



Moldable scenario builder

Master Thesis

Ivan Kravchenko

University of Bern

June 2020

Prof. Dr. Oscar Nierstrasz

Andrei Chiş & Nitish Patkar & Nataliia Stulova

Software Composition Group
Institute of Computer Science
University of Bern, Switzerland



Abstract

Current behavior-driven development (BDD) practices promise to engage more stakeholders in an agile software development process through the use of behavior specifications of the software product. However, current capabilities for behavior specification restrict possible feedback as they fail to reliably connect the specification with the corresponding implementation.

We analyzed 14 BDD tools to observe their limitations for facilitating feedback between multiple stakeholders. We observed that the existing BDD tools differ in characteristics regarding their support for a ubiquitous language and specification format. Despite the recent attempts to write more natural language specifications, the existing tools are largely developer-oriented and limit the engagement of other participants. The analyzed tools focus mostly on asserting input values against desired business output for a BDD scenario and much less on manipulating the output itself.

To tackle the aforementioned limitations, we present our prototype solution ‘Moldable scenario editor’ implemented in the Pharo environment. To achieve this, we allow BDD scenarios to return objects and adapt their representation to the perspective of non-technical stakeholders.

We strive to engage more participants in the agile software development process by giving them different ways to experiment with behavior specifications within an IDE. For instance, we use an embedded rich text editor to illustrate how plain textual specifications can leverage the corresponding implementation. Similarly, with a combination of graphical elements, we allow users to experiment with domain objects *i.e.*, compose new BDD tests without having to code.

We speculate that such an approach can bind behavior specifications more closely to the implementation, and facilitate effective collaboration among team members.

Contents

1	Introduction	1
2	BDD Tools Analysis	4
2.1	Discussion	6
2.1.1	General characteristics	6
2.1.2	Degree of support for ubiquitous language definition	7
2.1.3	Syntax enforcement to specify behavior	8
2.1.4	Parametrized scenario support	8
2.1.5	Support for code generation	9
2.1.6	Scenario description format	9
2.1.7	Scenario execution output	9
3	Moldable Scenario Builder	12
3.1	Agile artifacts	13
3.2	Case overview	13
3.3	Pharo	15
3.4	GToolkit	15
3.4.1	Gtoolkit Inspector	15
3.4.2	Spotter	15
3.4.3	Coder	15
3.4.4	Documenter	15
3.4.5	Bloc	16
3.4.6	gtExample	16
3.4.7	gtParametrizedExample	16
3.5	From documentation to testing	17
3.5.1	Scenario and examples	18
3.5.2	Scenario editor functions	18
4	Discussion and future work	24
4.1	Code decoupling and code generation	24

4.2	Scenario editing	26
4.3	Live object inspection	26
5	Conclusion	28
	Appendices	30
A	Solution overview	31
A.1	Configuring specification with pragmas	32
A.2	EParametrizedExample	32
A.3	EScenarioInputWidget	32
A.4	EWidget	33
A.5	EScenarioParameterSelectionElement	33
A.6	'Pick Examples' graphical element	33
A.7	Generating new inputs	35
A.8	'Edit' graphical element	36

Chapter 1

Introduction

Contrary to the traditional waterfall software development paradigm, agile methodologies promote evolutionary software delivery model through (i) close collaboration between project stakeholders, (ii) continuous control of requirements, design and solutions in each iteration, and (iii) rapid feedback on changes [2, 9, 10].

However, adopting agile methodologies in practice raises multiple important challenges such as (i) losing the sight of the big picture as increasing requirements complicate development, (ii) requirement management influences long and short term planning, and (iii) inadequate engagement of stakeholders into project activities affects feedback and product transformations [6, 16].

To address such challenges, the software engineering community uses distinct practices to engage more stakeholders in the software development process and to increase the confidence in the implemented functionality. For instance, Test-Driven Development¹ (TDD) aims to focus on testing over implementation. In TDD, development starts with writing tests corresponding to the expected system behavior. As the tests fail, developers have to continuously refactor the implementation until the tests do not produce any more errors.

Another such practice is a Domain-Driven Design (DDD).² In DDD, software developers aim to build a system that concentrates on delivering domain logic corresponding to the underlying domain model. To achieve this, DDD encourages developers to use the terms from a ubiquitous language. A ubiquitous language³ is a collection of non-ambiguous terms inside a certain domain that is common and understandable to all the participants of the process.

Behavior-Driven Development (BDD), on the other hand, is a combination of TDD and DDD that attempts to connect business requirements with a working software product. BDD promises to bring

¹<https://martinfowler.com/bliki/TestDrivenDevelopment.html>

²<https://martinfowler.com/bliki/DomainDrivenDesign.html>

³<https://martinfowler.com/bliki/UbiquitousLanguage.html>

together the advantages of both TDD and DDD in order to improve collaboration between project stakeholders. It encourages developers to use a ubiquitous language within project teams, and by specifying the system behavior using such a language, all team members can participate in building a common product vision. For instance, the specifications that are written using the ubiquitous language become a vehicle for communication, thereby leaving a positive impact on stakeholder collaboration. At the same time, the process of writing specifications follows the red-green-refactor style from TDD. The BDD software development process facilitates iterative scenario implementation and views scenarios as acceptance test candidates to make sure that the working system conforms to the expected behavior.

To describe system functionality, often BDD tools use scenarios. Scenarios are usually plain-text written sequences of actions designed to be easily understandable for business stakeholders, Q/A, and developers. Each scenario is composed of several steps that are bound to the underlying implementation.

However, the adoption of BDD in practice seems to be low due to the following reasons: (1) convincing team members to use BDD is difficult, (2) lack of experience leads to poorly written scenarios, (3) duplications in BDD specifications cause performance issues, (4) specifications become difficult to change for the purpose or fault correction, (5) need to maintain new BDD tests increases software development time, and in general (6) cost of product maintenance becomes higher [11, 12, 19].

The major potential improvement for BDD is a more transparent and understandable mechanism of reflecting requirements in the implementation and vice versa. However, tracing business requirements from high-level descriptions to actual implementation is costly and tedious [7]. One of the reasons is the absence of an established way to connect written behavior with the code. On the one hand, current BDD guidelines recommend writing scenario implementation and validation parts similar to the plain text description. On the other hand, they do not define how one can guarantee the connection of ubiquitous language terms between specification and implementation. Such a situation poorly encourages all participants of the agile process to take part in the creation of BDD scenarios [?]. Hence, the implemented behavior strongly depends on the developers' comprehension. This might restrict business stakeholders and other participants from an active cooperation during the development lifecycle *e.g.*, self experimenting with scenarios inputs and sharing the knowledge and experience regarding system robustness and extensibility. In this thesis, we experiment with an approach to connect specifications closely with the implementation. Particularly, we explore (i) 'living documentation' to specify behavior in an IDE, and (ii) 'living objects' to modify the behavior at system run time.

Consequently, we aim to answer the following research questions:

- **RQ₁**: *What are the limitations of the state-of-the-art BDD tools regarding software behavior specification?*

We analyzed 14 active open-source BDD tools and assessed them across 9 parameters.

- **RQ₂**: *How can we closely couple specification and implementation and what advantages does it give over existing tools?*

We present our prototype solution 'Moldable scenario editor' modeled in the Pharo environment.

The remainder of this thesis is structured as follows: In chapter 2 we report the state-of-the-art BDD tools and discuss their limitations. In chapter 3 we describe the building blocks of our solution, provide implementation details, and discuss the advantages over the existing tools. Finally, in chapter 4 we compare the novelties of our solution with the existing tools and discuss future work.

Chapter 2

BDD Tools Analysis

BDD has gathered a lot of attention from practitioners since its conceptual introduction. The focus of BDD so far has been on making behavior specification understandable for multiple project stakeholders. Comprehension of implementation details for non-technical stakeholders might be challenging, and BDD tools promise to address it by improving the readability of the executable specifications. However, readability is not the only reliable characteristic. To be widely used by the community, BDD tools have to correspond to the developer requirements as well. Such requirements include, but are not limited to: (i) demand for a close coupling between the specification and implementation details, and (ii) the possibility of automatic code generation from the specification.

Previous studies have assessed BDD tools from the perspective of their primary target users, support characteristics for BDD based on the input and output formats, and which phases of software development they support [11, 13, 18]. The captured values are mostly subjective and the reporting lacks concrete insights into the specific characteristics of the measured parameters. For instance, Okolnychyi *et al.* report whether the BDD tools support *business readable input*, but do not provide concrete details of the type and characteristics of the possible inputs.

We combined tools from the aforementioned studies and expanded the list with the new ones. Previously analyzed tools such as StoryQ,¹ JDave,² NBehave,³ Easyb,⁴ and BDDfy⁵ had to be excluded as they are either obsolete or no longer maintained [11, 13]. Eventually, we analyzed 14 existing BDD tools to characterize how they support behavior specification and reflect on our assessment across parameters described in Table 2.1.

¹<https://archive.codeplex.com/?p=storyq>

²<https://github.com/jdave/JDave>

³<https://github.com/nbehave/NBehave>

⁴<http://easyb.io/v1/index.html>

⁵<https://teststackbddfy.readthedocs.io/en/latest/>

Specifically, we take a look at: Cucumber,⁶ FitNesse,⁷ JBehave,⁸ Concordion,⁹ SpecFlow,¹⁰ Spock,¹¹ RSpec,¹² MSpec,¹³ LightBDD,¹⁴ ScalaTest,¹⁵ Specs2,¹⁶ JGiven,¹⁷ phpspec,¹⁸ and Gauge.¹⁹

The results of our tool comparison are summarized in Table 2.2. Additionally, the following information is also collected: (i) the URL to the homepage, (ii) the URL to the source code repository, (iii) the last update in the source code repository, (iv) number of contributors in the source code repository, and (v) number of stars of the source code repository (see Table 2.3). Data were gathered on 26th May 2020.

The following conventions are used in Table 2.2: every ‘Yes’ value is shown as ✓, whereas, ‘No’ as ✗.

-
- ⁶<https://cucumber.io/>
 - ⁷<http://fitnesse.org/>
 - ⁸<https://jbehave.org>
 - ⁹<https://concordion.org>
 - ¹⁰<https://specflow.org>
 - ¹¹<http://spockframework.org/>
 - ¹²<http://rspec.info>
 - ¹³<https://github.com/machine/machine.specifications>
 - ¹⁴<https://github.com/LightBDD/LightBDD>
 - ¹⁵<http://www.scalatest.org/>
 - ¹⁶<https://etorreborre.github.io/specs2/>
 - ¹⁷<http://jgiven.org/>
 - ¹⁸<http://www.phpspec.net/en/stable/>
 - ¹⁹<https://gauge.org>

Table 2.1: Tool analysis parameters

Nr	Parameter	Description	Recorded values
1	Degree of support for ubiquitous language definition	The tool encourages or enforces behavior specification using a ubiquitous language.	'High': tool specifies objects in specification using a binding layer, 'Medium': tool converts primitives from the specification into objects, 'N/A': no support.
2	Syntax enforcement to specify behavior	The tool forces users to write the specification in a specific format, <i>i.e.</i> , use specific keywords or annotations on methods to denote a state or the flow.	Yes/No
3	Interface to write scenario	Details of the environment where scenarios are specified	'Text editor': the user writes a scenario in any system text editor, without syntax support ('T'), 'IDE': there are certain extensions and plugins that allow for scenario writing directly in an IDE ('I'), 'Web interface': framework supports browser-based interface for writing specifications ('W').
4	Scenario description format	This characteristic shows how the scenario description is represented	'Semi-structured text': denotes auxiliary to code file, for example .spec, .feature etc. ('S') 'Table of values': a table-like structure ('T'), 'Code': reflects that the executable scenario is only backed up by code ('C')
5	Parameterized scenario support	Whether the tool allows more than one set of values to be injected in a scenario.	Yes/No
6	Parameterized primitive support	Whether the users can insert primitive types such as any of: string, boolean, number in parameters.	Yes/No
7	Parameterized object support	Whether the users can inject object values, in particular, objects from the domain model in parameters.	Yes/No
8	Support for code generation	The tool has any possible code generation methods such as any of: creating test method placeholders, extracting documentation from the test, or adding new values as parameters.	Yes/No
9	Test output format	This parameter collects information regarding the scenario output format.	Pass/Fail status ('S'), Output object ('O')

2.1 Discussion

Below we discuss the results of our tool analysis to answer RQ₁.

2.1.1 General characteristics

All analyzed tools are open-source and are actively developed on GitHub. We find that most tools allow behavior specification in plain text with added annotations which bind to tests or code snippets. Similarly, to avoid a lot of rewriting in case requirements change, several tools also encourage specifying behavior in

classes and corresponding methods complemented with plain text annotations. In any case, specifications are glued to an already existing testing framework. Among all analyzed tools we identified no support for graphical interfaces, that is, building the specification and/or implementation using graphical elements such as in state chart diagrams.

2.1.2 Degree of support for ubiquitous language definition

Writing specifications with a ubiquitous language is an effective approach for an unequivocal understanding of behavior. Tools that have such capabilities reduce scenario ambiguity and guarantee an understandable context of execution. According to our analysis, Concordion and Gauge seem to facilitate binding to a ubiquitous language to a higher degree. While neither specifies how exactly users should write specifications, they allow binding to the code:

- Concordion does not force one to use any predefined document or sentence structure. Instead, users link the text pieces with code using ‘commands’ (see Figure 2.1). This is usually done in a specific file called a fixture.
- Gauge proposes a similar idea. It defines aliases to an application model in user specifications, but its possibilities in this scope are more limited than Concordions. For example, using Gauge with Java, users can bind objects with Enum fields.

```
# Adding a contact in address book

To help support edit functions,
contacts first need to be added to an address book.
The system checks that contact name and phone number are unique.

Adding a ['Jane Doe'](#user=janeDoeUser())
into an ['empty Address Book'](#addressBook=emptyAddressBook())
will have the size of [1](-"=?=result.size")
```

Figure 2.1: Linking specification with code in Concordion

On the other hand, tools with ‘Medium’ support mostly provide: (i) primitives binding, (ii) binding multiple specifications in one table-like structure, and (iii) user-defined object mappers from primitives (see Figure 2.3). Figure 2.2 demonstrates primitives binding in a Cucumber environment for Java programming language. The parameters are defined as regular expressions in annotation, and then injected as strings inside a specification. Although primitive binders and mappers are handy, they are impractical to use with complex domain entities with many properties or entities that are hard to describe using only text *e.g.*, HTTP requests or 3D shapes.

In our opinion, the ideal and effective use of a ubiquitous language would be to implicitly connect terms from the textual specification with domain objects. None of the analyzed BDD tools facilitate such automatic binding. Although there have been several attempts to automate such a process, current frameworks largely lack this feature [14, 17].

```

@When("^User enters \"(.*)\" and \"(.*)\"$")
public void user_enters_UserName_and_Password(String username, String password)
    throws Throwable {
    driver.findElement(By.id("log")).sendKeys(username);
    driver.findElement(By.id("password")).sendKeys(password);
    driver.findElement(By.id("login")).click();
}

```

Figure 2.2: Defining parameters in a Cucumber specification

```

@ParameterType("red|blue|yellow")
public Color color(String color){ // type and name
    return new Color(color);      // mapping function
}

```

Figure 2.3: Binding parameter values to a mapper method

2.1.3 Syntax enforcement to specify behavior

A significant number of tools (8 out of 14) enforce a specific syntax for behavior specification. As such, it gives users the possibility to describe behavior in a unified manner, standardizing the structure. For example, jBehave enforces the *'Given..When..Then'* structure mentioned inside an auxiliary file (see Figure 2.4), whereas, mspec expects to write functions for *'Establish..Because..It'* in the corresponding programming language (see Figure 2.5).

```

Scenario: trader is not alerted below threshold

Given a stock of symbol STK1 and a threshold of 10.0
When the stock is traded at 5.0
Then the alert status should be OFF

Scenario: trader is alerted above threshold

Given a stock of symbol STK1 and a threshold of 10.0
When the stock is traded at 11.0
Then the alert status should be ON

```

Figure 2.4: jBehave specification structure

We speculate constrained syntax to be effective at validating specification and implementation consistency in constantly expanding requirements, rapid versioning, and regression testing.

2.1.4 Parametrized scenario support

Parameterized support is extremely helpful in situations when the acceptance tests are repeated with different input values. It helps developers write fewer tests for similar sets of assertions. We found that 10 out of 14 tools allow different values to be injected as parameters inside the same specification. Only

```
Establish context = () =>
    subject = new SecurityService();

Because of = () =>
    user_token = subject.Authenticate("username", "password");

It should indicate the users role = () =>
    user_token.Role.ShouldEqual(Roles.Admin);

It should have a unique session id = () =>
    user_token.SessionId.ShouldNotBeNull();
```

Figure 2.5: mspec specification structure

Concordion and ScalaTest allow us to define objects, but they follow different approaches.

We consider the lack of parameterized support among the majority of BDD tools as a weak point, as it forces developers to write more repetitive code structures and specifications. This might affect specification readability, hinder the implementation aspects, and increase overall development time in a response to a requirement change. We address such a problem in our solution, as explained later in chapter 4.

2.1.5 Support for code generation

Generated code saves a significant amount of time and prevents developers from making repetitive errors. We think that, in the context of BDD, the best code generation functionality should consider the scenario context. None of the inspected tools can generate executable code automatically from the specification. Instead, frameworks allow users to create boilerplate code as placeholders to be filled with right functionality by programmers. Three tools (Cucumber, JBehave, Specflow) that support code generation are based on semi-structured text, and only phpspec generates code from other code.

2.1.6 Scenario description format

The analyzed tools provide distinct ways to specify software behavior. Based on the current framework documentations, we only identified two formats: semi-structured text and code. Unfortunately, we did not find frameworks that specify graphical alternatives, for example, binding scenario steps using visual elements such as arrows and blocks. We believe that such functionality will improve user experience. We address our perception of interaction with graphical elements in chapter 3.

2.1.7 Scenario execution output

Our findings suggest that the analyzed tools assume that they generate outputs that are understandable by non-technical stakeholders, however, this is subject to evaluation. There are several points to consider: (i) the test output from CI/CD pipelines can be easily analyzed in case of failure, (ii) plain language with a combination of ubiquitous language in output help to further understand the context of implementation, and (iii) test output is used for report generation for business purposes.

We are particularly interested in how users explore specification execution results. It turns out to be


























Table 2.2: BDD Framework Comparison

Tool	Support degree for ubiquitous language definition	Syntax enforcement to specify behavior	Parameterized scenarios	Primitive support	Object support	Code generation	Interface to write scenario	Scenario description format	Test output format
Cucumber	Medium	✓	✓	✓	✗	✓	T, I, W	S	S
FitNesse	Medium	✗	✓	✓	✓	✗	T, I, W	S, T	S
JBehave	Medium	✓	✓	✓	✗	✓	T, I	S	S
Concordion	High	✗	✓	✓	✓	✗	T, I	S	S
SpecFlow	Medium	✓	✓	✓	✗	✓	T, I	S	S
Spock	Medium	✓	✓	✓	✗	✗	T, I	C	S
RSpec	Medium	✓	✗	✗	✗	✗	T, I	C	S
MSpec	Medium	✓	✗	✗	✗	✗	T, I	C	S
LightBDD	Medium	✗	✓	✓	✗	✗	T, I	C	S
ScalaTest	Medium	✓	✓	✓	✓	✗	T, I	C	S
Specs2	Medium	✗	✓	✓	✗	✗	T, I	C	S
JGiven	Medium	✓	✗	✗	✗	✗	T, I	C	S
phpspec	Medium	✗	✗	✗	✗	✓	T, I	C	S
Gauge	High	✗	✓	✓	✗	✗	T, I	S	S
Our solution	High	✗	✓	✓	✓	✓	I	S	S, O

an industry practice, that tests, reflecting specifications, are usually void methods as they do not return anything. Instead, developers rely on assertion logic inside the test. However, in such a situation, the accessibility to evaluate test results belongs either to programmers or to Q/A. We consider it to be a limiting factor for other participants of the process. We want to investigate the way users can interact with returned value from BDD specification examples concerning the domain model.

Additionally, we consider many tools specifications being constrained in readability ahead of execution. It is especially inherent to code-oriented tools, as the plain text and code, when mixed, may cause confusion and misunderstanding. Moreover, it is unlikely that non-technical stakeholders will take a look at such specifications, let alone understand the details. We address the topic of consuming the specification ahead of execution in the section 3.5.

Table 2.3: BDD Tools Repository Statistics

Tool	Homepage	Repository	Last update	Number of contributors	Stars
Cucumber			23.May.20	87	2910
FitNesse			10.May.20	104	1521
JBehave			24.May.20	62	240
Concordion			16.May.20	13	184
SpecFlow			25.May.20	132	1606
Spock			8.May.20	79	2649
RSpec			18.Dec.19	19	2633
MSpec	-		11.May.20	63	775
LightBDD	-		24.May.20	4	151
ScalaTest			14.May.20	32	897
Specs2	-		27.Apr.20	73	681
JGiven			22.May.20	34	287
phpspec			17.Dec.19	124	1690
Gauge			20.May.20	45	2188

Chapter 3

Moldable Scenario Builder

As discussed earlier, working with a ubiquitous language within existing BDD frameworks is still challenging. Most of the glue code is still written by developers, especially in tools that have only ‘Medium’ support. We want to engage other participants in the agile process without a need to write actual code. We speculate that by using our solution, the engagement of other domain experts can positively reflect on system reliance and robustness, decrease the communication gap with technical people, and provide new ways of experimenting with the domain model. Our approach engages stakeholders in playing and experimenting with the executable specification by using different ways to interact with programming objects and documentation. To answer RQ₂ we introduce our prototype implementation called ‘Moldable scenario editor’ that aims to solve the aforementioned problems.

Considering that BDD encourages you to write system behavior tests, it can be seen as contributing to functional testing to a certain degree. Functional testing verifies system functionality with a ‘black-box’ method, where test data is taken from software specification [1]. Performing functional testing is difficult as one often needs to have a deployed application or a set of applications in a test or a stage environment to ensure correct functionality. Stable support of such an environment can be tedious, as developers have to deal with the following problems [3, 5, 15]: (i) artifact versioning, (ii) test repeatability, and (iii) tracking changes made by function calls through multiple artifacts.

At the same time, functional testing indicates whether the system conforms to the intended behavior by allowing all team members to interact with the underlying domain model at run time. Moreover, testing the system in a ‘black-box’ mode requires the developers to abstract from its internal behavior and concentrate on the intermediary or resulting execution outputs. The same can be tested within the BDD approach. Our solution can potentially have a positive impact on the aforementioned challenges, while at the same time supporting system robustness at a level which functional testing assumes. We speculate that returning and inspecting domain objects from executable BDD specifications will exceed the experience of functional testing, and make domain models truly tangible and manipulable.

This section is structured as follows: first, we introduce our running example and demonstrate a requirements workflow modelled inside an IDE in section 3.2. Next, we describe the Pharo programming environment and introduce essential tools from GToolkit that we leveraged in our implementation (see section 3.3). Finally, in section 3.5 we show how our solution allows users to work with BDD tests and specifications.

3.1 Agile artifacts

To support the iterative approach, teams consume several agile artifacts, often a composable hierarchy of tasks. They begin with the decomposition of high-level requirements or product visions into the final low-level technical details. In this work we concentrate on the following particular agile artifact setup:

Epics. Epics are high-level large pieces of work, often delivered through multiple iterations or sprints. They are useful for starting the development process, imagining the product as a whole. One often uses epics to break the work down into artifacts, wherein each particular sub-artifact contributes to a common epic-defined goal. Usually, it is difficult to keep epics consistent and even more difficult to release them according to estimated deadlines [8].

Use cases. Use cases are acknowledged high-level descriptions of system actions from a user perspective. Use cases are often more detailed than epics, but do not contain low-level requirements. In use cases, the user is either a person, a role, an organization, or any other system. A use case is often described as a flow of steps with preconditions, a main success scenario, and post conditions, including success guarantees, extensions *etc.*

User stories. A user story is a concrete sequence of business-related actions that helps teams to identify *who* makes the action, *why* this action is important for the system and *what* value it introduces. They are most often written in a simple, easy-to-grasp language that emphasizes the desired outcome. Usually they are represented as a single sentence. In contrast to epics and use cases, user stories, due to their simplicity, are easy to estimate.

3.2 Case overview

Imagine that we need to create an address book and model simple functionality: add, edit, and search contacts inside of it. To show it as a part of the agile process, we first create wrapper classes for our agile artifacts and build a hierarchy of them as described below.

We start first by defining the *epics*. The Epics include actionable *use cases*. Then, we split the use cases into multiple *user stories*. The user stories are designed to contain multiple *scenarios* entities. Scenarios are concrete steps of actions inside the system that cover the system behavior specified in a user story and a use case. Unlike to the user story, they contain more technical information and confirm the user story

value with multiple assertions. Scenarios aggregate *examples* and *parameterized examples*. The notion of *examples* and *parameterized examples* is explained in subsection 3.4.7.

Imagine that we want to model the functionality of adding a contact in an address book. Here is how it could be organized according to our representation:

- *Epic*: Manage an address book.
- *Use case*: A user adds a contact inside a book.
- *User story 1*: As a User I can add a new contact inside an existing address book.
- *User story 2*: As a User I can add a new phone number to a contact.
- *Scenarios*: Contains the following examples for *user story 1*
 - Add a contact in an empty book
 - Add a contact with same name
 - (parameterized) Add a contact in a book

The above scaffolding shows how we can model requirements workflow inside an IDE without a need to integrate third-party tools (see Figure 3.1). One can trace requirements from epic to example inside an IDE, where each gray tab shows object representations.

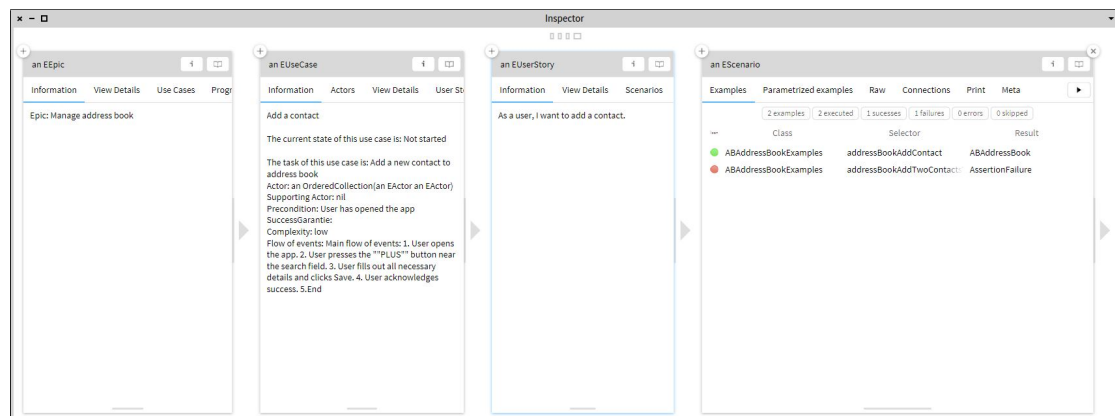


Figure 3.1: Tracing requirement from an epic to example

We speculate that such a requirements workflow inside an IDE would make development more effective as developers would spend less time finding the right piece of code that corresponds to the requirement specification. Additionally, non-technical stakeholders get a more transparent way to observe the state of development and inspect attached tests, documentation *etc.* (see Figure 3.5, Figure 3.10).

3.3 Pharo

Pharo is a modern dynamic development environment for a dialect of Smalltalk. The Pharo Virtual Machine allows developers to modify and execute the code on the fly. The Pharo environment offers specific features for development such as:

- *Do it and go* instantly executes any selected piece of code
- *Inspect it* executes selected piece of code and displays the returned result in an Inspector window
- *Print it* executes the selected piece of code and prints the result
- *Profile it* allows one to quickly profile the code

3.4 GToolkit

GToolkit is a moldable development environment that offers a unique, live coding experience.¹ It improves on existing Pharo tools such as Inspector and Coder. In particular, our solution uses the following GToolkit components:

3.4.1 Gtoolkit Inspector

The Inspector offers a more intuitive and rich interface than the usual Pharo Inspector. It can provide alternative user-defined representations of objects and ways to navigate through them. We create views for our domain objects to deliver varying degree of interaction for different types of users.

3.4.2 Spotter

Spotter is a universal search engine inside the Pharo environment. We use Spotter capabilities to create search-based widgets.

3.4.3 Coder

The Coder represents a code editor for static code manipulations. Our solution integrates Coder during a code generation process.

3.4.4 Documenter

Documenter combines documentation and code. Code snippets can be integrated inside Documenter and either be displayed as actual code or evaluated on the fly to generate graphical representations. We model a process of requirements documentation utilizing Documenter, allowing one to create clear and understandable specifications.

¹<https://gtoolkit.com/>

3.4.5 Bloc

Bloc is a standalone UI framework. Most of the GToolkit user interfaces, drawn graphics and representations use Bloc.² Our UI interfaces are also purely based on this library.

3.4.6 gtExample

GToolkit offers a `<gtExample>` pragma³ (*i.e.*, annotation over methods) that enables user interfaces to execute a piece code inside the annotated methods as if they were tests (see Figure 3.2). The figure shows a function annotated with such a pragma. On the user click on 'play' button it will execute and return a domain object. Such methods are not supposed to have arguments, rather, do a unit of work independently and return a result.

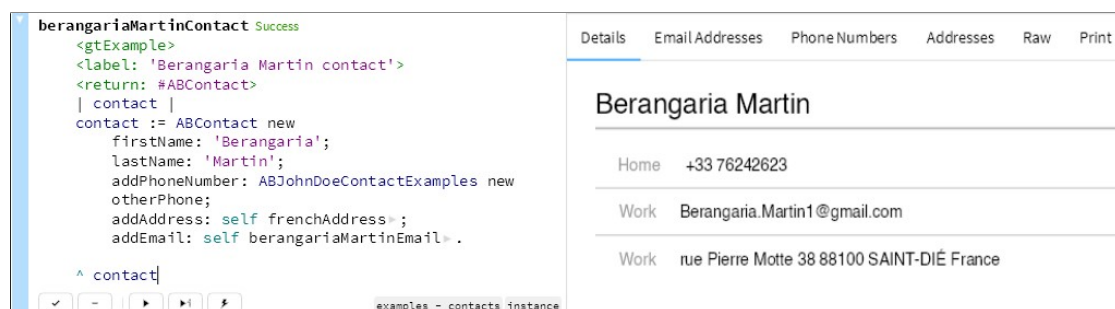


Figure 3.2: `gtExample` annotated method and execution result

In fact, not only can we use `<gtExamples>` to return domain objects, but also to contain assertions, *i.e.*, introduce testing capabilities. Since `gtExample` is a simple annotated method, we can further reuse them in other methods, thus allowing the tests and examples to be composable. As such, a method in Figure 3.3 makes an assertion and invokes another `gtExample` (`self addressBookWithFullTestContact`).

The concept of example reusability and composability is already demonstrated in the form of documentation examples in the GToolkit framework. For instance, `Spotter`, `Coder` and `Documenter` each have a separate sub-package 'Examples' that lists various `<gtExample>` methods that cover main library functionality.

3.4.7 gtParametrizedExample

`<gtParametrizedExample>` is another pragma that inherits from `<gtExample>`. It is the same executable method with the exception of having arguments in the method signature (see Figure 3.4). We aim to extend the IDE capabilities to support parameterized examples. Consequently, we intend to create

- new UI elements for working with executable parameterized examples, and

²<https://github.com/feenkcom/Bloc>

³<https://github.com/pharo-open-documentation/pharo-wiki/blob/master/General/Pragmas.md>

```

AddressBook > ABAddressBookExamples search
exampleSearchCompanyWithOneContact N/A
<gtExample>
<label: 'one contact search company'>
<description: 'Verify that searching company within one
contact is successful'>
<return:#ABAddressBook>
| book resultOneCompany |
book := self addressBookWithFullTestContact .
resultOneCompany := self exampleSearchCompany: 'Company'
in:> book.
self assert: resultOneCompany size equals: 1.
^ book

```

Figure 3.3: gtExample as a test

```

AddressBook > ABAddressBookExamples as yet unclassified
exampleSearchCompany: aCompany in: aBook
<gtParameterizedExample>
<gtExample>
<given: #aCompany ofType: #String in:#'
elementType:#input>
<given: #aBook ofType: #ABAddressBook in:
#ABAddressBookExamples
elementType: #search>
<return: #ABAddressBook>
<label: 'Parameterized search of a company in an address
book'>
| result |
result := aBook searchCompanies: aCompany.
^ result

```

Figure 3.4: gtParameterizedExample as a parameterized test

- infrastructure to combine tests and specifications under a single IDE interface

In the rest of this chapter, we refer to a <gtExample> as an *example* and to a <gtParameterizedExample> as a *parameterized example*.

3.5 From documentation to testing

We want to expand the way end users interact with tests. Our current analysis in chapter 2 shows that most frameworks rely on pass/fail output format. We speculate that with such format users might trust the test execution result as is. Through our solution we aim to let users inspect the results of test execution and try out new, possibly undiscovered inputs. An environment, oriented towards experimentation with input values for specification examples can potentially deliver a more comprehensive view of a problem in comparison to rich text editors or other IDEs, check for corner cases, or bring improvements to the domain model based on user impression.

3.5.1 Scenario and examples

Ideally, a specification is backed up with an executable scenario, represented as a test. In our case it is a `<gtExample>` annotated method with additional assertions. To give users another way to look at the test result, we let such examples return objects and inspect them right afterward. Figure 3.5 demonstrates how example `addAddressBookToContact` successfully executes (after clicking on circle button system runs the test and marks it green) and returns an `ABAddressBook` object with one contact.

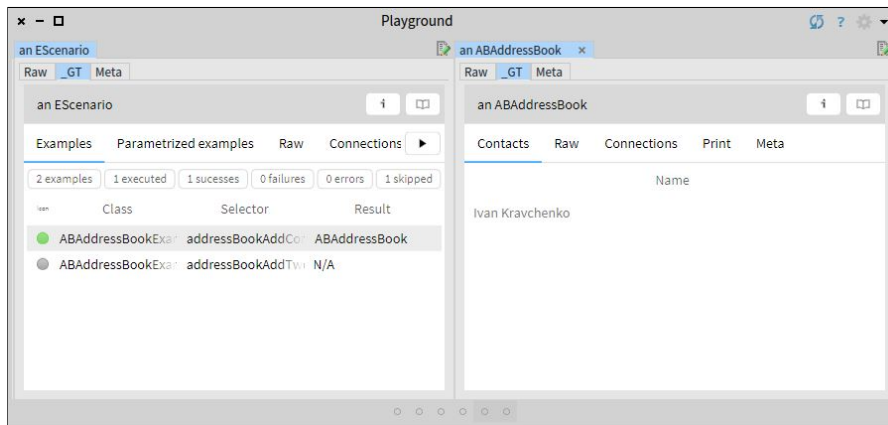


Figure 3.5: `gtExample` returns an object

Custom views. The tab 'Contacts' is a custom view plugged into this object. Such a view generates an enhanced Bloc graphical element that lists items (here, contacts in a book). The view is picked up by a GToolkit engine and rendered in the Inspector window among other tabs.

In this concrete example it only shows the first name and the last name, but can be designed to display any values. Inspector tabs are configurable *i.e.*, programmers can change them as they like, remove or add new ones. Additionally, end-users might provide faster feedback on what final system elements might look like without the need to build sophisticated UIs. Figure 3.6 demonstrates how contact information can be customized to look more appealing than the standard value inspection.

3.5.2 Scenario editor functions

To engage the users even more, we let them experiment with returned values. By using our editor, one can:

- execute tests against a chosen value,
- automatically and semi-automatically generate new examples,
- edit parameterized scenario properties in a user-friendly way, and
- generate new domain objects.

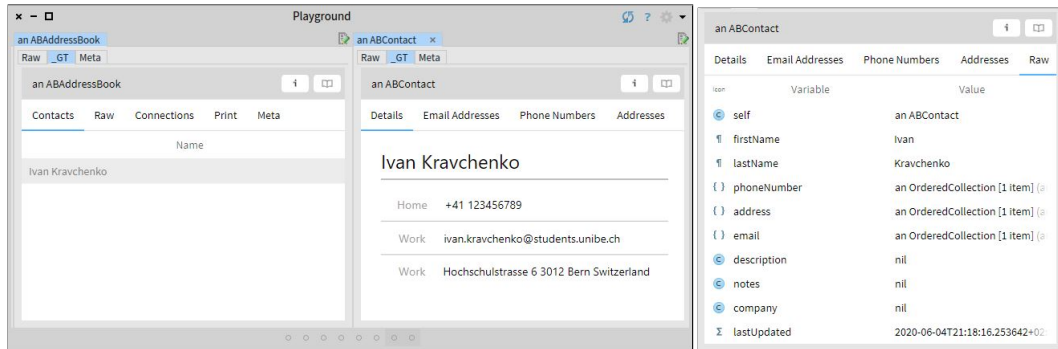


Figure 3.6: Custom and default Inspector views

Choosing values for parameterized example. To give users the ability to experiment with various test inputs, we introduce a specific Inspector view ‘Pick Examples’ that is attached to a parameterized example. The idea behind it is to provide a user-friendly input for any possible kind of test values. For example, a scenario ‘Search a contact in an address book’ can be illustrated with the following widgets (see Figure 3.7).

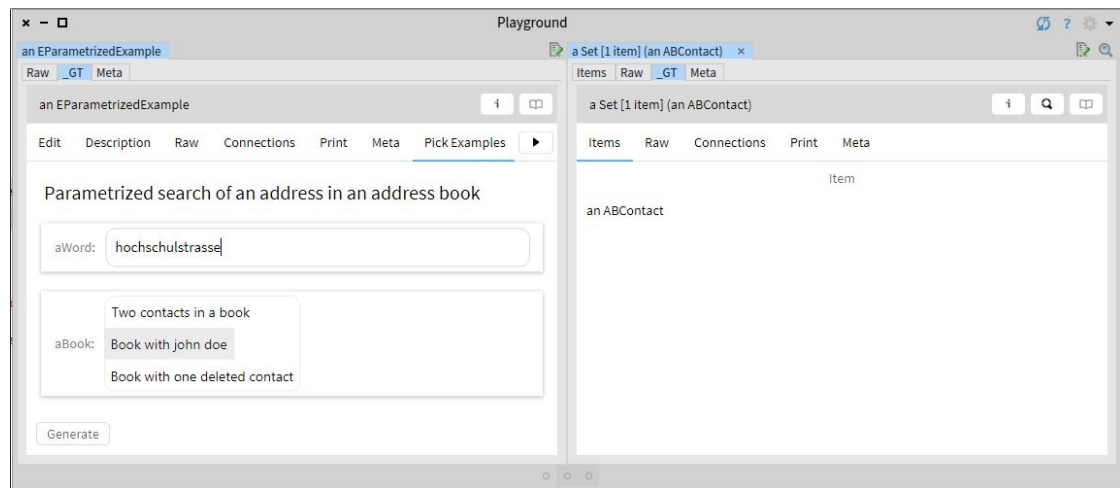


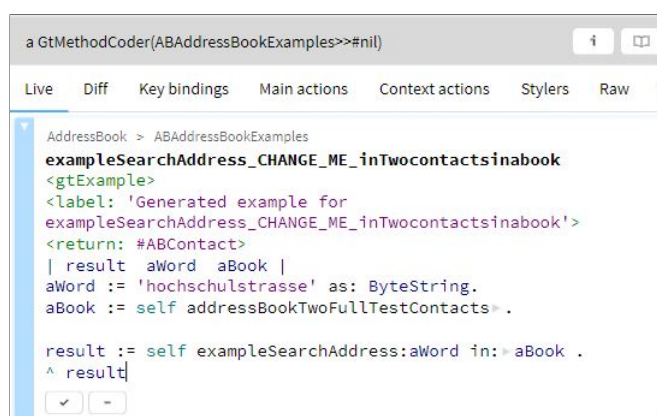
Figure 3.7: Selecting input values for contact search

Here, the parameterized example accepts two arguments (`aWord`, `aBook`). Two arguments are correspondingly rendered as widgets on the canvas. *Widgets* are self-contained graphical elements that expect a certain user input, either by manually entering the value or through interaction. In this case, two widgets will be created. The first text input widget only accepts a manual string input and second one a limited choice of available examples. When the user clicks on a ‘Run’ button on the upper right side, the view extracts the entered values and executes them against a defined functionality (in this context, searching a contact by address in a contact book). The result, a Set of contacts, similarly to the execution of a `<gtExample>`, will be shown in the Inspector.

The functionality of making widgets is handled by adding or removing a pragma(s) `<given:ofType:in:elementType:>` to a parameterized example. This pragma represents *what* (class) and *how* (input type) the user can enter something. The ‘Pick Examples’ view traverses pragmas inside a parameterized example and based on their values renders the corresponding UI elements.

Pharo is a dynamically typed language. This means that types of arguments passed inside a method are only known at run time. Such allows one to write Pharo code and get fewer compilation errors, but to expect more run time errors during actual code execution. To avoid possible difficulties with dynamic typing we consider specifications to strictly define the types of input and output values. Therefore, we want to be consistent with method contracts and check that input and output correspond to the provided description.

Generating new examples. We assume that at a particular moment, users might desire to save the results of their experiment with inputs. To facilitate that, we let non-technical stakeholders semi-automatically generate new tests by interacting with the UI and save them in a codebase. In Figure 3.7, when a user clicks on the ‘Generate’ button, the system will used to information from the execution context. The extracted information will be used to generate a valid method description and shown inside a Coder window. The Coder window permits one to change everything that is generated (*i.e.*, values, method name, pragmas) and subsequently saves the result in the Pharo environment (see Figure 3.8).



```
a GtMethodCoder(ABAddressBookExamples>>#nil)
Live Diff Key bindings Main actions Context actions Stylers Raw C
AddressBook > ABAddressBookExamples
exampleSearchAddress_CHANGE_ME_inTwocontactsinabook
<gtExample>
<label: 'Generated example for
exampleSearchAddress_CHANGE_ME_inTwocontactsinabook'>
<return: #ABContact>
| result aWord aBook |
aWord := 'hochschulstrasse' as: ByteString.
aBook := self addressBookTwoFullTestContacts.

result := self exampleSearchAddress:aWord in:aBook .
^ result
```

Figure 3.8: Generated new example with Coder

Editing a parameterized scenario. As the system evolves, so does the specification. Having a consistent comprehension of both is utterly important. Imagine, after a certain development period, business requires to alter the current contact search as described in subsection 3.5.2 by changing to another domain model of contact. We introduce another ‘Edit’ view, targeting to address the problem of changing requirements. Using such a view, one can change the input domain model class without touching the actual implementation (see Figure 3.9).

Among allowed changes, users can edit all widget parameters (*i.e.*, parameter name, expected returned

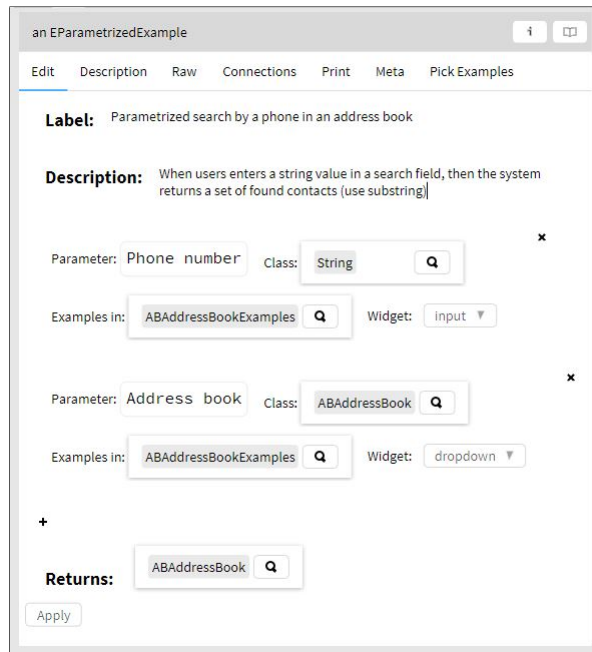


Figure 3.9: Editing an existing example

class, search scope, widget type), add new ones or remove existing ones, and change documentation for a particular parameterized example. The scenario editor stages all user changes and applies them upon the ‘Apply’ button click. After a successful change, the ‘Pick Examples’ view will re-instantiate to correspond to the renewed parameters.

New domain objects. The same search operation for a contact in an address book can have only a few provided examples. One might wish to test the search functionality on new, self-created domain objects. However, to create those would mean to interact with the implementation. We provide a way of using parameterized examples to generate new model instances. This can be useful in cases when non-technical people would like to create their entries for BDD specifications but have no prior programming experience. A parameterized example is simply a method with arguments, but with a combination of knowledge about the domain model one can program interfaces to let others create new instances, as shown in Figure 3.10, where a new contact is being created. We model such use cases using the same aforementioned widgets.

Writing scenarios with Documenter. To couple specification and implementation inside a single scenario even closer, we strived to compose a view that displays them both. Ideally, it should consist of a plain-text part, describing the behavior, supplemented with examples that confirm it. We model such a requirement for a scenario ‘Add a contact in an address book’ (see Figure 3.11).

Inside the view users are shown four elements:

- *Description and Scenario.* This is a Documenter view that embeds both the short and detailed textual

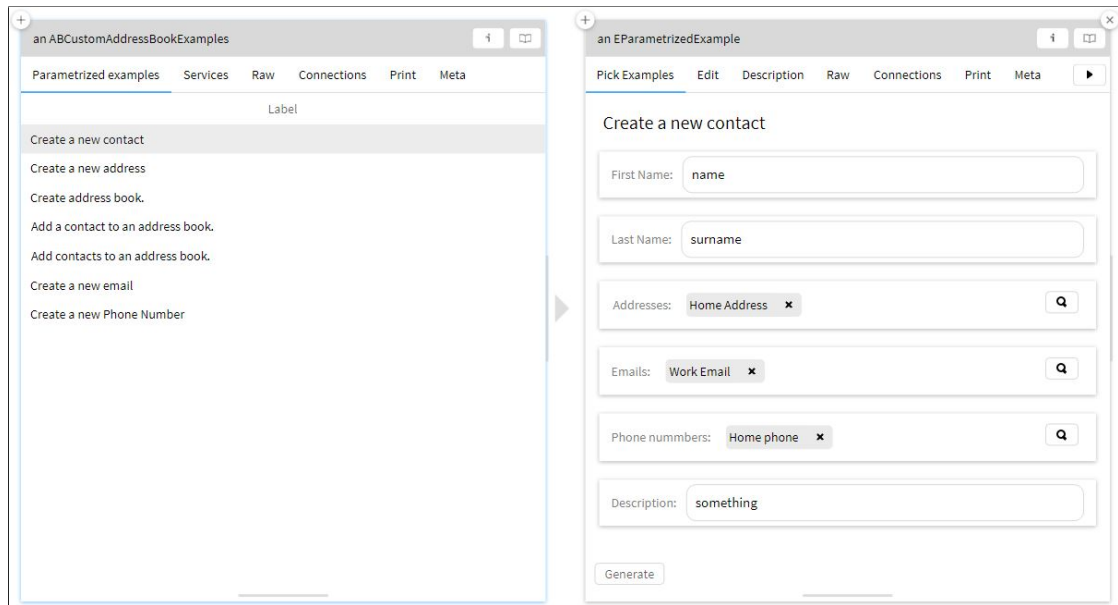


Figure 3.10: Creating an example for contact entity

description.

- *Simple example.* A Second Documenter is responsible for showing how the scenario should be used. In Figure 3.11, the text is additionally supplied with the actual Pharo objects that use custom UI representations. For example, one can inspect an actual `ABAddressBook` object view with one 'John Doe' contact, or a `ABContact` custom view for 'Berangaria Martin'.
- *Examples.* This view represents all examples the scenario contains. Examples snippets can be expanded and executed inside the same window.
- *Parametrized examples.* This view represents all parameterized examples the scenario contains.

With Documenter, one can create a comprehensive text document containing live examples. This allows us to insert and observe objects alongside the textual description, provide necessary code snippets, execute them and observe the result at the same time.

Description:
Adding a contact to an address book

Scenario:
Feature: Add a contact to an address book
Given: AddressBook `ABAddressBook` and a Contact `ABContact`
When: The user clicks on 'Add' button in a contact list
Then: the chosen contact should be added.

Simple example:
Imagine we have a simple `address book` with one contact:

Name

John Doe

We want to add another `contact`:

Berangaria Martin

Work: +33 75142315234

Work: Berangaria.Martin1@gmail.com

Work: rue Pierre Motte 38 88100 SAINT-DIÉ France

After the execution of `ABAddressBook>>#addContact:`

The size of book should increase by one.

Test case: `ABAddressBookExamples>>#testExampleAddContactToAddressBook`

Name

John Doe

Berangaria Martin

Examples:

AddressBook > ABAddressBookExamples
addressBookAddContact

AddressBook > ABAddressBookExamples
addressBookAddTwoContactsWithSameName

Parametrized Examples:

ABAddressBookExamples>>#exampleAddContact: toAddressBook:

```

AddressBook > ABAddressBookExamples
exampleAddContact: aContact toAddressBook: anAddressBook
<gtParametrizedExample>
<gtExample>
<label: 'Add a given contact to an address book'>
<description: 'Add a given contact to an address book'>
<given: #aContact ofType: #ABContact in: #ABContactExamples
elementType: #dropdown>
<given: #aBook ofType: #ABAddressBook in: #ABAddressBookExamples
elementType: #search>
<return: #ABAddressBook>
anAddressBook addContact: aContact.
^ anAddressBook

```

Save

Figure 3.11: Creating an example for contact entity

Chapter 4

Discussion and future work

The ‘Moldable scenario editor’ is a prototype solution that aims to change the perspective on how scenarios can be composed, modelled, and described. Using the scenario editor, one can explore similar, but different interactions with tests and specifications. Below we reflect on concrete differences our approach has over the existing BDD frameworks. Particularly we point out: (i) code decoupling and code generation, (ii) scenario editing, and (iii) live object inspection.

4.1 Code decoupling and code generation

Potentially, the usage of a certain framework-defined writing style makes it easier to generate code with prepared templates or snippets. Some of the previously studied BDD frameworks rely on strict scenario structure and are able to generate a partial implementation by creating method placeholders. For example, the JVM version of Cucumber suggests snippets (see Figure 4.1) if they are missing in project files. This

```
You can implement missing steps with the snippets below:
@Given("the following empty book")
public void the_following_empty_book() {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}
@When("I add a contact")
public void i_add_a_contact() {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}
@Then("the book size should be {int}")
public void the_book_size_should_be(Integer int1) {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}
```

Figure 4.1: Cucumber feature example

can be extremely useful when the requirements only start to shape up and the implementation details are yet to be discussed. Apart from that, it guarantees loose coupling between feature-like files and implementation, and allows implementation replacement according to a context-specific environment.

Imagine a following scenario: ‘add contact details data to address book database on mobile smartphone and send the synchronization message to the back-end server’. But there are different smartphone models with different mobile operating systems. They all have a unique platform architecture, programming guidelines, and design patterns. Here, using the same BDD feature file for every mobile platform can be viewed as the most productive solution. The implementation will be system-dependent, but the behavior will not change.

When editing a behavior specification, we do not force the user to write it in a strict syntactical or grammatical style. Instead, Scenarios just provide descriptions as a plain editable text. We cannot generate placeholders as the other frameworks, because we do not introduce enforced lexical elements. In contrast to existing solutions, we succeed in generating new inputs, as illustrated both in the form of examples and domain model instances (see Figure 4.2). Generating new inputs should allow non-technical stakeholders to independently explore the system and create new examples for BDD tests without seeking help from a developer.

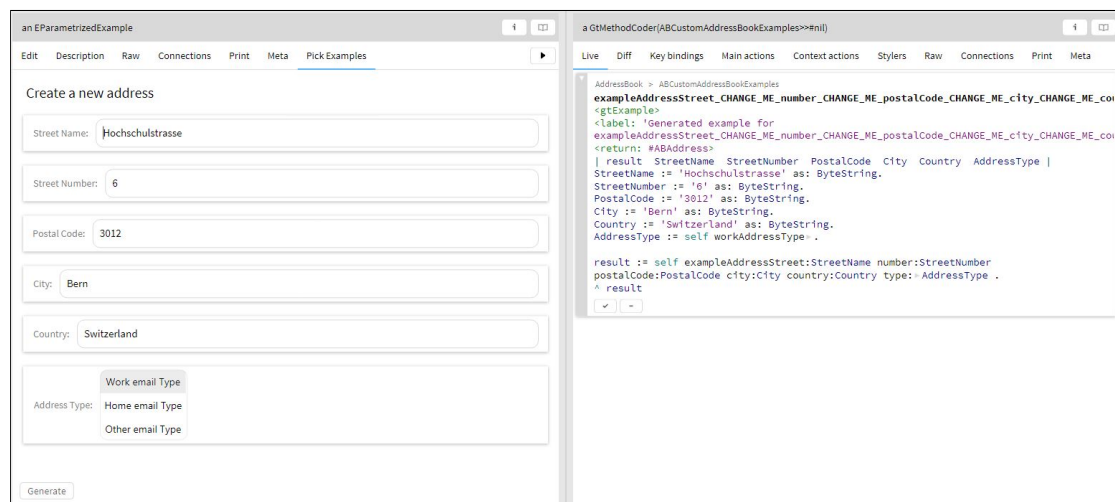


Figure 4.2: Generating new contact for address book

In the future, we plan to improve on generating executable specifications using terms from a ubiquitous language. Dictionaries and auxiliary parse methods can be used to achieve this. Programmers could, for instance, bind certain ubiquitous language terms with objects and methods by creating key-value entries. A key would be a ubiquitous language term, and a value its representation inside the software implementation. All together these entries form a dictionary that can later be used in parsing user-entered text. The parsing tool would either automatically or semi-automatically (with user confirmation) use the dictionary to construct an executable scenario from the specification with minimal programming effort.

4.2 Scenario editing

As the requirements change, so do the scenarios. Often they need to be edited not only in the plain-text annotated part but also in the implementation. Specifically, ‘Semi-structured text’ frameworks and frameworks with ‘Support for Behavior Driven syntax’ suffer here most: developers have to manually relate documentation text with code (IDE plugins partially solve such a problem) and vice versa.

The analyzed frameworks lack support for automatically binding parts of behavior specification such as connecting domain entities with concrete implementation stubs.

Our approach, in contrast, relies on merging scenario descriptions and examples in a single interface. This allows users to: (i) look at the problem within the same scope *i.e.*, interact with text and domain model at the same time (see Figure 3.11), and (ii) edit both in parallel to immediately observe the changes (see Figure 3.9). So far we only allow to configure example parameters and documentation as explained earlier. The current UI for editing the Parametrized Example needs improvements and is part of the future work.

Imagine that after requirements change the default method inside implementation needs to invoke another scenario(s). For example, functionality of adding a contact introduced in section 3.2 might be complemented with an additional argument validation or a third-party API system call. One possibility to expand our editor would be to allow editing of the internal structure of a parameterized example (*i.e.*, value assignment, method invocation) with minimal coding prerequisites: constitute a generic Coder-enhanced interface to work with entities that have `<label:>`, `<gtExample>` and `<gtParametrizedExample>` pragmas.

4.3 Live object inspection

BDD specifications act as a communication tool. As a supplement to functional testing, the BDD tool should indicate to the stakeholders that the system behaves as expected. We assume that just browsing test reports even with well-written documentation might be insufficient. At the same time, writing the code without mistakes is difficult. It is not uncommon that we can only discover mistakes either through logging/debugging, closer output observation, or by using various testing techniques. In relation to BDD, it could mean that the code behind written specifications may not completely implement the desired functionality.

Contrary to established practices of void test methods, we assume that returning domain objects from tests will give all stakeholders a better perception of the underlying domain model, reduce the number of misunderstandings, and accelerate the development. The same outcome can be achieved by specifically turning on debug mode inside an IDE for any other language and explicitly returning domain objects from tests.

The Inspector not only enables one to view objects as a set of fields and methods, but to traverse inside them, examine assigned values, and experiment on the fly (see Figure 3.6 and Figure 3.1). However, if we want returned values to be more readable by non-technical participants, a new, unified interaction paradigm

has to be introduced. A viable solution would be to let domain objects plug-in different views, renderable by the Inspector.

Chapter 5

Conclusion

The BDD community has been active in advertising the technology to the business community and developers, yet the adoption of the BDD practices in practice is far from ideal. We speculate that the reason lies in facilitating collaboration among diverse stakeholders. Particularly, popular frameworks have a great impact on the current perception of BDD.

Our analysis of existing BDD tools led us to the conclusion that most of them, although delivered as separate applications or libraries, are still not standalone products and are heavily dependent on test runners. Cucumber stands out among others, especially due to its simplicity, profound documentation, and extendability. Its *Given..when..then* syntax has strongly influenced how the ubiquitous language is used in several other instruments. However, the overall support for the ubiquitous language is still limited, few tools try to automatically generate tests using the plain text specifications, and many others ignore this aspect.

For the moment, we analyze the characteristics of the BDD tools without taking into account existing extensions. We are sure that the tools provide a comprehensive level of customization, from behavior description peculiarities to continuous delivery pipeline enhancements. Among prior BDD tool-related studies we did not discover material that examines BDD tools in combination with other community inventions. The current findings regarding tool limitations will certainly be different and are extendable with respect to existing community practices.

Our work expands the perception of how behavior specification can be written in different flavors that is practical for distinct project stakeholders. Through distinct object representations, one can create convenient domain-complementable and understandable interfaces that are interesting not only for the programmers but also for other non-technical stakeholders. By allowing users to graphically compose scenarios, the way of creating high-quality documentation will depend only on the writer's imagination. Together with various widgets arrangements, we highlight different ways to consume behavior specifications.

Although we demonstrate an example of modeling requirements workflow in an IDE with a custom scaffolding (*i.e.*, Epics...Use cases...User stories...Scenarios), it is still quite basic. A novel, standalone Pharo project for managing requirements should additionally help in reducing the gap between documentation and implementation. It can be either a tool that does not include any third-party-products or a proper and reusable integration model with several existing solutions such as Jira. We hope our experiment will show the different, previously unexplored ways of perceiving BDD specifications.

Appendices

Appendix A

Solution overview

We propose an experimental prototype that allows us to compose parameterized tests and directly see the execution result. Figure A.1 illustrates the basic internal structure. Below we provide more detailed information concerning classes and their relations with each other. Next, we describe the functionality of two main screens ‘Pick Examples’ and ‘Edit’ as well as the code generation process.

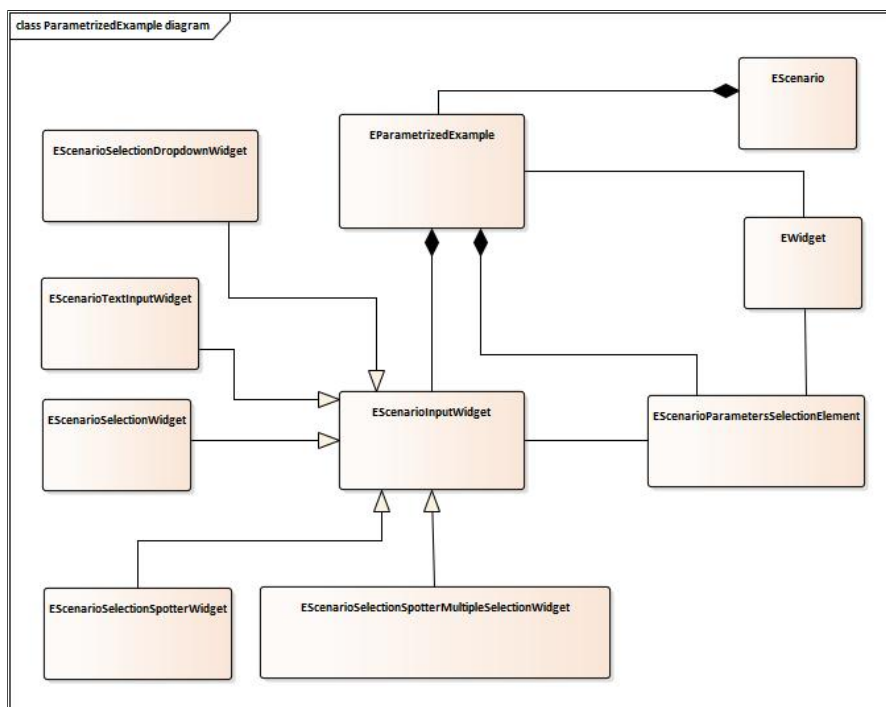


Figure A.1: Class Diagram

A.1 Configuring specification with pragmas

We want to avoid tight coupling to classes, and let additional attributes of an example to be processable by other tools. Therefore, we define them as pragmas inside the annotated method itself, and not within `EParametrizedExample`. For our implementation, we consider the method to have the following necessities (see Figure 3.4)

- it is annotated with pragmas `<gtExample>` and `<gtParametrizedExample>`,
- it has a label and description, also defined with pragmas `<label:>` and `<description:>`,
- it specifies information about input values: their name, class, scope and input type. We achieve this goal with a following pragma: `<given:ofType:in:elementType:>`,
- it describes the return class: `<return:>`

A.2 EParametrizedExample

The basic entity in our implementation is an `EParametrizedExample`. Every `EParametrizedExample` is a wrapper around a `<gtParametrizedExample>`, represented as a compiled method. The Compiled method is a static description of Pharo method, and in the scope of an `EParametrizedExample` saves the example in a single 'exampleMethod' instance variable [4]. `EParametrizedExample` knows how to work with the exampleMethod, recognize provided documentation, and is responsible for displaying the UI to user. As such, it creates `EWidget` instances based on provided pragmas, instantiates and configures the example selection and example edit views.

A.3 EScenarioInputWidget

This is an abstract class. The class responsibility only includes holding the information about edited value and scope. It is loosely coupled with an example so it can be instantiated independently. A widget knows how to initialize and show itself, act on user input, and return the selected value. So far we provide the following widgets:

- dropdown list,
- user text input,
- selection list,
- Spotter with single search,
- Spotter with multiple search,
- file choice

A widget's goal is not only to let users select a value from a subset of inputs but to provide several modes of interaction. For example, both 'Spotter with single search' and 'dropdown list' can use an equal

number of example sets. However, based on the different interaction patterns they require, one can already imagine how the future system will respond to such behavior. Another widget, ‘file choice’, suggests a combination of inputs: user can either manually provide a file location, or click on the chosen file element from the system file dialog window (see Figure A.2, Figure A.3).

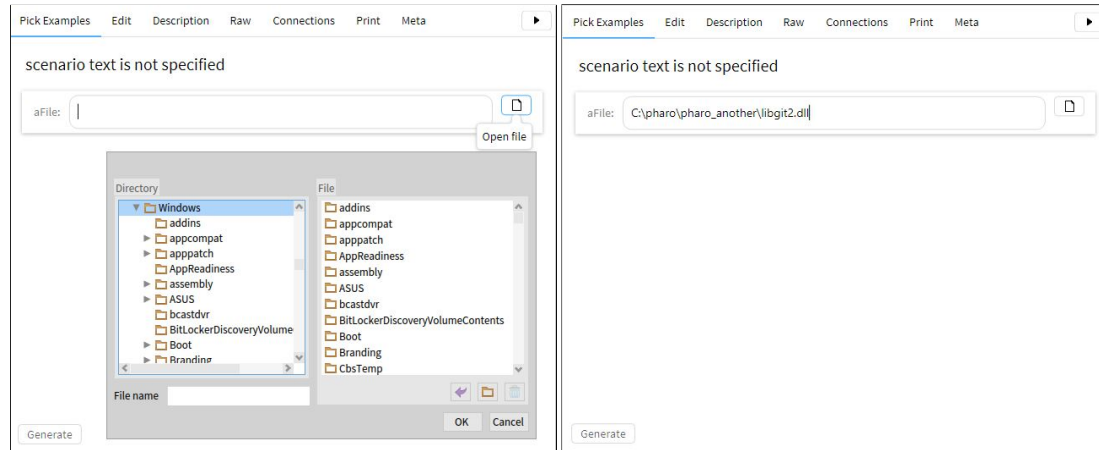


Figure A.2: Interaction: pressing a button opens a system dialog Figure A.3: Interaction: the same widget alternates user input with input field

A.4 EWidget

The EWidget class serves to propagate information parsed from the `<given:ofType:in:elementType:>` pragma in an EParametrizedExample. One can imagine it as a DTO.¹ The class knows only about the widget name, what type of object it can hold, and how the domain object, or parameter is called.

A.5 EScenarioParameterSelectionElement

EScenarioParameterSelectionElement represents the view element integrated inside of EScenarioInputWidget. At instantiation time it takes the EParametrizedExample, extracts a compiled method and according to its pragmas’ arguments, produces the UI for experimentation.

A.6 ‘Pick Examples’ graphical element

EParametrizedExample gives the user the possibility to experiment with examples. We will illustrate the process of view generation using the sequence diagram for Parametrized Example with the single input widget (see Figure A.5). The method prints the entered value in the Transcript(console) window (see Figure A.4).

¹<https://martinfowler.com/eaCatalog/dataTransferObject.html>

```

printString: str
  <gtExample>
  <gtParametrizedExample>
  <given: #aWord ofType: #String in: #' ' elementType:#input>
  <label: 'Print string'>
  <description: 'A simple parametrized examples that
  prints the string representation of an input parameter'>
  <return: #String>
  Transcript show: str asString.
  ^ str.

```

printing instance

Figure A.4: Parametrized example that prints the input

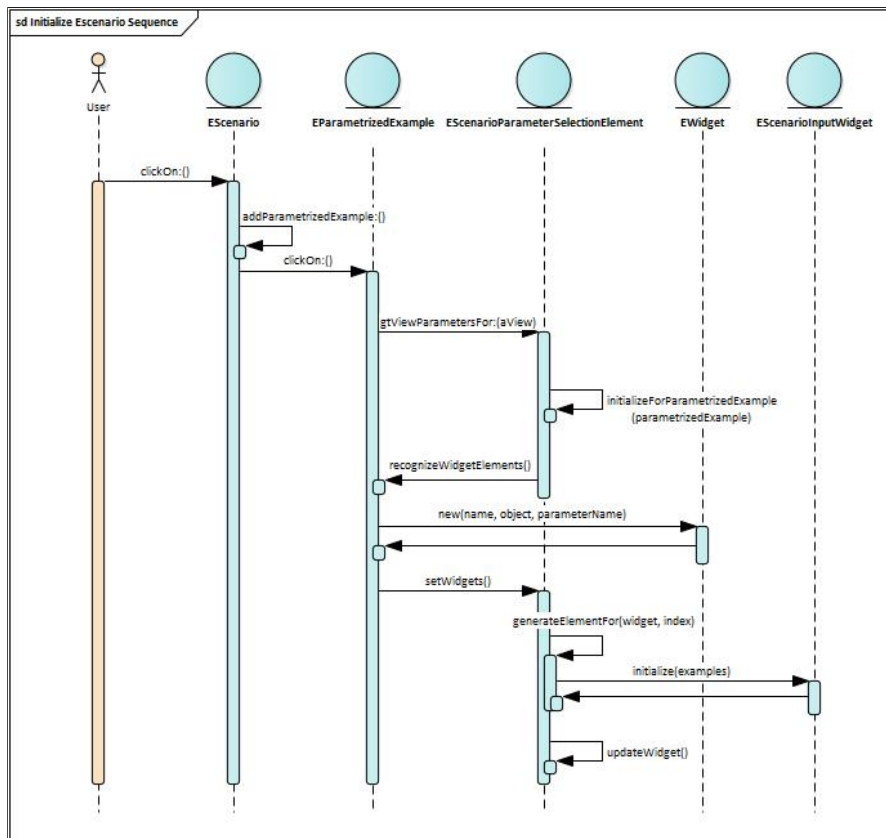


Figure A.5: Initialization of 'Pick Examples' view

Method `addParametrizedExample: anExample` is responsible for adding new examples inside the `EScenario`. A click on a tab 'Pick Examples' `EParametrizedExample` invokes the corresponding method for rendering `gtViewParametersFor:aView`. The idea behind this method is to return a UI ready for user interaction. This is done in several steps.

At first `EParametrizedExample` creates a new `EScenarioParameterSelectionElement` and tells it to instantiate a UI container filled with widgets (`initializeForParametrizedExample: EParametrizedExample`). Method

EScenarioParameterSelectionElement extracts pragma data from the compiledMethod assigned to example. By invoking EScenarioParameterSelectionElement>>recognizeWidgetElements a new EWidget for each <given:ofType:in:elementType:> pragma will be created.

Then, after the widget set is filled with EWidget instances, method generateElementFor:widget atIndex: index creates a new graphical representation for each of them. Knowing the widget type, it picks the right widget instance that inherits from the EScenarioInputWidget class (one instance of EScenarioTextInputWidget in this case). On the one hand, input widgets by design should not contain any predefined set of inputs. Other widgets, on the other hand, expect to give users a choice of input values. Input values are marked as <gtExample> and are collected inside a class marked with #in: argument of the aforementioned pragma. The input widget then makes a label from the #given: argument and attaches it to the container.

Finally, EParametrizedExample adds a generation button and a special ‘Phlow’ element *i.e.*, a scenario execution button (see Figure A.6).

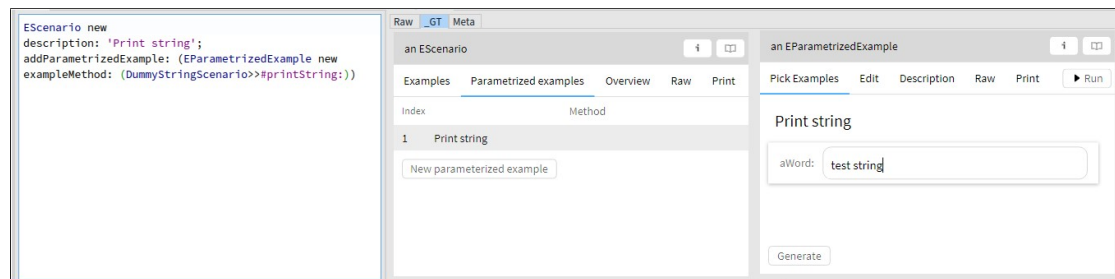


Figure A.6: Parametrized example that prints the input

Each action made by the user on ‘Pick examples’ view reflects on the underlying model. The scenario execution button listens to user clicks. Every time such event is fired, it will execute executeExample: ex withParams: prms – namely, gather the input values from the widgets as an array of arguments and invoke the compiled method with them. The user then inspects the execution result in Inspector window.

A.7 Generating new inputs

By letting users play with the examples, we do not force them to interact with code. In fact, the only technical part they need to know is related to widgets with manual input. For instance, the file input widget will show a Pharo error if the user enters the incorrect filepath. An environment, oriented to experimentation with entry values for specification examples (in our case they are parameterized examples) can potentially deliver a more comprehensive view of a problem, check for corner cases, or bring improvements to the domain model based on user impression.

The ‘generate’ button in ‘Pick Examples’ tab (see Figure A.6) serves exactly this purpose. Upon clicking, the EParametrizedExample>>generateMethodString will be invoked. It constitutes the string representation of the method, marks it as a <gtExample>, identifies the returned class, and provides the label with a description. Such a string representation will be instantiated by a Coder and shown in a separate

window. Inside the Coder window users can change the generated code as they like and keep the result. The method then will be checked by the compiler, saved and added inside a Pharo environment. Figure A.7 demonstrates what a possible generated method could look like.

```
AddressBook > DummyStringScenario
printString_CHANGE_ME_
<gtExample>
<label: 'Generated example for
printString_CHANGE_ME_'>
<return: #String>
| result aWord |
aWord := 'test string'.

result := self printString:>aWord .
^ result
```

Figure A.7: Generated example

A.8 ‘Edit’ graphical element

The scope of parameterized examples potential includes the example’s description, a number of parameters, their names and representations. The ‘Edit’ window tries to address changing these characteristics with relation to parametrized example (see Figure 3.9).

Similarly, as in the ‘Pick examples’ window, the ‘Edit’ element instantiates on a user click and uses the corresponding view class `EParametrizedExampleEditView`. The class then renders the UI with the following functionality:

- *Editing example label and description.* Editors for label and description fields are the instances of `Documenter`, and can potentially hold any textual or graphical information that `Documenter` supports.
- *Changing parameter representation.* We provide a fully customizable parameter editor, allowing to completely change parameter purpose when required. We permit changing the base parameter name, return class, example scope search and widget types.
- *Addition or removal of parameters.* Adding or removing a parameter correspondingly reflects changes in a `compiledMethod`.

Bibliography

- [1] V.r. Basili and R.w. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, SE-13(12):1278–1296, 1987. doi: 10.1109/tse.1987.232881.
- [2] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.
- [3] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Debugging distributed systems. *Queue*, Mar 2016. URL <https://dl.acm.org/doi/10.1145/2927299.2940294>.
- [4] Andrew P. Black, Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. *Pharo by Example*. Lulu.com, Raleigh, North Carolina, 2010. ISBN 978-3-952-33414-0.
- [5] Lisa Crispin and Janet Gregory. *Agile Testing - A Practical Guide for Testers and Agile Teams*. Pearson Education, Amsterdam, 2008. ISBN 978-0-321-61693-7.
- [6] Kim Dikert, Maria Paasivaara, and Casper Lassenius. Challenges and success factors for large-scale agile transformations: A systematic literature review. *Journal of Systems and Software*, 119:87–108, 2016.
- [7] O. Gotel, J. Cleland-Huang, J. Huffman Hayes, A. Zisman, A. Egyed, P. Grunbacher, and G. Antoniol. The quest for ubiquity: A roadmap for software and systems traceability research. *2012 20th IEEE International Requirements Engineering Conference (RE)*, 2012. doi: 10.1109/re.2012.6345841.
- [8] Ville Heikkilä, Kristian Rautiainen, and Slinger Jansen. A revelatory case study on scaling agile release planning. *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2010. doi: 10.1109/seaa.2010.37.
- [9] Irum Inayat, Siti Salwah Salim, Sabrina Marczak, Maya Daneva, and Shahaboddin Shamshirband. A systematic literature review on agile requirements engineering practices and challenges. *Computers in human behavior*, 51:915–929, 2015.
- [10] Samireh Jalali and Claes Wohlin. Agile practices in global software engineering—a systematic map. In *2010 5th IEEE International Conference on Global Software Engineering*, pages 45–54. IEEE, 2010.

- [11] Rakesh Kumar Lenka, Srikant Kumar, and Sunakshi Mamgain. Behavior driven development: Tools and challenges. In *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pages 1032–1037. IEEE, 2018.
- [12] Nikolaos Konstantinou Leonard Peter Binamungu, Suzanne Embury. Maintaining behaviour driven development specifications: Challenges and opportunities. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.
- [13] Anton Okolnychyi and Konrad Fögen. A study of tools for behavior-driven development. *Full-scale Software Engineering/Current Trends in Release Engineering*, page 7, 2016.
- [14] Rane and Prerana Pradeepkumar. Automatic generation of test cases for agile using natural language processing, Mar 2017. URL <https://vtechworks.lib.vt.edu/handle/10919/76680>.
- [15] Pilar Rodríguez, Alireza Haghhighatkah, Lucy Ellen Lwakatara, Susanna Teppola, Tanja Suomalainen, Juho Eskeli, Teemu Karvonen, Pasi Kuvaja, June M. Verner, Markku Oivo, and et al. Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, 123:263–291, 2017. doi: 10.1016/j.jss.2015.12.015.
- [16] Eva-Maria Schön, Dominique Winter, María José Escalona, and Jörg Thomaschewski. Key challenges in agile requirements engineering. In *International Conference on Agile Software Development*, pages 37–51. Springer, Cham, 2017.
- [17] Mathias Soeken, Robert Wille, Rolf Drechsler, and Robert Wille Rolf Drechsler Mathias Soeken. Assisted behavior driven development using natural language processing, May 2012. URL https://dl.acm.org/doi/10.1007/978-3-642-30561-0_19.
- [18] Carlos Solis and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387. IEEE, 2011.
- [19] Ruxandra Bob Tim Storer. Behave nicely! automatic generation of code for behaviour driven development test suites. *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019.