Employing Issues and Commits for In-Code Sentence Based Use Case Identification and Remodularization

Peter Berta   Michal Bystrický   Michal Krempaský   Valentino Vranić
Institute of Informatics, Information Systems and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 2, Bratislava, Slovakia
pepoberta@gmail.com, {michal.bystricky,vranic}@stuba.sk, kfckk@centrum.sk

Abstract

Use case driven modularization improves code comprehension and maintenance and provides another view on software alongside object-oriented modularization. However, approaches enabling use case driven modularization require to modularize code manually. In this paper, we propose an approach to employing issues and commits for in-code sentence based use case identification and remodularization. The approach aims at providing use case based perspective on the existing code. The sentences of use case steps are compared to sentences of issue descriptions, while the sentences generated from the source code of issue commits are compared to sentences generated from the corresponding methods in source code in order to quantify the similarity between use case steps and methods in source code using different similarity calculation algorithms. The resulting level of similarity is used to remodularize source code according to use cases. We conducted a study on the OpenCart open source e-shop employing 16 use cases. The approach achieved the recall of 3.37% and precision of 75%. The success of the approach strongly depends on issues and commits assigned to them. The results would be better especially for the code that natively employs use case driven modularization.

CCS Concepts • **Software and its engineering** → **Abstraction, modeling and modularity**; **Software reverse engineering**; **Software version control**; *Maintaining software*

Keywords use case, traceability links, information retrieval, natural language processing, text similarity, intent, modularization, remodularization, DCI, aspect-oriented programming

1. Introduction

The comprehension of what systems actually do is better in code organized according to concerns of use cases than in common object-oriented modularization, as is implied by multiple studies [3, 4, 9, 21]. Also, responding to changes is improved in case of specific change types [7]. However, code has to be modularized according to use cases manually. Unfortunately, automatic modularization would require to identify use cases in source code.

ECBS '17, August 31-September 1, 2017, Larnaca, Cyprus

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4843-0/17/08…$15.00

DOI: https://doi.org/10.1145/3123779.3123792

Since object-oriented code can be transformed into natural language [11, 15] or other way around, e.g., use cases can be transformed into object-oriented code using RSL (Requirements Specification Language) [30], recognizing which object-oriented code belongs to which use case can be achieved by employing syntactic, lexical, and semantic text similarity approaches. For example, Jaccard, Cosine, and Dice are common approaches to detect lexical similarity at the sentence level [24]. But what is crucial for detecting the context is the semantic similarity. By employing thesaurus, such as is in WordNet [16], it is possible to detect semantically similar words, i.e., synonyms, words with broader meaning so called hypernyms, and with more specific meaning so called hyponyms. This is implemented in tools such as Natural Language Toolkit [1].

Similarity between software artifacts was heavily studied before in approaches recovering traceability links, which employ information retrieval methods. For example, documentation to code traceability identification using latent semantic indexing [14], corpus [8], ontologies [27], or Wikipedia [13]. However, these approaches address software artifacts or documents similarity in general, but they do not deal with similarity between use cases and source code explicitly. There are many use case notations out of which most famous are Cockburn's [6] and Jacobson's [9]. Programming styles also vary significantly, even within the same programming language, with differences in naming conventions, modularity, commenting, etc.

There are also approaches aiming explicitly at traceability links between use cases and source code. But they do not offer automatic detection of traceability links from use cases and source code, but require external influence, such as human involvement [12, 28] or monitoring developers [18]. Automatic identification of traceability links is concurrent research problem.

Fortunately, issue tracking systems provide a solid knowledge base for use case identification in code. For example, domain specific concerns are present in pull request discussions [20], but they are observable in code, too. The relation between pull requests and corresponding code could be employed to recognize the similarity between use cases and code in order to overcome the difference in the abstraction level of use cases and source code.

In this paper, we propose an approach to employing issues and commits for in-code sentence based use case identification and remodularization. The rest of the paper is organized as follows. Section 2 explains the textual similarity between use cases and code on an example. Section 3 describes how the difference in the abstraction level of use cases and source code can be overcome. Section 4 proposes the methods for the in-code sentence based use case identification. Section 5 explains the use case based code remodularization. Section 6 presents the evaluation results. Section 7 discusses the results. Section 8 compares our approach to existing approaches. Section 9 concludes the paper and presents some challenges.

2. Textual Similarity Between Use Cases and Source Code

There is a significant similarity between use cases and source code at the text level. Specific words from use cases often appear in their implementation, e.g., in identifiers, string literals, file, and folder names. We will explore this on examples taken from our study of open source e-shop OpenCart.[1]

Consider the *Order Gift Certificate* use case:

```
1. The customer selects to purchase a gift
   certificate
2. The system prompts for name, email,
   recipient's name and email, gift certificate
   theme, and message
3. The system requires to confirm gift
   certificates are non-refundable
4. The customer enters the name, email, theme,
   message and confirm that gift certificates
   are non-refundable
5. Include "Show Cart"
6. Include "Place Order"
7. The system sends an e-mail with details
   how to redeem gift certificate
```

In step 2, the system prompts for information required to proceed with the order, i.e., name, email, recipient's information, theme, and message. This information can be found in implementation, too. Consider the following form:

```
<form action="<?php echo $action; ?>" method=...>
  <input type="text" name="to_name" value.../>
  <input type="text" name="to_email" value.../>
  <input type="text" name="from_name" value.../>
  <input type="text" name="from_email" value.../>
  <input type="radio" name="voucher_theme_id".../>
  <textarea name="message" cols="40" rows="5">...
  <input type="submit".../>
</form>
```

Since all the information from the use case step can be found in this form, it is probable that the form is a part of the use case step implementation. User steps usually refer to attributes and the attribute names can be found in user interface implementation. These, for example, include user interface templates, views, or forms.

After identifying the form, it is easy to look up the other steps, too. For example, the form is generated by the `ControllerAccountVoucher` class:

```
class ControllerAccountVoucher extends Controller {
  public function index() {
    $data['text_agree'] = "I understand that gift
      certificates are non-refundable."
    ...
    $data['action'] =
      $this->url->link('account/voucher', '', true);
    ...
    if ($this->customer->isLogged())
      $data['from_name'] =
        $this->customer->getFirstName(). ' '  .
          $this->customer->getLastName();
    ...
    $this->response->setOutput(
      $this->load->view('account/voucher', $data));
  } }
```

Here, the string literal with the message to confirm certificates are non-refundable is assigned to the `text_agree` variable, where multiple words match with step 3 of the *Order Gift Certificate* use case. More specifically, these words are: "understand," "gift certificate," and "non-refundable."

Notice also this line of the `ControllerAccountVoucher` class:

```
$this->customer->getFirstName()
```

The `getFirstName()` method is called to retrieve the customer's name. After removing the special characters, such as dollar signs, dashes, dots, apostrophes, and round, curly, and angle brackets, this line becomes a sentence:

```
This customer get(s) (the) first name.
```

Consider also the sentence in step 4 of the *Order Gift Certificate* use case:

```
The customer enters the name.
```

Obviously, these sentence are very similar. Word "get" is the synonym of word "enter" according to WordNet [16], words "customer" and "name" are present in both sentences, and "this" is interchangeable with "the" in this case. The degree of similarity between two sentences can be determined by *semantic similarity calculation algorithms*. For example, the sentences are similar up to 80% according to the semantic similarity toolkit [22] and up to 65% according to the similarity calculated according to semantic nets and corpus statistics [10].

However, semantic similarity calculation algorithms are not always capable of correctly identifying similar sentences. Consider the following sentence:

```
Load controller.
```

obtained from the following line of code:

```
$this->load->controller('common/header');
```

Comparing it to the following use case step:

```
The customer selects to purchase a gift certificate.
```

we get a similarity of 31% according to the similarity based on semantic nets and corpus statistics [10] and 21% according to the semantic similarity toolkit [22], but the sentences express something completely different.

Additionally, nouns in use cases can be used for recognition of domain model classes in architectural pattern Model-View-Controller. Continuing with our example, the domain model classes are Customer, Cart, and Email, because they appear as nouns in the use case and as classes in source code.

Next, when systems employ object relational mapping, they include attributes matching words in use cases. For example, the Voucher model has the following attributes: name, email, theme, message, amount, and date added. These attributes match with the words from step 2 of the *Order Gift Certificate* use case. Also, when looking at the similarity of words "voucher" and "certificate," according to WordNet [16], "document" is a hypernym for both of them. Based on the attributes and this hypernym, it is reasonable to assume that the Voucher model is a part of the use case step implementation.

Table 1 summarizes textual similarities between use cases and source code. Although use cases are written in different languages than source code, both use cases and source code contain domain

[1] The use cases along with their implementation are available at github.com/useion/opencart/tree/master/upload/catalog.

Table 1. Textual similarity between use cases and source code.

	Use case	Source code
Abstraction level	High	Low
Language	Natural	Programming
Technical details	No	Yes
Domain specific words	Yes	
Form	Structured	

specific words and have the structured forms. Natural language is free, but source code has to adhere specific syntax of programming language. Source code also contains technical details, such as algorithms. Use cases on the other hand do not. However, the most significant difference is in the abstraction level of use cases and source.

3. Overcoming the Difference in the Abstraction Level of Use Cases and Source Code

As we saw in the previous section, despite the tools are capable of identifying hypernyms and hyponyms, the difference in the abstraction level of use cases and source code significantly worsens the results. In the end, semantically similar sentences end up with unsatisfactory similarity result and completely unrelated sentences are recognized as similar. To mitigate this problem, we investigate employing issue descriptions and commits here.

Consider this construct in the `sendVoucher()` method of the `ModelSaleVoucher` class:

```
$data['text_greeting'] =
  sprintf($this->language->get('text_greeting'),
    $this->currency->format($voucher_info['amount'],
    $this->config->get('config_currency')));
```

Exactly the same code appears in the changed code of a commit assigned to the following issue[2]:

> Sending a voucher by e-mail leads to an error like this: PHP Notice: Undefined index: in {server}\system\library\cart\currency.php on line 25 and lines 26,27 and 30.
> This is because the voucher is not created from an order.
> Now it takes the currency value from config.

This relationship can be used to identify the following use case step in the code:

```
The system sends an e-mail with details
   how to redeem a gift certificate
```

What enables this is that this step is similar to a specific sentence in the issue description:

```
Sending a voucher by e-mail leads to an error
```

The similarity of these sentences is 48% according to the similarity based on semantic nets and corpus statistics [10] and 50% according to the semantic similarity toolkit [22]. Notice, a gift certificate can be used for discount, while a voucher can be used to obtain a particular product. Despite the words "voucher" and "certificate" have different meaning, they share the same hypernym, "document". Such relations can be used to overcome the difference in the abstraction level of use cases and source code.

4. Use Case Identification in Source Code

In this section, we present three methods of identifying use cases in source code. The methods are based on synonym, hypernym, and hyponym analysis (Section 4.1), sentence similarity between use cases and code (Section 4.2), and semantic similarity between issues and commits (Section 4.3). Here, each method is presented conceptually, but all three methods —including their combination— were actually implemented and evaluated, which is described in Section 6. Use case remodularization, which is explained in Section 5, is based on the results of these methods.

4.1 Synonym, Hypernym, and Hyponym Analysis

The first method of identifying use cases in source code is based on the similarity between all the words from use case steps along with their synonyms, hypernyms, and hyponyms and all the words from a particular method in source code except the reserved words of the corresponding language. The result of this calculation is a percentage similarity between a given use case step and method in source code.

The reserved words of a programming language are not taken into consideration when calculating the similarity because they could deteriorate the results. For example, consider the "return" keyword in PHP and the *Return Product* use case. Word "return" would be found in all methods if this keyword was not removed. Also, we take into consideration only nouns, verbs, and adjectives because other word classes, such as conjunctions, prepositions, articles, etc., do not identify classes and methods in source code.

Given a use case step and a method in source code from the system under considerations, this method of identifying use cases in source code works as follows:

1. All nouns, verbs, and adjectives are extracted from the use case step.
2. The words are converted into lemmas.[3]
3. The plural nouns are converted into their singular form.
4. Synonyms, hypernyms, and hyponyms are retrieved for each such word.
5. The reserved words are removed from the method in source code (the reserved words depend on the programming language).
6. The similarity of the use case step and method is expressed as a percentage of equal words to all nouns, verbs, and adjectives in the use case step.

This method of identifying use cases in source code is applied to all use case steps in all use cases and all methods in the system. To choose only the methods in source code with a high similarity to use case steps, to each use case step, we applied *the algorithm for calculating the biggest difference*:

1. The methods are ordered by their similarity in ascending order.
2. The differences between the similarities of each n-th and $n+1$-th method are calculated.
3. If the biggest difference is between the similarities of n-th and $n+1$-th method, the methods after the $n+1$-th method and with over the 50% similarity are selected.

We considered various methods of selecting methods in source code with the high similarity. For example, selecting methods in source code above mean, average, or above specific percent. However, the algorithm for calculating the biggest difference takes into consideration also the sets with a lot of methods in source code having low or high number of similarity.

4.2 Sentence Similarity Between Use Cases and Code

As we observed in Section 2, the sentences created from code can be similar to the sentences from use cases. In this section, we present

[2] The issue is available at github.com/opencart/opencart/pull/4822.

[3] Lemmas are word forms selected to represent sets of the words with the same root.

a method of identifying use cases in source code that generates sentences from code and compares them to the sentences from use cases using a semantic similarity calculation algorithm.

The similarity between a use case step and a method in source code is calculated as follows:

1. The following types of sentences are created from code:
   (a) From the method name: Class name + method name (e.g., given the `ControllerAccountVoucher` class and `add()` method, the sentence would be "Controller account voucher add.")
   (b) From the method calls: Object name + method name (e.g., given the `$order->add(...)` call, the sentence would be "Order add.")
2. The sentences from the use case step are compared with the sentences created from code according to a semantic similarity calculation algorithm (no specific algorithm is enforced).
3. The maximum similarity of all the similarities for the method, i.e., between method name, its assignments, calls, and the use case step, is sentence similarity between use cases and code.
4. The algorithm for calculating the biggest difference is applied to the use case step after calculating all the similarities for each method.

4.3 Semantic Similarity Between Issues and Commits

Issues in change management systems, such as GitHub, are described in few sentences or paragraphs. After an issue description is formulated, it is analyzed by developers who attempt to resolve it. If they resolve the issue, they assign commits to it. This relation between the issue description and the code that was changed from the commit is used further. In this section, we present a method of identifying use cases in source code based on relations between issues and commits.

The method of identifying use cases in source code between a use case step and a method in source code works as follows (see also Fig. 1):

1. Sentences are generated from the method (as in Section 4.2).
2. Five issues that have the highest semantic similarity between the issue name and the use case name are selected for the given use case.
3. Sentences from the changed methods of commits assigned to the issues are generated.
4. For each issue, the similarity is calculated by generating the sentences from the code changed in issue commits (as in Section 4.2) in two ways:
   (a) Between the sentences from a particular issue description and use case step
   (b) Between the sentences from the code changed in issue commits and the sentences from the method based on a semantic similarity calculation algorithm.
5. The similarity is calculated as the average of the maximum similarity from steps 4a and 4b.
6. The algorithm for calculating the biggest difference is applied to the use case step after calculating all the similarities for each method.

5. Use Case Remodularization

Different approaches to preserving use cases in code rely on different use case representation. Thus, use cases are represented by classes

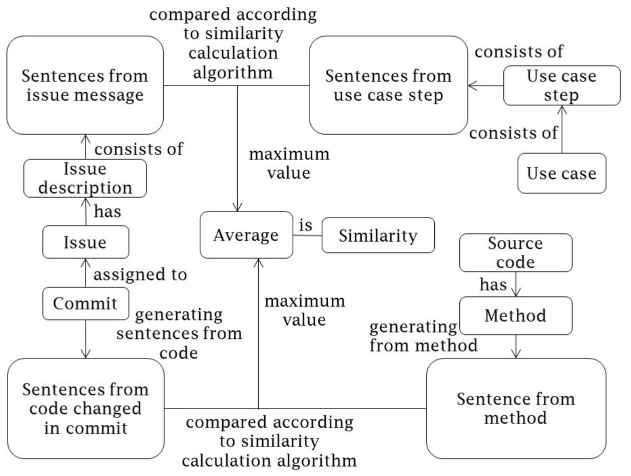
compared according to similarity calculation algorithm
Sentences from issue message
Sentences from use case step
consists of
Use case step
consists of
Use case
consists of
Issue description
has
Issue
assigned to
Commit
maximum value
Average
is
Similarity
Source code
has
Method
maximum value
generating sentences from code
generating from method
Sentences from code changed in commit
compared according to similarity calculation algorithm
Sentence from method

Figure 1. Similarity based on relations between issues and commits.

in InFlow [4], partial classes in literal inter-language use case driven modularization [3, 5], roles in DCI [21], or aspects in aspect-oriented software development with use cases [9]. Since our methods of identifying use cases in source code yield methods (operations) in source code related to use case steps, it is not possible to employ InFlow and DCI because DCI uses roles to organize code, and applying InFlow would require to affect code to a large extent, e.g., by moving method bodies, changing method parameters, etc. However, use cases can be represented by aspects [9], though methods similar to multiple use cases would be duplicated in multiple aspects and these duplicates would necessarily get into an unsynchronized state on the course of changes. Therefore, we employ literal inter-language use case driven modularization,[4] since this approach features synchronization of duplicate code.

In literal inter-language use case driven modularization, each use case along with its implementation is in a *use case file*. Since use case files are written in Markdown [5], partial classes in different programming languages can be mixed in one file. This is the use case file for the *Order Gift Certificate* use case introduced in Section 2:

~~~
# Use case Order Gift Certificate

## Main scenario
...

# Code

## controller/account/voucher.php
```php
<?php
class ControllerAccountVoucher extends Controller {
  public function index() { ... }
  public function success() { ... }
  protected function validate() { ... } }
```

## model/extension/total/voucher_theme.php
```php
<?php
class ModelExtensionTotalVoucherTheme extends Model {
  public function getVoucherThemes(...) { } }
~~~

[4] The project site is available at useion.com.

~~~
```

## controller/checkout/confirm.php
```php
<?php
class ControllerCheckoutConfirm extends Controller {
  public function index() { ... } }
```

## model/checkout/order.php
```php
<?php
class ModelCheckoutOrder extends Model {
  public function addOrder($data) { ... } }
```

## model/extension/total/voucher.php
```php
<?php
class ModelExtensionTotalVoucher extends Model {
  public function addVoucher(...) { ... } }
```
~~~

As can be seen, a use case file contains the use case itself followed by partial classes that implement it. It is the methods of these partial classes that are being identified by our method (presented in the previous section). Subsequently, these methods are used in the remodularization process. Any changes to these files are synchronized with the executable code and vice versa, which is covered in detail elsewhere [3, 5].

6. Evaluation

We evaluated the methods of identifying use cases in source code proposed in Section 4 on a study of the Catalog module of the OpenCart open source e-shop system (introduced in Section 2) employing 16 use cases and 141 methods in 38 classes (with each class in its own file). The study considered the following use cases:[5]

- *Login*
- *Change Password*
- *Subscribe To Newsletter*
- *Register Account*
- *Show Wishlist*
- *Wish Product*
- *Remove Product*
- *Add Product*
- *Register Affiliate Account*
- *Estimate Shipping And Taxes*
- *Place Order*
- *Use Reward Points*
- *Show Order Detail*
- *Use Coupon Code*
- *See Reviews Of Product*
- *Review Product*

The module is written in PHP and uses Model-View-Controller architectural pattern.

[5] The use cases and their implementation is available at github.com/useion/code-uc-synon-ident/tree/master/examplem.

We implemented all three methods of identifying use cases in source code described in Section 4. We also implemented the use case remodularization script (Section 5).[6] in JavaScript using Node.js. Since NLTK is written in Python and our methods in JavaScript, we interfaced them using remote procedure call client in JavaScript and server in Python.

Synonyms, hypernyms, and hyponyms are obtained from WordNet using NLTK. Word classes are determined by the standard Treebank part-of-speech tagger (maxent_treebank_pos_tagger) of NLTK and lemmas are determined by the stem package of NLTK. Plural nouns are converted into singular nouns using Pattern.[7] We implemented and compared two similarity calculation algorithms: Levenshtein's distance[8] and sentence similarity based on semantic nets and corpus statistics [10].[9]

The methods of identifying use cases in source code are expected to find correct relations between use case steps and methods in source code. We determine this by measuring a percentage of correctly identified methods over all correctly identified methods (recall) and a percentage of correctly identified methods over all identified methods (precision). Sections 6.1 to 6.4 present the results of calculating the similarity based on each of the three methods of identifying use cases in source code and on the combination of these methods. Section 6.5 presents the overall results. Section 6.6 identifies threats to validity.

6.1 Synonym, Hypernym, and Hyponym Analysis

The results of counting correct and incorrect relations between use case steps and methods in source code using the similarity based on synonyms, hypernyms, and hyponyms are summarized in Table 2.[10] The legend is as follows:

Lemmas	Lemmas only
Synonyms	Lemmas and synonyms
Hypernyms	Lemmas, synonyms, and hypernyms
Hyponyms	Lemmas, synonyms, hypernyms, and hyponyms

In the case of relying on lemmas only, the method of identifying use cases in source code was able to detect 40.40% of correct relations, though 22 methods were identified on average incorrectly, which is the precision of 10.05%. Adding synonyms helped to detect correct relations about 10.16%, but the number of incorrect relations increased on average about 34 lowering the precision to 7.02%. As was expected, adding hypernyms and hyponyms into account helped even more with detecting correct relations—about 8.03%—but the number of incorrectly identified relations increased a little lowering the precision to 6.73%.

6.2 Sentence Similarity Between Use Cases and Code

Table 3 shows the results of the sentence similarity between use cases and code.[11] The legend is as follows:

[6] The implementation of the methods is available at github.com/useion/code-uc-synon-ident and github.com/useion/code-uc-sentences-ident. The use case remodularization script is available at github.com/useion/remodularize.

[7] Pattern is available at pypi.python.org/pypi/Pattern.

[8] Levenshtein's distance is implemented in the similarity module of Node.js available at https://www.npmjs.com/package/similarity.

[9] The script implementing the sentence similarity based on semantic nets and corpus statistics is available at github.com/sujitpal/nltk-examples/blob/master/src/semantic/short_sentence_similarity.py.

[10] The results for each use case can be found at github.com/useion/code-uc-synon-ident/blob/master/Output-m-50.xlsx.

[11] The results for each use case can be found at github.com/useion/code-uc-sentences-ident/blob/master/Output-50.xlsx (see the `Output-direct` columns).

Table 2. The results for the method of identifying use cases in source code based on synonyms, hypernyms, and hyponyms.

	Recall [%]	Precision [%]
Lemmas	40.40	10.05
Synonyms	50.56	7.02
Hypernyms	58.59	6.73
Hyponyms	58.59	6.86

DirectLeven:	the sentence similarity between use cases and code using Levenshtein's distance
DirectLi:	the sentence similarity between use cases and code using semantic nets and corpus statistics [10]

Table 3. The results for the sentence similarity between use cases and code.

	Recall [%]	Precision [%]
DirectLeven	13.13	20
DirectLi	11.11	13.58

Levenshtein's distance calculation is not a semantics based similarity calculation algorithm, because it is based on measurement of difference between two string sequences, but DirectLi is. Surprisingly, the string based similarity calculation algorithm identified more correct relations than the semantic based one about 2.02% with precision 20. Furthermore, the semantics based similarity calculation algorithm identified more incorrect relations lowering the precision to 13.58%. Therefore, DirectLeven performed better than DirectLi.

6.3 Similarity Based on Relations Between Issues and Commits

The results for the method of identifying use cases in source code based on relations between issues and commits are displayed in Table 4. [12] The legend is as follows:

IssueLeven:	the method of identifying use cases in source code based on relations between issues and commits using Levenshtein's distance
IssueLi:	the method of identifying use cases in source code based on relations between issues and commits using semantic nets and corpus statistics [10]

Since our study is based on OpenCart, we used OpenCart's issues and commits to populate our database with the relations between issues and commits. The semantics based similarity calculation algorithm found more correct relations about 22.23% with both having almost a same precision. Therefore, IssueLi performed better than IssueLeven here.

Although NLTK was in RAM and GitHub requests and the similarity measures were cached, it took approximately one hour for each use case to calculate results the of IssueLi. This is because the issue descriptions contained a lot of sentences that had to be compared with use case sentences and all the sentences from the changed code of issue commits compared with the sentences from source code of the application.

[12] The results for each use case can be found at github.com/useion/code-uc-sentences-ident/blob/master/Output-50.xlsx (see the `Output-issue` columns).

Table 4. The results for the similarity based on relations between issues and commits.

	Recall [%]	Precision [%]
IssueLeven	27.27	3.46
IssueLi	49.5	3.96

6.4 Combining the Methods of identifying Use Cases in Source Code

The methods of identifying use cases in source code evaluated in Sections 6.1, 6.2, and 6.3 were combined in different configurations and the results are presented in Table 5. Only those relations are considered which are identified in all methods of identifying use cases in source code in a particular combination.[13] We used the following configurations:

All	Lemmas, Synonym, Hypernym, Hyponym, DirectLi, DirectLeven, IssueLi, IssueLeven
Li	DirectLi, IssueLi
Leven	DirectLeven, IssueLeven

Table 5. Recall and precision of the combinations of methods of identifying use cases in source code.

	Recall [%]	Precision [%]
All	3.37	75
Li	6.06	45.15
Leven	3.03	75

The *All* configuration decreased the number of incorrectly identified relations almost to zero raising the peak of precision to 75 as was expected, but it also dramatically decreased the number of correct relations to 3.37%. Comparing the *Leven* and *Li* configurations, the results indicate that *Leven* is better than *Li*, although its recall is lower than in *Li*, which was not expected, because *Leven* does not employ semantic similarity calculation algorithm.

6.5 Overall Results

Table 6 presents normalized average values of recall and precision attributes. The data was normalized by standard score. The higher the number is, the better at use case identification in code particular approach is. The best results were achieved using All, as was expected. Slightly behind are Leven, Hyponyms, and Hypernyms. Leven has high precision and Hyponyms and Hypernyms have high recall. The sentence similarity between use cases and code ended up the worst with recall approximately 12% and precision 16%. The similarity based on relations between issues and commits strongly depends on the issues.

6.6 Threats to Validity

There are several internal threats to validity we consider significant. First, different programmers could consider different methods to be part of use case implementation, which could affect the results. Second, more precise results could be achieved with a better similarity calculation algorithm. Third, using a dictionary other than WordNet, or a different part-of-speech tagger could affect the results.

We also consider as significant the following external threats to validity. First, complex systems, systems of other domains, or systems employing different architecture could affect the results. In particular, better results would be achieved in systems having

[13] The script that combines the results is available at github.com/useion/code-uc-sentences-ident/blob/master/merge.js.

Table 6. Overall results.

All	0.349
Leven	0.341
Hyponyms	0.33
Hypernyms	0.327
Synonyms	0.156
IssueLi	0.078
Lemmas	-0.01
Li	-0.13
DirectLi	-0.59
DirectLeven	-0.43
IssueLeven	-0.42

use cases represented in code. Second, high level programming languages could raise the abstraction level of code identifiers and, thus, they could be closer to use cases possibly positively affecting the results. Third, using a different use case notation could affect the results too. Fourth, in case of the similarity based on relations between issues and commits, a change management system other than Github could affect the results, too. Also, processing more issues of other projects could improve the results.

7. Discussion

The methods of identifying use cases in source code does not provide sufficiently precise results for automatic use case driven remodularization. As a result, an expert still has to be involved in selecting a relevant implementation to achieve use case driven remodularization. However, the identified implementation and generation of use case modules greatly helps with remodularization.[14] Having the possibility to see another perspective of software according to use cases improves source code comprehension and maintenance [3–5].

The success of our method strongly depends on issues and commits assigned to them. It could help to have a convention, that would enforce to include which use case step is related to which issue in its description. It would also help if use case driven modularization was employed in source code from the beginning. This would improve the success of our method significantly since whole sentences from code would be matched with use case steps.

Even if the methods of identifying use cases in source code would be precise enough, in their current implementation, calculating similarity for only one use case takes an hour, which makes them unsuitable for a continuous automated code remodularization.

8. Related Work

There are three broader areas of research related to our work: recovering traceability links, similarity between natural language and code, and use case based code remodularization. As far as the similarity between text and code is concerned, a semi-automatic methods of use case identification in code were reported [19, 26]. The methods are based on generating execution traces according to a static code analysis. However, an expert has to be involved in order to assign the execution traces to actual use cases.

Recovering traceability links among software artifacts was deeply studied before. For example, there are approaches using latent semantic indexing [14] or other information retrieval methods, such as vector space model and probabilistic network model, employing ontologies [27], or Wikipedia [13]. Our work differs from these approaches because we employ issues and commits to find specific relations between use cases and source code. For this, we create specific sentences from source code and compare them by different similarity calculation algorithms.

Automatic identification of traceability links is concurrent research problem. Therefore, a human expert is still necessary to identify traceability links, as our study showed, too. Involving an expert at early stages or after the identification of traceability links was studied before [28]. Our approach also requires manual review of identified relations.

There are several approaches capable of determining the similarity of sentences. Some of them, such as Jaccard, Cosine and Dice's approach [23], the semantic similarity toolkit [22], or the similarity based on semantic nets and corpus statistics [10], treat the similarity at the lexical level. More recent approaches employ the combination of syntactic, lexical, and semantic similarity [2]. The experience from detecting inflected forms of last names [25] might be applicable as well if the approach is to be extended to synthetic languages. In our approach, we used Levenshtein's distance and determining sentence similarity based on semantic nets and corpus statistics [10].

In the method of similarity measurement between code fragments [29], the similarity is measured according to the semantic relatedness of corresponding textual descriptions obtained from StackOverflow. The method is correct approximately up to 85%. Although this method is successful in determining program similarity, this is different from finding similarity between natural language and code. However, the idea can be used to improve the results of our method.

In the semi-automatic method of code remodularization according to features [17], the features have to be identified and execution traces have to be recorded at program run time manually. Consequently, a tool moves classes to packages that represent the corresponding features. However, this method requires a considerable human effort. This method organizes code into packages, but not at the granularity of use case steps as we do.

9. Conclusion and Challenges

In this paper, we propose an approach of employing issues and commits for in-code sentence based use case identification and remodularization. The approach aims at providing use case based perspective on the existing code. The sentences of use case steps are compared to sentences of issue descriptions, while the sentences generated from the source code of issue commits are compared to sentences generated from the corresponding methods in source code in order to quantify the similarity between use case steps and methods in source code using different similarity calculation algorithms. We evaluated the approach on a study of the OpenCart e-shop application employing 16 use cases. The approach achieved the recall of 3.37% and precision of 75%. The success of the approach strongly depends on issues and commits assigned to them.

We find multiple points to be challenging. One of them is raising the recall and precision of the presented approach in order to improve use case identification in source code. This could be achieved by covering more sentence types as they are identified in source code, but embracing some sentence types might actually deteriorate results. Another challenge is in decreasing the time needed to calculate the similarity. Finally, it could also help if developers could write use case expressions in issue descriptions and represent use cases in code, which is not trivial to achieve.

Acknowledgments

This work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under the grant No. VG 1/0752/14, by the internal grant scheme of the Slovak University of Technology in

[14] The input source code is available at github.com/useion/code-uc-synon-ident/tree/master/examplem/implementation.
The output source code after remodularization is available at github.com/useion/remodularize/tree/master/out.

Bratislava in support of the teams preparing proposals for Horizon 2020 (V. Vranić), by SOFTEC PRO SOCIETY in cooperation with Softec and Centaur (P. Berta), and by Research & Development Operational Programme for the project Research of Methods for Acquisition, Analysis and Personalized Conveying of Information and Knowledge, ITMS 26240220039, co-funded by the ERDF.

References

[1] S. Bird. NLTK: The natural language toolkit. In *Proceedings of COLING/ACL on Interactive Presentation Sessions, COLING-ACL '06*, Sydney, Australia, 2006.

[2] M. Blšták and V. Rozinajová. Automatic question generation based on analysis of sentence structure. In *Proceedings of 19th International Conference on Text, Speech, and Dialogue, TSD 2016*, Brno, Czech Republic, 2016. Springer.

[3] M. Bystrický and V. Vranić. Literal inter-language use case driven modularization. In *MODULARITY Companion 2016, Companion Proceedings of 15th International Conference on Modularity, Modularity 2016*, M&#225;laga, Spain, 2016. ACM.

[4] M. Bystrický and V. Vranić. Preserving use case flows in source code: Approach, context, and challenges. *Computer Science and Information Systems Journal (ComSIS)*, 14(2):423–445, 2017.

[5] M. Bystrický and V. Vranić. Use case driven modularization as a basis for test driven modularization. In *Proceedings of 6th Workshop on Advances in Programming Languages (WAPL'17)*, Prague, Czech Republic, 2017. IEEE.

[6] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[7] F. d'Amorim and P. Borba. Modularity analysis of use case implementations. *Journal of Systems and Software*, 85(4):1012–1027, 2012.

[8] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk. Enhancing software traceability by automatically expanding corpora with relevant documentation. In *Proceedings of 2013 IEEE International Conference on Software Maintenance, ICSM 2013*, Eindhoven, Netherlands, 2013. IEEE.

[9] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2004.

[10] Y. Li, D. McLean, Z. A. Bandar, J. D. O'Shea, and K. Crockett. Sentence similarity based on semantic nets and corpus statistics. *IEEE Transactions on Knowledge and Data Engineering*, 18(8):1138–1150, 2006.

[11] H. Liu and H. Lieberman. Metafor: Visualizing stories as code. In *Proceedings of 10th International Conference on Intelligent User Interfaces, IUI '05*, San Diego, California, USA, 2005. ACM.

[12] G. A. D. Lucca, A. R. Fasolino, and U. de Carlini. Recovering use case models from object-oriented code: A thread-based approach. In *Proceedings of 7th Working Conference on Reverse Engineering, WCRE 2000*. IEEE, 2000.

[13] A. Mahmoud, N. Niu, and S. Xu. A semantic relatedness approach for traceability link recovery. In *Proceedings of 2012 20th IEEE International Conference on Program Comprehension, ICPC 2012*, Passau, Germany, 2012. IEEE.

[14] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of 25th International Conference on Software Engineering, ICSE 2003*, Portland, Oregon, USA, 2003. ACM.

[15] P. W. McBurney and C. McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, 2016.

[16] G. A. Miller. WordNet: A lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[17] A. Olszak and B. N. Jørgensen. Remodularizing java programs for comprehension of features. In *Proceedings of 1st International Workshop on Feature-Oriented Software Development, FOSD '09*, Denver, Colorado, USA, 2009. ACM.

[18] I. Omoronyia, G. Sindre, M. Roper, J. Ferguson, and M. Wood. Use case to source code traceability: The developer navigation view point. In *Proceedings of 2009 17th IEEE International Requirements Engineering Conference, RE 2009*, Atlanta, Georgia, USA, 2009. IEEE.

[19] T. Qin, L. Zhang, Z. Zhou, D. Hao, and J. Sun. Discovering use cases from source code using the branch-reserving call graph. In *Proceedings of 10th Asia-Pacific Software Engineering Conference, APSEC 2003*, Chiang Mai, 2003. IEEE CS.

[20] M. M. Rahman and C. K. Roy. An insight into the pull requests of github. In *Proceedings of 11th Working Conference on Mining Software Repositories, MSR 2014*, Hyderabad, India, 2014. ACM.

[21] T. Reenskaug and J. O. Coplien. The DCI architecture: A new vision of object-oriented programming. Artima Developer, 2009. http://www.artima.com/articles/dci_vision.html.

[22] V. Rus, M. Lintean, R. Banjade, N. Niraula, and D. Stefanescu. SEMILAR: The semantic similarity toolkit. In *Proceedings of 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations, ACL 2013*, Sofia, Bulgaria, 2013.

[23] S. M. Saad and S. S. Kamarudin. Comparative analysis of similarity measures for sentence level semantic measurement of text. In *2013 IEEE International Conference on Control System, Computing and Engineering, ICCSCE 2013*. IEEE, 2013.

[24] S. M. Saad and S. S. Kamarudin. Comparative analysis of similarity measures for sentence level semantic measurement of text. In *Proceedings of 2013 IEEE International Conference on Control System, Computing and Engineering, ICCSCE 2013*. IEEE, 2013.

[25] D. Zahoranský and I. Polášek. Text search of surnames in some slavic and other morphologically rich languages using rule based phonetic algorithms. *IEEE/ACM Transactions on Audio, Speech and Language Processing*, 23(3):553–563, 2015.

[26] L. Zhang, T. Qin, Z. Zhou, D. Hao, and J. Sun. Identifying use cases in source code. *Journal of Systems and Software*, 79(11):1588 – 1598, 2006.

[27] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev. Ontological approach for the semantic recovery of traceability links between software artefacts. *IET Software*, 2(3):185–203, 2008.

[28] J. Zhou, Y. Lu, and K. Lundqvist. A context-based information retrieval technique for recovering use-case-to-source-code trace links in embedded software systems. In *Proceedings of 2013 39th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2013*, 2013.

[29] M. Zilberstein and E. Yahav. Leveraging a corpus of natural language descriptions for program similarity. In *Proceedings of 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016*, Amsterdam, Netherlands, 2016. ACM.

[30] M. Śmiałek, W. Nowakowski, N. Jarzębowski, and A. Ambroziewicz. From use cases and their relationships to code. In *Proceedings of 2012 2nd IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012*, Santander, Spain, 2012. IEEE.