Development Environment for Literal Inter-Language Use Case Driven Modularization

Michal Bystrický Valentino Vranić

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 2, Bratislava, Slovakia
{michal.bystricky,vranic}@stuba.sk

Abstract

Commonly, during programming the code related to use cases becomes scattered across different modules and at the same time the code related to different use cases becomes tangled. This way, it is hard to follow the intent, which is otherwise well comprehensible in use cases. In this paper, we demonstrate a development environment for literal inter-language use case driven modularization. The environment enables to preserve use cases and their steps and have each use case text and related code focused in one file. For this, code instrumentation at three levels is involved: continuous processing, preprocessing, and execution. The approach itself requires also execution control provided by a dedicated framework. Many aspects of the program can be controlled directly from the use case text. At the same time, it is comprehensible to a wide range of stakeholders. A layered 3D layout of the use case dependencies is provided in the environment (`https://bitbucket.org/bystricky/literal-use-cases`, `https://www.youtube.com/watch?v=R4ArqH4ZdgI`).

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—Languages; D.2.6 [*Software Engineering*]: Programming Environments; D.2.10 [*Software Engineering*]: Design; D.3.4 [*Programming Languages*]: Processors—Preprocessors

General Terms Design, Documentation, Languages

Keywords use case, modularization, flow of events, intent, DCI, aspect-oriented programming

1. Introduction

Commonly, during programming the code related to use cases becomes scattered across different modules and at the same time the code related to different use cases becomes tangled. This way, it is hard to follow the intent, which is otherwise well comprehensible in use cases [15]. Moreover, modern software systems are rarely developed in one language. For example, in web applications different software languages are used, some of which serve the purpose of expressing the application logic and are usually referred to as programming languages, while other, markup languages are used to express the user interface to be rendered by web browsers.

Attempts to modularize code according to use cases [6, 9] have demonstrated that this is not fully achievable without introducing some kind of code instrumentation. We proposed literal inter-language use case driven modularization [3] that involves only as much of code instrumentation as necessary to achieve literal use case modularization while allowing for this to be balanced with differently modularized code as desired.[1] Thus, the approach is non-invasive and adaptable. Here, we present our literal inter-language use case driven modularization development environment.[2]

Section 2 presents briefly literal inter-language use case driven modularization. Section 3 describes the role of the use case text. Section 4 explains how are use case steps implemented. Section 5 discusses the related work. Section 6 concludes the paper.

2. The Approach in a Nutshell

In our approach, each use case is maintained in a separate file written in the programming language that plays the role of the *bearing language*. In our current implementation, it is JavaScript. However, we have successfully experimented with Java as the bearing language. Although the file itself could play the role of a module, we have chosen to use classes for this. Thus, each use case is implemented within a class (referred to as *use case class* further).

The *use case text* is placed at the top of the file as a comment. The Markdown markup language is used to denote the structure of the use case text. Cockburn's use case notation [5] is employed. The use case text is comprehensible to a wide range of stakeholders, but it also plays an active role in the program. All three use case relationships—include, extend, and generalization/specialization—can be expressed directly in the use case text and without the need to use the actual code. Use case step parameters can be expressed in the use case text, too.

The use case text is followed by the actual use case step implementation. Each use case step is implemented as a use case class method (referred to as *use case step method* further). Some of the

[1] While we rely on some aspects of our approach presentation as provided in our LaMOD'16 workshop paper [3], our focus here is on the technical aspects of the supporting development environment not covered by the LaMOD'16 workshop paper.

[2] See `https://bitbucket.org/bystricky/literal-use-cases` (download) and `https://www.youtube.com/watch?v=R4ArqH4ZdgI` (video demonstration).

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
© 2016 ACM. 978-1-4503-4033-5/16/03...
http://dx.doi.org/10.1145/2892664.2893465

application logic is programmed in the bearing language, but other languages, including markup languages to express the user interface, can be used, too. For this, each such *fragment* is embedded into the bearing language code using comments. A fragment actually represents a *virtual file* whose name is provided after the opening comment symbol.

Code instrumentation at three levels is involved: continuous processing, preprocessing, and execution. Continuous processing ensures all equally named virtual files are kept the same by propagating the changes to any one of them to all other ones.

In preprocessing, virtual files are merged before the code can be processed further by the corresponding common language tools (compiled and executed or interpreted, depending on the language). Also, the use case text is transformed into an internal form, the input parameters are included in the actual use case step code, use case relationships are translated into code, and the predefined, generic main method necessary as a starting point for the application is copied.

The execution of the processed and preprocessed code is controlled by the framework. In this, the framework relies on the internal form of the use case text generated by the preprocessor. Individual use cases are activated from the user interface controls.

Our development environment adopts a typical integrated development environment layout with a web based user interface. In addition to a code editor, file browser, code inspector, and console output, it provides also a layered 3D layout of the use case dependencies (see the next section). The server side is implemented in JavaScript and uses the Node.js and Express.js frameworks. The client side is written in JavaScript with the AngularJS framework and Twitter Bootstrap, a CSS framework. CodeMirror is used as a code editor. The layered 3D layout is implemented in Three.js, a WebGL based JavaScript library. The internal form of the use case text is in JSON. We used the environment to develop 25 use cases of a real web application [3].

3. Writing the Use Case Text

Here is an example of the use case text part of a use case implementation in our development environment:

```
/**
 * Usecase Show Articles
 * =====================
 *
 * A user wants to display the list of articles.
 *
 * Actors
 * ------
 * User: any user visiting our site
 *
 * Triggers
 * --------
 * When the "saving article" extension point is
     reached
 *
 * Main success scenario
 * ---------------------
 * 1. User selects to show articles
 * 2. System displays articles
 *
 * Postconditions
 * --------------
 * User can see articles.
 */
```

As can be seen, the use case text is partitioned into the sections according to Cockburn's notation. The main part in each use case is its main success scenario. The steps are used to form the names of use case step methods in the camel case format (omitting the actor). This is used to bind use case steps to their implementation (explained in the next section).

Use case steps can have input parameters that are typically bound to user interface controls in their implementation. However, use case steps serve a more sophisticated purpose: to express the include relationship between use cases (which can be visualized in a layered 3D layout as depicted in Figure 1). For this, the `Include` keyword with the use case name as a parameter can be used, e.g.:

7. Include `'Show articles'`.

Alternatively, if the use case step implementation is omitted, the use case name can be provided right after the actor name (the notation is not case sensitive):

7. User show articles.

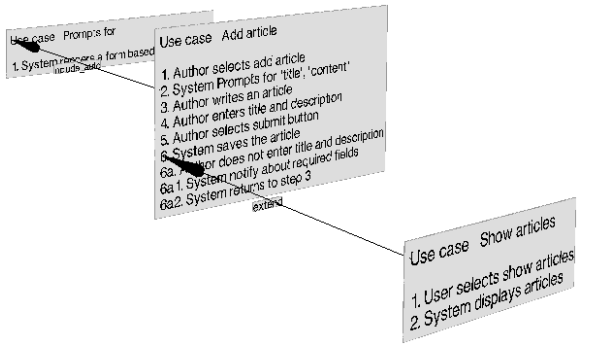

Figure 1. The layered 3D layout of the use case dependencies.

Technically, each include relationship can be expressed as an extend relationship. For this, an extension point has to be exposed instead of invoking the corresponding use case —in this case *Show Articles*—directly:

```
* Extension points
* ------------------
* 'saving article': step '6.'
```

The *Show Articles* use case would be set to be activated upon reaching this extension point:

```
* Triggers
* ---------
* When the 'saving article' extension point is reached
```

Only parts of the use case text are active. For example, the trigger description is scanned by the preprocessor for the *extension point* string and a string in simple quotation marks. The rest is ignored. We continue experimenting with different forms that the use case text can take. While we are interested in improving its expressiveness, we strive for the flexibility in its syntax. Its relaxed form enables to let developers use their own style of expressing use cases making an illusion that the development environment understands whole expressions while it effectively picks out the essential and leaves out the rest.

4. Use Case Step Coding

The use case text is followed by the actual step implementation. The names of use case step methods are formed by transforming the

use case step text into the camel case format (omitting the actor). This is used to bind use case steps to their implementation.

The step implementation gathers all the code fragments of each step. Each code fragment in the step implementation may be in a different programming language. For example, the code for the second step of the *Show Articles* use case is written in the bearing language JavaScript and two other languages, PHP and HTML:

```
this.displaysArticles = function (done) {
  /** Partial model/Article.php
    <?php
    class Article {
      public $title;
      public $content;
      public static function find() { ... } }
  */
  /** Partial controler/getArticles.php
    <?php
    require 'Article.php';
    echo json_encode(Article::find());
  */

  var _this = this;
  get(this.metadata.pathDir+'/getArticles.php', function (
       data) {
    var articles = JSON.parse(data);

    /** Partial view/articles-list.html
      {% for(var i=0, article; article=o.articles[i
           ]; i++){ %}
        <div class="article" data-id="{%=i%}">
          <h3>{%=article.title%}</h3>
          <p>{%=article.content%}</p>
        </div>
      {% } %}
    */

    get(_this.metadata.pathDir+'/articles.html', function (
         data) {
      var gen = tmpl(data, {articles: articles});
      document.getElementById('content').innerHTML= gen;
    });
  });
}
```

Each language is maintained within one or more virtual files each of which is in the form of a comment starting with the `Partial` word followed by the *virtual file path*.

On file saving, the preprocessor extracts and merges virtual files into real files based on their virtual file paths. How the code in virtual files is merged is language dependent. For example, the elements of the equally named classes in equally named PHP virtual files would be merged into one resulting class. Consider the step of the *Add Article* use case in which an article is being saved. From the perspective of this use case, the article title, content, and the actual save operation are the only things that matter, so the corresponding partial class looks as follows:

```
/** Partial Article.php
  <?php
  class Article {
    public $title;
    public $content;
    public function save() { ... }
  }
*/
```

The *Review Article* use case contains a step where an article is obtained by its ID. This is a different perspective on the same `Article` class:

```
/** Partial Article.php
  <?php
  class Article {
    public static function getById($id) { ... }
  }
*/
```

The preprocessor merges these two virtual files:

```
<?php
class Article {
  /* Fragment Add Article */
  public $title;
  public $content;
  public function save() { ... }

  /* Fragment Review Article */
  public static function getById($id) { ... }
}
```

As can be seen, use cases are traceable from the generated code, though developers normally do not get in touch with it. This may be used in refactoring conventional code into the code modularized by use cases.

5. Related Work

Our development environment is related to several other experimental development environments. Object Teams [8] enables to program with roles and their bundles called teams. Teams gather partial classes in order to create specific functionality, which is similar to our partial classes bundled into use case classes, but there are no indications that Object Teams support use case steps.

The ReDSeeDS development environment [16] enables to transform use cases written in a requirements specification language called RSL into UML models and Java code. This is somewhat similar to the translation of the use case text into the internal form to be used by the framework, but in our approach, the use case text does not have to follow so rigid syntax. Furthermore, ReDSeeDS does not preserve use cases in code (nor in UML models).

Dynamic code structuring [11] implemented within Sieve Source Code Editor [10, 14] makes possible to have different perspectives on code through an explicit concern representation. Use cases could be one such perspective, but not at the level of individual use case steps.

None of the mentioned development environments supports multiple languages. In general, our approach, with its roots in our prior work [2], is related to DCI [6, 13], aspect-oriented programming with use cases [9], and subject-oriented programming [12] and symmetric aspect-oriented composition in general [4, 7], which is covered elsewhere [3].

6. Conclusions and Challenges

The development environment for literal inter-language use case driven modularization was demonstrated in this paper. The environment enables to preserve use cases and their steps and have each use case text and related code focused in one file. Code instrumentation

at three levels is involved: continuous processing, preprocessing, and execution.

Many aspects of the program can be controlled directly from the use case text. At the same time, it is comprehensible to a wide range of stakeholders. This brings us close to the ideas of end-user software engineering [1]. By gradually transferring the control to the use case text, we can make the whole approach even more accessible to non-professionals. To improve working with extensive applications, the layered 3D layout of the use case dependencies could be made editable and presented using stereoscopic means.

Acknowledgments

The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under grants VG 1/0734/16 and VG 1/0774/16. It 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.

References

[1] 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.

[2] M. Bystrický and V. 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*, Brno, Czech Republic, 2015. IEEE CS.

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

[4] 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.

[5] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.

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

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

[8] S. Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007. ISSN 1570-5838.

[9] I. Jacobson and N. Pan-Wei. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2004.

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

[11] M. Nosáľ, J. Porubän, and M. Nosáľ. Concern-oriented source code projections. In *Proceedings of 2013 Federated Conference on Computer Science and Information Systems, FedCSIS 2013*, Kraków, Poland, 2013. IEEE.

[12] 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.

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

[14] M. Sulír and M. Nosáľ. 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.

[15] V. Vranić, J. Porubän, M. Bystrický, T. Frťala, I. Polášek, M. Nosáľ, and J. Lang. Challenges in preserving intent comprehensibility in software. *Acta Polytechnica Hungarica*, 12(7):57–75, 2015.

[16] M. Śmiałek, N. Jarzębowski, and W. Nowakowski. Translation of use case scenarios to Java code. *Computer Science*, 13(4):35–52, 2012.