# 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

Use cases are a practically proven choice to concisely and precisely express what highly interactive systems do. Several attempts have been made to modularize code according to use cases. None of these approaches is able to gather all the code related to a use case in one module and to reflect its steps. However, to allow for code to be modularized according to use case flows and their steps, an instrumentation environment is necessary. In this paper, literal multi-language use case coding based on defragmentation is proposed. The approach aims at fully preserving use case flows in as comprehensible form as possible. The steps of use case flows appear directly in the code as comments. Despite being comments, the steps are active, which is achieved by a dedicated preprocessor and framework. The detailed step implementation gathers all the code fragments of each step each of which may be in a different programming language.

***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.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages; 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

In all but simplest programs, code is commonly not gathered in one unit. Different approaches to programming employ different units of modularization. In object-oriented programming, these include classes and methods, but also packages at the higher level and code blocks at the lower level. The aim of code modularization is to enable the separation of concerns, which is necessary in any organized thinking and problem solving [8]. As such, it should improve code comprehension, but this is not always the case since modularization is often put into the service of technical concerns, such as maintainability and reuse.

Undoubtedly, documentation improves code comprehension, but it introduces another artifact to be maintained and synchronized with code, including maintaining the links between to the documented code [16, 21]. With internal documentation, commonly known as comments, at least the documentation to code links cease to be an issue. Literate programming [17] brings internal documentation to another level. The documentation takes the primacy over the code and the code appears as if it is being embedded into the documentation. The structure of such documentation necessarily copies the code modularization along with the comprehensibility problems it brings.

But what kind of modularization is appropriate for documentation? With respect to common, highly interactive software systems, such as virtually all custom developed information systems, use cases are a practically proven choice to concisely and precisely express what such systems do. Even agile and lean approaches to software development, which are quite reluctant regarding documentation, recognize their usefulness, though often in a shorter form or in an exemplary style of user stories. The idea of describing the interaction between the user and the system in the form of steps has proven over time to be easily readable and comprehensible even by end users At the same time, use cases resemble algorithms [6] with use case step sequences—known as flows of events, use case flows, or simply flows—as their crucial part. This dual nature makes use cases a very good basis for self documented code. This idea might be observed in modeling, too: UML provides the collaboration element as a way of modularizing models according to use cases.

While use cases might not be appropriate for certain less interactive kinds of systems, such as batch processing or embedded systems, or where interaction is well defined and not likely to change [6], we focus here exactly on highly interactive systems in which it pays off to engage use cases in exploring interactivity prior to investing any significant effort into coding [6]. Another point worth consideration are changes. They mainly come as change requests, which are usually phrased in terms of the system usage, which is the language very close to use cases. In the code modularized according to use cases, tracing the affected places would be much easier.

Several attempts have been made to modularize code according to use cases ranging from improving their traceability [10, 13], through concentrating code of each use case into one module using aspect-oriented programming [14, 15], to preserving use case steps [6], including our own prior work [2]. None of these approaches is able to gather all the code related to a use case in one module and to reflect its steps. In spite of that, DCI (Data, Context and Interaction) [6] is particularly interesting in how it manages to come close to the ideal of having human comprehensible use case



*MODULARITY Companion'16*, March 14–17, 2016, Málaga, Spain

http://dx.doi.org/10.1145/2892664.2892680

representation in pure code by employing role based programming in common object-oriented programming languages without having to modify them. However, use case flows remain fragmented across the roles [2]. Our prior work took initial steps in a little bit different direction, acknowledging the necessity of providing an instrumentation environment to allow for code to be modularized according to use case flows and their steps [2].

The rest of the paper presents a novel approach to inter-language use case driven modularization. Section 2 presents the basics of having use cases as code modules. Section 3 describes how multiple languages are employed in implementing use case steps. Section 4 explains what kind of processing is necessary for this. Section 5 describes how are the use case relationships supported. Section 6 discusses our approach in a broader context. Section 7 describes the implementation and evaluation. Section 8 discusses the related work. Section 9 concludes the paper.

## 2. Use Cases as Code Modules

To overcome use case flow fragmentation, we propose to maintain a whole use case in a single file. Here is an example:

```
/**
 * UseCase Add Article
 * ===================
 *
 * A user adds a new article into the database.
 *
 * Preconditions
 * --------------
 * Author has logged onto the system
 *
 * Actors
 * ------
 * Author: Writes an article
 *
 * Main success scenario
 * ---------------------
 * 1. Author selects to add article
 * 2. System prompts for the 'title' and 'content'
 * 3. Author writes an article
 * 4. Author enters the title and content of the article
 * 5. Author selects to submit
 * 6. System saves the article
 * 7. System displays the article
 *
 * Extensions
 * ----------
 * 4a. Author doesn't enter the title and description
 * 4a1. System notifies about required fields
 * 4a2. System returns to step 3
 *
 * Postconditions
 * ---------------
 * User can see the article
 */

(function () {
  this.authorHasLoggedOntoTheSystem = function (done) {...};
  this.selectsAddArticle = function (done) {...};
  this.promptsFor = function (args, done) {...};
  this.writesAnArticle = function (done) {...};
   .
   .
});
```

Since the use case text from provides a good overview suitable for both software developers and all other stakeholders, it 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.

A use case may contain several flows. Cockburn's notation [5] is employed here. The main flow is denoted as *Main success scenario*. Extension flows are introduced in the *Extension* section. Each extension is indicated by the triggering step number and then followed by the sequence of alternative steps. The above example contains an extension that resolves the problem with the unprovided input.

The use case text is followed by the implementation of steps each of which is represented by a method (in our example, the method bodies are omitted). All *use case step methods* for one use case, including its extension flow steps, are gathered in one class. This *use case class* is implemented in JavaScript using its function approach. This is peculiar to JavaScript. In case of a class based language such as Java, the common class construct would be used.

The names of use case step methods are formed by transforming the use case step text into the camel case format. This is used to bind use case steps to their implementation.

Each precondition and postcondition is introduced at a separate line in the corresponding section. Unlike with steps, their implementation may be omitted (in our example, only the postcondition is implemented). If included, the same approach as with use case step methods is used, including the lexical binding mechanism. Use case short description is not active in the implementation.

The use case step methods have one or two input parameters. The `done` parameter is a callback method to return the values in asynchronous calls.

The user input may be indicated directly in use case steps. Single quotation marks are used for this. Consider step 2 in our example. The *title* and *content* are the input parameters whose value is to be provided by the user. This is just a more convenient way than introducing them directly as the corresponding use case method parameters:

```
this.promptsForTitleAndContent =
   function (args = ['title', 'content'], done) {...};
```

The use case steps that represent actor actions such as "User selects an article." cannot be expressed literally. They are captured by the listeners attached to appropriate controls. For the step we mentioned, a listener would be attached to the select article button. When the article button is clicked, the listener executes the corresponding code. Afterwards, the execution proceeds with the next use case step.

## 3. Engaging Multiple Languages

In web applications, front end and back end are typically implemented in different languages due to development team experience, programmer or other costs, libraries support, or performance, and this is so in our case, too. Consider the implementation of step 6 of the use case introduced the previous section:

```
this.savesTheArticle = function (done) {
  /** Partial Article.php
    <?php
    class Article {
      public $createdAt;
      public $publishedAt;
      public $title;
      public $content;
      public function save() { ... } }
  */
  /** Partial saveArticle.php
    <?php
    require 'Article.php';
    $article = new Article();
    $article->title = $POST['title'];
    $article->content = $POST['content'];
    $article->createdAt = new DateTime();
    echo $article->save();
  */
  var _this = this;
  get(_this.metadata.pathDir+'/saveArticle.php',function(result){
    result = JSON.parse(result);
    /** Partial success.css
      .alert { border: 1px solid red }
```

```
    */
    /** Partial success.html
      <link rel="stylesheet" type="text/css" href="success.css">
      <div class="alert alert-success" role="alert">Successfully
           saved</div>
    */
    if (!result.err) {
      get(_this.metadata.pathDir+'/success.html',function(data){
        var gen = tmpl(data, {});
        document.getElementById('result').innerHTML = gen;
      });
    } else { ... }
    done();
    _this.continue([
      document.getElementById('article-title').value,
      document.getElementById('article-content').value
    ], function () {});
  });
};
```

Inside the JavaScript code, there are *code fragments* implemented in three other languages: PHP, HTML, and CSS. JavaScript plays the role of a bearing language. The code fragments contain the parts of code of regular modules related to the particular step.

Physically, the respective regular modules would normally be contained within their files, so the code fragments can be seen as partial, *virtual files*. Each virtual file is delimited by comments that indicate the actual file it contributes to. By this, the virtual files can be extracted and merged into actual files that can be processed further by the corresponding common language tools (compiled and executed or interpreted, depending on the language).

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. In CSS, the same is applied with respect to selectors and declarations: all declarations related to the same selector are merged under this selector.

## 4. Processing

To achieve the literal inter-language use case driven modularization as described in previous sections, code instrumentation at three levels is involved: continuous processing, preprocessing, and execution. Use cases may involve the same virtual files. For example, several use cases may employ the same code for document printing. For convenience, they are editable from any of these use cases. However, editing manually all the virtual file copies would not be convenient. Therefore, 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.

Before the code can be processed further by the corresponding common language tools (compiled and executed or interpreted, depending on the language), it is necessary to merge the virtual files, as discussed in the previous section, It is also necessary to extract the information on the order in which the use case step methods are to be executed. This is performed according to the lexical binding between the use case steps in the use case text and the corresponding methods (recall the camel case, Section 2). Preprocessing involves making the input parameters included in the actual use case step code as described in Section 2. It also involves translating use case relationships—discussed in the next section—into code. Finally, preprocessing involves copying the predefined, generic main method necessary as a starting point for the application.

Use cases are activated from the user interface controls. These are actually defined directly in use case code, as has been discussed in the previous section. To execute use case steps in the corresponding order, the underlying framework is involved. It relies on the order of the steps extracted during preprocessing. Note that the framework can be entirely replaced by weaving additional code at the preprocessing level.

## 5. Use Case Relationships

All three relationships between use cases in use case modeling—include, extend, and generalization/specialization—are supported by our approach. For this, special syntax is used in the use case text. Input parameters demonstrated in Section 2 are a part of this.

A use case can activate another use case in a procedural call fashion via the *include* relationship. This is implemented in the `continue` method of the framework, which is normally used to indicate that the step has finished and to make the control flow proceed to the next step. This behavior is altered to executing another use case by providing this use case name as an argument to the `continue` method. A more convenient way is simply to omit the step implementation. This would make the preprocessor interpret the step text as an include construct and generate the corresponding code. Either the `Include` keyword with the use case name as a parameter can be used:

```
Include 'Find an article'.
```

or the use case name can be provided right after the actor name (the notation is not case sensitive):

```
User find an article
```

The *extend* relationship enables for a use case to affect another use case in one of its exposed extension points in an asymmetric aspect-oriented way. This is implemented similarly to the include relationship: the `continue` method calls the extending use case when the extension point and trigger are reached. Again, a more convenient way is to define the extension points and triggers in the use case text and let the preprocessor generate the corresponding code. Suppose the *Review an Article* use case is extended by the *Find an Article* use case. For this, the *Review an Article* use case defines the corresponding extension point:

```
'Finding article': step '3.'
```

The *Find an Article* use case, in the body part Triggers, must define it should be triggered when the `finding article` extension point is reached:

```
When the extension point 'Finding article' is reached.
```

The *generalization/specialization* relationship makes possible for a use case to reuse an existing use case. In this, it can modify some of its behavior. The modification can be defined explicitly by use case steps similarly to method overriding. We employed this approach. In code, it corresponds directly to inheritance and method overriding, as use cases are implemented as classes and their steps as methods. As with two other relationships, generalization/specialization is also supported by the preprocessor. The generalization/specialization is indicated directly in the use case text header:

```
UseCase Add Article with Review specializes Add Article
```

The specialized use case implements only the affected steps, while the use case text must contain also the inherited steps with the `(inherited)` prefix, e.g.:

```
(inherited) System prompts for the title and content.
```

## 6. Broader Context

The approach proposed here can be used in different ways. First of all, the scope of its application can vary. As has been mentioned in the introduction, use case driven modularization is not appropriate

in all cases. Moreover, it may be appropriate for only a part of the system. The reasons may be organizational, but probably the most interesting case is employing use case driven modularization for a new functionality with an existing code base. If the framework is imported and the main method is omitted, there is no interaction with such code. The existing code can be called from within use case step implementation.

What functionality is to be considered a use case is something that could be debated over. Although it is evident that overusing of the include relationship leads to the pitfall of functional decomposition, the need for distinguishing use case granularity is generally accepted be it via so-called include-only use cases [15], lower layer use cases [5], or habits [6]. Our approach enforces no methodological viewpoint in this respect and the technically is capable of supporting layered use cases via the include relationship.

Not enforcing any methodological viewpoint makes our approach fit different kinds of software development processes. Of course, additional benefit may be expected in use case driven processes and this includes agile and lean approaches as has been explained in the introduction. Furthermore, the approach suits well the iterative and incremental way of software development enabling developers to experiment with implementation while having the actual use case text right in front of their eyes. However, nothing prevents one from developing use cases in the waterfall way.

The approach is not limited to particular software languages (programming and markup), including the bearing language. In fact, we experimented with Java as the bearing language, too.

As has been explained in the previous section, use cases can be reused using the three kinds of relationships between them. In a transition to another language, the use case text is reusable directly.

The preprocessed code is human readable. It is annotated by use cases. In fact, this is a clue to semi-automatic refactoring of the existing code that is not modularized according to use cases.

Refactoring can be achieved by reversing the transformation performed by the preprocessor and we have actually implemented a support for it. Parts of existing code are manually annotated with steps of use cases. Based on these annotations, the implementation for use cases is generated. The parts of existing code appear in use case implementation as virtual files for particular use case step. Then, at runtime, a dedicated module ensures that the existing code is executed according to the use case text.

## 7. Implementation and Evaluation

To support our approach to literal inter-language use case driven modularization we created a dedicated web based development environment [3].[1] 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.

We reimplemented a significant portion of a real web application (hockey academy) using our approach. We implemented the whole content management system with 25 use cases out of the total of 110 use cases in the application. Having both implementations, we have been able to assess the complexity of following a use case in code and complexity of making a change to a use case implementation. Each of these was measured by three metrics: the number of files that had to be open, number of lines of code operated upon, and number of context switches that occurred. Having to look elsewhere than at the next statement in source code while following a particular thread of thoughts is counted as a context switch [2].

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

While in our approach a use case can be followed or changed within a few context switches and files (depending on the number of related use cases), the conventional approach requires 7–12 context switches and 5–8 files to open on average. In following a use case in code, the number of lines to be operated upon (read) varies. There are cases in which it is almost double in our approach. This is probably because of the use case text being included in the code and due to the framework specifics. In making a change to a use case implementation, the number of lines to be operated upon (read, written, or changed) is comparable.

There are several threats to validity of the results obtained by our study. From the perspective of internal validity, the threats include considering only a limited number of metrics and not considering other programming languages and development environments with different tracing capabilities. From the perspective of external validity, the threats include not considering systems of different sizes nor of different types (less interactive systems).

It is reasonable to expect that our approach would perform better compared to DCI and aspect-oriented development with use cases as has been demonstrated for the precursor approach we developed earlier [2].

## 8. Related Work

DCI [6, 20] differentiates between domain objects and use cases. Use cases are decoupled from domain objects and are implemented against the roles to be played by domain objects. Similar decoupling can be achieved with our approach, too, by applying it only partially. However, differently than with our approach, in DCI, use case flows in code are not continuous and tracing within the file that contains the use case implementation is necessary, as has been demonstrated in our prior work [2]. While DCI requires no particular tools, as with our approach, dedicated frameworks may be necessary, such as Apache Zest (formerly Qi4J) in case of Java [23]. Unlike with our approach, no effort is taken in DCI to engage multiple languages in one file.

In aspect-oriented programming with use cases [14, 15], aspects implement so-called use case slices, i.e., an aspect gathers all the elements of a system a particular use case operates upon. The aspect methods may be organized to reflect the use case steps, which would be similar to our approach, but, unlike with the use case steps in our approach, the ordering of the methods has no effect with respect to the program behavior. Also, this approach does not address the engagement of multiple languages.

It has been demonstrated that source code can be generated from use cases [9, 24], but that requires strictly adhering to the specific language for expressing use cases. In the end, the code is not modularized according to the use cases it was generated from.

Merging use case related parts recalls partial classes form subject-oriented programming [19] and symmetric aspect-oriented composition in general [4, 11]. Each partial class treats one concern (subject) in order to improve code comprehension and maintainability.

There are also some broader implications of the approach proposed here. The language used to express use cases in the comment part may be viewed as a domain-specific language. This domain-specific language is enabled by a preprocessor that is actually a generator in the sense of generative programming [7]. Combining different languages and different styles of programming is related to multi-paradigm programming [22].

Our development environment is related to several other experimental development environments. Object Teams [12] enable to program with roles and their bundles called teams. This seems to highly correspond to DCI, but there are no indications of a support for use case steps. ReDSeeDS [24] transforms use cases written in a particular specification language (RSL) into UML models and Java

code, but they vanish there. Dynamic code structuring [18] 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.

## 9. Conclusions and Further Work

In this paper, literal multi-language use case coding based on defragmentation is proposed. The approach aims at fully preserving use case flows in as comprehensible form as possible. The steps of use case flows appear directly in the code as comments. Despite being comments, the steps are active, which is achieved by a dedicated preprocessor and framework. The detailed step implementation gathers all the code fragments of each step each of which may be in a different programming language.

The need for taking different perspectives on code is reflected in modern integrated development environments. Dynamic code structuring [18] addresses this need directly in the code itself based on explicit concern representation. It would be interesting to explore how the approach proposed here would accommodate this or some other way of providing multiple code views.

As we discussed in our previous work [2], being able to see use case steps directly in code and moreover to program in terms of use case steps would make the intent expressed by the code more comprehensible and easier to maintain. Such code is potentially readable and even maintainable by end users, which is in line with the current trend of end-user software engineering [1]. In our approach, transferring the control to the use case text can make code even more accessible to end-users.

## 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*, pages 201–211, 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, Aug. 2015. IEEE CS.

[3] M. Bystrický and V. Vranić. Development environment for literal inter-language use case driven modularization. In *Proceedings of Modularity 2016 Demos & Posters*, 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] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[8] E. W. Dijkstra. On the role of scientific thought. Technical Report EWD 447, The University of Texas at Austin, USA, 1974. `http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF`.

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

[10] J. Greppel and V. Vranić. An opportunistic approach to retaining use cases in object-oriented source code. In *Proceedings of 4th Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2015*, Brno, Czech Republic, Aug. 2015. IEEE CS.

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

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

[13] R. Hirschfeld, M. Perscheid, and M. Haupt. Explicit use-case representation in object-oriented programming languages. In *Proceedings of 7th Symposium on Dynamic Languages, DLS'11*, pages 51–60, Portland, Oregon, USA, 2011. ACM.

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

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

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

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

[18] 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*, pages 1541–1544, Kraków, Poland, 2013. IEEE.

[19] 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*, July 1994.

[20] 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`.

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

[22] V. Vranić. Towards multi-paradigm software development. *Journal of Computing and Information Technology (CIT)*, 10(2):133–147, 2002.

[23] J. Zat'ko 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.

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