Proceedings of the Federated Conference on Computer Science and Information Systems pp. 693–696
DOI: 10.15439/2017F343
ISSN 2300-5963 ACSIS, Vol. 11

Use Case Driven Modularization as a Basis for Test Driven Modularization

Michal Bystrický and 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
Email: {michal.bystricky,vranic}@stuba.sk

***Abstract*—While in waterfall-like processes changes are expected to happen mostly after the main development has finished, agile approaches have incorporated response to changes into the main development itself, which raises the importance of the ability to respond to changes effectively to a sine qua non. Changes are specified from the perspective of how users actually use systems, i.e., usage scenarios, which does not correspond to a common object-oriented code modularization. In their complete form, usage scenarios can be directly observed in user acceptance tests. Unit tests reveal parts of usage scenarios, too. Logically, tests follow the modularization of the code they are related to. Thus, in common object-oriented code, user acceptance tests, which play a very important role in any kind of software development process and which follow the procedural modularization, would be scattered and, consequently, hard to maintain. In this paper, we propose a new approach capable of achieving test driven modularization, i.e., organizing code according to tests. Besides pure test driven modularization, which can be based on user acceptance tests, unit tests, or both, the approach also enables combining use case and test driven modularization.**

Keywords: modularization, use case, user acceptance test, unit test, test driven development, Cucumber

I. Introduction

WHILE in waterfall-like processes changes are expected to happen mostly after the main development has finished, agile approaches have incorporated response to changes into the main development itself, which raises the importance of the ability to respond to changes effectively to a sine qua non. Changes are specified from the perspective of how users actually use systems, i.e., usage scenarios, which does not correspond to a common object-oriented code modularization.

In their complete form, usage scenarios can be directly observed in user acceptance tests. Unit tests reveal parts of usage scenarios, too. Test driven development [18] and its red-green-refactor loop makes tests very close to code, but the developers have to switch between tests and code hundreds of times to make their test "green." Although a test itself is contained within a small number of mock modules, the tested code remains spread throughout many modules, which significantly increases tracing.

Logically, tests follow the modularization of the code they are related to. Thus, in common object-oriented code, user acceptance tests, which play a very important role in any kind of software development process and which follow the procedural modularization, would be scattered and, consequently, hard to maintain. This is of particular importance, since it is known that user acceptance tests immensely improve code comprehension, as can be seen in the Cucumber approach [7].

User acceptance tests are highly related to use cases [22], and there are several approaches capable of preserving use cases in code, such as DCI (Data, Context and Interaction) [4], aspect-oriented software development with use cases [12], and our own approach of inter-language use case driven modularization [1]. However, all these approaches fail to fully support expressing user acceptance tests because user acceptance tests are actually based on user interfaces, and good use cases are kept independent of user interface details.

Use case driven modularization and mechanisms that are used to achieve it remain a good basis for establishing a new kind of modularization—test driven modularization—which we propose in this paper. Keeping user interface code coupled with the corresponding application logic code is essential for this kind of modularization. Modern user interfaces tend to be written in dedicated languages and this is where inter-language use case driven modularization, which enables mixing fragments of code written in different programming languages in so-called virtual files with their continuous merging into compilable and executable units [1], provides the necessary capabilities missing in other approaches to preserving use cases in code.

The rest of the paper is organized as follows. Section II explains briefly how inter-language use case driven modularization is implemented. Section III proposes the new approach that enables test driven modularization. Section IV compares our approach to related work. Section V concludes the paper.

II. Inter-Language Use Case Driven Modularization

Several approaches capable of organizing code according to use cases are available. However, since each language is intended for its specific use, an approach suitable for test driven modularization has to support mixing different languages in program modules. Literal inter-language use case driven modularization [1] enables exactly this. In this approach, code for a use case is located in a *use case module*, which is simply a file: a *use case file*. On top of the use case file there is a

use case in its text form, a *use case text*, written in form of a comment. Under the use case text is a class which represents the use case, and its methods represent the use case steps. A framework executes the methods representing the use case steps based on the use case text. The mapping between the text of the steps and the corresponding methods is maintained with a naming convention: method names are actually derived from the use case step text by turning it into the camel case format.

The use of different languages is enabled by so-called *virtual files*. Virtual files are defined using particular comment conventions directly within the methods that implement use case steps. All virtual files are extracted out of use case files, merged, and saved by a *preprocessor*, which also resolves possible virtual file duplicities and gathers the code from partial virtual files that contain partial namespaces or classes. The preprocessor saves the files containing the merged code, which are actual files, to their *virtual file paths* as indicated in the corresponding virtual files. Subsequently, the merged code can be executed, which may require a compilation depending on the programming language.

Additions and alterations of virtual files in use case files are propagated to the merged code and to other use case files by the preprocessor. Accordingly, deletions and alterations in the merged code are propagated to use case files, too. Since there is no way of knowing to which use case are additions to the merged code related, these additions are not propagated to virtual files in use case files. All changes to use case files and merged code, including the direct changes by developers, are displayed to developers and logged. This process of change propagation is actually *synchronization*, as will be referred to further.

III. Including Tests in Use Cases

Test driven modularization aims at organizing code according to tests keeping their representation in code. For this, the synchronization mechanism from inter-language use case driven modularization was employed accompanied by the new mechanism of use case coverage calculation that we propose here.

Via test driven modularization, our approach provides yet another view on software under development. This view may be combined with literal inter-language use case driven modularization, but what is always available is the modularization of the merged files (recall Section II). Developers can switch between these modularizations and get the perspective that suits best their current needs. Of course, all three modularizations have to be maintained. However, the tools capable of synchronizing changes can automate this process.

Section III-A explains the details of writing tests within use case modules. To enable keeping track of how well tests cover use cases, the approach embraces a continuous calculation of this value, which is described in Section III-B. Writing user acceptance tests is described in Section III-C. Test representation is described in Section III-D. Section III-E provides a brief information on the experience we have in applying our approach.

A. Tests in Use Case Modules

In literal inter-language use case driven modularization, use case steps appear in the classes representing use cases as comments along with the related code. but comments are hard to write and maintain and development environments do not support code completion and syntax highlighting for code inside of comments. To address this problem, we propose to use the Markdown format in use case files. Consider the *Add Product into Cart* use case:

````
# Use case Add Product into Cart
## Main scenario
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) {...} })
```

# Tests
## tests/features/cart.feature
```feature
Feature: Shopping cart

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

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

## tests/unit/cart.js
```js
...
Cart.empty();
assert(
  Cart.getAll().length === 0,
   "The empty cart should have
   zero items");
Cart.add("1");
assert(
  Cart.getAll().length === 1,
   "...");
```
````

As can be seen, the actual use case implementation follows the use case text. This part constitutes a separate section in Markdown. It consists of virtual files (recall Section II) with virtual file paths represented by second level Markdown headers.

The tests for the use case, placed into a separate, *test section* of the use case file, follow the code section. The test section consists of virtual files, thus the same synchronization mechanisms apply to tests as in the code section. The example contains two tests: the *Adding Products into Cart* user acceptance test written in the Cucumber's Gherkin language [7] and the `Cart` unit test.

For the merged files, we used common object-oriented modularization with the Model-View-Controller architectural pattern, but any other kind of modularization, such as functional or procedural, can be used as well. Use case driven or test driven modularization can be built upon any kind of underlying modularization.

If a test from the test section is moved to the top of a use case file, the preprocessor treats virtual files in this section as use

cases in the use case section in use case driven modularization, where each line of the test is treated as a use case step in the main flow.

B. Use Case Coverage

Each use case should be covered by the corresponding code. This can be measured as a percentage of the words from the use case found in the declarations residing in its code. Also, each use case should be covered by the corresponding tests. In the same way, the coverage of use cases by the tests can be measured as a percentage of the words from the use case found in the declarations residing in its tests.

With each change in use case files, the preprocessor recalculates and displays in its console output the coverage for each use case step along with the words that are missing in the code and in the tests. The missing words are the words present in use cases, but not covered by code or tests. Conjunctions, prepositions, and articles can be ignored, which can be specified in the *ignored words file*.

Missing words can also be pseudo-covered by code if they are included in comments inside of four asterisk symbols (denoting the bold font in Markdown). In the same way, they can be pseudo-covered by tests, too. Consider this virtual file as an example:

```
## view/bank-transfer.html
```html
<!-- **provide bank transfer instructions** -->
<p>Please send the payment to the address below.</p>
```
```

Although the "provides" word is missing in the corresponding use case step, the preprocessor captures its form "provide" introduced in the comment and considers it as being covered by code. Here, the difference algorithm [15] is used. The similarity of 70% or higher is considered to indicate the words are the same.

By presenting the percentage of how well use cases are covered in both code and tests encourages developers to work on increasing it, which is achieved by a better test and use case driven modularization.

Conveniently, the links between use cases and code—i.e., between a use use case step and a line of code or a line of test—can be visualized by using our tool (see the relationship walkthrough video at https://youtu.be/N1hbu3K0yp4).

C. Writing User Acceptance Tests

As can be seen from the *Add Product into Cart* use case example introduced at the beginning of this section, user acceptance tests closely follow use cases, which makes them appropriate for use case driven modularization. User acceptance tests can also be written in the form of use cases to ensure they cover the corresponding use cases fully. The main steps of a user acceptance test would then actually be the same as the steps of a use case. Each such step is followed by a sequence of actual test steps that specify the corresponding testing actions. This is the first step of the test from our *Add Product into Cart* use case example: "User selects to add a product into cart" is followed by the following actual test steps:

```
When I click on "Add into cart"
Then I should see "Test product"
```

The test steps are then expressed by code. Here is the code for the use case step:

```
this.When(/^user selects to add a product
            into cart$/, function () {
  this.clickOn("Add into cart");
  this.shouldSee("Test product");
```

D. Writing Unit Tests

Unit tests follow the modularization of object-oriented code which is different than use case driven modularization. However, unit tests are capable of testing use case steps at least partially. Consider the following example of the test for the "System saves the product into cart" step of the *Add Product into Cart* use case:

```
function testSaveProductIntoCart() {
  product = {...}
  assert(Cart.save(product) === true,
    "System should save the product into cart");
```

A use case step can be implemented as an exception, too. For this, the `Cart.save()` call in the `testSaveProductIntoCart()` function should be wrapped into a try-catch block. If the call fails, an assert function will raise an exception with the "System should save the product into cart" message from the use case step. Notice that all the words are the same as in the use case step except for "should," which is characteristic for tests.

E. Experience with the Approach

Being encouraged by the positive results of two studies of our approach to use case driven modularization, one of which embraced our own e-shop application with its seven use cases (including the use case presented in Section III-A), while the other one consisted of remodularizing the well-known OpenCart e-commerce platform into 55 use cases,[1] so far, we successfully applied our approach to test driven modularization to our own e-shop application. We implemented a combined use case driven and test driven modularization version and a pure test driven modularization version (based on user acceptance tests).[2]

IV. Related Work

DCI [4], [19], aspect-oriented software development with use cases [11], [12], InFlow [2], and behavioral programming [8] preserve use cases in code. Each approach enforces a specific use case representation in code, e.g., DCI does this via roles, while aspect-oriented software development with use cases employs aspects. In our approach, use case representation can be freely chosen while the approach still achieves use case representation in code. Approaches to generating object-oriented code from use cases [6], [23] do not actually represent use cases in code.

[1] See github.com/useion/opencart.

[2] See youtu.be/zK8QKsIOkOg and github.com/useion/useion-e-shop (the test modules can be found in the context/behavioral folder). More information is available at useion.com.

Software artifacts were combined previously in literate programming [14], too. Literate programming brings code into documentation, but not tests into code. The code is extracted and combined by the noweb tool [13], which is basically a preprocessor or code generator [5]. This is similar to our approach and literate programming is even capable of expressing use case modules as our approach. However, without synchronization, code duplication would occur, which would be unbearable.

Code defragmentation can be achieved also by applying Object Teams [10], subject-oriented programming [17], and symmetric aspect-oriented composition [3], [9], but none of these approaches achieves this at the level of multiple languages, as we do.

Different structuring of code provides different views on software. Although dynamic structuring has been reported [16], [21], it was not achieved at the file system level, but using a particular editor. Our approach provides use case and test views, which are synchronized at the file system level and can be used simultaneously.

The Cucumber project [7] enables test execution based on the text specification written in Gherkin, but each step has to be expressed in code. This is similar to our approach where code can be structured according to the text specification of use cases while each their step has to be expressed in code.

While in our approach a web based interface is used to display the links between use cases or user acceptance tests and code, exposing them directly in the development environment using, for example, information tags [20] could help in making developers pay more attention to use case coverage.

V. Conclusions

In this paper, we propose a new approach capable of achieving test driven modularization. It enables organizing code according to tests. The approach employs inter-language use case driven modularization, which provides a good basis for keeping code modularized according to user acceptance tests and enables mixing different kinds of languages, which is particularly useful when dedicated languages are used for user interface development. Besides pure test driven modularization, which can be based on user acceptance tests, unit tests, or both, the approach also enables combining use case and test driven modularization.

Acknowledgments

The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under the grant No. VG 1/0752/14. This contribution/publication is also a partial result of the 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. Michal Bystrický was supported by the STU Grant scheme for Support of Young Researchers.

References

[1] 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. doi.org/10.1145/2892664.2893465.

[2] 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. doi.org/10.2298/CSIS151101005B.

[3] J. Bálik and V. Vranić. Symmetric aspect-orientation: Some practical consequences. In *Proceedings of NEMARA 2012: International Workshop on Next Generation Modularity Approaches for Requirements and Architecture, at AOSD 2012*, Potsdam, Germany, 2012. ACM. doi.org/10.1145/2162004.2162007.

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

[5] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[6] J. Franců and P. Hnětynka. Automated code generation from system requirements in natural language. *e-Informatica Software Engineering Journal*, 3(1):72–88, 2009.

[7] S. Garg. *Cucumber Cookbook*. Packt Publishing, 2015.

[8] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Communications of the ACM*, 55(7):90–100, July 2012. doi.org/10.1145/2209249.2209270.

[9] W. H. Harrison, H. L. Ossher, and P. L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Research, 2002.

[10] S. Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.

[11] I. Jacobson. Use cases and aspects – working seamlessly together. *Journal of Object Technology*, 2(4), 2003. doi.org/10.5381/jot.2003.2.4.c1.

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

[13] A. Johnson and B. Johnson. Literate programming using (noweb). *Linux Journal*, 1997(42es), 1997.

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

[15] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[16] M. Nosál'. Sieve source code editor. https://github.com/MilanNosal/sieve-source-code-editor, 2015.

[17] H. Ossher, W. Harrison, F. Budinsky, and I. Simmonds. Subject-oriented programming: Supporting decentralized development of objects. In *Proceedings of 7th IBM Conference on Object-Oriented Technology*, 1994.

[18] M. Rahman and J. Gao. A reusable automated acceptance testing architecture for microservices in behavior-driven development. In *2015 IEEE Symposium on Service-Oriented System Engineering, SOSE 2015*, 2015. doi.org/10.1109/SOSE.2015.55.

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

[20] K. Rástočný and M. Bieliková. Empirical metadata maintenance in source code development process. In *4th Eastern European Regional Conference on the Engineering of Computer Based Systems*, 2015. doi.org/10.1109/ECBS-EERC.2015.13.

[21] M. Sulír and M. Nosál'. Sharing developers' mental models through source code annotations. In *Proceedings of 2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015*, Łódź, Poland, 2015. IEEE. doi.org/10.15439/2015F301.

[22] P. Zielczynski. Traceability from use cases to test cases, 2006. IBM developerWorks, https://www.ibm.com/developerworks/rational/library/04/r-3217/.

[23] M. Śmiałek, N. Jarzębowski, and W. Nowakowski. Translation of use case scenarios to Java code. *Computer Science*, 13(4):35–52, 2012. doi.org/10.7494/csci.2012.13.4.35.