Ghengo: Test case generation based on natural language using BDD

Niklas Brocker
niklas.brocker@web.de
Hochschule der Medien
Stuttgart, Germany

Abstract

In agile software development using Behavior-Driven Development (BDD), clients are involved in the entire development process. Their requirements are recorded in natural language. This guarantees good products that meet customer needs. However, BDD also has a weak point which is in the area of testing. To verify that a system meets the requirements captured in natural language, developers must manually translate them into code at each step of development. This is time-consuming. It also creates many code snippets that interact with each other and make maintainability difficult. If the associated test code could be generated automatically based on the feature files, the development process would become easier and more efficient. With this in mind, this paper presents a prototype that can transform the natural language text of a feature file into test suites for a Django API. In contrast to existing approaches, the software should be able to generate complete test cases without an existing implementation. To do this, the prototype uses a compiler that analyses the meaning and data of the text.

The prototype is tested using a simulation in which various feature files are converted. This work's analysis shows that the resulting software is capable of generating tests without an existing implementation. The speed is at a similar level to previous work. The prototype also offers interesting possibilities for further development. However, it must be used in a real-life project to allow a detailed evaluation.

Keywords: Compiler, BDD, NLP, Agile, Generation, Tests

1 Introduction

1.1 Motivation

Agile methods are very common in the field of software development. This includes Behavior-Driven Development (BDD), a technique that developers use in about 20 percent of engineering processes as of 2020 [4][20]. There, requirements are recorded in natural language and clients are involved in every step of the development process - an approach that ensures good products and satisfied customers [3][5].

However, Behavior-Driven Development also has a weak point which lies in the area of testing. In order to check whether a system fulfills the requirements recorded in natural language, developers are forced to translate these requirements manually into code. The requirements are usually available as feature files written in the *Gherkin* language [3][5].

In addition, developers have to repeat this translation work very often in the context of BDD. The reason: The feature files describe different scenarios in individual steps (Given, When, Then). Developers must write the corresponding test code for each of these steps. In the test-first approach which is usually chosen here, these tests are already created before the product code is implemented. Developers have to ask themselves at each step: what would the code look like that could achieve or test what each step describes? Unlike traditional test cases, this creates many small code snippets for the steps. Each of these code snippets can interact with any other which leads to worse maintainability. But the key disadvantage is that developers have to spend a lot of time translating feature files to tests. This causes costs and delays the completion of a product [5][27].

If the feature files could be used to automatically generate the associated test code, this would make the development process more efficient and easier. To achieve better maintainability, it would be desirable to create complete test cases instead of code snippets.

1.2 Related Work

There are already some papers that look into test generation.

Pynguin, Evosuite and Randoop generate code based on existing implementation. They work with a test-last approach and generate tests that can be used to check whether a system still works as desired after changes have been made [15][22][25]. So it is not possible to use these three approaches to generate tests in BDD. First, it cannot be assumed that the implementation already exists. Second, tests in BDD are supposed to check the requirements of the system.

There are also papers that have looked at code generation in BDD or in the agile development environment.

- **behave_nicely** is a software that can generate code snippets of the individual steps using feature files. Thus, developers no longer have to worry about their maintenance [30].
- Soeken et al. have dealt with the question of how information can be extracted from natural language. The result is software that generates tests semi-automatically. A user enters the steps into a dialog and receives code snippets that represent them. Then developers can

correct and reuse these snippets. The interaction of human and machine is therefore necessary here [28].

- **Kirby** uses feature files to generate complete test cases. The generation is performed in cycles and in several steps. In the first step, the outline of the tests is created. In the next step, developers must create parts of the implementation. Kirby can then fill in further parts of the tests. This is repeated until the implementation is finished and Kirby has generated the tests completely. This process is based on the Red-Green-Refactor cycle which is common in Test-Driven Development (TDD) [17][18].

The prototype uses and combines some ideas of these papers.

1.3 Goal

This paper will present a prototype that converts feature files to tests in Python and allows testing their defined requirements. The resulting tests should be executable for a common test library in Python.

The prototype should be usable in an agile, test-first approach and thus be able to work without prior implementation. To make this possible, it specializes in one use case: generating tests for a Django API. In the implementation of this prototype, ideas from the papers mentioned in section 1.2 are used.

Figure 1 shows the setup. A customer has requirements for a Django API. These are recorded in a feature file. The prototype evaluates this file and generates Python tests. These tests verify that the API meets the requirements.

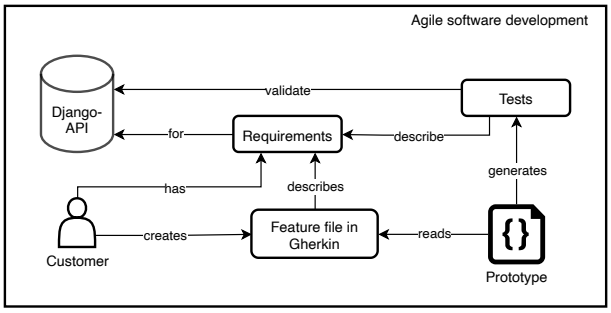

Figure 1. Setup of generation

Normally, in agile software development the customer would describe the requirements from a user's point of view [26]. In such an application, there would also be a frontend. This, however, is not part of this paper.

In order to achieve the formulated goals, it must be clarified how the prototype can process the input. It must be possible to translate the text into an internal structure that describes it. The prototype should use the text to answer questions like: which sections of the text contain data that needs to be extracted? How can data be extracted? What does a sentence mean? How can sentences be categorized so that translating the meaning of a sentence to code becomes possible? When these questions have been answered, the next step is to find a suitable approach for transforming text to tests. It is important that the software is extensible for other programming languages and test libraries while remaining fast. The unoptimized version of Kirby (see section 1.2) needs 5 seconds per scenario - a speed which the prototype should be able to undercut [17, p. 59].

2 Conceptual design

This section presents the conceptual design of the prototype called Ghengo (IPA: ['gɛŋgoʊ]). It deals with practical implementation questions and the architecture of the prototype.

Ghengo receives the requirements for the Django API in a feature file and must translate these requirements into tests. This can be done with a compiler. Compiling in Ghengo will consist of lexical analysis by a lexer, syntactic analysis by a parser, and code generation (see figure 2) [10, p. 31][31, p. 1]. Type checking is not required because Gherkin is a language without typing [3].

Before conceptual decisions on the individual steps of the compilation are explained in greater detail, it must be determined for which test framework the generated code is executable.

2.1 Test framework

A widely used test framework for Python is the library PyTest [21]. It should be able to execute the code output. PyTest and Gherkin offer some similar features. Gherkin tags are one example for these similarities. They give developers the ability to execute only specific parts of a feature file. In PyTest, there is a decorator that does the same thing. Listing 1 and listing 2 show their similarity. There are also similar functions for data tables in Gherkin and PyTest.

```
1 @pytest.mark.foo
2 def test_only_sometimes():
3   pass
```

Listing 1. Pytest test case with decorator

```
1 # language: de
2 Funktionalität:
3   @foo
4   Szenario:
5     Gegeben sei ein Auftrag 1
```

Listing 2. Tag in feature-file

For the creation of Django models within tests the library *pytest-factoryboy* is useful [7]. It is assumed at this point

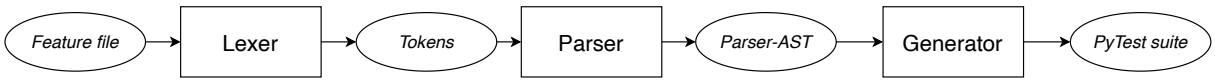

Figure 2. Steps of Ghengo-Compiler

Tag	Feature	Background	Examples
ScenarioOutline	Scenario	And	But
Given	When	Then	DataTable
DocString	Language	Comment	Rule
Language			

Table 1. Tokens for keywords from Gherkin

that the library will be available when working with Ghengo later.

2.2 Lexer

In the first step of compilation, the prototype's lexer must tokenize Gherkin text from a feature file. Since in Gherkin each line of text typically starts with a keyword, it makes sense to perform the lexical analysis required for the conversion on a line-by-line basis [3].

The lexer creates tokens which are assigned keywords or patterns. The tokens also indicate which sections of the text belong to them. These sections are also called lexemes [9, p. 111].

Table 1 shows all tokens representing keywords from Gherkin. Their meanings are described in the Gherkin documentation [3].

Ghengo supports German and English as input, but works best with German. The patterns for the individual tokens differ depending on the selected language. If the term *Scenario* was used in English, the equivalent in German would be *Szenario*. Tokens must react equally well to both depending on the selected language.

In addition to tokens for keywords from Gherkin, Ghengo uses some tokens that are not explicit keywords:

- **Empty** - an empty line
- **Description** - free text
- **EndOfLine** (in short EOL) - end of a line
- **EndOfFile** (in short EOF) - end of a file

The process of lexical analysis will be illustrated using the feature file from listing 2.

First, the lexer looks at line 1 or the text *# language: de* and checks which token matches this text. In this case it is the *Language* token to which the complete text of this line can be assigned as lexeme. So the lexer creates the token. In the first line, it has checked the entire text. To mark the end of the line, the lexer adds an *EOL* token.

The lexer repeats this process for lines 2 to 4. Because of the *Language* token it knows the chosen language and discovers the German keywords *Funktionalität* and *Szenario*. The lines result in the tokens *Feature*, *EOL*, *Tag*, *EOL*, *Scenario* and *EOL*.

In line 5, the lexer first recognizes a *Given* token. However, the lexeme of the token is only "Gegeben sei". Following this there is free text to which no token has been assigned yet. Accordingly the tokens *Given*, *Description* and *EOL* are created from the line.

The lexical analysis is finished. To indicate the end of the file, the lexer creates a *EOF* token. The lexer creates a list from the resulting tokens which can be used further.

2.3 Parser

In the next step, the parser of the prototype must perform the syntactic analysis [31, p. 1]. For this purpose, it receives the tokens created during the lexical analysis (see section 2.2), checks their syntax and creates an abstract syntax tree (AST) - a tree representing the feature file. The basis for the analysis is the context-free grammar of Gherkin which was created using the documentation of Gherkin [3]. Ghengo uses a top-down parser since there is a single non-terminal symbol on the left-hand side of these rules in each case, and ambiguity and left-hand recursion can be ruled out [24][31, p. 9].

Such a Gherkin analyzing parser already exists for Python [2]. This parser did not prove successful in initial tests, so Ghengo uses its own parser. Ghengo's parser works recursively, as it is easier to implement and provides better error localization than a table-based parser [13, p. 6]. People using Ghengo should receive detailed error messages and suggestions for improvement in case of incorrect syntax.

To implement its own parser, Ghengo needs three components or classes: the **operators** of the rules, the **non-terminals** and the **terminals**.

There are the following **operators** which are classes in Ghengo [31, p. 7 ff]:

- **Optional** is [ ] in the extended Backus-Naur-Form (EBNF). The class contains optional symbols.
- **Repeatable** is { } in EBNF. Child symbols can be repeated multiple times.
- **OneOf** is | in EBNF. The class represents the logical OR.
- **Chain** specifies symbols that must follow each other.

That way, Ghengo can define the production rules of Gherkin. Listing 3 shows how it notates a production rule in code.

The class for **non-terminals** specifies how they can be replaced. **TerminalSymbols** are wrappers around the tokens

```
gherkin_rule = Chain([
  Optional(language_rule),
  Optional(feature_rule),
  TerminalSymbol(EOF),
])
```

Listing 3. Example for definition of production rule within Ghengo (EBNF: $S = [LANGUAGE],\ [FEATURE],\ ''eof''$)

Data from parser AST	Needed for...
Name of Feature	Name of test suite
Name of ScenarioDefinitions	Name for test cases
Text of steps	Generation of statements
Datatables	Generation of statements

Table 2. Data from AST of the parser and their intended use for the generator

that can be seen in table 1. The above classes can be used to map Gherkin's grammar in code and perform recursive syntactic analysis.

After the analysis is complete, an AST is available. Table 2 contains the most important information and how it will be used later on.

Ghengo must loop through each ScenarioDefinition (e.g. Scenario, ScenarioOutline) from the AST. For each definition a test case is created, whose name depends on the name of the ScenarioDefinition. The content or statements of the tests are created using the steps. A feature is represented by a test suite [3].

2.4 Code Generator

The code generator performs the largest and most complicated part of the translation from Gherkin to test cases. It receives the AST of the parser (from now on called **parser AST**), has to extract information from it and create test cases for PyTest.

The parser AST specifies the structure of the tests, e.g. a test suite is created from a feature and a test case is formed from a scenario. In addition, the steps in the parser AST contain information that Ghengo must analyze. This analysis is divided into two areas:

- **Analysis of meaning**: What does a phrase mean? Ghengo categorizes phrases into different *actions*. Actions are commonly used in Django tests. Examples are:
  - Creating an instance of a model
  - Making a request to the API of the project
  - Analyzing the response, the database or a previously created model instance

  Ghengo has to analyse which of the actions is meant with a given sentence.

Ghengo's supported action	Step type
Creating model instance	Given
Sending request to API	When
Checking database	Then
Analyzing response	Then
Checking a model instance	Then

Table 3. Assignment of supported actions to step types

- **Analysis of the data**: What data is included in the text?

The structure of the tests, the meaning and the data are translated into suites using a generation technique. Since the type of generation technique has an impact on all other steps mentioned at this point, the first question addressed must be which technique Ghengo's generator should use.

2.4.1 Generation technique.

The code generator receives the parser AST and has to transform it to code. The generated code is expected to be very dynamic because it is based on natural language. Because of that, Ghengo uses the generation technique of transformation [14, p. 124 ff][29, p. 149, 151]. The generator transforms the AST of the parser. A second tree is created that represents the code to be generated. To distinguish the two trees, the second tree will be called **output AST** from now on.

For the transformation Ghengo uses the information mentioned in the introduction. In the last step the transformation to code and the suite are created considering the global settings. This process can be seen in figure 3.

2.4.2 Determine meaning.

To enable Ghengo to fill the test cases or the output AST, its generator must determine the meaning of the texts from the steps of the parser AST. The question here is what a user means with a phrase. A first categorization is required.

The type of a step already gives the generator some useful information. *Given* represents bringing the system to a desired state, *When* represents an action by the person using it, and *Then* is a keyword for checking the system [3]. Table 3 shows which steps match some of the mentioned actions in section 2.4.

For each step type, the generator creates associated Tiler instances (e.g. *GivenTiler* or *WhenTiler*). Tilers apply natural language processing (NLP) to their step using the *Spacy* library which contains models for different languages [16].

Each Tiler has different *Converter* classes associated to it (see table 3). Each Converter represents an action and can convert a step to statements for that action. Tilers search for the most suitable Converter and take statements from it which the code generator attaches to the test case. After the prototype has run through all the steps in a scenario, it assigns the test case to the suite.

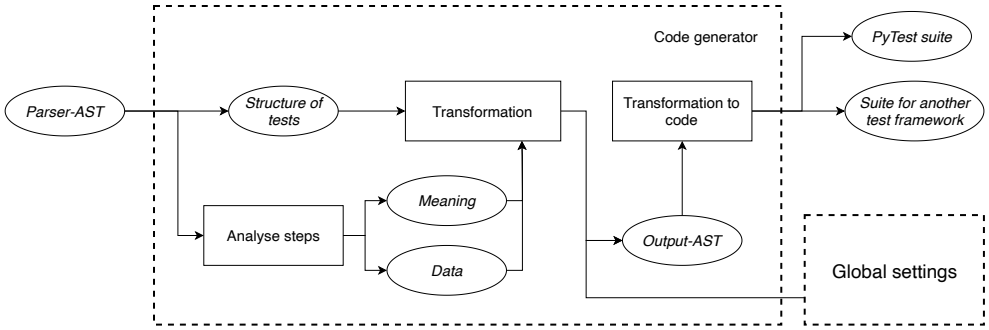

Figure 3. Code generation process with Ghengo after choosing the generation technique

2.4.3 Extract data. The categorization described in section 2.4.2 has clarified which action the user means. The associated Converter must now extract data from the text and embed it in statements. The Converter uses information about an existing Django project. Ghengo gets this information by the specification of a path to a Django project.

Ghengo uses two types of data: names and values. Using names, Ghengo creates references to existing or planned implementations in Django (e.g. Django models). Values are directly put into the resulting tests. These values are usually defined as data types of the programming language being generated which in this work is Python. Some examples are Boolean, integer or string. In the following they are called **native values**.

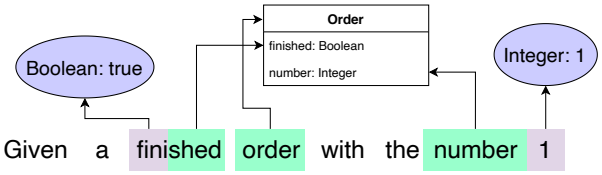

Figure 4. Data in a sample sentence

Figure 4 shows the data that Ghengo pulls from the sentence "Given a finished order with the number 1". The words *finished*, *order*, and *number* are names. *order* references the model *Order* and the words *finished* and *number* reference the fields *finished* and *number*. The word *finished* implies a native value - the Boolean *True*. The number *1* represents an integer.

This example shows that tokens can represent names and/or native values. The operation method of Converters is always the same: They run through all tokens created by NLP and analyze them. Converters look for tokens that contain data needed for an action. Each Converter knows by its action which data is necessary and which is optional.

First, the Converter tries to find names in the text. Soeken et al. evaluate nouns as references to classes, adjectives as references to attributes and verbs as references to methods (see section 1.2) [28]. Since Ghengo specializes in Django and Converters specialize in specific actions, a slightly modified approach can be used. When it identifies interesting tokens the following happens:

1. The Converter removes all tokens that are stop words. They do not contain useful information [23, p. 27].
2. The first token is always a keyword from Gherkin (Given, When etc.) and is always ignored.
3. There are "blocked" tokens. They usually describe required data that Ghengo uses in other places. In figure 4 the word *order* would be blocked because it already describes the model.
4. Converters search (with some exceptions) for nouns, adjectives, verbs, proper nouns and adverbs. They all describe names that Ghengo processes further. Ghengo gets the information about the word types by using POS tagging from Spacy.

Afterwards the Converter has a list of tokens representing names. Ghengo now tries to find the references. Example: *order* should be assigned to the model *Order* and *number* should be assigned to the model field *number*. For this purpose Ghengo uses two classes: **Lookout** and **Translator**.

Usually code is written in English but Ghengo supports German as an input. So Ghengo uses the **Translator** class to translate texts. It uses the DeepL API [1]. The prototype stores the results in a cache, so it needs to send fewer requests. The cache ensures good performance for repeated translations.

Lookout classes are responsible for searching. They compare different words and texts, determine their similarity and use the Translator in combination with lemmatization. Ghengo uses them for two problems: **finding words** (e.g. to determine the CRUD method of a request) and **finding matching references in the implementation**. Lookout classes provide the ability to return a fallback object if they do not find a matching result.

For example, when analyzing an existing implementation of the given Django project, the Lookout uses the Translator. For the sentence "Gegeben sei ein Auftrag" (engl. "Given an order"), the Lookout translates the word *Auftrag* (engl. order)

using the Translator and searches through all implemented models. If there is a model with a similar name, it will be returned. If no matching model exists, the lookout returns a fallback object that guesses the name. If there was no matching implementation yet, Ghengo would guess the name *Order*. This allows Ghengo to work prior to any implementation.

Lookout classes almost exclusively return so-called **Wrappers**. They wrap the object that is actually of interest. If a fallback is necessary, wrapper objects are empty and fill themselves with important information (e.g. the name). Figure 5 shows a suitable class diagram. There the wrapper classes have the function *get_fields* which returns all fields of the model. In case of an already implemented model, they can search for its fields. The *ModelWrapper*, on the other hand, returns an empty list due to lack of implementation. Ghengo works with Lookouts for models, model fields, Django apps, API endpoints, Serializer fields, and Viewsets. Lookouts can find tokens for CRUD methods, file extensions (e.g. txt or png) and comparison operators (e.g. == or <=).

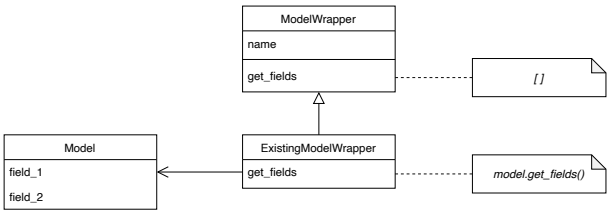

Figure 5. Class diagram for wrappers

Using Lookout classes, Ghengo can determine which objects the names reference. For determining native values, **ExtractorOutput** classes are used. They receive a token (also called **source**) representing a name (e.g., from the previous examples *number* or *finished*) and use it to create a native value (e.g., 1 or True). ExtractorOutput classes can specialize in different data types to produce particularly good results. If Ghengo has no information about the data type, the ExtractorOutput must draw conclusions about the data type based on the source.

Converters use only Extractor classes that have information about the area of application of the native value (e.g. a Boolean for a field in a model). Extractors use an ExtractorOutput class and get a native value that they can customize to the area of application.

The following list describes the data types that the native values supported by Ghengo can take.

- **String**. ExtractorOutput classes recognize strings if words after the source are marked with quotes. An example is *Given an order named **"foo"***.
- **Boolean**. The classes recognize Boolean values by negation. If the source or the associated verb is negated, the value is *False*. In the sentence *Given an order that is **not** finished*, *finished* is negated with the word *not*. So here the output would be *False*. When guessing, Ghengo looks for words like *True* or *False*. If, while guessing, the source is a verb or an adjective, Ghengo checks its negation.
- **Numbers (float, integer, decimal)**. Numbers are always child elements of the source in the dependency tree. So the word *number* in the sentence *Given an order with number 1* is the head of *1*. If Ghengo has no information about the data type, it looks for text that it can cast from a string to a number. Ghengo also processes written out numbers like *two*, *twelve* or *none*.
- **Variables**. If the prototype knows that a variable must be created from the source, it searches for proper names, numbers or text in quotation marks. This results in a variable with a suitable name. In the sentence *Given a user Bob*, *Bob* would be a proper name leading to the variable *bob*. If Ghengo does not know anything about the type, it looks for text with quotes and the characters <> (e.g. "<foo>").
  Some Converters also allow to define an ID. In the sentence "Given a finished order 1", Ghengo interprets the *1* as an ID and creates a variable for it in the output AST. If a sentence in another step references this order (e.g. "When order 1 is deleted"), Ghengo recognizes the context (the model *Order*) as well as the ID and creates an associated reference to this variable.
- **List of strings, numbers or variables**. The prototype supports the listing of multiple strings, numbers or variables (e.g. *Given an order with files 1, 2 and 3*). These lists always lead to dependencies which Ghengo can run over in the dependency tree [12][19]. This does not work if Ghengo has to guess the data type.
- **Data structures (dict, set, tuple)**. Ghengo only partially supports data structures. Their description in natural language is difficult, the creation of understandable steps therefore a challenge. Output classes recognize data structures reliably only if they are quoted (e.g. "(1, 2)". Ghengo does not support nested data structures.
- **GenerationWarning**. If an Extractor detects that an ExtractorOutput had problems creating a native value, it creates a so-called **GenerationWarning** and passes it to the Converter. So, if there are any errors, Ghengo will catch them and inform the developer who can fix the issues in the generated code.

Let's go over an example that summarizes everything covered so far. Given the following steps:

- **Given** a finished order 1
- **When** order 1 is deleted

The Converter for creating models carries out the first step. It recognizes *order* and identifies the word as a reference to the model *Order* using Translator and Lookout. It is followed

by *1*. Since it is a number, Ghengo interprets this as an ID that the Converter remembers. Additionally, it detects the word *finished*. Using Translator and Lookout, the prototype concludes that it references a BooleanField from Django. So it has to find a Boolean value using the source *finished* and calls the appropriate Extractor or ExtractorOutput. The word is not negated. That means, the native value must be *True*. From all this information, the Converter creates a statement in the output AST. It must create an order with the field *finished* and its value *True*. The order is assigned to the variable that Ghengo created using the ID.

The second step is performed by a Converter that specializes in requests. It first recognizes the word *order*. That means that the request is related to the *Order* model. It also discovers the ID *1* and searches the statements in the output AST for a matching variable in the context *order*. The Converter can detect the CRUD method using the word *deleted*. It uses Lookout and Translator to find a DELETE route for the order model and checks what data the endpoint expects. In this case, the route needs a primary key that identifies the order to be deleted. The Converter uses the variable of the order for this purpose. One possibility for the resulting code is shown in listing 4.

```
1  order_1 = order_factory(finished=True)
2  client = APIClient()
3  client.delete(reverse('orders-detail',
   ↪ {'pk': order_1.pk}))
```

Listing 4. Example for generated code

2.4.4 Create Output-AST.

Python offers the possibility to create an (output) AST for Python code [6]. However, Ghengo does not use it, because this would limit the extensibility for different programming languages and libraries.

Instead, the output AST is created using various **transformation classes** that describe common keywords from programming languages. Here are some of these classes:

- **TestSuite**. The class includes everything that belongs in the particular suite. So the first step in creating the output AST is to instantiate this class.
- **TestCase**. This allows the generator to create individual test cases. They are normally assigned to a TestSuite.
- **Statements**. They represent instructions. Test cases usually consist of several statements. Each Gherkin step results in one or more statements.
- **Expression**. Expressions stand for values.
- **Variable**. A Variable contains a value.
- **VariableReference**. This class allows referencing a previously defined variable. Using such a class opens the option to remove unreferenced variables from the output AST at the end of generation to make the code more readable.
- **Import**. Often you have to import other files or modules to use methods and functions. This class represents such imports.

The following section describes the main functions shared by the transformation classes.

2.4.5 Transformation to code.

The transformation to code is the main feature of the transformation classes. The classes contain small templates that determine the syntax of the code to be generated. Transformation classes improve the readability of the code by splitting long lines and removing unneeded variables.

```
1  def test_all_users():
2    user_1 =
     ↪  create_user(email="foo@local.local",
     ↪  first_name="Max")
3    user_2 = create_user()
4    assert len(get_all_users()) == 2
```

Listing 5. Nicht verwendete Variablen und lange Zeilen

Lines 2 and 3 in listing 5 contain *user_1* and *user_2*, two variables that are not used in the test. The code would be readable more easily without them.

The code transformation process is created in a way that Ghengo will be able to support not only the PyTest library but also other test frameworks in the future. Ghengo uses some transformation classes that are specialized in certain circumstances (e.g. when PyTest was selected as output). This means the prototype must react when transforming to code if certain classes match the global settings and use them instead (see also figure 3).

An example: Given a class for creating an expression that creates a model entry (*ModelExpression*). However, the creation works differently with test library A and with test library B. Accordingly, there will be different classes for each use case. So Ghengo has to react flexibly when creating the output AST and use the classes that match the selected settings. For test library A this will be *TestAModelExpression* and for test library B *TestBModelExpression*.

Finally, transformation classes should also listen to events - especially when a TestCase or TestSuite is extended. For example, this would be the case when a code snippet is used that requires an import. Once the statement with this code snippet is added to the TestCase, it can react and add an Import instance in the AST.

3 Evaluation

After implementing the prototype, three questions arise: Does the translation from Gherkin to PyTest work? Is the

resulting test code valid? How is the performance when generating the tests and what takes the longest? This can be clarified in the context of a simulation.

The simulation imitates the process from figure 1. To validate the generation against existing implementation, the simulation starts with a minimal Django project presented by section 3.1. The requirements for the system are extended in several steps. As the requirements change, new feature files are created. Ghengo has to translate the files and generate tests in the simulation. The resulting tests will be analyzed to answer the questions put above.

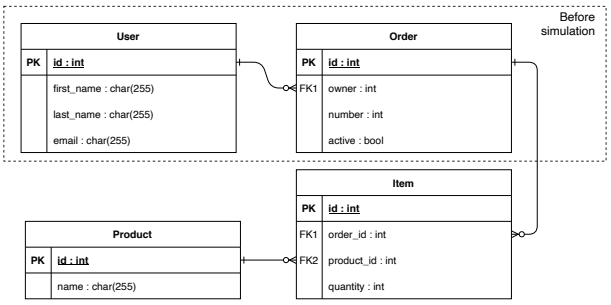

Figure 6. Database setup after step S-4

3.1 Setup

In the simulation, a Django API should be created for a small online store. For this purpose, a database structure already exists which can be seen in the box in in figure 6.

The project already contains some API endpoints to create, delete and get orders. Authentication is built into the endpoints so that the system recognizes which user sent the request.

So far, the project consists only of the implementation and does not contain any tests. With the simulation the online shop is developed further in the context of BDD.

3.2 Simulation steps

The simulation goes through several steps:

- S-1 **Integration of Ghengo into the project**. The prototype must be integrated into the project before it can be used to generate tests.
- S-2 **Requirements for existing system**. The development on the project begins. The first step is to record requirements for the existing system in a feature file. These requirements are used to create tests that verify whether the previous implementation of the system works as intended.
- S-3 **New field**. The first extension of the system is created. The *Order* model gets an additional field that describes whether an order is active. Finished or canceled orders count as inactive and should not be included in the GET requests.
- S-4 **New API calls and new model**. The online store now implements the first products represented by a new model: *Product*. Order and Product are linked via a M2M relationship using the *Item* model. An Item represents a product in an order and also contains the selected quantity. Figure 6 shows the new database structure. New CRUD methods for the *Product* model are added to the API. There is also one route to assign a product to an order.
- S-5 **Permissions**. Not every user should be able to perform all actions. For this reason there are the permissions: “Can create product” and “Can change product”. Only users with the corresponding permissions are allowed to create or edit products.
- S-6 **Customization of an API call**. The API gets an extension: While users had to create an order first and then add the respective product, both will now be possible in one step. When creating an order, it should be possible to specify associated products and their quantities. The online store then automatically creates the appropriate *Item* entries.

3.3 Execution

Subsequently, the previously defined steps were carried out in practice: A developer hands over feature files to the prototype that describe these steps. Ghengo reads them and generates test cases. The developer checks the resulting code and then integrates it into the project’s existing test files.

During execution the generated code was analyzed. To get a better insight into the performance, Ghengo converts each feature file 20 times. Variations are compensated for with the help of the average values determined in the process.

The duration was examined using the Python module *time* [8]. A Mac computer with a 2.8 GHz quad-core Intel Core i7 processor and 16GB memory ran Ghengo. The computer was manufactured in 2017.

4 Discussion

This section evaluates the prototype based on the experience gained in the simulation and identifies its strengths and its weaknesses. The following questions are addressed: Does Ghengo integrate well into a project? Does the transformation from natural language to tests work? Is the generated code valid? And: How is the performance?

Integration: The simulation showed that Ghengo can be easily integrated into an existing project. However, developers must copy the prototype folder into a Django project and install its dependencies. An automated process would be desirable at this point and would simplify the integration. Ghengo thus shows the qualities of a library, but it still lacks the finishing touches at this point.

Transformation from language to tests: The simulation showed that Ghengo can generate tests from natural

language. Simple sentences were correctly transformed even without existing implementation. Errors that initially occurred in lists of native values and in guessing names could be fixed by writing the corresponding implementation. After that Ghengo was able to use the information from the project correctly.

Nevertheless, it cannot be ruled out that Ghengo might have difficulties with certain texts under real-life conditions. Possibly, the prototype would then be confronted with sentences not considered during its development. Perhaps it would also have to process sentences in which names, references and native values are conveyed differently than previously assumed. This could lead to inaccuracies and incorrect transformation - problems that need further clarification in the future because it is outside the scope of this paper.

Furthermore, what was already hinted at in section 2.4.3 has been confirmed: Ghengo has problems with nested data structures and indicates them to developers using GenerationWarnings.

Code: The generated Python code for the PyTest library was valid in every step of the simulation. If Ghengo has to guess parts of the code, it compensates the lacking parts with placeholders. In most IDEs, warnings pop up for these placeholders. This is to be expected and even desired, because it tells developers that something needs to be adjusted.

Ghengo also ensured good readability of the code in all steps of the simulation. No unused variables appeared in any generated test case. Ghengo split long lines into several short ones.

Performance: In the simulation, the generation per scenario took only 1 to 2 seconds. If Ghengo encounters nested data structures, it needs 3 seconds per scenario (see step S-6). Within the overall very fast generation process, the *Lookout* classes that have to apply NLP multiple times and translate texts take the most of the time. It has also been shown that extracting data or creating the statements takes less time than determining the meaning or finding the best action.

To achieve a good performance for translations, the prototype uses a cache. Without a cache, Ghengo has to execute multiple requests against the DeepL API and takes longer for test generation. During simulation, the cache proved its worth and initially solved existing problems.

The simulation also showed that loading the Spacy models takes an average of 4 to 5 seconds. If the feature file is small, this can take up to 60 percent of the generation time.

On the other hand, missing implementation has no effect on performance. In fact, Ghengo can be expected to be faster before implementation, since it has to search through a larger code base afterwards.

Summary: In summary, Ghengo is able to generate test cases in PyTest for Django from natural language. Furthermore, the simulation has shown that the prototype works correctly even in the absence of an implementation, thus providing exactly what is needed in the test-first environment of agile software development. It has also been shown that the prototype can generate test cases with less information than methods described in previous work that rely on the red-green-refactor cycle process in TDD [17, p. 26 ff][18]. Due to fast generation and good performance the prototype promises efficient work. However, Ghengo can still be improved in some areas. Examples include nested data structures, recognizing lists without implementation, translating text without an external API, and loading Spacy models. It is also to be expected that Ghengo processes certain formulations incorrectly which were not considered during its development.

5 Conclusion and Future Work

With the creation of the prototype, the goal of this work has been achieved: It could be shown that it is possible to generate suitable test cases using feature files written in natural language. The specialization on Django and the implicit information about the step types allow Ghengo to generate test code even without an existing implementation. Thus, the prototype achieves better results than generation concepts described in previous publications and is particularly suitable for agile work.

If there is already an implementation, Ghengo can reliably generate test cases. Under these conditions, the prototype recognizes e.g. listings of native values. In all cases, it is faster than the software from previous work and generates readable code that transforms the described requirements from feature files.

However, Ghengo has problems with the implementation of complex requirements. And it cannot be ruled out that further weaknesses could become apparent under real-life conditions. Such problems would have to be tackled in further development cycles - in accordance with the basic principles of generative programming [11, p. 328 ff].

For use in real-life projects, the prototype should be available as a Python package. Alternatively, a user interface (UI) could be provided to developers. Another option would be the integration into an IDE. UI or IDE would have the advantage that the prototype would not have to reload the Spacy models with each generation.

In real-life projects, it is common for product owners to define requirements from the front-end perspective [26]. Feature files then describe requirements that affect both frontend and backend. Approaches for further development of the prototype can also be derived from this: One could extend Ghengo to generate tests not only for Django, but also for frontend technologies like React or Vue. A generation for both platforms (frontend and backend) at the same time would be the next step. In addition, extended versions of the prototype could also support other test frameworks.

Finally, development potential in the processing of tests should be addressed: The prototype's use could be further improved if it could evaluate existing test files in the future and use the information they contain to update subsequent tests. Thus, managing test cases with a mixture of generated and developer-created code would be easier. In its current version, Ghengo does not look at existing test code. The prototype only uses information from the feature file and outputs complete test suites. If requirements or an existing feature file change, the generated file must be compared with the previous version. Developers are faced with many questions: What must be retained from the old file and what can be taken over from the new version? Which parts of the code originate from Ghengo and which parts do not? Is it possible to recognize what has been adapted in the old file?

In conclusion, Ghengo can generate tests with little information about a project. In an initial simulation, it met the expectations placed on it. And it offers interesting starting points for further development.

References

[1] [n.d.]. *DeepL API.* Retrieved October 5, 2021 from https://www.deepl.com/de/docs-api/

[2] 2019. *Gherkin for Python.* Retrieved September 9, 2021 from https://github.com/cucumber/gherkin-python

[3] 2019. *Gherkin Reference - Cucumber Documentation.* Retrieved September 1, 2021 from https://cucumber.io/docs/gherkin/reference/

[4] 2020. *14th annual State of Agile Report.* Retrieved August 18, 2021 from https://www.qagile.pl/wp-content/uploads/2020/06/14th-annual-state-of-agile-report.pdf

[5] 2020. *BDD Pros and Cons.* Retrieved October 28, 2021 from http://www.automated-testing.com/bdd/bdd-pros-and-cons/

[6] 2021. *ast — Abstract Syntax Trees.* Retrieved September 10, 2021 from https://docs.python.org/3/library/ast.html

[7] 2021. *factory_boy integration with the pytest runner.* Retrieved September 27, 2021 from https://github.com/pytest-dev/pytest-factoryboy

[8] 2021. *time — Time access and conversions.* Retrieved October 3, 2021 from https://docs.python.org/3/library/time.html

[9] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[10] Charles Consel, Fabien Latry, Laurent Réveillère, and Pierre Cointe. 2005. A Generative Programming Approach to Developing DSL Compilers. In *Generative Programming and Component Engineering*, Robert Glück and Michael Lowry (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 29–46.

[11] Krzysztof Czarnecki. 2005. Overview of Generative Software Development. In *Unconventional Programming Paradigms*, Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 326–341.

[12] Francesco Elia. 2020. *Constituency Parsing vs Dependency Parsing.* Retrieved September 3, 2021 from https://www.baeldung.com/cs/constituency-vs-dependency-parsing

[13] Joachim Fischer, Klaus Ahrens, and Ingmar Eveslage. 2014. *Compilerbau.* Retrieved August 22, 2021 from https://www.informatik.hu-berlin.de/de/forschung/gebiete/sam/Lehre/Compilerbau/vorlesungsfolien/CB14_8

[14] Martin Fowler and Rebecca Parsons. 2011. *Domain-Specific Languages.* Pearson Education, Inc.

[15] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software *(ESEC/FSE '11).* Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[16] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. *spaCy: Industrial-strength Natural Language Processing in Python.* https://doi.org/10.5281/zenodo.1212303

[17] Sunil Kamalakar. 2013. *Automatically Generating Tests from Natural Language Descriptions of Software Behavior.* Retrieved October 30, 2021 from https://vtechworks.lib.vt.edu/bitstream/handle/10919/23907/Sunil_Kamalakar_F_T_2013.pdf

[18] Tun Khine. 2019. *Red, Green, Refactor!* Retrieved September 2, 2021 from https://medium.com/@tunkhine126/red-green-refactor-42b5b643b506

[19] Lucas Kohorst. 2019. *Constituency vs. Dependency Parsing.* Retrieved September 3, 2021 from https://lucaskohorst.medium.com/constituency-vs-dependency-parsing-8601986e5a52

[20] Prof. Dr. Ayelt Komus and Moritz Kuberg. 2017. *Status Quo Agile.* Retrieved August 18, 2021 from https://www.gpm-ipma.de/fileadmin/user_upload/GPM/Know-How/Studie_Status_Quo_Agile_2017.pdf

[21] Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. 2004. pytest 6.2.4. https://github.com/pytest-dev/pytest

[22] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated Unit Test Generation for Python. *Lecture Notes in Computer Science* (2020), 9–24. https://doi.org/10.1007/978-3-030-59762-7_2

[23] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2009. *Introduction to Information Retrieval.* Cambridge University Press. https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf

[24] Marc Moreno Maza. 2004. *LL(1) Grammars.* Retrieved August 22, 2021 from https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Syntax.html/node14.html

[25] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07).* Association for Computing Machinery, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902

[26] Max Rehkopf. 2021. *User Storys mit Beispielen und Vorlage.* Retrieved August 18, 2021 from https://www.atlassian.com/de/agile/project-management/user-stories

[27] Benno Rice, Richard Jones, and Jens Engel. 2017. *Tutorial.* Retrieved August 18, 2021 from https://behave.readthedocs.io/en/stable/tutorial.html

[28] Mathias Soeken, Robert Wille, and Rolf Drechsler. 2012. Assisted Behavior Driven Development Using Natural Language Processing. In *Objects, Models, Components, Patterns*, Carlo A. Furia and Sebastian Nanz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 269–287.

[29] Thomas Stahl, Markus Völter, Sven Efftinge, and Arno Haase. 2007. *Modellgetriebene Softwareentwicklung.* dpunkt.verlag GmbH.

[30] Tim Storer and Ruxandra Bob. 2019. 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), 228–237.

[31] Niklaus Wirth. 2012. *Grundlagen und Techniken des Compilerbaus 3.A.:.* Oldenbourg Wissenschaftsverlag. https://doi.org/doi:10.1524/9783486719741