Use Case Driven Modularization

Michal Bystrický*
Institute of Informatics, Information Systems and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 2, 842 16 Bratislava, Slovakia
michal.bystricky@stuba.sk

Abstract

How well is software comprehensible and maintainable is highly dependent on code modularity. Common object-oriented modularity, for instance, puts forward technical concerns such as reusability, but mystifies other concerns, especially usage scenarios, which greatly help with comprehending software systems, but they are completely dissolved in code. Although there are approaches able to modularize code according to use cases, they fail to gather all the code related to a use case in one module and to reflect its steps. Such modularization would improve software comprehensibility and maintainability. In this extended abstract, we briefly introduce use case driven modularization approaches achieving this. Multiple studies were conducted to evaluate the proposed approaches. Use case driven modularization requires less effort to follow code and to apply a change in code than common object-oriented modularization. DCI and aspect-oriented software development with use cases add more complexity to following a use case flow in code than the use case driven modularization approach.

Categories and Subject Descriptors

D.1.5 [**Object-oriented Programming**]; D.2.3 [**Coding Tools and Techniques**]: Object-oriented programming; D.2.6 [**Programming Environments**]: Integrated environments; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Documentation; D.2.11 [**Software Architectures**]: Patterns

Keywords

modularization, use case, user acceptance test, test driven development

1. Introduction

What is typical for open source systems, on top of which a lot of software systems are built, is that the code serves as a documentation while no other documentation is available or it is very limited. Without the documentation, it is very hard to go through hundreds of thousands lines of code to comprehend what is program actually doing. This is because of a high detail captured in code spread through hundreds of files and tens of folders. With the documentation, the situation is slightly better. However, the documentation often cannot keep up with code over time, because code is constantly changing in agile approaches, and documentation becomes obsoleted at some point. A way to tackle this may be to connect the documentation to the code [12], but these connections have to be maintained, too. Another better way may be to bring the documentation into the code, similarity as literate programming do bring programmers' thoughts into code [13], but this is not a real documentation, ultimately introducing even more confusion [16].

The high detail captured in code spread through many files and folders negatively impacts code maintainability, too. The root of any change lies in its change request, which is typically formulated from the perspective of how system is used, i.e., usage scenarios. Numerous changes in tens of files spread through multiple directories have to be applied in code with each commit, because code modularity puts forward rather technical concerns, but mystifies other concerns, such as usage scenarios, which greatly help to comprehend and maintain software systems, but they are completely dissolved in code.

There are many approaches aiming to modularize code according to use cases [8, 9, 10, 11, 15], however these approaches are not capable of gathering all the code related to a use case in one module and to reflect its steps, eventually failing to capture usage scenarios in code. In this extended abstract, we briefly introduce a set of use case driven modularization approaches gathering all the code related to a use case in one module and reflecting its steps. We also introduce a new approach achieving test driven modularization.

The structure of this extended abstract is as follows. Section 2 presents an approach to identifying use case flows in common object-oriented code. Section 3 presents an approach preserving use case flows in code called InFlow. Section 4 presents literal preserving of use case flows, which pushes the idea further. Use cases appear right in code. Section 5 introduces an approach achieving test driven modularization. This approach modularizes code according to tests and it is compatible

*Recommended by thesis supervisor: Assoc. Prof. Valentino Vranić
To be defended at the Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava on [To be specified later], 2018.

with the former approaches. Section 6 presents an evaluation of the approaches. The evaluation is based on multiple studies, parts of which will be briefly presented throughout this abstract as well. Section 7 concludes this thesis and presents remaining challenges.

2. Identifying Use Case Flows in Code

There is a significant textual similarity between *use case flows*, also called use case scenarios or simply use cases, and code. Table 1 summarizes the textual difference between use cases and source code [1]. Both use cases and code include certain domain specific words and their form is structured, though they are written in different languages. Use cases are written in natural language which is a free, non-structured language, however in the case of code, it has to adhere specific syntax of programming language. On the top of that, code also includes implementation and technical details, e.g., algorithms, but use cases do not. The major difference is the abstraction level though. While use cases are at the high level, code is at the low level.

Table 1: The Textual Difference Between Use Cases and Source Code [1]

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

Since both use cases and code include domain specific words, it is reasonable to assume that one can be identified in the other. We propose a new method of in-code use case identification in code [1], in which we employ issues and commits to find the relation between use cases and code. The relation of issues, at the side of natural language, and their assigned commits, at the side of code, is used. Sentences from issue descriptions are compared to sentences from use case steps, and sentences from commits assigned to particular issue are compared to code. Based on the relation, we can calculate the similarity between use case steps are issue descriptions, and link them with code which is changed in assigned commits. For sentence comparisons we used the following sentence similarity algorithms: Levenshtein's distance and sentence similarity based on semantic nets and corpus statistics [14].

3. Preserving Use Case Flows in Code

We continue in making use case flows more closer to code in this section and further. Fig. 1 shows use case *Listen a Stream* of an audio streaming service on the left side and its corresponding code on the right side. This use case describes what does it mean to play an audio stream in the main success scenario and handling the error status of reaching maximum number of listeners in the alternate flow (Extensions). The arrows connect use case steps to corresponding statements in code.

As you can see, each use case is represented by a *use case class*, in this case, the use case is represented by the `ListenStream` class in code. Step 1 activates the use case for which we reserved special method `run`. In Step 2, the system prompts for credit cart details, because the audio streaming service is payed. This is represented in code as rendering of a form. In Step 3, the user enters credit card details, which is basically filling up the rendered form. Their action of submitting the form is routed to the corresponding method of which declaration represents this step. Notice that the statements in methods must be executed in the order of use case steps, otherwise the functionality would be changed, this is how we ensure the order of use case steps is preserved.

In object-oriented modularization, this code would be typically scattered across multiple files and folders, but in this case, the code related to a use case is in one place and the intent is readable right from the code. Potentially, such code can be understood also by end users [2].

These use case classes are attached to the domain model using a special annotation called *InFlow*, which is also the name of the approach [6]. What this annotation do is redirecting control flow from a particular domain object to the corresponding method of the use case object. Note that the context of the domain object and the use case object are accessible in the corresponding method implementation. The annotation employs metaprogramming to achieve this, but we implemented it in Ruby, AspectJ, and PHP using design patterns.

4. Literal Preserving of Use Case Flows

The idea of preserving use cases from the previous section is pushed to its limits in the next approach [4, 7] briefly introduced in this section. The use case texts are included right in code, literally preserving them in code, while retaining the benefits of object-oriented decomposition. Consider use case *Add Product into Cart* [7]:

````
# Use case Add Product into Cart

## Main flow
1. User selects to add a product into cart
2. System saves the product into cart
3. System notifies user about updating
   shopping cart
4. Include "Show Cart"

# Code

## view/product-detail.html
```html
<h3>{%=o.product.name%}</h3>
<p>{%#o.product.description%}</p>
<div>{%=o.product.price%}</div>
<a id="add-into-cart">Add into cart</a>
```

## model/Cart.js
```js
({ add: function (id) {...} })
```

## controller/public.js
```js
(function () {
  this.addIntoCart = function (event) {
    require({
      Cart:   "model/Cart.js"
    ...
```
````

As can be seen, the use case text is on the top of this Markdown file, also called a *use case file*, and below it, is implementation. The implementation is consisted of

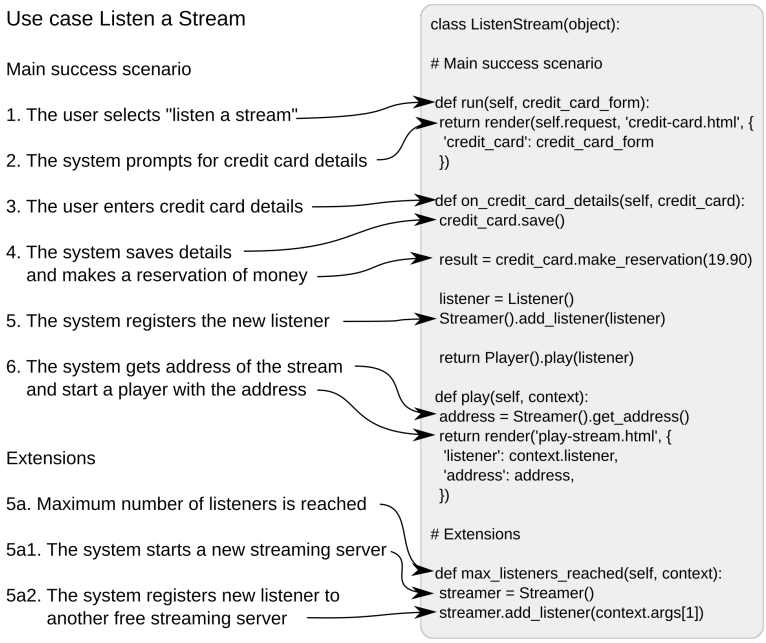

Figure 1: Preserving use case flows in source code [6].

multiple partial parts, e.g., partial classes, each of which can be in different programming language.

Changes to these use case files are propagated to the *executed code*, also called *merged code*, and changes to the executed code are propagated back to the use case files, as can be seen in Fig. 2 [5]. This is how we achieve inter-language modularization.

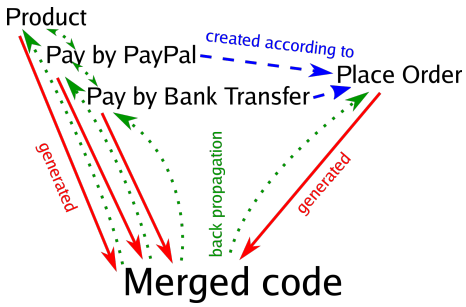

Figure 2: Synchronizing changes among multiple views on software [5].

The modularization of the executed code can be of any kind, but the approach works well when the use case driven approaches are used for the executed code, such as Data-Context-Interaction [8, 15] or aspect oriented software development with use cases [10, 11]. Since our studies were based on real systems, we used object-oriented modularization of the executed code in these examples.

A special representation of use cases in code was explored, too, where each method, also called a *use case step method*, in a use case class represents one use case step. Use case step methods have names resembling sentences of use case steps in the camel case format. For this, a framework was developed [3].[1][2] The framework executes the use case methods in the order of how it is specified in the use case text. A basic text analysis of the use case text is performed for this.

The example representation of use case *Add Article* follows [4]:

```
class AddArticle {
  selectsAddArticle () {...}
  promptsFor (args = ['title', 'content']) {...}
  writesAnArticle () {...}
  ...
}
```

As can be seen from this example, in step *The system prompts for the article title and content* of this use case represented by the second method in the example class, we also experimented to extract attributes `title` and `content` from the use case step and provide them in the use case method as parameters. These can be used to change behavior right from the use case text, and potentially, bring end-users closer to programming [2]. We name this approach literal inter-language use case driven modularization and denote it further as *Literal*.[3]

5. From Preserving Use Cases to Preserving Tests

Continuing with the example from the previous section, a new section of *Tests* is introduced in the use case file to preserve user acceptance and unit tests [5], in this case, user acceptance test *Adding products into the cart*:

[1] See `https://bitbucket.org/bystricky/literal-use-cases`
[2] See `https://www.youtube.com/watch?v=R4ArqH4ZdgI`.
[3] See `http://useion.com/`.

```
# Tests

## tests/features/cart.feature
‘‘‘feature
Feature: Shopping cart

Scenario: Adding products into the cart
 Given I am on the test page
 When I click on "Add into cart"
 Then I should see "Test product"
 And I should see "120 EUR"
...
‘‘‘

## tests/unit/cart.js

‘‘‘js
...
Cart.empty();
assert(
 Cart.getAll().length === 0, "The cart should
   be empty");
Cart.add({...});
assert(Cart.getAll().length === 1, "...");
...
‘‘‘
```

As can be seen, the user acceptance test covers one scenario of the *Add Product into Cart* use case. Unit test *Cart* partially covers methods executed in the context of this use case. Notice, it is clear which parts of use case are implemented and tested.

Pure test views can be created by placing a test on the top of the Markdown file and the code for the test would be below. This will enable to modularize code entirely according to tests achieving test driven modularization [5]. We further denote this approach as *TDM*. Note, both use case driven and test driven modularizations can be used simultaneously.

6. Evaluation

We evaluated the proposed approaches as well as the state of the art approaches: Data-Context-Interaction further denoted as DCI, aspect-oriented software development with use cases denoted as AOSD-UC, and also, common object-oriented approach with Model-View-Controller denoted as MVC. The evaluation was based on multiple studies: an audio streaming service consisting of 12 use cases, a real world application of content management system of 25 use cases, an open source e-shop application OpenCart consisting of 55 use cases. In total, the studies embraced 92 use cases.

We considered an effort to comprehend a use case and an effort to make a change, such as, integrating an extension flow with the main flow, adding, removing, adjusting steps in the main flow or an extension flow, but also commits from version control systems. The effort is measured by the number of context switches, the number of lines of code, and the number of opened files, which are needed to be overcome to comprehend a use case or apply a change. A context switch is a necessity to look elsewhere than at the next statement when following particular use case flow in code.

Following a use case flow is easier in InFlow than in DCI and AOSD-UC, though InFlow introduces a lot of code indirection, which could eventually lower the comprehension. DCI scatters use case implementation into roles [17], which made very hard to follow a use case also because DCI does not preserve all the steps of use cases compared to InFlow and AOSD-UC. InFlow annotation enables to see traceability links from domain model to use cases, but this is not possible in DCI nor AOSD-UC. AOSD-UC is better in making a change than InFlow, because it exposes all the behavior for a use case in one aspect, but InFlow is better than DCI.

MVC, compared to Literal, requires much more effort to follow and change a use case, because use cases are scattered in many modules in MVC. But the overall results speak in favor of TDM, because user acceptance test modules are covering just one flow of use cases, therefore the test modules are smaller and better comprehensible. Also, test driven modularization includes the test right in code, which reduces switching between tests and code in test driven development.

The approach of in-code use case identification in code was evaluated on a study of the OpenCart e-shop application employing 16 use cases written in an object-oriented way. The approach achieved the recall of 3.37% and precision of 75%, but the success of the approach strongly depends on issues and commits assigned to them. It would help, if use case driven modularization was applied from the beginning. Although the results do not seem good, finding at least one module per use case greatly helps in locating use cases in code.

7. Conclusions and Challenges

How well is software comprehensible and maintainable is highly dependent on code modularity. Unfortunately, use cases, which help very much with comprehending system are completely dissolved in code. This is because common modularization puts forward technical concerns, such as reusability. Approaches able to modularize code according to use cases tackle this problem, but they fail to gather all the code related to a use case in one module and to reflect its steps. In this extended abstract, we introduced a set of use case driven modularization approaches achieving this.

Multiple studies were conducted to evaluate the approaches. Use case driven modularization requires less effort to follow code and to apply a change in code than common object-oriented modularization. DCI and aspect-oriented software development with use cases add more complexity to following a use case flow in code than our approach, called InFlow. InFlow, however, introduces a lot of code indirection which can worsen code comprehensibility.

Test driven modularization employing user acceptance tests in code reduces switching between tests and code in test driven development. User acceptance test driven modularization also lower the complexity of modules compared to the use case modules, because a user acceptance test describes only one usage scenario while a use case describes multiple usage scenarios, therefore user acceptance test modules are smaller and more comprehensible.

We consider multiple remaining points to be challenging. Use case driven modularization brings end-users closer to programming, as we showed, but they still have to cope with programming constructs. It would help if programming languages employed natural language con-

structs. Next, adjusting use cases to be more closer to code, and not the other way around, may bring a different view on software, which could be the next promising step towards end-user software engineering. Further, use case driven modularization is superior to common object-oriented modularization, though test driven modularization is even better. It would be interesting to explore other views such as page flow view and their comparison with use case views and test views.

Acknowledgements. This work was partially supported by grants Scientific Grant Agency of Slovak Republic (VEGA) No. VG 1/1221/12, VG 1/0752/14, VG 1/0734/16, VG 1/0774/16, STU Grant scheme for Support of Young Researchers (M. Bystrický), 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, the internal grant scheme of the Slovak University of Technology in Bratislava in support of the teams preparing proposals for Horizon 2020 (V. Vranić), and SOFTEC PRO SOCIETY in cooperation with Softec and Centaur (P. Berta).

References

[1] P. Berta, M. Bystrický, M. Krempaský, and V. Vranić. Employing issues and commits for in-code sentence based use case identification and remodularization. In *Proceedings of 5th European Conference on the Engineering of Computer Based Systems, ECBS 2017*, Larnaca, Cyprus, 2017. ACM.

[2] M. M. Burnett and B. A. Myers. Future of end-user software engineering: Beyond the silos. In *Proceedings of Future of Software Engineering, FOSE 2014*, Hyderabad, India, 2014. ACM.

[3] M. Bystrický and V. Vranić. Development environment for literal inter-language use case driven modularization. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 12–15, New York, NY, USA, 2016. ACM.

[4] M. Bystrický and V. Vranić. Literal inter-language use case driven modularization. In *Proceedings of LaMOD'16: Language Modularity À La Mode, workshop, Modularity 2016*, Málaga, Spain, 2016. ACM.

[5] M. Bystrický and V. Vranić. Modularizing code by use cases and tests for better maintainability. In *Proceedings of Programming 2017 Demos*, Brussels, Belgium, 2017. ACM.

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

[7] 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.

[8] J. Coplien and G. Bjørnvig. *Lean Architecture for Agile Software Development*. Wiley, 2010.

[9] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, July 2012.

[10] I. Jacobson. Use cases and aspects – working seamlessly together. *Journal of Object Technology*, 2(4), July–August 2003.

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

[12] A. Kacofegitis and N. Churcher. Theme-based literate programming. In *Proceedings of 9th Asia-Pacific Software Engineering Conference, APSEC 2012*. IEEE, 2002.

[13] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

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

[15] T. Reenskaug and J. O. Coplien. The DCI architecture: A new vision of object-oriented programming. Artima Developer, 2009.

[16] M. Smith. *Towards Modern Literate Programming*. Honours project report, University of Canterbury, Christchurch, New Zeland, 2001.

[17] J. Zaťko and V. Vranić. Assessing the DCI approach to preserving use cases in code: Qi4J and beyond. In *Proceedings of IEEE 19th International Conference on Intelligent Engineering Systems, INES 2015*, Bratislava, Slovakia, 2015. IEEE.

Selected Papers by the Author

Michal Bystrický and Valentino Vranić. Use Case Driven Modularization as a Basis for Test Driven Modularization. In *6th Workshop on Advances in Programming Languages (WAPL'17)*, September 2017, Prague, Czech Republic, IEEE.

Peter Berta, Michal Bystrický, Michal Krempaský and Valentino Vranić. Employing Issues and Commits for In-Code Sentence Based Use Case Identification and Remodularization. In *Proceedings of 5th European Conference on the Engineering of Computer Based Systems, ECBS 2017*, September 2017, Larnaca, Cyprus, ACM.

Michal Bystrický and Valentino Vranić. Modularizing Code by Use Cases and Tests for Better Maintainability. In *<Programming> '17 Companion, <Programming> 2017, <Programming> 2017 Demos*, April 2017, Brussels, Belgium, ACM.

Michal Bystrický and Valentino Vranić. Literal Inter-Language Use Case Driven Modularization. In *MODULARITY Companion 2016, Companion Proceedings of the 15th International Conference on Modularity, Modularity 2016, LaMOD'16: Language Modularity A La Mode*, workshop, March 2016, Málaga, Spain, ACM.

Michal Bystrický and Valentino Vranić. Development Environment for Literal Inter-Language Use Case Driven Modularization. In *MODULARITY Companion 2016, Companion Proceedings of the 15th International Conference on Modularity, Modularity 2016, Modularity 2016 Demos & Posters*, March 2016, Málaga, Spain, ACM.

Michal Bystrický and Valentino Vranić. Preserving Use Case Flows in Source Code: Approach, Context, and Challenges. *Computer Science and Information Systems Journal (ComSIS)*, 2015.

Valentino Vranić, Jaroslav Porubän, Michal Bystrický, Tomáš Frťala, Ivan Polášek, Milan Nosáľ, and Ján Lang. Challenges in Preserving Intent Comprehensibility in Software. *Acta Polytechnica Hungarica*. 12(7): 57-75, 2015.

Michal Bystrický and Valentino Vranić. Preserving Use Case Flows in Source Code. In *Proceedings of 4th Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2015*, September 2015, Brno, Czech Republic, IEEE Computer Society.