41]) can help software organizations analyzing large specification bases. In our small and medium-size specification bases, the execution time was less than 2 min to identify refactoring candidates. In an examination (by the authors) with a specification base of 500 specifications, refactoring candidates were identified in less than 5 min using NCS and SR. This indicates that our proposed approach scales well, even for large specification bases. Further studies are needed to investigate how well our approach performs in comparison to ML-based approaches with increasing sizes of specification bases.

According to Erb's classification [53], our proposed approach can be classified as "semi-automated", since human intervention is required when changing the specifications. A GUI tool could be developed to facilitate the refactoring process with a side-by-side comparison that highlights the similar parts between specifications. However, the development of such a tool was out of the scope of this study.

5.2. *Difference in size of specification pairs*

The industrial evaluation (in Tables 1 and 2) showed that the measures produced one false negative (i.e., marked with no similarity but have similarity) and two false positives (i.e., marked as similar but have no similarity) results. The analysis revealed that all three cases (pairs 27–12, 30–28 in Table A.8 and pair 8–10 in Table A.9) have one common trait; a considerable difference in the number of lines in the specifications (difference of five or more lines). Telles et al. [33] also noted that NCD is impacted by the differences in the sizes of the texts compared for similarity. The false positive pairs may waste practitioners' time by suggesting a pair of refactoring candidates that are not similar. The false negative pairs may result in missing out on refactoring opportunities.

5.3. *Specification suite's diversity*

A diverse test suite can increase the likelihood of identifying defects in a system [54]. A by-product of the two proposed measures is that practitioners can quickly recognize their specification suite's diversity, e.g., a large number of lower NCS or SR value means the BDD specification suite is diverse and has fewer redundant specifications. This is an additional advantage of using these measures to identify the diversity of the specification suite.

5.4. *Choice of compressors*

Different compression algorithms produced slightly different NCS values for the same specification pairs. We have not investigated in detail the impact of different compression algorithms. However, we tried three different algorithms (Blosc,[4] zlib,[5] and gzip[6]), and there were no noticeable differences in the results. An implementation of NCS with the three algorithms above is provided by the authors [41]. Later, we selected zlib as a compressor for our study as it is suitable for smaller text-based data-sets [36]. The results of this study are calculated using zlib.

Zlib is a general-purpose compression algorithm that is commonly used in computer applications. The following factors dictate the choice of zlib as our preferred compression algorithm:

- Existing literature suggests zlib as suitable for small text-based data compression [36].
- Existing studies have used zlib with NCD to classify small natural language data [55].
- A study on identifying similarity/dissimilarity of test-data identified zlib as a better compressor than Blosc when used for NCD measurements [35].

To our knowledge, no previous study exists that measures the effect of different compression algorithms on BDD specifications. Further studies are required to evaluate different compression algorithms' effectiveness for capturing similarity of BDD specifications.

5.5. *Contribution to practice and research*

From a *practical perspective*, the study

- provides two measures (NCS and SR) that practitioners can use to identify refactoring candidates;
- describes a four-step semi-automated approach for the refactoring of BDD specifications that supports practitioners when refactoring BDD specifications.

[4] https://blosc.org/.
[5] https://zlib.net/.
[6] https://www.gzip.org/.

From a *research perspective*, the study

- is exploratory and extends the existing body of knowledge on the refactoring of text-based BDD specifications;
- involves data from real industry cases as well as practitioners from industry throughout the study;
- describes the research context and data in detail so that the research community can evaluate and compare new techniques for refactoring in the BDD context.

6. Validity threats

In our discussion of validity threats, we follow Petersen and Gencel's suggestions [56] and address validity threats with respect to a participatory worldview (Action Research).

Theoretical validity: Theoretical validity takes into account if the study can capture what was intended and whether other factors may have impacted the investigation. A primary concern is regarding the selection of specifications to conduct this study. The specifications were written by developers developing and using the BDD based specifications for the first time. However, some of them had several years of product development experience in the same organization and domain. Since the two measures (NCS and SR) were applied to test new cases, there is a possibility that the maturity level of specifications may impact the identification of candidates for refactoring. A more experienced team (with multiple years of experience in BDD) may have designed the specifications to utilize less text but cover more functionality, thus influencing the similarity and maintainability of the specifications. The second concern is whether both measures (NCS and SR) actually capture similarity. To address this concern, we have used different methods to evaluate the similarity results of NCS and similarity ratio using BDD specifications, such as comparing standard lines in specification pairs and comparing the outcome with opinions from industry experts.

Interpretive validity: Interpretive validity deals with the conclusions drawn from results, whether these conclusions are correct, and not based on researchers' bias. The foremost concern to interpretive validity is based on the usefulness of NCS and SR. Since this study was exploratory, it was not straightforward to determine if these measures effectively identify refactoring candidates. To address this concern, we have asked the experts to (i) determine if the specification pairs were similar (same pairs that were marked similar/dissimilar by NCS and SR) and (ii) evaluate if specifications are candidates for refactoring. Expert opinion was used to evaluate the effectiveness of our proposed measures. This also gave us some indications regarding the impact of differences in the sizes of specifications, scalability of measures, and the impact of compression algorithms.

In Action Research, researchers are part of the organization and are actively involved in performing the study. Staron suggested a validity threat for such cases called the "John Henry effect" [57]. This effect refers to setting a baseline for comparison that makes the new approach look effective. In our study, the choice of threshold value for similarity may be subjected to this threat. To mitigate this threat, we selected a threshold value that was in the middle of the values provided in the literature. Furthermore, we discussed the suitability of the threshold value with the practitioners.

Hypothesis guessing is a crucial threat to the interpretive validity of studies using Action Research [57]. To mitigate this threat, we disclosed the used similarity measures and how they are computed from the practitioners involved in the study. Furthermore, they were not shown any results until their manual evaluations were completed.

Another threat to validity concerns the researchers' involvement in the data collection and evaluation. The data collection was conducted using forms in an excel sheet to avoid information loss and external bias. Each practitioner recorded his/her response independently. Furthermore, the data collection results are shared as part of this study to reduce the threats to interpretive validity.

Generalizability: Generalizability refers to the general applicability of results over different settings and contexts. The two industrial cases are from two different products but the same organization. However, the cases have different contexts (programming language, tools, process, etc.) and development teams. The results of this study might differ if the study were carried out in a different organization. However, the study captures the general advantages and disadvantages associated with the usage of similarity measures for identifying refactoring candidates from BDD specifications. Considering this study's exploratory nature, we believe that the results are interesting for researchers and practitioners in other software organizations, even though their generalizability is limited.

A critical threat to the study's generalizability deals with the implementation of the scripts by one of the researchers. There is a chance that the implementation is biased or aligned to support the results of this study. The scripts implemented by the researcher are made available as a part of the study to deal with this threat. Furthermore, a second independent researcher also reviewed the scripts to evaluate the implementation of the approaches using the script.

Repeatability: Repeatability is concerned with data collection, analysis, and steps followed during the study and whether these aspects are described in sufficient detail. The data collection was conducted using automated scripts. A link is provided to the scripts implementing the measures, and the steps for their analysis are documented in detail. Furthermore, the collected data is published for the sake of comparison if the study is repeated in a comparable context and setting. Selection biases are a threat to validity in Action Research when researchers are a part of the researched organization [57]. To mitigate this threat, selection criterion for study participants were defined to ensure that only experienced practitioners who have worked on the product and in the domain were recruited.

Descriptive validity: Descriptive validity deals with the factual accuracy of the account by the researchers [58]. To mitigate this threat, data collection and data analysis were performed systematically using automated scripts and tools. We used automated scripts for calculating NCS and SR values. These generated NCS and SR values were then stored in spreadsheets. The data analysis was performed using the four-step process described in Section 3. Similarly, the data collection concerning evaluation using software practitioners and text classification was conducted using automated tools and scripts. The scripts, the collected data, and well-defined steps for the data analysis may mitigate this threat to the validity of our investigation.

7. Conclusion

Software engineering researchers have explored the idea of refactoring an artifact to improve its maintainability for several years. New artifacts of various types (text, tables, etc.) in software development are introduced with changing technologies and processes. BDD specifications are one of these new artifacts and are gaining popularity in industry.

Recent research has pointed out that refactoring is a crucial activity of BDD, however, there are very few studies on the refactoring of BDD specifications. A vast majority of the existing studies do not operate on the BDD specifications. They perform refactoring in BDD by utilizing the test-code hooks, identifying parts where new code is introduced, or using IDE-plugins to perform refactoring of FIT tables. In this study, we have proposed approaches to support the refactoring of BDD specifications and evaluated the identification of refactoring candidates.

We have shown that similarity measures (NCS and SR) can support identifying refactoring candidates in a BDD specification base.

We have furthermore proposed a four-step approach for pre-processing, measuring (using NCS and SR), ranking, and identifying refactoring candidates. The approach and the two proposed measures were successfully evaluated using two industrial projects.

Table A.8
Case 1: Similarity values and assessment of "candidate for refactoring" for specification pairs based on NCS, SR, Machine learning classifier and practitioners' views. **AC** = A candidate for refactoring, NAC = Not a candidate for refactoring.

Pair	NCS value	SR value	Practitioners' view	ML classifier	NCS on refactoring	SR on refactoring	Practitioners on refactoring
49–47	0.809	0.60	**Similar**	**Similar**	**AC**	**AC**	**AC**
38–35	0.811	0.667	**Similar**	**Similar**	**AC**	**AC**	**AC**
59–60	0.639	0.8	**Similar**	Not Similar	**AC**	**AC**	**AC**
8–5	0.615	0.583	**Similar**	Not Similar	**AC**	**AC**	**AC**
4–10	0.608	0.583	**Similar**	Not Similar	**AC**	**AC**	**AC**
27–12	0.537	0.3	**Similar**	Not Similar	**AC**	NAC	**AC**
17–23	0.563	0.583	**Similar**	**Similar**	**AC**	**AC**	**AC**
11–14	0.667	0.883	**Similar**	Not Similar	**AC**	**AC**	**AC**
26–7	0.571	0.778	**Similar**	Not Similar	**AC**	**AC**	**AC**
1–17	0.623	0.7	**Similar**	Not Similar	**AC**	**AC**	**AC**
70–63	0.485	0.334	Not Similar	Not Similar	NAC	NAC	NAC
1–14	0.490	0.30	Not Similar	Not Similar	NAC	NAC	NAC
72–63	0.490	0.40	Not Similar	Not Similar	NAC	NAC	NAC
17–19	0.472	0.417	Not Similar	Not Similar	NAC	NAC	NAC
26–18	0.470	0.25	Not Similar	Not Similar	NAC	NAC	NAC
21–13	0.459	0.1667	Not Similar	Not Similar	NAC	NAC	NAC
30–28	0.40	0.50	Not Similar	Not Similar	NAC	**AC**	NAC
11–3	0.479	0.384	Not Similar	Not Similar	NAC	NAC	NAC
49–7	0.349	0.2	Not Similar	Not Similar	NAC	NAC	NAC
66–37	0.381	0	Not Similar	Not Similar	NAC	NAC	NAC
13–48	0.406	0.2	Not Similar	Not Similar	NAC	NAC	NAC
34–35	0.392	0	Not Similar	Not Similar	NAC	NAC	NAC
3–18	0.384	0.125	Not Similar	Not Similar	NAC	NAC	NAC
27–53	0.362	0.1667	Not Similar	Not Similar	NAC	NAC	NAC
41–37	0.386	0	Not Similar	Not Similar	NAC	NAC	NAC
50–60	0.440	0.1667	Not Similar	Not Similar	NAC	NAC	NAC
16–40	0.349	0	Not Similar	Not Similar	NAC	NAC	NAC
39–3	0.129	0	Not Similar	Not Similar	NAC	NAC	NAC
1–44	0.187	0	Not Similar	Not Similar	NAC	NAC	NAC
70–21	0.20	0	Not Similar	Not Similar	NAC	NAC	NAC
29–14	0.297	0	Not Similar	Not Similar	NAC	NAC	NAC
4–34	0.261	0	Not Similar	Not Similar	NAC	NAC	NAC

Table A.9
Case 2: Similarity values and assessment of "candidate for refactoring" for specification pairs based on NCS, SR, Machine learning classifier and practitioners' views. **AC** = A candidate for refactoring, NAC = Not a candidate for refactoring.

Pair	NCS value	SR value	Practitioners' view	ML classifier	NCS on refactoring	SR on refactoring	Practitioners on refactoring
13-12	0.745	0.833	**Similar**	**Similar**	**AC**	**AC**	**AC**
1-2	0.698	0.667	**Similar**	**Similar**	**AC**	**AC**	**AC**
9-8	0.687	0.50	**Similar**	**Similar**	**AC**	**AC**	**AC**
6-8	0.673	0.6	**Similar**	Not Similar	**AC**	**AC**	**AC**
5-2	0.643	0.6	**Similar**	Not Similar	**AC**	**AC**	**AC**
3-2	0.621	0.667	**Similar**	**Similar**	**AC**	**AC**	**AC**
6-7	0.608	0.6	**Similar**	**Similar**	**AC**	**AC**	**AC**
10-11	0.632	0.8	**Similar**	**Similar**	**AC**	**AC**	**AC**
8–10	0.642	0.2	Not Similar	Not Similar	**AC**	NAC	NAC
7–11	0.330	0.125	Not Similar	Not Similar	NAC	NAC	NAC
6–10	0.330	0.4	Not Similar	Not Similar	NAC	NAC	NAC
9–10	0.30	0.33	Not Similar	Not Similar	NAC	NAC	NAC
15–7	0.296	0	Not Similar	Not Similar	NAC	NAC	NAC
4–7	0.296	0	Not Similar	Not Similar	NAC	NAC	NAC
3–12	0.294	0	Not Similar	Not Similar	NAC	NAC	NAC
9–13	0.290	0	Not Similar	Not Similar	NAC	NAC	NAC
9–12	0.288	0	Not Similar	Not Similar	NAC	NAC	NAC
1–14	0.286	0	Not Similar	Not Similar	NAC	NAC	NAC
14–13	0.280	0	Not Similar	Not Similar	NAC	NAC	NAC
14–12	0.275	0	Not Similar	Not Similar	NAC	NAC	NAC
15–12	0.251	0	Not Similar	Not Similar	NAC	NAC	NAC
15–13	0.246	0	Not Similar	Not Similar	NAC	NAC	NAC
4–13	0.232	0	Not Similar	Not Similar	NAC	NAC	NAC
4–12	0.222	0	Not Similar	Not Similar	NAC	NAC	NAC

We found that the proposed semi-automatic approach identified refactoring candidates accurately and far quicker than software practitioners. For small to medium-size specification bases, the identification of refactoring candidates took less than 2 min. The approach also scales well. Identifying refactoring candidates in a specification base with 500 BDD specifications took less than 5 min. Overall, our evaluation showed encouraging results and the practitioners' feedback was positive.

NCD and SR can also help assessing a BDD specification base's diversity; the higher the overall similarity, the lower the diversity. Higher diversity in BDD specifications results in higher coverage of product features during the validation activities.

As BDD is a relatively new field, in the future, we plan to extend our approach by automating the refactoring techniques in Table 6. Furthermore, we plan to improve our measurement and develop a graphical interface to facilitate the refactoring process by supporting software practitioners.

CRediT authorship contribution statement

Mohsin Irshad: Conceptualization, Methodology, Investigation, Formal analysis, Writing - original draft, Writing - review & editing. **Jürgen Börstler:** Conceptualization, Methodology, Writing - review & editing, Supervision. **Kai Petersen:** Writing - review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix. Tables

See Tables A.8 and A.9.

References

[1] D. North, Behavior modification: The evolution of behavior-driven development, Better Softw. 8 (3) (2006) 8–12.
[2] M.M. Moe, Comparative study of test-driven development (TDD), behavior-driven development (BDD) and acceptance test–driven development (ATDD), Int. J. Trend Sci. Res. Dev. (2019) 231–234.
[3] C. Solis, X. Wang, A study of the characteristics of behaviour driven development, in: Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications, IEEE, 2011, pp. 383–387.
[4] A.Z. Yang, D.A. da Costa, Y. Zou, Predicting co-changes between functionality specifications and source code in behavior driven development, in: Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories, 2019, pp. 534–544.
[5] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 2018.
[6] M. Kim, T. Zimmermann, N. Nagappan, A field study of refactoring challenges and benefits, in: Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, 2012, pp. 1–11.
[7] E. Mealy, P. Strooper, Evaluating software refactoring tool support, in: Proceedings of the Australian Software Engineering Conference (ASWEC'06), 2006, pp. 10–19.
[8] C. Abid, V. Alizadeh, M. Kessentini, T.d.N. Ferreira, D. Dig, 30 years of software refactoring research: A systematic literature review, 2020, arXiv preprint arXiv:2007.02194.
[9] R. Borg, M. Kropp, Automated acceptance test refactoring, in: Proceedings of the 4th Workshop on Refactoring Tools, 2011, pp. 15–21.
[10] L.P. Binamungu, S.M. Embury, N. Konstantinou, Maintaining behaviour driven development specifications: Challenges and opportunities, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2018, pp. 175–184.
[11] S. Bruschi, M.K. Le Xiao, G. Jimenez-Maggiora, Behavior driven development (BDD) a case study in healthtech, in: Proceedings of the Pacific Northwest Software Quality Conference, 2020, pp. 1–12.
[12] A. Egbreghts, A literature review of behavior driven development using grounded theory, in: 27th Twente Student Conference on IT, 2017, Available at: https://pdfs.semanticscholar.org/4f03/ec0675d08cfd1ecdbaac3361a29d756ce656.pdf.
[13] S. Suan, An Automated Assistant for Reducing Duplication in Living Documentation (Master's thesis), School of Computer Science, University of Manchester, 2015.
[14] D. North, Whats in a story? 2020, http://dannorth.net/whats-in-a-story/, accessed: 2020-01-02.
[15] A.I. Anton, Successful software projects need requirements planning, IEEE Softw. 20 (3) (2003) 44.
[16] N.M.A. Pulido, Applying Behavior Driven Development Practices and Tools to Low-Code Technology (Ph.D. thesis), 2019.
[17] E.C.S. Santos, D.M. Beder, R.A.D. Penteado, A Study of test techniques for integration with domain driven design, in: Proceedings of the 12th International Conference on Information Technology-New Generations, 2015, pp. 373–378.
[18] S.-T. Lai, F.-Y. Leu, W.C.-C. Chu, Combining IID with BDD to enhance the critical quality of security functional requirements, in: Proceedings of the Ninth International Conference on Broadband and Wireless Computing, Communication and Applications, 2014, pp. 292–299.
[19] R. Mugridge, W. Cunningham, Fit for Developing Software: Framework for Integrated Tests, Pearson Education, 2005.
[20] M. Bures, M. Filipsky, I. Jelinek, Identification of potential reusable subroutines in recorded automated test scripts, Int. J. Softw. Eng. Knowl. Eng. 28 (01) (2018) 3–36.
[21] M. Rahman, J. Gao, A reusable automated acceptance testing architecture for microservices in behavior-driven development, in: Proceedings of the 2015 IEEE Symposium on Service-Oriented System Engineering, 2015, pp. 321–325.
[22] C. Sathawornwichit, S. Hosono, Consistency reflection for automatic update of testing environment, in: Proceedings of the 2012 IEEE Asia-Pacific Services Computing Conference, 2012, pp. 335–340.
[23] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empir. Softw. Eng. 14 (2) (2009) 131.
[24] K. Petersen, C. Gencel, N. Asghari, D. Baca, S. Betz, Action research as a model for industry-academia collaboration in the software engineering context, in: Proceedings of the 2014 International Workshop on Long-Term Industrial Collaboration on Software Engineering, 2014, pp. 55–62.
[25] D.S. Cruzes, M.G. Jaatun, T.D. Oyetoyan, Challenges and approaches of performing canonical action research in software security, in: Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security, 2018, pp. 1–11.
[26] P.A. Nielsen, G. Tjørnehøj, Social networks in software process improvement, J. Softw. Maint. Evol.: Res. Pract. 22 (1) (2010) 33–51.
[27] K. Bang, M.A. Kanstrup, A. Kjems, J. Stage, Adoption of UX evaluation in practice: An action research study in a software organization, in: Proceedings of the IFIP Conference on Human-Computer Interaction, 2017, pp. 169–188.
[28] G.I. Susman, R.D. Evered, An assessment of the scientific merits of action research, Adm. Sci. Q. (1978) 582–603.
[29] A.A.B. Baqais, M. Alshayeb, Automatic software refactoring: a systematic literature review, Softw. Qual. J. 28 (2) (2020) 459–502.
[30] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, D. Dig, Accurate and efficient refactoring detection in commit history, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 483–494.
[31] M. Abebe, C.-J. Yoo, Trends, opportunities and challenges of software refactoring: A systematic literature review, Int. J. Softw. Eng. Appl. 8 (6) (2014) 299–318.
[32] P.M. Vitányi, F.J. Balbach, R.L. Cilibrasi, M. Li, Normalized information distance, in: Information Theory and Statistical Learning, Springer, 2009, pp. 45–82.
[33] G.P. Telles, R. Minghim, F.V. Paulovich, Normalized compression distance for visual analysis of document collections, Comput. Graph. 31 (3) (2007) 327–337.
[34] E. Rogstad, L. Briand, R. Torkar, Test case selection for black-box regression testing of database applications, Inf. Softw. Technol. 55 (10) (2013) 1781–1795.
[35] R. Feldt, S. Poulding, D. Clark, S. Yoo, Test set diameter: Quantifying the diversity of sets of test cases, in: Proceedings of the IEEE International Conference on Software Testing, Verification and Validation, 2016, pp. 223–233.
[36] I. Ivanov, C. Hantova, M. Nisheva, P.L. Stanchev, P. Ein-Dor, Software library for authorship identification, Digit. Present. Preserv. Cult. Sci. Herit. (2015) 91–97.
[37] J.-C. Corrales, Behavioral Matchmaking for Service Retrieval (Ph.D. thesis), Université de Versailles-Saint Quentin en Yvelines, 2008.
[38] S.-S. Choi, S.-H. Cha, C.C. Tappert, A survey of binary similarity and distance measures, J. Syst. Cybern. Inform. 8 (1) (2010) 43–48.
[39] M. Girardi, B. Ibrahim, A similarity measure for retrieving software artifacts, in: SEKE, 1994, pp. 478–485.
[40] T. Kwon, Z. Su, Modeling high-level behavior patterns for precise similarity analysis of software, in: Proceedings of the 11th IEEE 11th International Conference on Data Mining, 2011, pp. 1134–1139.
[41] M. Irshad, Data of paper, 2020, URL https://drive.google.com/file/d/1to8bDQhF3Dv8rdUUJh-UNSZmPjejiCrR/view?usp=sharing.
[42] Online random picker, 2019, https://miniwebtool.com/random-picker/, accessed: 2019-05-13.
[43] W. Maalej, Z. Kurtanović, H. Nabil, C. Stanik, On the automatic classification of app reviews, Requir. Eng. 21 (3) (2016) 311–331.
[44] O. Ormandjieva, I. Hussain, L. Kosseim, Toward a text classification system for the quality assessment of software requirements written in natural language, in: Fourth International Workshop on Software Quality Assurance: In Conjunction with the 6th ESEC/FSE Joint Meeting, 2007, pp. 39–45.
[45] B.Y. Pratama, R. Sarno, Personality classification based on Twitter text using Naive Bayes, KNN and SVM, in: Proceedings of the International Conference on Data and Software Engineering, 2015, pp. 170–174.
[46] K. Soumya George, S. Joseph, Text classification by augmenting bag of words (BOW) representation with co-occurrence feature, IOSR J. Comput. Eng. 16 (1) (2014) 34–38.
[47] T. Mens, T. Tourwé, A survey of software refactoring, IEEE Trans. Softw. Eng. 30 (2) (2004) 126–139.
[48] Z. Xing, E. Stroulia, UMLDiff: an algorithm for object-oriented design differencing, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, 2005, pp. 54–65.
[49] D.J. Weller-Fahy, B.J. Borghetti, A.A. Sodemann, A survey of distance and similarity measures used within network intrusion anomaly detection, IEEE Commun. Surv. Tutor. 17 (1) (2014) 70–91.
[50] S. Sorlin, C. Solnon, J.-M. Jolion, A generic graph distance measure based on multivalent matchings, in: Applied Graph Theory in Computer Vision and Pattern Recognition, Springer, 2007, pp. 151–181.

[51] R.E. Caballero, S.A. Demurjian, Towards the formalization of a reusability framework for refactoring, in: International Conference on Software Reuse, Springer, 2002, pp. 293–308.

[52] Acceptance testing best practices, 2020, https://github.com/archfirst/acceptance-testing-best-practices, accessed: 2020-08-10.

[53] S. Erb, A Survey of Software Refactoring Tools (Master thesis), Baden-Württemberg Cooperative State University, Karlsruhe, Germany, 2010.

[54] H. Hemmati, A. Arcuri, L. Briand, Reducing the cost of model-based testing through test case diversity, in: Proceedings of the IFIP International Conference on Testing Software and Systems, 2010, pp. 63–78.

[55] J.K. Van Dam, V. Zaytsev, Software language identification with natural language classifiers, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, IEEE, 2016, pp. 624–628.

[56] K. Petersen, C. Gencel, Worldviews, research methods, and their relationship to validity in empirical software engineering research, in: Proceedings of the Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement, 2013, pp. 81–89.

[57] M. Staron, Action Research in Software Engineering, Springer, 2020.

[58] J. Maxwell, Understanding and validity in qualitative research, Harv. Educ. Rev. 62 (3) (1992) 279–301.