Interactive Behavior-driven Development: a Low-code Perspective

Nitish Patkar* Andrei Chiş† Nataliia Stulova* Oscar Nierstrasz*
Software Composition Grroup
University of Bern
Switzerland
Email: * https://scg.unibe.ch/staff, † https://feenk.com

***Abstract*—Within behavior-driven development (BDD), different types of stakeholders collaborate in creating scenarios that specify application behavior. The current workflow for BDD expects non-technical stakeholders to use an integrated development environment (IDE) to write textual scenarios in the Gherkin language and verify application behavior using test passed/failed reports. Research to date shows that this approach leads non-technical stakeholders to perceive BDD as an overhead in addition to the testing.**

In this vision paper, we propose an alternative approach to specify and verify application behavior visually, interactively, and collaboratively within an IDE. Instead of writing textual scenarios, non-technical stakeholders compose, edit, and save scenarios by using tailored graphical interfaces that allow them to manipulate involved domain objects. Upon executing such interactively composed scenarios, all stakeholders verify the application behavior by inspecting domain-specific representations of run-time domain objects instead of a test run report. Such a low code approach to BDD has the potential to enable non-technical stakeholders to engage more harmoniously in behavior specification and validation together with technical stakeholders within an IDE. There are two main contributions of this work: (i) we present an analysis of the features of 13 BDD tools, (ii) we describe a prototype implementation of our approach, and (iii) we outline our plan to conduct a large-scale developer survey to evaluate our approach to highlight the perceived benefits over the existing approach.

***Index Terms*—bdd, behavior-driven development, collaborative development, acceptance testing, visual programming, end-user programming**

I. Introduction

Behavior-driven development (BDD) is an approach that drives development teams to specify "live", executable, and testable requirements. Within BDD, non-technical stakeholders specify application behavior through scenarios that everybody in a team can understand [1]. Non-technical stakeholders often leverage a constrained natural language, *i.e.,* Gherkin, to write scenarios. For example, the scenario in lines 5-8 of Listing 1 asserts the sum of two numbers for an arithmetic calculator application to have a particular value.

```
1 Feature: Basic arithmetic operations
2 As a user
3 I want to use a calculator to add numbers
4 So that I don't need to add them myself
5 Scenario: Add two numbers -2 and 3
6 Given I have a Calculator
7 When I add -2 and 3
8 Then the result should be 1
```

Listing 1: A sample feature description with a scenario

The BDD frameworks then tie the scenarios to acceptance test cases (also called step definitions, glue code, or fixtures) to verify the specified functionality. Listing 2 shows an example expansion. The developers need to fill in the body of glue code methods (lines 10,14, and 18).

```
1 public class CalculatorRunSteps {
2 private int total;
3 private Calculator calculator;
4 @Before
5 private void init() {
6 total = -999;
7 }
8 @Given("I have a calculator")
9 public void initializeCalculator() throws Throwable {
10 calculator = new Calculator();
11 }
12 @When("I add {int} and {int}")
13 public void testAdd(int num1, int num2) throws Throwable {
14 total = calculator.add(num1, num2);
15 }
16 @Then("the result should be {int}")
17 public void validateResult(int result) throws Throwable {
18 Assert.assertThat(total, Matchers.equalTo(result));
19 }
20 }
```

Listing 2: Glue code for the scenario from Listing 1

Finally, developers implement the logic for the calculator application:

```
1 public class Calculator {
2 public int add(int a, int b) {
3   return a + b;
4 }
5 }
```

Listing 3: Implementation for the functionality from Listing 1

Without any exception, the existing BDD frameworks expect non-technical stakeholders to use an IDE to write scenarios and verify behavior through test run reports. An analysis of 20 open-source Ruby projects revealed that the majority of the scenario specifications were added with the source code, or long after it had been written [2]. In another recent survey, the respondents with software engineer and business analyst roles highlighted the shortcomings of the current BDD practices [3]. From both these empirical studies, we learn that the practitioners do not strictly perform BDD, probably due

to the workflow supported by the existing BDD frameworks. Additionally, when the requirements change, a lot of manual effort is needed to maintain the textual scenarios, to manually propagate the changes to acceptance tests, leading the practitioners to perceive BDD as only an additional task to writing unit tests [3], [2]. Furthermore, in our manual inspection of scenario specification files from 23 open-source GitHub projects, and after contacting the contributors to these files, we learned that it was developers who specified the behavior. We speculate that the current support for behavior specification and verification in IDEs is the limiting factor for poor acceptance of BDD in practice, *i.e.,* pushing developers to specify, implement, and verify the behavior. Our goal is to propose an alternative approach to BDD that engages non-technical stakeholders equally in the BDD process and addresses the following limitations of the current BDD process:

- *behavior specification*: the textual format for scenario specification is poor in conveying information about the domain, and
- *behavior verification*: test run reports are provided at a wrong level of abstraction for non-technical stakeholders, compared to specifications.

We discuss a LowCode solution for business and technical stakeholders to specify and verify application behavior. The main contributions of this work are as follows:

- We survey the documentation of 13 BDD frameworks to assess their characteristics in support of behavior specification and verification. We found that existing BDD frameworks only provide text editors to specify scenarios and test cases. Likewise, all the frameworks only output test run reports to verify the specified behavior.
- We propose an approach that, through tailored graphical interfaces, enables non-technical stakeholders to create scenarios, and through custom visual representation of the domain objects, enables non-technical stakeholders to verify behavior collaboratively with developers.
- We present an advanced prototype implementation of our approach in the Glamorous Toolkit IDE,[1] to demonstrate the feasibility of our approach. We are currently conducting a comprehensive usability evaluation of our approach with a large-scale developer survey.

The remainder of the paper is structured as follows: section II describes a typical BDD process using a running example, and section III presents a discussion concerning the characteristics of existing BDD frameworks to engage multiple stakeholders in the BDD process. Next, in section IV, we present the underlying idea and main building blocks of our approach, and finally, in section VII we summarize the main contributions and planned future work.

II. Background and Motivation

BDD builds on a combination of multiple existing methodologies and concepts, such as test-driven development (TDD) and ubiquitous languages from domain-driven design (DDD), which enable teams to develop software systems collaboratively [4], [5], [6], [7]. The term ubiquitous language refers to the practice of building and using a common language between business and technical stakeholders [7]. This language consists of terms from the application domain that are used consistently in all aspects of a project, *e.g.,* in conversations, in requirements, and in the source code. Within the classical BDD workflow, non-technical stakeholders specify the desired application behavior through textual scenarios that make use of such a ubiquitous language [1].

A. BDD workflow

BDD supports an agile software development process in which requirements are commonly structured into *epics* (a set of requirements that together deliver greater business value and touch the same portion of the product, whether functional or logical, typically split into several user stories), *user stories* (informal, natural language descriptions of one or more functionalities of a software system), and corresponding *scenarios* (instances of stories with concrete values) [8]. Figure 1 summarizes the approach proposed by most available BDD frameworks that automate the BDD process. The boxes represent the main steps, while the arrows indicate the direction of the workflow.

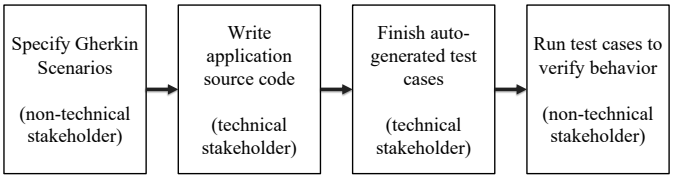

Fig. 1: Typical BDD process

First, non-technical stakeholders specify requirements as a number of scenarios in Gherkin, a domain-specific language [1]. They describe application behavior in the terms agreed upon by the project team, *i.e.,* using the ubiquitous language [7]. A typical Gherkin template splits a scenario into three core elements: *Given* (*i.e.,* a context assumed for this scenario execution), *When* (*i.e.,* an action or event that occurs in the given context), and *Then* (the expected outcome of the system for the provided action and context). An element can have additional context, expressed in the template by the word *And*.

Next, developers implement the desired functionality as the application source code. The BDD frameworks then generate *step definitions*, *i.e.,* code snippets that connect each scenario step to the corresponding executable test code. Developers map appropriate input parameters from the *Given...When...Then* statements to the step definitions, and also implement the requirements specified in scenarios. The names of classes and methods conform to the vocabulary (nouns and verbs) of the ubiquitous language defined for the project.

Finally, when non-technical stakeholders execute the acceptance tests, the BDD frameworks present them with the test run status, *i.e.,* success or failure. Non-technical stakeholders

[1]https://gtoolkit.com/

are required to specify the input parameters for tests and also the expected output values, see Listing 1.

B. Motivating example and issues with the workflow

We introduce a running example to illustrate a few critical issues with the commonly-followed BDD workflow. Let us consider that we need to update an existing invoicing system for a restaurant and consequently verify if the new invoices are calculated correctly. The invoicing system allows its users to add menu items to an order, and indicate if those items were consumed inside the restaurant or were ordered for take away. The new invoices should correctly reflect a change in VAT calculation. A different value added tax (VAT) should apply for the same menu item depending on whether it is for take away or on-site consumption.[2]

Menu item	On site VAT	Takeaway VAT %
Black coffee	19	19
Cappuccino	19	7
Pizza Margherita	19	7

The invoice contains the total cost of ordered menu items and a VAT. The invoicing example is a little more complicated than a calculator application and involves complex domain objects, such as of type `Invoice` and `Order`. To verify the user story "As a waiter, I want to be able to view the total price before printing the invoice," one can write scenarios[3] such as:

```
Scenario 1: Customers place an order to take away
(only milk products)
Given an empty order
When the waiter adds Cappuccino to the empty order
And a cup of Cappuccino costs 4 EUR
And a cup of Cappuccino is taxed at 7%
And the waiter generates the Invoice for the order
Then the total invoice price is 7.28 EUR

Scenario 2: Customers place an order to take away
(combination of non-milk and milk products)
Given an empty order
When the waiter adds a Cappuccino and a black coffee
to the empty order
And a cup of Cappuccino costs 4 EUR
And a cup of black coffee costs 3 EUR
And a cup of Cappuccino is taxed at 7%
And a cup of black coffee is taxed at 19%
And the waiter generates the Invoice for the order
Then the total invoice price is 7.85 EUR
```

Listing 4: Sample scenarios for invoicing application

The final price in the *Then* statement in the first scenario is voluntarily incorrect.

The typical current BDD workflow faces two issues here. First, this workflow leads non-technical stakeholders to write numerous scenarios with minor variations, such as in input parameter values, and requires them to specify the test assertions. The latest version of Gherkin supports background and data tables, which allow non-technical stakeholders to specify various combinations of input and corresponding expected outputs that can be provided to a single scenario.[4] Background and data tables help to reduce redundancy in textual scenarios. However, our manual inspection of 23 open-source projects on GitHub (with more than 500 stars and primary language Java) that use Gherkin specifications lead us to conclude that data tables are rather moderately used in scenario specifications. For instance, from 23 repositories, we analyzed around 1509 feature files of which only 568 used tables. There were on average about 1.7 columns per table, whereas there were on average about 2.6 rows per table, *i.e.,* quite tiny tables that may not be very helpful in reducing the redundancy in scenarios.

Second, using the test run status as a means to verify behavior obscures details of logical mistakes made in the scenario specification. A non-technical stakeholder manually had to calculate the expected results during specification. Although it is a common practice in testing in general, it can lead to software run-time errors that are difficult to locate in the textual specifications.

We analyzed 13 BDD frameworks to provide documented proof of our claims of the currently adopted BDD workflow shortcomings in section III. Our analysis is solely based on the features of the tools and not on their actual use in practice — we are currently conducting a large-scale developer survey to obtain the data on it.

III. BDD TOOL ANALYSIS

Only a few prior studies have evaluated BDD tools. Lenka *et al.* analyzed five BDD tools and classified them either as testing tools or test automation frameworks, essentially supporting the view of BDD tools as being testing tools [9]. Solis *et al.* analysed seven BDD tools according to six parameters, such as the supported programming languages and supported software development phases [10]. They observed poor support for BDD in the planning phase, *i.e.,* the analysed tools did not support the creation of features or user stories. Finally, Okolnychyi *et al.* analyzed five BDD tools to characterize their support for BDD in terms of ubiquitous language creation and automated scenario execution [11]. They compared the tools based on their primary target users and specific tool features, such as support for mocking third-party libraries. Like Solis *et al.*, they observed that the support for ubiquitous language definition is limited.

These studies do not establish criteria to measure to what degree BDD tools enable collaboration among both technical and non-technical stakeholders.

A. Tool comparison

Previously analyzed tools such as StoryQ [12], JDave [13], NBehave [14], Easyb [15], and BDDfy [16] are either obsolete or no longer maintained [11], [9]. We analyzed 13 BDD tools that are currently actively maintained by their creators to observe to what degree they enable collaboration among stakeholders. In particular, we studied how the IDE integration enables behavior specification and verification for non-technical

[2]The German federal government recently proposed changes to the value-added tax (VAT) system: accessed November 12, 2020, https://www.hellotax.com/blog/new-vat-rates-germany/

[3]In subsection A and subsection B, we list the complete requirements for such an application decomposed into epics, user stories, and corresponding scenarios.

[4]"Gherkin reference," https://github.com/cucumber/common/blob/main/gherkin/CHANGELOG.md

TABLE I: BDD tool comparison

Tool	Input type				Parameter type		Specification interface		Output type	
	Plain text	Markdown	Table	Code	Primitive	Object	Textual	Graphical	Run Status	Report
Cucumber	✓	-	✓	-	✓	✓	✓	-	✓	✓
JBehave	✓	-	-	-	nc	nc	✓	-	✓	-
Concordion	✓	✓	✓	-	✓	✓	✓	-	✓	-
SpecFlow	✓	-	✓	-	✓	✓	✓	-	✓	✓
Spock	-	-	-	✓	✓	✓	✓	-	✓	-
RSpeck	-	-	-	✓	✓	nc	✓	-	✓	nc
MSpec	-	-	-	✓	✓	nc	✓	-	✓	-
LightBDD	-	-	-	✓	✓	nc	✓	-	✓	✓
ScalaTest	-	-	-	✓	✓	✓	✓	-	✓	-
Specs2	-	-	-	✓	✓	nc	✓	-	✓	✓
JGiven	-	-	-	✓	✓	nc	✓	-	✓	✓
phpspec	-	-	-	✓	nc	✓	✓	-	✓	✓
Gauge	-	✓	✓	-	✓	nc	✓	-	✓	✓

stakeholders. We analysed Cucumber [17], JBehave [18], Concordion [19], SpecFlow [20], Spock [21], RSpec [22], MSpec [23], LightBDD [24], ScalaTest [25], Specs2 [26], JGiven [27], phpspec [28], and Gauge [29]. All analyzed tools are open-source and are actively maintained on GitHub. We define our assessment parameters in subsection III-B. The results of our tool comparison are summarized in Table I. The symbol "✓" denotes that the value is "true," whereas "nc" means "not clear from the documentation."

B. Comparison parameters

The aforementioned 13 BDD tools are available as IDE plugins. We evaluated the IDE plugins according to six parameters to understand how these plugins enable specification and verification of the behavior, in other words, what opportunities non-technical stakeholders have in IDE plugins to specify and verify the application behavior.

- *Input type for specifying the scenario.* (i) "Plain text," which means a specification is written as a natural language text, (ii) "Markdown text," which means a specification is written in a Markdown format, (iii) "Table," which means a specification accepts input values in a tabular format, or (iv) "Code," meaning a specification is written in some programming language but is enhanced with annotations.
- *Type of parameters in the glue code.* (i) "Primitive," such as strings, numbers, or boolean values as input parameters, or (ii) "Object," which means a scenario can take domain objects as inputs.
- *Specification interface.* (i) "Textual," which means specifications can be written only as text, or (ii) "Graphical," which means specifications can be composed by using graphical elements.
- *Output type.* (i) "Test run status," which means the tool only indicates a pass or fail status for tests, or (ii) "Report," which means the tool provides alternatives to customize test run reports so that the output is readable by non-technical stakeholders.

C. Results

Type of input. Specifications are written either textually (5 tools) or as test cases enhanced with annotations, such as `[Given]` (8 tools). The textual specifications are written either with Gherkin syntax, in a Markdown format, or a combination of both. Data tables with input and expected output values for behavior tests are supported in a total of 4 tools, which means data tables are not supported universally across all tools.

Support for parameterized scenarios. We observed that only primitive types, such as strings or numbers, are allowed as inputs to the scenarios. Data tables help to specify input parameters concisely. However, how much helpful are the data tables to specify complex domain objects with numerous attributes has not yet been studied.

Specification interface. All the analyzed tools support textual specification only; no tool allows specifications to be composed in any other way, *e.g.,* graphically.

Type of output. All analyzed tools provide two ways to output their results: (i) test results that indicate how many tests are passing and failing, and (ii) test reports that can be customized with formats (*i.e.,* charts and graphs) and color schemes (*i.e.,* indicating passed and failed tests in different colors).

IV. Our approach in a nutshell

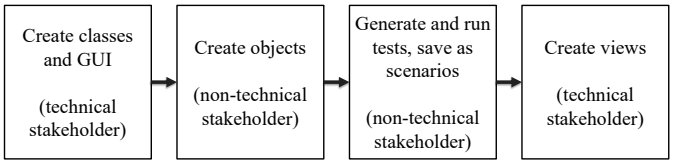

Fig. 2: Proposed BDD process

Different stakeholders have different goals, therefore an IDE should be adapted to support distinct stakeholders, *i.e.,* depending on the context of the task within a BDD process. Developers need appropriate support for creating and editing

classes and methods. Non-technical stakeholders need custom interfaces (*e.g.,* graphical) so that they can create and use run-time objects to invoke operations on them, *i.e.,* to compose, run, and save scenarios.

Glamorous Toolkit is an advanced IDE that pioneered the idea of adapting tools, such as object inspector or debugger, to a specific development context. For example, an object inspector in Glamorous Toolkit allows an object to define a set of multiple interchangeable presentations (*i.e., views*) capturing interesting aspects of that object in various development contexts [30]. Likewise, Glamorous Toolkit extends the concept of test cases to *examples* that return a domain object instead of the test assertion status. For the scenario in Listing 4, in addition to the test assertions, with *examples*, one could return the resulting object of type `Invoice`, and subsequently explore it using a dedicated graphical representation (details in section V).

To improve the BDD workflow for both technical and non-technical stakeholders, we reuse the functionalities of Glamorous Toolkit IDE and adapt those as specific building blocks of the IDE-based BDD process. In Figure 2, we outline our proposed BDD workflow. Developers will create classes and implement the behavior of those classes in the methods as before. However, instead of specifying behavior and updating test cases, they only need to insert the assertions in the fully-generated test cases. Developers will also create graphical interfaces for object creation and views to explore details of run-time objects visually. Non-technical stakeholders, on the other hand, will use graphical interfaces to compose and save scenarios. If they wish, they can also insert the assertions in the fully-generated test cases. In other words, non-technical stakeholders do not need to write textual scenarios. Instead of test run status, they will use a domain-specific representation of the involved objects to verify the implemented behavior. Technical stakeholders, on the other hand, need to implement a GUI for object creation and object representation. Our suggested approach, in principle, can be adapted to any IDE with a rich-enough object inspector and the possibility to have Glamorous Toolkit IDE-like examples or in any testing framework that would allow tests to return objects, and not just the success or failure status.

A. Building blocks

Below we explain the main building blocks of our approach: views, graphical interfaces, and examples.

Views. A view is a domain-specific representation of an object. Any number of views can be attached to an object. The right hand windows in Figure 6 and Figure 7 show different views of an `Invoice` object. A printable view (see, Figure 6) shows the `Invoice` object in its final printable version, whereas a composition view (see, Figure 7) shows the composition of several domain objects that compose an `Invoice`. It is primarily the developer's responsibility to create views that are useful for other stakeholders. Creating views does not require much effort, *e.g.,* on an average 12 lines of code for a view in Glamorous Toolkit IDE. To verify this claim, kindly refer to the replication package.[5] The readme file in the replication package provides instructions to verify this number. Such domain-specific representations of the resulting domain objects offer a different way of inspecting the results of running examples compared to simply analyzing the test run reports.

Graphical interfaces. Most IDEs support creation and manipulation of objects programmatically. Very few IDEs, such as BluJ,[6] enable non-technical stakeholders to create and manipulate objects interactively, *i.e.,* objects can be dynamically created, the contents of fields are displayed and their methods can be invoked through provided graphical interfaces. However, these graphical interfaces are generic and cannot be customized to adapt to a particular application domain. Glamorous Toolkit IDE is built using a graphical framework (*i.e.,* Bloc) that enables the creation of customizable graphical interfaces for various types of objects. To support BDD, developers can build tailored graphical interfaces to enable object creation and manipulation for non-technical stakeholders. Non-technical stakeholders can also use these interfaces to: (1) provide run-time objects to test cases and execute those test cases, and (2) save the current selection of input domain objects and results of example execution as individual scenarios.

Examples. Examples are individually executable pieces of source code just like test cases except that instead of simply reporting success or failure, they return a domain object as a result. For instance, an example method may create and return an object of type `Cappuccino` or create an `Order` with two `Cappuccino` instances, check whether the `Order` object contains two `Cappuccino` instances, and return the resulting `Order` object for further inspection.

```
orderWithCoffee
<gtExample>
<label: 'create order with coffee'>
<description: 'Create an order'>
|order coffee|
coffee := Coffee new.
order := self emptyOrder add: coffee.
self assert: order size equals: 1.
^ order
```

Listing 5: An example that creates an order with coffee

Examples can be chained together so that domain objects flow through a series of examples emulating complex scenarios. For instance, Listing 5 creates an `Order` with a `Coffee` by first calling Listing 6 that creates an empty `Order` object. This makes examples reusable. In other words, such chained examples represent a concrete scenario. By using graphical interfaces, non-technical stakeholders can create and save several examples by providing appropriate domain objects, *i.e.,*`Order` and `Coffee`. Examples are useful to create both simple domain objects, *i.e.,* domain objects that do not contain

[5]https://figshare.com/s/1ab8123e66845f50ffb2

[6]https://www.bluej.org/

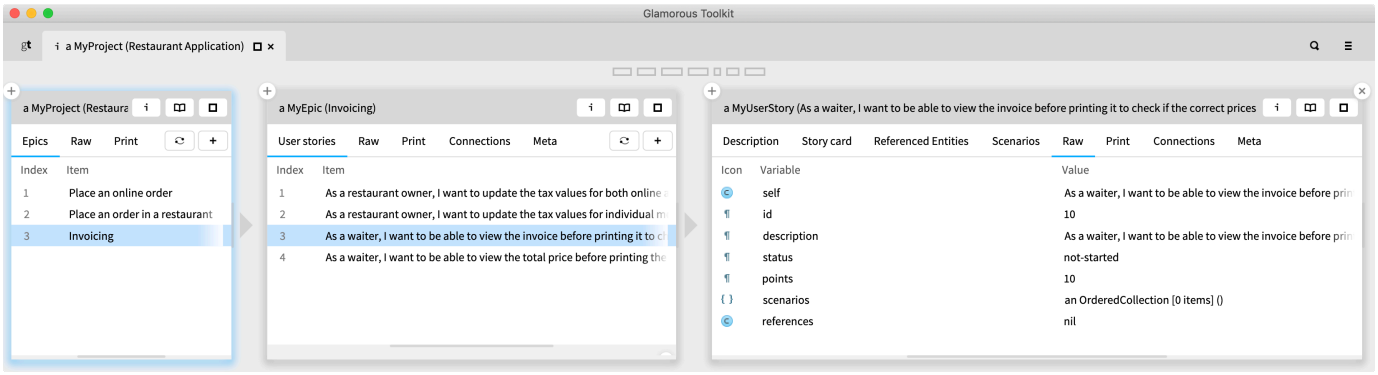

Fig. 3: Custom requirements hierarchy and a user story view

or require other domain objects (*e.g.,* an empty `Order` object), and complex domain objects, *i.e.,* domain objects that contain other domain objects (*e.g.,* an `Order` with a `Coffee` object).

```
1 emptyOrder
2 <gtExample>
3 <label: 'create an empty order'>
4 <description: 'Create an empty order'>
5 | order |
6 order := Order new.
7 self assert: order size equals: 0.
8 ^ order
```

Listing 6: An example that creates an empty order

Additionally, examples can be given a description that is understandable by all the stakeholders, see line 4 of Listing 6.

V. Our proposed BDD workflow

Existing literature discussed the benefits of creating requirements as first-class citizens in an IDE [31]. The authors demonstrated how one should create several requirements formats (*e.g.,* epics and user stories) as classes and concrete requirements as objects of those classes. In this case, they leveraged the has-relationship to model epics to have several user stories.

We follow a similar approach and we adapt here in Figure 3 a list view (*i.e.,* a view for an object of type `MyProject` built by developers), which displays the titles of epics for our invoicing example. When users click on a particular `MyEpic` object, they can see corresponding `MyUserStory` objects in the second window. When they click on a specific `MyUserStory` object in the second window, among other things, they can see in the third window a "raw" view that displays internal representation details, such as status and story points, about that user story. All three windows in Figure 3 are object inspectors with several dedicated views built by developers. In other words, we have reused an object inspector to visually inspect requirements, while at the same time the requirements are directly tied to code. To verify the implementation for the included screenshots, kindly refer to the replication package. In this vision paper, we extend the same idea for user stories to have several scenarios.

Let us consider that we have two stakeholders, Bob, who is a non-technical domain expert, and Melinda, the developer. As scenarios elaborate a specific user story, by using the discussed building blocks, *i.e.,* graphical interfaces, examples, and views, Bob can interactively create scenarios as first-class citizens in an IDE as described next. Note that except for the test case generation, no other step in the workflow is automated — each step still requires manual effort from the concerned stakeholders, however the type of interaction with the system is different compared to the existing BDD workflow. Our proposed workflow divides the process of scenario creation and verification into the following four steps: (1) create classes and graphical user interface (GUI), (2) create domain objects, (3) generate test cases and save as scenarios, and (4) create views. We detail below each step.

A. Create classes and GUI

The user story "As a waiter, I want to add menu items to prepare an order" contains domain concepts, such as `Waiter`, `MenuItem`, and `Order`. This user story also specifies expected behavior, *i.e.,* an order is created by adding menu items to it. Melinda creates classes for the domain concepts involved in a particular user story and builds graphical interfaces that would enable Bob to create objects for these classes. For instance, the graphical interface in Figure 4 enables creating objects of type `MenuItem`. This interface enables Bob to create a `Cappuccino` object, and provide details, such as the price and VAT.[7] Melinda also implements the methods that define behavior for each class.

B. Create domain objects

Now, Bob can create several instances of concerned classes by using the provided graphical interfaces. When he clicks on the "Generate" button in Figure 4, the corresponding example method is created, see the right hand window. When anyone executes this newly created example method, it always returns the same `Cappuccino` object with selected price and tax. Once the basic domain objects are created, they can be used to create more complex domain objects. For example, an `Order`

[7] Pawson explored a similar idea in the Naked Object approach, in which complete core business objects are exposed behaviorally to a user. The Naked object framework automatically creates a user interface that supports CRUD (create, read, update, delete) operations for domain objects to bootstrap a business application [32].

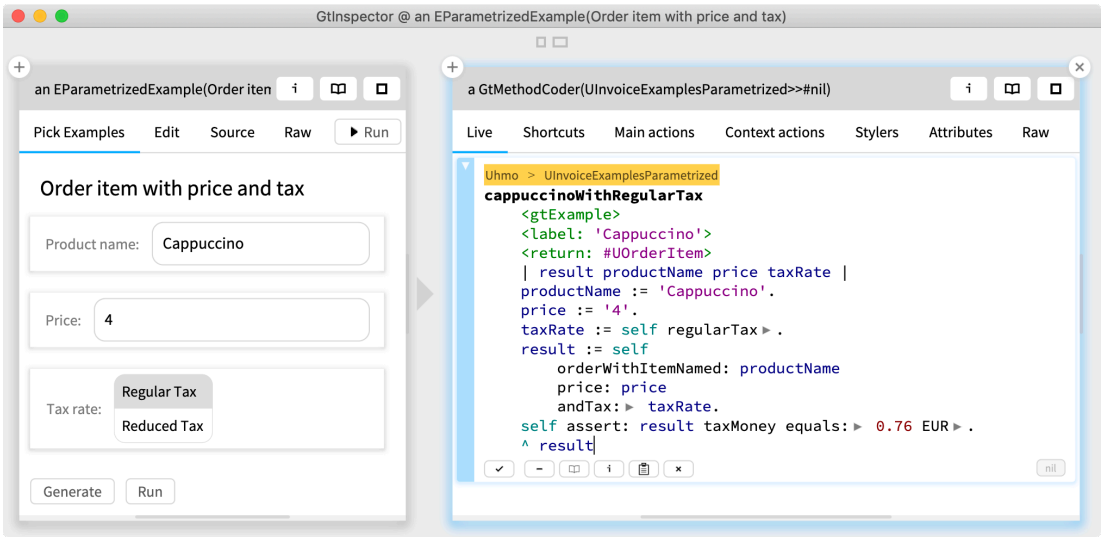

Fig. 4: A GUI to create a simple domain object and save the example as an operation

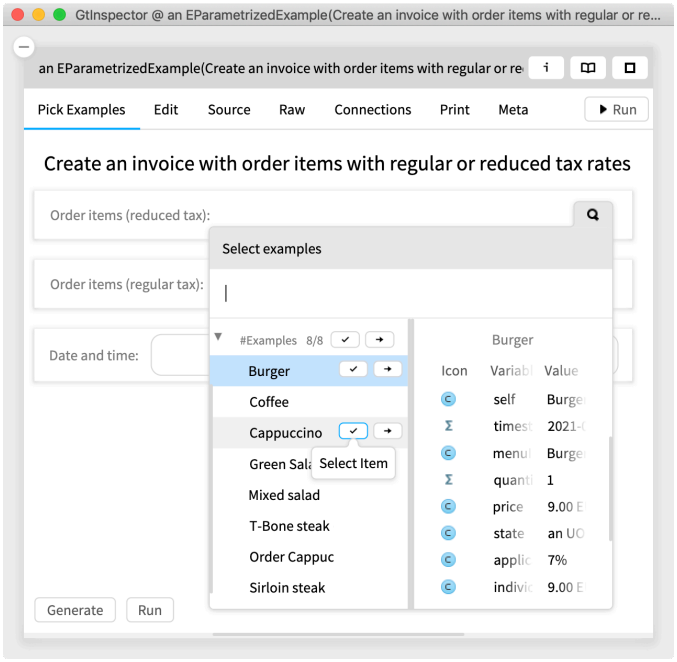

Fig. 5: A GUI to select simple domain objects

could be composed from various `MenuItems`. To enable Bob to create such complex domain objects, Melinda creates a tailored graphical interface. For instance, the graphical interface in Figure 5 is populated with various already created simple objects, *i.e.,*`Cappuccino` and `Coffee`, that appear as a list in a drop-down menu. Bob uses this interface to create complex objects and save the selection as another example method (see Figure 6). Here, Bob creates an `Invoice` object for an `Order` with two menu items (*i.e.,*`Cappuccino` and `Coffee`). However, instead of "Generate," now he clicks on the "Run" button to explore the resulting `Invoice` object visually.

C. Create test cases and save as a scenario

The tailored graphical interfaces, shown in Figure 4 and Figure 6, essentially enable Bob to create both simple and complex domain objects and also generate an example method that when executed returns a specific domain object. Examples represent a concrete scenario. To save an example method as a scenario, Bob clicks on the "✓" button in the right hand side window of Figure 4. This saves the newly created scenario for a particular user story, and Bob can always access it from one of the views for a `MyUserStory` (see Figure 8). Melinda or Bob add assertions to this newly generated example method to test the specified behavior in the respective methods of the domain classes.

With this approach, instead of writing scenarios textually, Bob could interactively create simple domain objects (*e.g.,*`Cappuccino` and `Coffee`) and use those to create complex domain objects (*e.g.,*`Order`). He could save the selection of `Cappuccino` and `Coffee` to an `Order` as an example method, which will return the same `Order` instance with `Cappuccino` and `Coffee` when executed. This example method is attached to `MyScenario` object.

D. Create views

Both Bob and Melinda need different representations of domain objects to accomplish distinct tasks. For instance, Bob needs to determine whether the correct number of menu items are added to an `Order` object, whether correct prices and tax rates are applied to each `MenuItem` object, and whether the final price is accurately calculated in `Invoice` object. He uses the printable representation of the `Invoice` object in Figure 6 that fulfills his needs. Likewise, Melinda needs to understand how an `Invoice` object is constructed. She uses the composition presentation of the `Invoice` object in Figure 7 to explore how it is made up of other objects, such as of type `Cappuccino`, with their corresponding properties, such as applied tax rates. Note that the process of creating

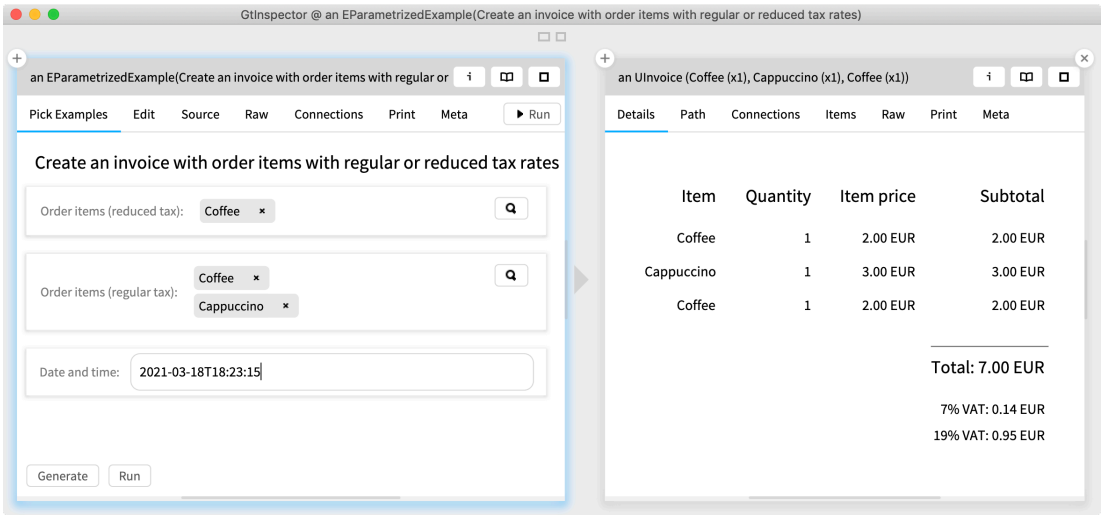

Fig. 6: A GUI to create complex domain object and explore the resulting object with printable view

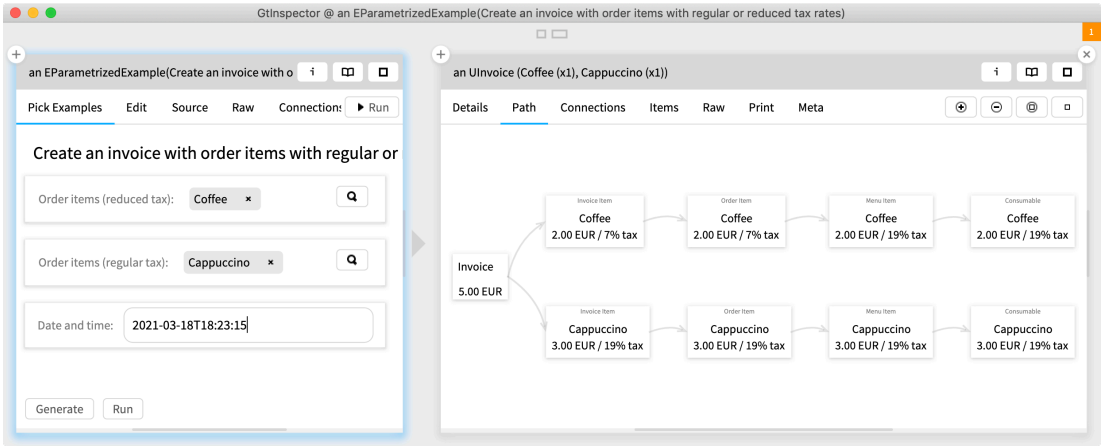

Fig. 7: A GUI to create complex domain object and explore the resulting object with composition view

objects and views is iterative and incremental— views can be designed as the necessity arises to explore some specific details of a specific domain object. Theoretically, Melinda could create the printable invoice view when she first created the class `Invoice`.

With this approach the application behavior becomes verifiable by stakeholders by inspecting domain objects instead of reading a test report. Notably, this approach does not eliminate the need for test cases. The example methods serve as test cases, but augments them with domain-specific representations of the involved run-time objects.

E. Support in IDE for multiple stakeholders

Both Bob and Melinda are supported in creating and exploring various scenarios. To engage different types of stakeholders in the BDD process, the IDE must provide suitable navigation mechanisms and interfaces that are appropriate for the needs of distinct stakeholders. A model navigation mechanism that enables Melinda to navigate between numerous scenarios is shown in Figure 8. Here, scenarios (*i.e.,* description of a

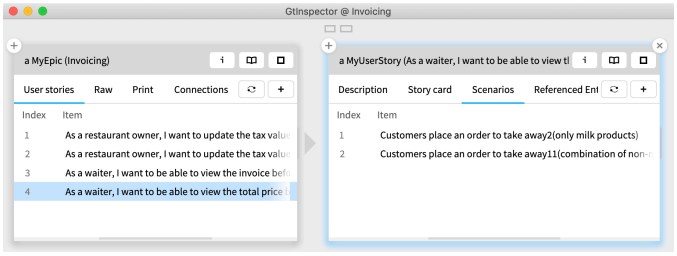

Fig. 8: Scenarios are attached to a user story

`Scenario` object) corresponding to a specific user story are collected and displayed in one of the views. On the other hand, Melinda explores scenarios pertaining to a specific class (see Figure 9).

Earlier we saw how Bob can compose and run examples using graphical interfaces. Melinda, on the other hand, can call example methods like any other method. An object inspector allows Melinda to inspect a specific representation of a domain object using a view suitable for her needs. The example

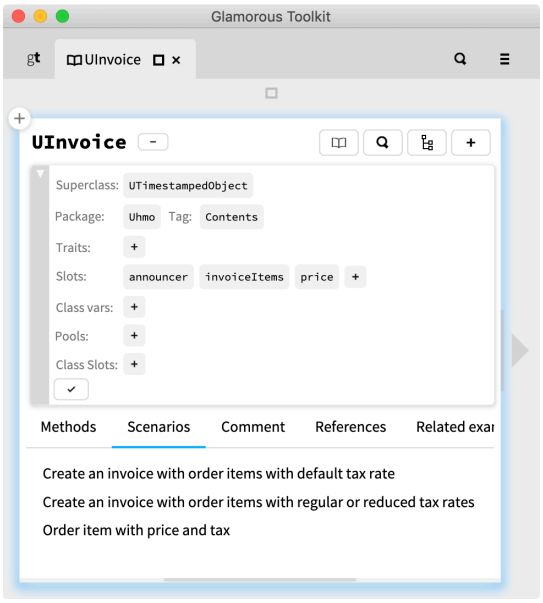

Fig. 9: Scenarios are attached to a class for efficient navigation

method in Listing 7, when executed returns an `Invoice` object. Melinda can decide to explore other representations of an `Invoice`, for instance, with a composition view, such as the one in Figure 7.

```
self
invoiceForOrderItemsReducedTax: {
self cappuccinoOrderItem }
regularTax: {
self cappuccinoOrderItem.
self coffeeOrderItem }
atTimestamp: '2020-11-17T18:53:50' asDateAndTime.
```

Listing 7: Invoking an example method

F. Summary

The custom graphical interfaces to support non-technical stakeholders to create and manipulate domain objects can help us to improve the current BDD workflow support in an IDE. Our building blocks align with the idea of low code development platforms (LCDPs) that, to engage multiple stakeholders in software development process, leverage model-driven engineering principles and provide infrastructure, such as graphical interfaces, and automatic code generation to develop entirely functioning applications [33]. Our approach pushes the idea of LCDP to complement BDD and solve the identified issues with the current workflow. Note that, we did not present a tool in this vision paper, but rather demonstrated how to adapt an IDE to better support non-technical stakeholders to do BDD with our proposed building blocks. The screenshots included are specific to the running example and will vary greatly depending on the context of other applications.

VI. Evaluation

To evaluate whether our approach to LowCode BDD is usable and effective compared to the existing BDD workflows supported by most available tools (*i.e.,* write textual scenarios, complete test cases, write the source code, execute test cases, and repeat), we have designed a controlled experiment. In the experiment, we will compare a baseline development process (*e.g.,* a typical BDD workflow supported in Cucumber) with our proposed low-code workflow, and ask the participants about their feedback on various factors, such as perceived usefulness and perceived benefits of our approach. We have identified about 96 contributors to feature files from 23 open source repositories mentioned earlier. Interested candidates from these 96 contributors will become subjects in our controlled experiment. We believe that these candidates having previous experience with BDD in reputed projects will be a perfect fit to experiment with our novel approach to BDD.

VII. Conclusion

In this paper, we argued that the current tool support for BDD, specifically in the available IDE plugins, lacks essential features to efficiently engage non-technical stakeholders in the BDD process. Current tools have limited opportunities for scenario specification, as well as to process the output of running tests. We have proposed an alternative BDD process to engage both technical and non-technical stakeholders in specifying and verifying the application behavior. We demonstrate through a running example of invoicing system for restaurants how non-technical stakeholders can visually compose behavior tests and discover inconsistencies in the underlying domain model through an inspectable output. Our proposed building blocks for an IDE allow to better integrate both technical and non-technical stakeholders in the BDD process. We are actively pursuing the evaluation of our approach through a controlled experiment with experienced contributors in reputed open source projects using BDD.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assistance" (SNSF project no. 200020-181973, Feb. 1, 2019 - April 30, 2022). We thank Norbert Seyff, Sebastiano Panichella, and Andrea de Sorbo for reviewing the manuscript. We thank Adwait Chandorkar for providing statistical insights into the usage of BDD tools in open-source BDD projects.

References

[1] M. Wynne, A. Hellesoy, and S. Tooke, *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.

[2] F. Zampetti, A. Di Sorbo, C. A. Visaggio, G. Canfora, and M. Di Penta, "Demystifying the adoption of behavior-driven development in open source projects," *Information and Software Technology*, p. 106311, 2020.

[3] L. P. Binamungu, S. M. Embury, and 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.

[4] T. R. Silva, J.-L. Hak, and M. Winckler, "Testing prototypes and final user interfaces through an ontological perspective for behavior-driven development," in *Human-Centered and Error-Resilient Systems Development*. Springer, 2016, pp. 86–107.

[5] N. Nascimento, A. R. Santos, A. Sales, and R. Chanin, "Behavior-driven development: A case study on its impacts on agile development teams," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 109–116.
[6] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.
[7] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.
[8] N. Bik, G. Lucassen, and S. Brinkkemper, "A reference method for user story requirements in agile systems development," in *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*. IEEE, 2017, pp. 292–298.
[9] R. K. Lenka, S. Kumar, and S. Mamgain, "Behavior driven development: Tools and challenges," in *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. IEEE, 2018, pp. 1032–1037.
[10] C. Solis and X. Wang, "A study of the characteristics of behaviour driven development," in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2011, pp. 383–387.
[11] A. Okolnychyi and K. Fögen, "A study of tools for behavior-driven development," *Full-scale Software Engineering/Current Trends in Release Engineering*, p. 7, 2016.
[12] S. (https://archive.codeplex.com/?p=storyq). Accessed: 2020-06-19.
[13] JDave. Tool repository at https://github.com/jdave/JDave. Accessed: 2020-06-19.
[14] NBehave. Tool repository at https://github.com/nbehave/NBehave. Accessed: 2020-06-19.
[15] E. (http://easyb.io/v1/index.html). Accessed: 2020-06-19.
[16] B. (https://teststackbddfy.readthedocs.io/en/latest/). Accessed: 2020-06-19.
[17] C. (https://cucumber.io/). Tool repository at https://github.com/cucumber/cucumber. Accessed: 2020-06-19.
[18] J. (https://jbehave.org). Tool repository at https://github.com/jbehave/jbehave-core. Accessed: 2020-06-19.
[19] C. (https://concordion.org). Accessed: 2020-06-19.
[20] S. (https://specflow.org). Tool repository at https://github.com/SpecFlowOSS/SpecFlow. Accessed: 2020-06-19.
[21] S. (http://spockframework.org/). Tool repository at https://github.com/spockframework/spock. Accessed: 2020-06-19.
[22] R. (http://rspec.info). Tool repository at https://github.com/rspec. Accessed: 2020-06-19.
[23] MSpec. Tool repository at https://github.com/machine/machine.specifications. Accessed: 2020-06-19.
[24] LightBDD. Tool repository at https://github.com/LightBDD/LightBDD. Accessed: 2020-06-19.
[25] S. (http://www.scalatest.org/). Tool repository at https://github.com/scalatest/scalatest. Accessed: 2020-06-19.
[26] Specs2. Tool repository at https://etorreborre.github.io/specs2/. Accessed: 2020-06-19.
[27] J. (http://jgiven.org). Tool repository at https://github.com/TNG/JGiven. Accessed: 2020-06-19.
[28] phpspec (http://www.phpspec.net/en/stable/). Tool repository at https://github.com/phpspec/phpspec. Accessed: 2020-06-19.
[29] G. (https://gauge.org). Tool repository at https://github.com/getgauge/gauge. Accessed: 2020-06-19.
[30] A. Chiş, O. Nierstrasz, A. Syrel, and T. Gîrba, "The moldable inspector," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2015, pp. 44–60.
[31] N. Patkar, "Moldable requirements," in *Benevol 2020: Proceedings of the 19th Belgium-Netherlands software evolution workshop*, 2020, p. To appear.
[32] R. Pawson and R. Matthews, "Naked objects: a technique for designing more expressive systems," *ACM SIGPLAN Notices*, vol. 36, no. 12, pp. 61–67, 2001.
[33] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, "Supporting the understanding and comparison of low-code development platforms," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2020, pp. 171–178.

APPENDIX

A. Model requirements for an invoicing system

- *Epic 1*: Placing an online order
  - *User story 1*: As a registered or unregistered user, I want to explore the online menu.
  - *User story 2*: As a registered or unregistered user, I want to add the menu items to the shopping cart.
  - *User story 3*: As a registered user, I want to do one click payment.
  - *User story 4*: As an unregistered user, I want to enter my delivery and payment details to make the payment.
- *Epic 2*: Placing an order in a restaurant
  - *User story 5*: As a waiter, I want to enter the selected menu items by guests to prepare an order.
  - *User story 6*: As a waiter, I want to be able to forward the received order to the chef.
  - *User story 7*: As a waiter, I want to be able to customize the order by modifying the default menu items.
- *Epic 3*: Invoicing
  - *User story 8*: As a restaurant owner, I want to update the tax values for both online and offline orders, in case they legally change.
  - *User story 9*: As a restaurant owner, I want to update the tax values for individual menu items.
  - *User story 10*: As a waiter, I want to be able to view the invoice before printing it to check if the correct prices and tax is applied.
  - *User story 11*: As a waiter, I want to be able to view the total price before printing the invoice.

B. Model BDD scenarios

Let us consider the following scenarios for *User story 11*:

```
Scenario 1: Customers place an order to take away
(only milk products)
Given an empty order
When the waiter adds Cappuccino to the empty order
And a cup of Cappuccino costs 4 EUR
And a cup of Cappuccino is taxed at 7%
And the waiter generates the Invoice for the order
Then the total invoice price is 7.28 EUR

Scenario 2: Customers place an order to take away
(combination of non-milk and milk products)
Given an empty order
When the waiter adds a Cappuccino and a black coffee
to the empty order
And a cup of Cappuccino costs 4 EUR
And a cup of black coffee costs 3 EUR
And a cup of Cappuccino is taxed at 7%
And a cup of black coffee is taxed at 19%
And the waiter generates the Invoice for the order
Then the total invoice price is 7.85 EUR

Scenario 3: Customer place an order to take away
(no milk products)
Given an empty order
When the waiter adds black coffee to the empty order
And a cup of black coffee costs 3 EUR
And a cup of black coffee is taxed at 19%
And the waiter generates the Invoice for the order
Then the total invoice price is 3.57 EUR

Scenario 4: Customer place an order to to take away
(combination of no-milk drink and food)
Given an empty order
When the waiter adds a black coffee and a pizza
margherita to the empty order
And a cup of black coffee costs 3 EUR
And pizza margherita costs 5 EUR
And a cup of black coffee is taxed at 19%
And pizza margherita is taxed at 7%
And the waiter generates the Invoice for the order
Then the total invoice price is 5.92 EUR
```

Listing 8: A sample feature description with a scenario