Assisted Behavior Driven Development Using Natural Language Processing

Mathias Soeken[1], Robert Wille[1], and Rolf Drechsler[1,2]

[1] Institute of Computer Science, University of Bremen
Group of Computer Architecture, D-28359 Bremen, Germany
{msoeken,rwille,drechsle}@informatik.uni-bremen.de
[2] Cyber-Physical Systems
DFKI GmbH, D-28359 Bremen, Germany
rolf.drechsler@dfki.de

Abstract. In *Behavior Driven Development* (BDD), acceptance tests provide the starting point for the software design flow and serve as a basis for the communication between designers and stakeholders. In this agile software development technique, acceptance tests are written in natural language in order to ensure a common understanding between all members of the project. As a consequence, mapping the sentences to actual source code is the first step of the design flow, which is usually done manually.
However, the scenarios described by the acceptance tests provide enough information in order to *automatize* the extraction of both the structure of the implementation and the test cases. In this work, we propose an assisted flow for BDD where the user enters into a dialog with the computer which suggests code pieces extracted from the sentences. For this purpose, natural language processing techniques are exploited. This allows for a semi-automatic transformation from acceptance tests to source code stubs and thus provides a first step towards an automatization of BDD.

1 Introduction

Historically, software testing has been a post-processing step in the classical waterfall model. After the actual software has been created, usually a team of test engineers writes test cases (e.g. *unit tests*) in order to validate the correctness of the implemented code. In the movement of agile software engineering, the test effort is already incorporated at an earlier point in the development process. In particular, *Test Driven Development* (TDD) [1] employs so-called *acceptance tests* as the starting point for the development process. These acceptance tests represent all scenarios which have to be realized by the final system. While the test cases fail initially before any code has been written, the desired software system is considered complete (*accepted*) if all acceptance tests pass.

It has to be noted that acceptance tests are different from unit tests and are not meant as an alternative. While unit tests check the correct implementation of single atomic components in the software, acceptance tests check a scenario of the system as a whole without considering *how* the system is actually implemented. As a result, it is often summarized that a unit test validates that the software does the *thing* right, whereas an acceptance tests checks whether the software does the *right* thing [2]. Consequently unit tests are generally written by the developers and the stakeholders will not take notice of them, whereas acceptance tests are written by the stakeholders and are discussed with the developers as part of the specification.

Recently, *Behavior Driven Development* (BDD) has been proposed [3] as a result of problems that arose with TDD when applying agile software practices. A major obstacle for programmers has often been to find a good starting point or to determine which facets need to be tested and which not. As a result, it has been noticed that the language used for describing the tests, i.e. class names and operation names, plays an important role both for writing test cases and for finding bugs in case of a failing test. Inspired by [4], for this purpose BDD uses *natural language* as a ubiquitous communication mean to describe the acceptance tests by means of scenarios. In fact, the natural language ensures a common understanding of the system to be developed between all members of the project – particularly between the designers and the stakeholders.

Based on the scenarios which are described by the acceptance tests, the designers map the sentences to actual code by implementing the test cases and code skeletons in the first step. Usually, this is a manual and thus time-consuming and error-prone process. However, all the information that is necessary to perform these steps is in principle already included in the natural description.

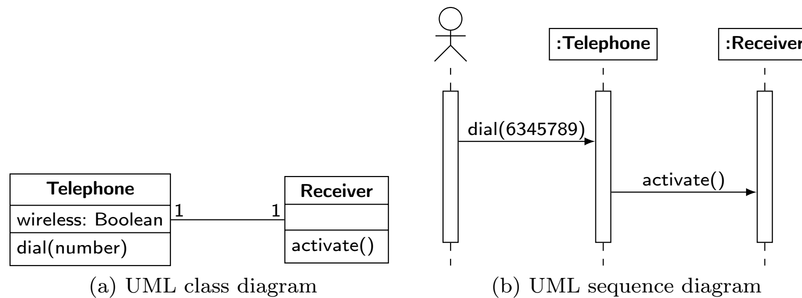
In this work, we propose a methodology which assists the designer in these first steps by semi-automatically extracting design information from the sentences using natural language processing techniques. We propose a design flow where the user enters into a dialog with the computer. In an interactive manner, the program processes sentence by sentence and suggests to create code blocks such as classes, attributes, and operations. The user can then accept or refuse these suggestions. Furthermore, the suggestions by the computer can be revised which leads to a training of the computer program and a better understanding of following sentences.

Using this new design flow, the following advantages arise.

- Having only scenarios described in natural language, the first steps towards writing the overall structure of the whole system can be cumbersome. However, analyzing the scenarios step by step assisted by a computer program allows for a smoother start into the design process.
- Descriptions in natural language bear the risk of misunderstandings, e.g. due to ambiguities. These risks can be minimized when the description is parsed by natural language processing techniques, because what a computer program might misunderstand is also likely to be misunderstood by another designer or stakeholder.
- Unlike previous work (cf. Sect. 6) where the result of the text processing is given after the whole text has been parsed, our approach provides the user with feedback after every sentence being parsed. As a result, the user can retrace the decisions of the tool and intervene if necessary.

For the implementation of the proposed approach, we enhance the *Cucumber* tool [2]. The sentences of each scenario are parsed and are transformed into actual code required for the subsequent implementation of the system. Furthermore, also user interactions are written in natural language as pre-defined background scenarios inside the *Cucumber* tool, which lead to a seamless user experience.

We have evaluated our approach in a case study where we have used a candy machine whose specification is provided by means of six use case scenarios in natural language. Using only a few user interactions, it is possible to generate the whole class diagram and test cases with the assistance of the computer. As a consequence, the proposed flow allows for a semi-automatic transformation from acceptance tests to source code stubs and thus provides a first step towards an automatization of BDD.



(a) UML class diagram (b) UML sequence diagram

Fig. 1. UML class and sequence diagram

The paper is structured as follows. The necessary background of methodologies used in this work is provided in the next section. Section 3 illustrates the general idea while Sect. 4 gives a more detailed insight into the extraction techniques. The results of the case study are presented in Sect. 5. Furthermore, Sect. 6 discusses related work and conclusions are drawn in Sect. 7.

2 Preliminaries

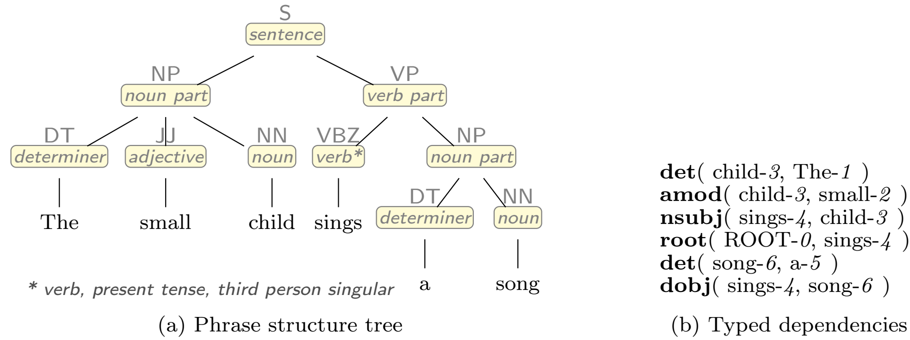
In this work, the *Unified Modeling Language* (UML) is applied to represent the code skeletons and test cases which are semi-automatically derived from natural language. Besides that, we are also exploiting language processing tools. To keep the paper self-contained, the underlying concepts of UML and the applied tools are briefly reviewed in the following.

2.1 Unified Modeling Language

In this section, we briefly review the basic UML concepts which are considered in this work. A detailed overview of the UML is provided in [5].

Class Diagrams. A UML *class diagram* is used to represent the structure of a system. The main component of a class diagram is a *class* that describes an atomic entity of the model. A class itself consists of *attributes* and *operations*. Attributes describe the information which is stored by the class (e.g. member variables). Operations define possible actions that can be executed e.g. in order to modify the values of attributes. Classes can be set into relation via *associations*. The type of a relation is expressed by *multiplicities* that are added to each association-end.

Example 1. Figure 1(a) shows a UML class diagram specifying a simple telephone. The class diagram consists of the two classes *Telephone* and *Receiver*. The class *Telephone* has an attribute *wireless* of type *Boolean*. The receiver is related to the telephone which is expressed by an association. As expressed by the multiplicities, each telephone has one receiver and vice versa. Both classes have an operation, i.e. the telephone can dial a number and the receiver can be activated.



(a) Phrase structure tree (b) Typed dependencies

Fig. 2. Application of the Stanford Parser

Sequence Diagrams. The dynamic flow caused by operation calls can be visualized by sequence diagrams. Sequence diagrams offer the possibility to represent particular scenarios based on the model provided by the class diagram. Hence, several sequence diagrams exist for a given class diagram. In the sequence diagram, instances of the classes, i.e. objects, are extended by *life lines* that express the time of creation and destruction in the scenario. Arrows indicate operations that are called on an object, and are drawn from the caller to the callee. Besides objects also actors from the outside environment can be part of the sequence diagram.

Example 2. A sequence diagram is depicted in Fig. 1(b). In that scenario, first a number is dialed from an actor in the outside environment, before the telephone activates the receiver.

In this work, class diagrams and sequence diagrams are applied to represent the semi-automatically determined code skeletons and test cases, respectively.

2.2 Stanford Parser

The Stanford Parser is an open source software compilation published by the Stanford Natural Language Processing (NLP) Group [6]. It parses sentences in different languages and returns a *phrase structure tree* (PST) representing the semantic structure of the sentence. A PST is an acyclic tree with one root vertex representing a given sentence. Non-terminal and terminal vertices (i.e. leafs) represent the grammatical structure and the atomic words of this sentence, respectively. A simple PST for the sentence "The small child sings a song" is given by means of Fig. 2(a). As can be seen all leafs are connected to distinct vertices that classify the *tag* of the respective word, e.g. nouns and verbs. These word tags are further grouped and connected by other vertices labeled with a tag classifying a part of the sentence, e.g. as noun parts or verb parts. The classifier tags are abbreviated in the PST, however, in Fig. 2(a) the full classifier is annotated to the vertices. For details on how a PST is extracted from a sentence, the reader is referred to [7].

Besides the PST, the Stanford Parser also provides *typed dependencies* [8] which are very helpful in natural language processing. Typed dependencies are tuples which describe the semantic correlation between words in the sentence. Figure 2(b) lists all typed dependencies for the sentence considered in Fig. 2(a). For example, the nouns are assigned their articles using the **det** relation. Note that the numbers after the word refer to the position in the text, which is necessary if a word occurs more than once

in a sentence. Two further important relations are **nsubj** and **dobj** that allow for the extraction of the typical *subject-verb-object* form. In this case it connects the verb *sings* with both its subject and object.

In this work, the Stanford Parser is applied to process the structure of the sentences describing a scenario.

2.3 WordNet

WordNet [9], developed by the Princeton University, is a large lexical database of English that is designed for use under program control. It groups nouns, verbs, adjectives, and adverbs into sets of cognitive synonyms, each representing a lexicalized concept. Each word in the database can have several *senses* that describe different meanings of the word. In total, WordNet consists of more than 90,000 different word senses, and more than 166,000 pairs that connect different senses with a semantic meaning.

Further, each sense is assigned a small description text which makes the precise meaning of the word in that context obvious. Frequency counts provide an indication of how often the word is used in common practice. The database does not only distinguish between the word forms noun, verb, adjective, and adverb, but further categorizes each word into sub-domains. Those categories are e.g. *artifact*, *person*, or *quantity* for a noun.

In this work, WordNet is applied to determine the semantics of the sentences describing a scenario.

3 General Idea and Proposed Approach

As outlined in Sect. 1, behavior driven development puts acceptance tests to be realized in the focus of the design flow. These acceptance tests are provided as scenarios written in natural language. The typical BDD design process, as it is applied today, involves the steps illustrated by means of Fig. 3:

1. Write a *scenario* describing a certain behavior in natural language.
2. Write a *step definition* for each sentence (i.e. for each step) in the scenario which connects the natural language to actual code. Since the sentences in a scenario are written in natural language, they have to be implemented as code by the designer. For this purpose, step definitions are written that consist of a regular expression and a block of code. Whenever a step matches a regular expression, the respective code block of that step definition is executed.
3. Write a *code skeleton* such that the code inside the step definition is compilable.
4. *Implement* the operations in the code skeleton such that the scenario passes.

Example 3. Figure 4 shows an example of the BDD flow as it is employed in the *Cucumber* tool [2]. Here, one scenario is provided and eventually implemented in Ruby [10]. An example of a scenario is given in Fig. 4(a) describing the process of initiating a telephone call. For the first sentence, a step definition is created using Ruby as depicted in Fig. 4(b). Inside the step definition code, it is written what should be executed when the step is processed by the *Cucumber* tool. However, the class *Telephone* as well as the operation *pickUp* do not exist yet. Thus, a code skeleton is manually generated

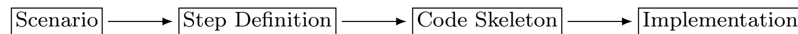


Fig. 3. Behavior Driven Development flow

Scenario: *Placing a call*

* Ada picks up the receiver from the telephone
* She dials the number 6-345-789
* The telephone places a call

(a) Scenario

```
Given /^Ada picks up the receiver from the telephone$/ do
  @telephone = Telephone.new
  @receiver = @telephone.pickUp
end
```

(b) Step Definition

```
class Telephone
  def pickUp
  end
end

class Receiver
end
```

(c) Code Skeleton

```
class Telephone
  attr_reader :receiver

  def initialize
    @receiver = Receiver.new
  end

  def pickUp
    @receiver
  end
end
```

(d) Implementation

Fig. 4. BDD example

in the next step as shown in Fig. 4(c). Then, the step definition compiles. Finally, as illustrated in Fig. 4(d), the code skeleton is implemented in the last step. After this procedure has been applied to all remaining sentences, the whole scenario passes representing a complete implementation of this scenario.

So far, all steps are performed manually. Obviously, the scenario is the starting point for the BDD flow and thus always needs to be created manually. However, when observing the design flow as depicted in Fig. 3, the following conclusion can be drawn. The creation of the step definition and the provision of a code skeleton can in fact be automatized, since the sentences given in the scenario often provide enough information for an automatic determination of these components. For example:

– Regular nouns in sentences usually are realized as objects in the system, and therefore, they can automatically be represented by classes.
– Proper nouns usually represent actors from the outside environment who interact with the system.
– Adjectives in sentences usually provide further information about the respective objects. Thus, they can automatically be represented by attributes of classes.
– Verbs in sentences usually describe actions in a scenario, and can therefore automatically be represented by operations of classes. Additionally, they provide information when an operation is called and by whom.

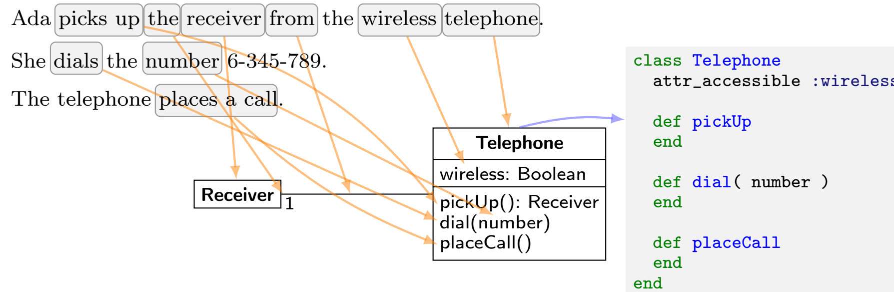


Fig. 5. Extracting class diagrams from scenarios for the generation of code skeletons

In this work, we propose a BDD methodology which exploits such information in order to semi-automatically generate step definitions and code skeletons from scenarios given in natural language. For this purpose, we are making use of UML class diagrams and UML sequence diagrams that are proper abstractions of code skeletons and step definitions, respectively, from which the required pieces for the BDD flow can easily be generated. In the following, the general idea is briefly illustrated in Figs. 5 and 6 by means of the telephone scenario given in Fig. 4.

First the creation of a class diagram, i.e. a code skeleton, is considered. Using only the first sentence in Fig. 5 for example, the following information can automatically be extracted:

- The sentence contains three nouns. Since Ada is a proper noun, it is treated as an actor and not as a component of the system. Accordingly, classes are only created for receiver and telephone.
- The adjective wireless can be identified as related to telephone and thus is extracted as attribute for the respective class.
- The verb pick up is specified to be an operation of telephone.
- The preposition from indicates a relationship between the receiver and the telephone. Since the sentence states "*the* receiver from the wireless telephone" it can be concluded that a telephone can only have one receiver, in contrast to "*a* receiver from the wireless telephone" which would indicate more than one receiver.

Further information can be determined from the remaining two sentences. The fragments dials and places a call indicate further operations of the telephone. The phone number after number can be detected as a parameter for the *dial* operation. This eventually leads to the class diagram shown in Fig. 5, which can be used to generate the code skeleton in the desired language.

Moreover, the order of the sentences and their actions described in it provide the basis to automatically determine a test case. This is done by automatically creating a step definition for each sentence in the scenario. From the first sentence, it is known that Ada, i.e. an actor, invokes the pick up operation. The noun She in the second sentence refers again to Ada. Thus it can be concluded that the same actor invokes dial with the parameter 6-345-789 in the next step. The last sentence states that at the end of this scenario the telephone invokes the operation place a call. All steps can be summarized in a sequence diagram which can be used to generate step definitions in the desired language as depicted in Fig. 6.

As illustrated by this example, step definitions and code skeletons can automatically be generated even if the scenario is provided in natural language. We are aware

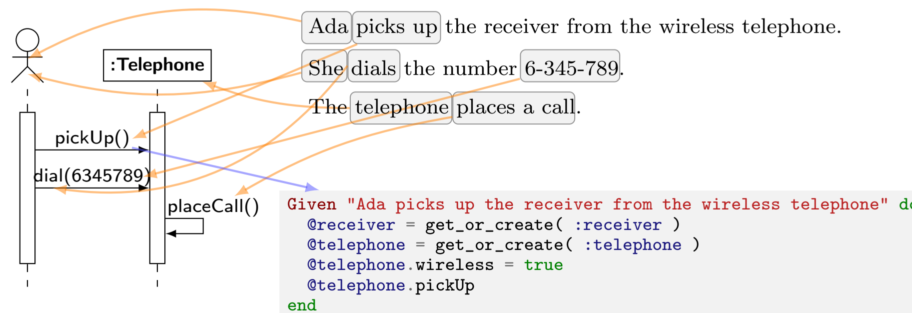


Fig. 6. Extracting sequence diagrams from scenarios for the generation of step definitions

that sentences in natural language might be ambiguous or incomplete and thus a fully automatic determination would not always lead to the desired result. However, as discussed in detail in the next section, the application of today's language processing tools in combination with ontologies already shows very good results. In addition, we propose an interactive flow where the designer enters into a dialog with the computer. In this flow, the computer is guiding the designer through the scenario while creating the UML diagrams step by step. During this process, the designer can refuse the automatically generated structures or provide the program with further information which cannot be extracted from the sentence or the ontology. In some cases this can even lead to a rephrasing of sentences in the scenario, e.g. in the presence of ambiguities. Then, the proposed approach also advances the design understanding within the development. If a sentence is misunderstood by the computer program, the same may also apply to other designers.

Overall, an approach is presented which significantly increases the efficiency of behavior driven development. As illustrated in Fig. 7, instead of manually creating step definitions and code skeletons, automatically generated suggestions from the proposed method just have to be revised or confirmed. The generated code skeleton is then used as the basis for the implementation, which remains the only non-automatic step. A further advantage of the new flow is that this implementation can immediately be validated against the scenario as also the step definitions have been generated automatically.

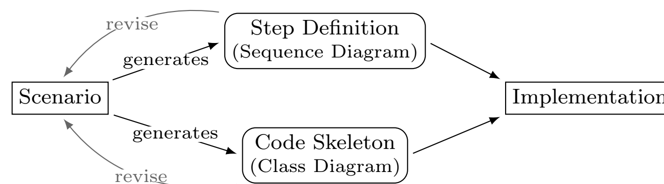


Fig. 7. Proposed flow

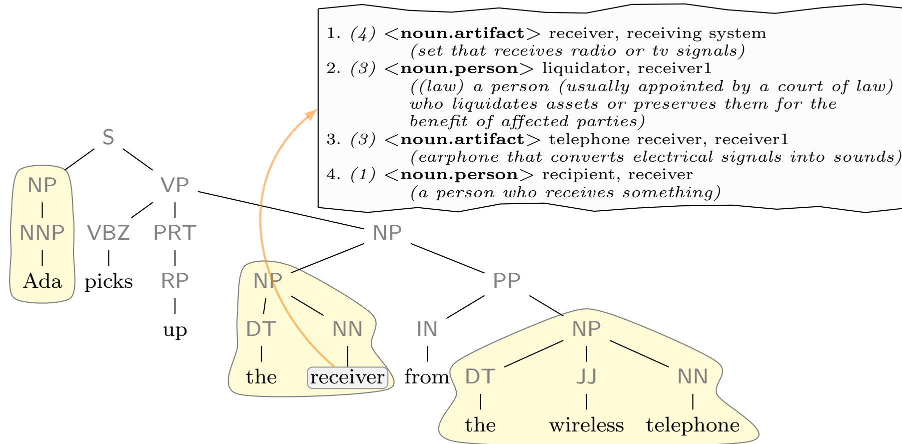


Fig. 8. Extraction of nouns using a PST and the WordNet dictionary

4 Semi-automatic Extraction of Information

As described above, the core of the proposed approach is the semi-automatic determination of UML class and sequence diagrams from a given scenario in natural language. Language processing tools and ontologies are exploited for this purpose. This section provides details on how the required information is extracted from the given scenarios. The determination of classes, attributes, and operations as well as their arrangement in a UML class diagram is initially described. Afterwards, the extraction of the actors of the system and the order of their actions are described which lead to the desired sequence diagram.

4.1 Classes

Components of the system to be implemented will be represented as classes in a UML class diagram. In order to extract these classes, the sentences provided by the scenario are parsed by the Stanford Parser reviewed in Sect. 2.2. This leads to the PST from which parts of the sentence related to a noun are extracted. Initially all *noun parts* (labeled NP in the PST) are extracted, i.e. nouns together with possible adjectives and articles. Afterwards, they are subdivided into *proper nouns* (labeled NNP in the PST) and *nouns* (labeled NN in the PST). Proper nouns are ignored – they represent actors and are required later in order to create the sequence diagram. All remaining nouns are further considered by the WordNet dictionary (cf. Sect. 2.3) in order to check whether they represent further actors of the system or actual components for which classes have to be created.

Example 4. Figure 8 shows the PST for the sentence "Ada picks up the receiver from the wireless telephone". This sentence is composed of three noun parts from which Ada is discarded since it is a proper noun. For the remaining two nouns, a further check is performed using WordNet. In case of receiver, this exemplary leads to the *lexical file information* as given in the page excerpt in the upper part of Fig. 8. The four entries are ordered by their *frequency counts* providing an indication about the commonly used semantic of this word. As can be seen, receiver might been used as a

Background:

* Consider coin.

Scenario:

* Bob picks up a coin.

(a) Example scenario with background

Phrase	Meaning
Consider *noun*.	Considers *noun* as a class
noun is a person.	Considers *noun* as an actor
Ignore *noun*.	Does not consider *noun* neither as class nor as an actor

(b) Possible phrases for interaction

Fig. 9. User interaction for nouns

person (denoted by *noun.person*) which would imply an actor of the system and no creation of a class. However, since its use as an object (denoted by *noun.artifact*) has a higher frequency count, this semantic is chosen, i.e. receiver is considered a component and thus a respective class is created. The same check is applied to telephone.

User Interaction. Although the automatic classification of nouns and thus the creation of classes works very well for many cases, two problems may occur: (1) the frequency counts of WordNet lead to a wrong decision or (2) the considered word cannot be classified using WordNet. In both cases, the user has to intervene.

A trivial approach is a simple modification of the resulting class diagram by the user, e.g. the removal of a class when it was wrongly interpreted as component. Besides that, also an interactive learning scheme can be applied. The latter has been seamlessly implemented into the *Cucumber* BDD flow [2] where scenarios are usually grouped as *features*. Each feature additionally can be enriched by a *background* section which is processed prior to each scenario. We use *pre-defined* background steps for providing additional information (again in natural language) that help the automatic approach to correctly retrieve the meaning of a word or to assume a context of a scenario. The following example illustrates the principle.

Example 5. Consider the sentence "Bob picks up a coin". The word coin is specified as a *noun.possession* by WordNet and therefore cannot be classified as class or actor. Thus in order to set the context the designer can additionally provide more background as shown in the *Cucumber* feature illustrated in Fig. 9(a). Due to the phrase "Consider coin", the background of coin is clearly set to a component, i.e. a class is generated for it.

Other background phrases that can be used are given in Fig. 9(b). By applying a phrase, the approach automatically learns additional information which can later be applied in other scenarios as well.

4.2 Attributes

The determination of the noun parts as illustrated in Fig. 8 does not only enable the extraction of classes, but also of their corresponding attributes. For this purpose, all vertices representing adjectives are extracted (labeled JJ in the PST) and are simply connected to the corresponding class. By default Boolean attributes are assumed. If the adjective additionally is prefixed by adverbs such as *very*, *slightly*, or *almost*, an integer attribute is assumed instead of the Boolean type. These cases are explicitly emphasized by the proposed approach. The designer may transform this classification later, e.g. to an enumerated type.

Example 6. Consider the noun part the wireless telephone in the example from Fig. 8. From the corresponding noun, a class with the name *Telephone* is extracted. Additionally, the class is enriched by a Boolean attribute *wireless* due to the adjective.

Further attributes can be extracted from other constructs of the sentence. As an example, consider the phrase "the product 12". The word 12 is classified as *cardinal number* (labeled CD in the PST). If this appears after a noun in a noun part, it is implied that the respective class has an attribute *id* of type integer. This also works with floating point numbers. A similar rule applies for sentence parts set in quotes. For example, consider the phrase "the song "Wonderful Tonight"". In this case it does not make sense to treat the words in quotes as normal words – wonderful and tonight should clearly not be considered as adjective and noun, respectively. Instead the whole quote is extracted from the sentence before parsing and it is stored that the word song can have an additional identifier. This finally leads to an attribute *name* of type string.

User Interaction. All information that is automatically extracted from the sentences in the scenarios as described above can also be provided explicitly by pre-defined sentences in the background section. The sentence "A *noun* can be *adjective*" adds an attribute *adjective* of type Boolean to the class representing *noun*. In a similar manner, for the sentence "A *noun* has an id" or "A *noun* has a name", attributes *id* of type integer or *name* of type string, respectively, are added to the class. In contrast, the consideration of certain attributes can be omitted by the sentences "A *noun* cannot be *adjective*", "A *noun* has no id", and "A *noun* has no name".

Further, enumeration types can be added to a class explicitly by the sentence "The *name* of a *noun* can be *value*, ..., or *value*". In this case, an attribute *name* is added to the class representing *noun* providing an enumeration for each value.

Example 7. Consider again the sentence "Ada picks up the receiver from the wireless telephone". By default the tool extracts an attribute *wireless* of type Boolean. When adding the sentence "The type of a telephone is wireless or wired" to the background section, an enumeration named *type* is added as an attribute to the class *Telephone* having the two values *wired* and *wireless*. In this case, the adjective wireless is not extracted for the class *Telephone* as it already appears as a value in the enumeration.

4.3 Operations

As outlined in Sect. 3, verbs are usually a good indicator of an operation to be extracted from a sentence. However, in order to assign a verb and, therefore, an operation to the corresponding class, the PST alone is not sufficient. For example, in the sentence "Ada picks up the receiver from the wireless telephone" it is not obvious whether the operation pick up belongs to the receiver or the telephone. As a solution, we additionally make use of the typed dependencies in the sentence (cf. Sect. 2.2). This allows for relating the verb to its subject and object in the typical *subject-verb-object* (SVO) phrase. Thus, the first step consists of extracting the SVO relation in the sentence. In a next step, it is determined whether the subject or the object in the sentence is the class to which the operation should be assigned. If one of the nouns (subject or object) has been identified as an actor and the other one as a class, then this decision is easy. In other cases, user interaction might be required. The following example illustrates the principle.

```
1 nsubj( picks-2, Ada-1 )
2 root( ROOT-0, picks-2 )
3 prt( picks-2, up-3 )
4 det( receiver-5, the-4 )
5 dobj( picks-2, receiver-5 )
6 det( telephone-9, the-7 )
7 amod( telephone-9, wireless-8 )
8 prep_from( receiver-5, telephone-9 )
```

Fig. 10. Typed dependencies for sentence from Fig. 8

Example 8. Consider again the sentence in Fig. 8. The verb `picks` in the sentence is easily identified in the PST by searching for a vertex labeled VBZ (verb, present tense). The typed dependencies for the same sentence are given by means of Fig. 10. With the relations **nsubj** (Line 1, nominal subject) and **dobj** (Line 5, direct object), the SVO relation *Ada-picks-receiver* can be determined. The relation **prt** (Line 3, phrasal verb particle) allows for completing the verb to *picks up* which results in the operation name *pickUp*. Note that the base form of the verb *picks* can be identified using a WordNet query. Using this information, the operation *pickUp* is added to the class *Receiver*, since *Ada* is already classified as an actor.

However, as already depicted in Fig. 5, this is not the right decision. It makes more sense that *pickUp* is an operation of the telephone that returns a receiver. The information for taking this decision is, however, already included in the typed dependencies. This is due to the preposition on the word `receiver`, which is indicated by the relation **prep_from** (Line 8, prepositional modifier). Further, this relation returns the correct link to the word `telephone`. Since *Telephone* also has been classified as a class, the operation *pickUp* is added to this class. Further, due to the preposition, *Receiver* can be identified as return type for that operation.

User Interaction. If operations should not be generated for a class, the user can write the sentence "A *noun* does not *verb*". Note that the noun should be the noun representing the class name the operation is assigned. If for example the operation *pickUp* should not be added to the class *Telephone*, the user would add "A telephone does not pick up" as a step to the background section. This seems inconvenient at first glance, since Ada is picking up the receiver in the sentence. However, those decisions are usually taken after seeing the result of the automatic translation.

4.4 Generation of Step Definitions

After the class diagram has been created, the same actors and classes with their attributes and operations can be used to generate the step definitions. While the scenarios serve as the outline in which the steps are executed, the step definitions describe the actual code that has to be executed in that step.

For this purpose, consider again the sentence "Ada picks up the receiver from the wireless telephone". The automatically generated step definition is illustrated in Fig. 6. First, for each class extracted from the sentence a respective object is created. This is done by making use of the factory design pattern [11]. In particular, two cases may occur, i.e. either a new object has to be created or the step refers to a possibly already existing object. The functions *create* and *get_or_create* are made available for these two cases. The article of the noun can be used to determine which function to choose, which is described in detail in the remainder of the section. Besides that, attribute values are assigned and operations are transformed into respective operation calls on the objects.

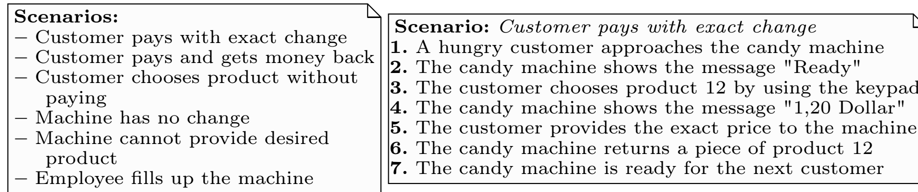


Fig. 11. Use case scenarios for a candy machine

The Role of Articles in Test Cases. The article in front of a noun can be used to determine whether a new object has to be created or whether the noun references a possibly existing object. This is illustrated by the following example.

Example 9. Consider the following two sentences.

"A telephone starts ringing. A telephone stops ringing"

This scenario indicates that possibly two telephones are involved, one that starts ringing and one that stops ringing. If, however, the scenario was specified as

"A telephone starts ringing. *The* telephone stops ringing",

it is obvious that the telephone in the second sentence is the same one as in the first sentence.

In the generation of test cases, we apply the following rule. For nouns with an undetermined article, always a new instance is generated, i.e. the factory function *create* is used. Otherwise, for nouns with an determined article, it is first tried to find the latest created instance for the respective type. If this is not possible, an instance is generated. This behavior is implemented in the factory function *get_ or_ create*.

A similar effect is noticeable when names are used for the actors. Then, the nouns *She* or *He* have to be assigned accordingly.

Example 10. Consider the following scenario.

"Ada and Bob play soccer. She is the goal keeper. He shoots the ball."

In this scenario, *She* refers to Ada, and *He* refers to Bob.

To automatically determine the relation of words such as *She*, *He* and also *her* or *his*, the first names of the actors have to be assigned a gender. WordNet is not capable of doing this. However, probably other dictionaries suitable for that purpose can be used for this problem. In the meanwhile, we make use of user interaction in form of background steps such as "Ada is a woman" or "Bob is a man".

5 Case Study

We implemented the approach on top of the *Cucumber* tool [2] in Ruby and applied it to semi-automatically design a simple candy machine specified by six acceptance tests. These acceptance tests, provided by means of scenarios, are summarized in the left-hand side of Fig. 11. Due to page limitations, we cannot provide and discuss all scenarios in detail. As a consequence, we demonstrate the usage of the proposed approach for the first scenario (provided in the right-hand side of Fig. 11) only. For this purpose, the

output of the approach as well as the resulting parts for the class diagram are sketched for each sentence in the following. The overall class diagram is created as union of all parts.

A hungry customer approaches the candy machine.
Customer has been detected as actor. Detected class *CandyMachine* without attributes. Detected operation *approach* for class *CandyMachine*.

CandyMachine
approach()

The first sentence was correctly processed by the approach, i.e. customer was correctly identified as an actor and candy machine as a component leading to the creation of a class. The two words candy and machine have correctly been identified as compound noun, since both belong to the same noun part (NN) in the PST. Further, the automatic approach has originally created an operation *approach* for the corresponding verb in the sentence. However, in the role of the designer, we decided not to use this operation in our class. For this purpose, a sentence "A candy machine does not approach" has been added to the background section in the feature description for the *Cucumber* tool.

The candy machine shows the message "Ready".
Detected class *CandyMachine* without attributes. I do not know how to categorize *message* as actor or class. I know it as *communication*. Do you mean message as in a communication (usually brief) that is written or spoken or signaled; "he sent a three-word message"?

In the second sentence, the approach was not able to determine whether or not a class should be created for the noun message. It is neither classified as *person* nor *artifact* in the WordNet database, but as *communication*. The approach informs the user about that and also prints out the corresponding WordNet information. Based on that, the user can take a decision. In the considered case, a class should be created for message. This is achieved by adding the sentence "Consider message" to the background section. Processing the sentences again including this additional information leads to the following result:

The candy machine shows the message "Ready".
Detected class *CandyMachine* without attributes. Detected class *Message* with attribute *name*. Detected operation *show* with parameter *message* for class *CandyMachine*.

CandyMachine
show(m: Message)

Message
name: String

Now, everything has been detected correctly. The class *Message* automatically gets the attribute *name* because of the identifier "Ready" in the sentence.

The customer chooses product 12 by using the keypad.
Customer has been detected as actor. Detected class *Product* with attribute *id*. Detected class *Keypad* without attributes. Detected operation *choose* with parameter *product* for class *Keypad*.

KeyPad
choose(p: Product)

Product
id: Integer

In the third sentence, the operation *choose* is added to the class *KeyPad* because of the preposition in the sentence. Since the object in the sentence is product, the operation gets a respective parameter. An attribute *id* is added to the class *Product* due to the number after the noun.

The fourth sentence "The candy machine shows the message "1,20 Dollar"" is equivalent to the second sentence in the scenario when considering structure extraction. In fact, the sentences will even generate the same step definition, since the Cucumber tool automatically extracts regular expressions for words in quotes.

In the fifth sentence, the user has to manually interact since the word price cannot be classified precisely. After it has been added as considered to the background section, the tool proceeds as follows:

The customer provides the exact price to the candy machine.
Customer has been detected as actor. Detected class *Price* with attributes *exact*. Detected class *CandyMachine* without attributes. Detected operation *provide* with parameter *price* for class *CandyMachine*.

CandyMachine
provide(p: Price)

Price
exact: Boolean

Since the adjective exact is associated to the noun price, it appears as an attribute for the class *Price*.

In a similar fashion, the remaining sentences and scenarios have been processed. Eventually, this led to a class diagram that consists of 6 classes with 3 attributes and 7 operations. In total, 9 sentences were added to the background section for a total of 40 sentences in 6 scenarios. Analogously, 18 step definitions have been created which cover all sentences in all scenarios and allow the execution of the acceptance tests.

Overall, step definitions and code skeletons of a complete system, the considered candy machine, have been semi-automatically generated by the proposed approach. For this purpose, each sentence was iteratively processed. In case of uncertainties, the user entered into a dialog with the computer. Compared to an entirely manual flow, this represents a significant improvement considering that the designer is automatically served with several options which she/he can easily refine.

6 Related Work

The proposed approach is a significant step towards an automatization of BDD. In doing so, our solution aligns with other approaches aiming at that goal – in particular in the domain of UML. As an example, the work presented in [12] extracts UML class diagram from specifications in natural language which, afterwards are used to generate code skeletons. The authors make use of a structure similar to the PST, but not of typed dependencies and not of a lexical database for classification. Further, the input language must be written in simple English and follow a certain sentence structure. User interaction is not intended in this approach.

The tool named *REBUILDER UML* [13] uses natural language as constituents for object oriented data modelling by using an approach based on case-based reasoning. However, only class diagrams are supported by this operation. Similarly, the tool *LOLITA* [14] generates an object model from a text in natural language. However, the tool only identifies objects from text and cannot further distinguish between other elements such as classes, attributes, and operations.

Class diagrams can also be extracted from natural language text using the tool *CM-BUILDER* [15]. Also, here dynamic aspects are not considered. Furthermore, the specification is considered as a whole which impedes user interaction and the result is always the complete class diagram such that subsequent modifications are cumbersome.

In [16] a method for generating executable test benches from a textual requirements specification is proposed. For this purpose, a subset of the English language called *textual normal form* has been designed that can be transformed into UML class diagrams which can be translated into classification trees according the *Classification Tree Method for Embedded Systems* (CTM/ES) [17]. These classification trees are finally used to generate the resulting executable test benches which can be utilized in a formal verification environment. However, besides that a different domain is addressed in this approach, the designer is limited to a restricted subset of the English language.

7 Conclusions

In this work, we proposed an assisted flow for BDD where the user enters into a dialog with the computer in order to semi-automatically generate step definitions and code-skeletons from a given scenario. For this purpose, natural language processing tools are exploited. A case study illustrated the application. Instead of going through the established BDD steps manually, the designer is automatically served with options which easily can be refined.

The proposed approach is a significant step towards an automatization of BDD. Moreover, while the case study focuses on acceptance test within the BDD scheme, the results of the proposed approach also motivate a consideration of general natural language system specifications. The proposed methodology provides a basis for further work in this direction.

Acknowledgments. This work was supported by the German Research Foundation (DFG) (DR 287/23-1).

References

1. Beck, K.: Test Driven Development. By Example. Addison-Wesley Longman, Amsterdam (November 2003)
2. Wynne, M., Hellesøy, A.: The Cucumber Book: Behaviour-Driven Development for Testers and Developers. The Pragmatic Bookshelf (January 2012)
3. North, D.: Behavior Modification: The evolution of behavior-driven development. Better Software **8**(3) (March 2006)
4. Evans, E.J.: Domain-Driven-Design: Tackling Complexity in the Heart of Software. Addison-Wesley Longman, Amsterdam (August 2003)
5. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language reference manual. Addison-Wesley Longman, Essex, UK (January 1999)
6. Jurafsky, D., Martin, J.H.: Speech and Language Processing. Pearson Prentice Hall (2008)
7. Klein, D., Manning, C.D.: Accurate Unlexicalized Parsing. In: Annual Meeting of the Association for Computational Linguistics. (July 2003) 423–430
8. de Marneffe, M.C., MacCartney, B., Manning, C.D.: Generating Typed Dependency Parses from Phrase Structure Parses. In: Int'l Conf. on Language Ressources and Evaluation. (May 2006) 449–454
9. Miller, G.A.: WordNet: A Lexical Database for English. Communications of the ACM **38**(11) (November 1995) 39–41
10. Flanagan, D., Matsumoto, Y.: The Ruby Programming Language. O'Reilly Media (January 2008)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Amsterdam (November 1994)
12. Bajwa, I.S., Samad, A., Mumtaz, S.: Object Oriented Software Modeling Using NLP Based Knowledge Extraction. European Journal of Scientific Research **35**(1) (January 2009)
13. Oliviera, A., Seco, N., Gomes, P.: A CBR Approach to Text to Class Diagram Translation. In: TCBR Workshop at the European Conf. on Case-Based Reasoning. (September 2006)
14. Mich, L., Garigliano, R.: A Linguistic Approach to the Development of Object Oriented Systems using the NL System LOLITA. In: Int'l Symp. on Object-Oriented Methodologies and Systems. Volume 858 of Lecture Notes in Computer Science., Springer (September 1994) 371–386
15. Harmain, H.M., Gaizauskas, R.J.: CM-Builder: A Natural Language-Based CASE Tool for Object-Oriented Analysis. Journal of Automated Software Engineering **10**(2) (April 2003) 157–181
16. Müller, W., Bol, A., Krupp, A., Lundkvist, O.: Generation of Executable Testbenches from Natural Language Requirement Specifications for Embedded Real-Time Systems. In: Int'l Conf. on Distributed, Parallel and Biologically Inspired Systems. (September 2010) 78–89
17. Grochtmann, M., Grimm, K.: Classification trees for partition testing. Software Testing, Verification and Reliability **3**(2) (June 1993) 63–82