An Opportunistic Approach to Retaining Use Cases in Object-Oriented Source Code

Ján Greppel and Valentino Vranić
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Bratislava, Slovakia
E-mail: jangreppel@gmail.com, vranic@stuba.sk

***Abstract*—Use cases are widely used to express what software systems are supposed to provide in terms of an interaction between the users and the system. Without a particular effort to preserve them, use cases dissolve in source code turning the task of locating their implementation into a major problem of change request realization. While aspect-oriented programming offers one solution to retaining use cases, there are also opportunities in the mainstream object-oriented programming languages that have no reliable support for aspect-oriented programming. This paper brings an approach to retaining use cases in source code by object-oriented means that employs three techniques: traits, the Event pattern (extracted from the Zend's framework EventManager component), and the Front Controller pattern. By these techniques, all possible kinds of use cases with the exception of specialization use cases are addressed: peer use cases, extension use cases, and inclusion use cases. Use cases are not fully modularized as in aspect-oriented software development with use cases, but the approach ensures their parts can be easily traced. The approach has been evaluated with respect to traceability on an online shop application demonstrating how a use case can be added, removed, or altered.**

***Keywords*-use case; design pattern; Front Controller; DCI; aspect-oriented programming**

I. Introduction

Use cases are widely used to express what software systems are supposed to provide in terms of an interaction between the users and the system. Using the natural language they remain comprehensible virtually to all stakeholders, while still being very precise. In this, they resemble stepwise natural language algorithm descriptions (albeit they are not supposed to cover the actual algorithms) getting close to source code. In fact, especially in less formal and highly agile settings use cases may be even directly implemented.

Once implemented, use cases remain useful in testing (as a basis for so-called test cases), but perhaps even more in maintenance. Change requests can be easily related to use cases to understand what has to be adjusted. Eventually, the changes have to be implemented. Unfortunately, without a particular effort to preserve them, use cases simply dissolve in source code with the functionality they specify being rearranged according to the given programming language modularization elements (like classes and methods in object-oriented programming) turning the task of locating places in code that have to be modified into a major problem of change request realization.

This paper brings an approach to retaining use cases in source code using typical programming features available in the mainstream software development to adjust source code structure to use cases. Section II discusses what can be retained out of use cases and mentions some approaches that enable this. Section III presents the core of our approach. Section IV explains how our approach improves traceability of use cases in source code. Section V explains how we evaluated our approach. Section VI puts our approach into the context of related work. Section VII concludes the paper.

II. Use Cases: What Can Be Retained

Despite some differences in use case modeling notations [1], use cases are typically expressed as partially ordered sequences of steps. Each use case is related to one or more actors that in most case represent users. Also, a use case may include another use case and may be extended by another use case. While the include relationship is simply analogous to an operation call, the extend relationship has a more complicated semantics that was often misunderstood until the advent of aspect-oriented programming in terms of which the extending use case represents an aspect that affects the base use case, which is unaware of being affected [2]. One more relationship that may appear between use cases is generalization/specialization, which is not used that often and which in terms of object-oriented programming has its counterpart in inheritance.

Object-oriented programming can clearly modularize fragments of use cases as methods and can directly express the include relationship. These methods are being organized into classes that are in turn common to several use cases. Thus, object-orientated programming—without specific efforts—falls short not even in preserving the extend relationship, but also in preserving peer use cases (use cases that are not in any kind of relationship).

This is where aspect-oriented programming can help, as proposed by Jacobson and Ng in their aspect-oriented software development with use cases [2]. Speaking of the AspectJ style of aspect-oriented programming, aspects are not only able to affect methods, but they also can introduce new methods into classes using a construct called inter-type declaration. Classes remain shared by use cases, but they contain act merely as placeholders possibly containing some basic elements and functionality. An aspect joins all fragments of a use case introducing each of them into the corresponding class. By this, an aspect actually represents a source code model counter part for a use case.

Aspect-oriented constructs spread throughout many programming languages. However, language extensions and frameworks through which are these constructs provided are often not capable of following the base, object-oriented language development and soon become obsolete, so the question remains how to retain as much as possible of use cases using the standard features of the base programming language.

III. Techniques to Retain Use Cases

The overall approach we propose here to support retaining use cases by object-oriented means consists of three techniques: traits, the Event pattern, and the Front Controller pattern. The process is depicted in Figure 1. The techniques are applied to each use case that is to be implemented. Which techniques apply depend on the position of the use in the model, i.e. whether it is a base use case (peer), include use case, or extension use case. As use case may be a peer use case with respect to some use cases, while at the same time being included by another use case, several—or even all—techniques may apply to the same use case.

A. Peer Use Cases and Traits

A trait is a mechanism for code reuse which is implemented in various object-oriented languages. It is intended to break some limitations of single inheritance and to reuse sets of methods independently of class hierarchies. A trait is similar to a class, but its intention is to group functionality rather then to inherit it. PHP, which is currently very popular programming language in the development of web applications, supports traits [3]. Peer use cases can be retained as traits. Assume the following example that could be a part of an online shop application:

```
trait RegisterUC {
   public function createUser() {
      // ...
   }
}
trait EditProfileUC {
   public function updateUser() {
      // ...
   }
}
class User {
   use RegisterUC;
   use EditProfileUC;
}
```

With this definition, the `User` class becomes equivalent to a class containing the `register()`, `createUser()`, `editProfile()`, and `updateUser()` methods. This is similar to inter-type declarations in the aspect-oriented software development with use cases, albeit the composition is specified the opposite way: within the affected class, and not in use cases.

B. Extension Use Case and The Event Pattern

Extension use cases affect other use cases in extension points that are exposed by them. Extension points are usually just labels of particular use case steps or ranges of use case steps. Once these points are defined in the code as one-liners (one line of code), extension use case can be attached to them. This can be emulated by the object-oriented design pattern with one object holding mappings of defined points and attached behavior that we call the Event pattern. The pattern represents the essence of the Zend framework's EventManager component, which is akin to the Observer pattern and applied in the broader context of event-driven architectures [4] (essential to complex event processing [5]). Figure 2 shows the structure of the Event pattern. Here's an example implementation:

```
// Define triggers in client class
class App {
   em = EventManager()

   function foo() {
      print "Something 1 ..."
      this.em.trigger(new Event(E1))
      print "Something 2 ..."
      this.em.trigger(new Event(E2))
   }
}

// Attach behavior to triggers
app = new App()
app.attach(E1, function() {
   print "Attached on E1!"
});

// Run triggers in client object
app.foo()

// Results of app.foo():
// Something 1 ...
// Attached on E1!
// Something 2 ...
```

Figure 3 depicts a situation where the Event pattern can be applied. Suppose that in an online shop application a kind of access control has to be provided to check whether users are not just guests in order to limit their rights as some functionality is reserved only to logged users. This is not the core business of the Administer Products and Edit Profile use cases, so an extension use case called Access Control is applied.

C. Use Cases and Repeated Functionality

A use case captures interaction between a user and a system. Obviously, we cannot write code to define what

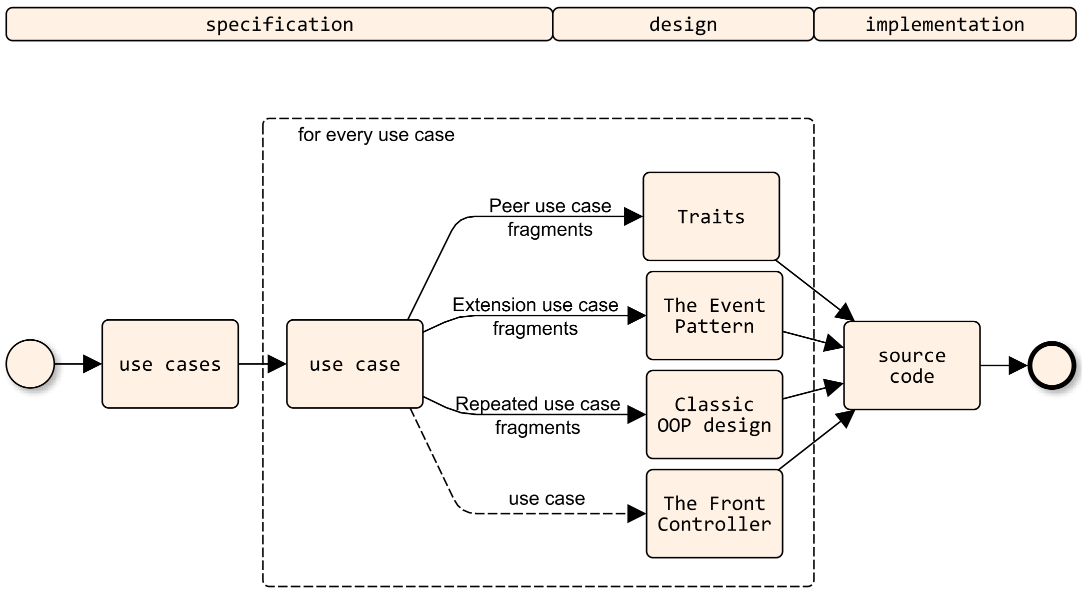
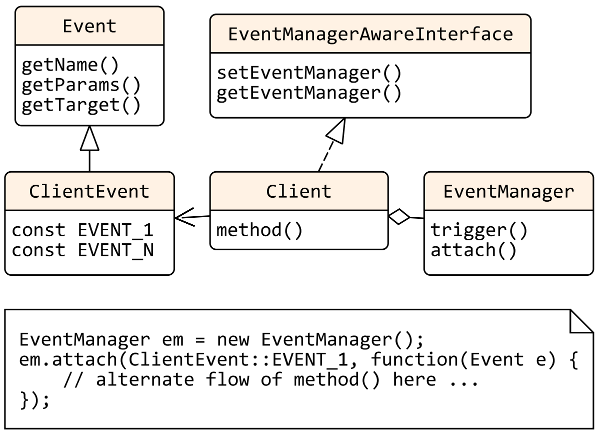


Figure 1. The process of retaining use cases at source code level.



```
EventManager em = new EventManager();
em.attach(ClientEvent::EVENT_1, function(Event e) {
    // alternate flow of method() here ...
});
```

Figure 2. The Event pattern class diagram.

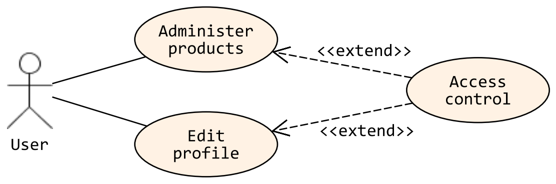


Figure 3. An extension use case.

users do: we can only write code to define what the system does. Therefore, to retain use cases in source code literally is not possible. However, a use case can be refined into a set of functionality pieces that respond to user actions described in the use case. These functionality pieces can repeat, but we would like each one implemented only once in accordance with the DRY principle (Don't Repeat Yourself): "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [6].

Such a functionality piece can represent a completely new code, code similar to the existing functionality, or an existing (repeated) code. Consider again our online shop example. One of its use cases is Register:

User: guest
1. The user selects to register.
2. The system asks for login and credentials.
3. The user fills in the information and submits it.
4. The system
a) validates the information
b) creates the new account
c) logs in user
5. The use case ends.
Alternative scenario 1:
(if the filled in information is empty or in a wrong format)
4. The system
a) validates the information
b) displays error message
c) (step 2 again)

The other use case we're going to consider is Edit Profile:

User: buyer, seller
Precondition: User is logged in.
1. The user selects to edit their profile.
2. The system shows the user's information.
3. The user changes the information and submits it.

4. The system
a) validates the information
b) saves the changes
5. The use case ends.
Alternative scenario 1:
(if the filled in information is empty or in wrong format)
4. The system
a) validates the information
b) displays error message
c) (step 2 again)

We can see that both use cases are rendering the form in their step 2: in the Register use case in order to acquire login credentials, in the Edit Form use case in order to provide the user profile information ready for editing. Validation of the sent data in step 4a is another repeated functionality piece that is found in both use cases. Yet another repeated functionality piece is displaying the error message.

In general, identifying repeated functionality often includes looking at each use case from the technical perspective and identifying the similar functionality pieces that can be implemented in a generic way that can substitute them all. Here is the Register use case implementation that uses a generic form to be rendered in a custom way using its methods **`define`**`()`, `isValid()`, and `addMessage()`:

```
use \\App\\Form;
use \\App\\Messenger;

trait RegisterUC {
   public function register() {
      \$f = new Form();

      \$f->define([
         'name' => [
            'type' => 'text',
            'text' => 'Your login name or email:',
         ],
         'passwd' => [
            'type' => 'text',
            'text' => 'Password:',
         ],
         'submit' => [
            'type' => 'submit',
            'text' => 'OK',
         ],
      ]);

      if (\$f->isValid()) {
         // Creates a new account
         // ...
      } else {
         \$m = Messenger::getInstance();
         \$m->addMessage(\$f->gerErrors());
      }
   }
}
```

It should be noted that functionality pieces can only be identified as repeated if the use case part is on the side of the system. One could assume that filling the information in and submitting it, as it happens in step 3 in both use cases) is a repeated functionality piece, too. However, this step is "executed" on the user side and therefore cannot be implemented in code and thereof cannot be retained at the source code level.

With just two use cases it is easy to spot repeated functionality pieces. However, with the increasing number of use cases, a table with all functionality pieces that are refined from use cases becomes necessary (see Table I as an exempla).

Table I
THE LIST OF FUNCTIONALITY PIECES REFINED FROM USE CASES.

Functionality	Repeated
ask for login and credentials	1x
show the user's information	1x
create a new account	1x
log the user in	1x
render the information	2x
validate the information	2x
display an error message	2x

D. *Organizing Use Cases with the Front Controller Pattern*

In addition to retaining the parts of use cases (i.e., pieces of functionality), the Front Controller pattern [7] can be used to organize whole use cases (or at least larger blocks of use cases) into controllers. This pattern, as is shown in Figure 4, provides a centralized way of handling requests. Each request made by a user will be caught by Front Controller and (based on the actual request) dispatched to the corresponding controller capable of handling it. Therefore, upon a user action, use cases as controllers will be the first classes in the application to be executed.

IV. TRACEABILITY OF USE CASES IN SOURCE CODE

An important purpose of retaining use cases in source code is to provide an easy way of tracing the parts of the actual use cases in source code. To be able to do this effectively, some additional means are necessary in our approach.

Each use case has the main actor and name, both of which can serve as pivot points for finding its implementation. The basic concept is shown in Figure 5. A matrix of actors and entities points to use case. For a given use case, the corresponding controller method is the central point towards all use case parts.

It should be pointed out that sometimes entities cannot be easily classified and therefore tracing down the use case can become ambiguous. As an example consider the Check Shipped Orders use case. It can be assigned either to the `Checks` or `Orders` entity. In such a case, the current most prevalent entity should be chosen. In our example, there are more use cases related to the `Orders` than to the `Checks`

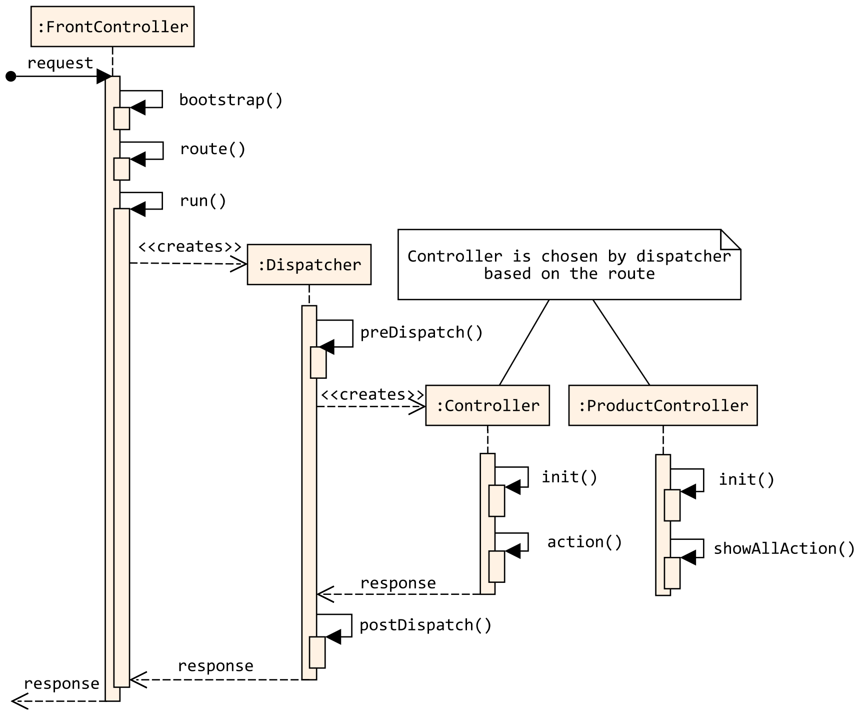


Figure 4. The Front Controller pattern.

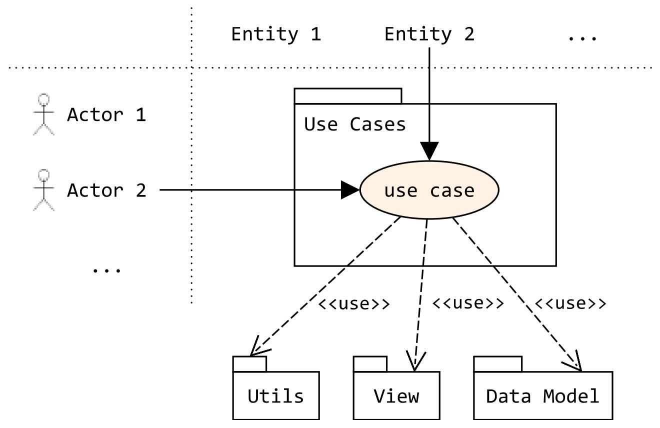


Figure 5. Locating a use case and its parts in source code.

entity, and therefore the Check Shipped Orders use case will be assigned to the `Orders` entity.

In the directory structure of our online shop application each directory serves a specific purpose related to retaining use cases:

- the *DataModel* directory contains classes that are responsible for inserting, updating, deleting and selecting of persistent data in relational-databases, text files, etc.
- the *ExtUseCase* directory contains extensions which are attached to base use cases, defined as classes with `attach()` method
- the *Libs* directory contains all third-party application interfaces and tools which are used in the system; they are mostly general-purpose libraries, not dependent on system that is developed in any way, such as MVC frameworks, ORM libraries, logging libraries, libraries communicating with online services, etc.
- the *UseCase* directory is divided into actors with each use cases implemented as a method in a given class that groups similar use cases (i.e., similar methods) together
- the *Utils* directory contains implementation of repeated functionality pieces that are used several times in the `UseCase` section; examples include forms, bank connection wrappers, flash messages, etc.
- *View* directory contains classes which render graphical interface.

Assume we have to identify the Add Product use case:

User: seller
Precondition: The user is logged in as a seller.
1. The user selects to add a new product.
2. The system prompts the user to fill necessary information.
3. The user fills in the information and submits it.
4. The system
a) validates the information
b) creates the new product
c) notifies user about the creation of a new product
d) shows list of all products added by the current

user
5. The use case ends.
Alternative scenario:
(if the filled in information is empty or in a wrong format)
4. The system
a) validates the information
b) displays error message
c) (step 3 again)

Since the user is a seller and title of the use case is Add Product, we may expect the use case to be implemented in the `add()` method in the `/UseCase/Seller/Product.php` file. The actual implementation of this method might look as follows:

```
class Products {
   function show() {
      \$form = new ProductForm();
      \$form->setData(\$this->getPost());

      // Validate the information
      if (\$form->isValid()) {
         // Create the new product
         ProductsDM::insert(\$this->getPost());

         // Notify the user about
         // the creation of a new product
         Messenger::getInstance()->
           addMessage('Product added');

         // Show the list of all products
         // added by the current user
         \$this->dispatch('Products',
           'showListOfCurrentUser');
         return;
      }
      // Show the form (prompts the user
      // to fill the necessary information)
      \$this->view = \$form->render();
   }
   function showListOfCurrentUser() {
      // ...
   }
}
```

Rendering, validating and retrieving user data from a form is all implemented in the `Form` class, which is in the `Utils` directory as a repeated functionality piece (since it's used in several use cases). The same is true for user notification by the `Messenger` singleton instance.

New product creation is handled by the `ProductDM` class, which inserts the provided data into a relational database. A part of the use case is also another use case: `showListOfCurrentUser()`, which is called using the `dispatch()` method. Figure 6 shows all different parts of the Add Product use case that are retained at source code level.

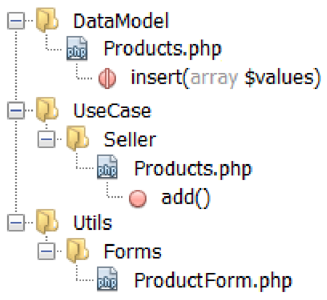


Figure 6. Parts of the Add Product use case.

V. Evaluation

We have evaluated our approach with respect to use case traceability on an online shop application mentioned throughout this paper. The full set of use cases and the actual application have been developed in the PHP programming language.

The most frequent actions with respect to retaining use cases are:

- to find the use case implementation or its relevant part in source code
- to add another use case to the existing set of use cases
- to remove a given use case from a software product (its source code)
- to alter or change one use case in a particular way

Since the first action is included in the other three (e.g., to delete a use case, it has to be located first), we focused on the last three actions.

A. *Adding Another Use Case*

When listing products in the online shop, there is no information about users' opinions to the products and therefore new functionality to add review and list reviews to a product could be useful. This functionality forms another use case (though a smaller one) that should be implemented separately for easier maintenance. The following artifacts have to be added:

- form `/Utils/Forms/ReviewForm.php` with the rating, description, and hidden fields for the user ID
- method `showReviewForProduct()` in view `/View/Product.php` to show the existing comments and to display the form
- data model `/DataModel/ProductReviewsDM.php` to insert, delete and select reviews from the persistent storage

In addition, the `showReviewForProduct()` method call in the `/View/Product.php` view has to be added in `/View/Product.php` to provide a list of reviews and the form for providing the product review details.

B. Removing a Use Case

Let's suppose that the way users register has to change substantially. Instead of altering the Register use case, the developers decide to remove it completely and to implement the new registration from scratch. For this case, the following artifacts have to be removed:

- form `/Utils/Forms/LoginForm.php`
- method `register()` in the `/UseCase/Guest/Users.php` file
- few lines of source code in the `/View/Layout/Main.php` file rendering the link to the registration page

C. Altering a Use Case

Originally, the Add Product use case supported attaching only one picture to one product. Suppose that after the product release users demanded at least three pictures per a product resulting in a change request: to be able to attach five pictures to one product and to show the gallery of these pictures in the details of the product.

This change request affects parts of two use cases: Add Product and Create Order. In these use cases, the following artifacts have to be changed:

- form `/Utils/Forms/ProductForm.php`—to add four more buttons for image upload and the validation for each one
- method `showOne()` in view `/View/Product.php`—to create four thumbnail pictures under the large one and to link them to the JavaScript slideshow

D. Discussion

What is critical in applying this approach is the ability to discern different parts of the use case and implement it in appropriate places of source code. For example, the validation of the date format should be performed within the form validation, and not in the data model. Or the functionality that is very often changed in one place should be captured in an extension use case. When wrong decisions about "what to implement where" are made, source code will not be compendious. Therefore, a strong understanding of the presented concepts and their role is crucial to sound use case implementation.

The approach proposed here is based on the needs of systems that are built on client–server architecture with the focus on complex enterprise demands. In such systems, data models, forms, views, and control blocks are very easy to find and therefore easy to implement at source code level. On the other hand, some back-end systems, for example, that have almost no presentation layer and get only a minimal input from the user, might not be appropriate for this approach. This is not because this approach is not capable of handling this type of systems, but because its main advantage of making use case code comprehensible and traceable despite their parts are implemented in different places will be lost.

From the software process point of view, after the release of an application to production (or after a shippable product increment in agile approaches), new functionality will be added and existing will be improved. Each software artifact being added increases the probability that refactoring will be needed. For example, some functionality pieces may become repeated or perhaps some use cases will get merged making new extensions apparent. That being said, refactoring of existing source code with each use case implementation is an important implicit part of the approach proposed here; otherwise source code will not be compendious.

VI. Related Work

Aspect-oriented development with use cases [2] relies fully on the capabilities of aspect-oriented programming to implement use cases. As has been discussed in Section II, this approach is successful in modularizing both peer and extension use cases. In the approach proposed in this paper, this is achieved purely by object-oriented means fully supported by the mainstream programming languages independently of aspect-oriented programming language extensions, which are often available only at the prototype level or are obsolete with respect to the host language development. The proposed here in this paper involves traits, which can be viewed as an aspect-oriented mechanism in a broader sense of aspect-orientation [8].

The DCI (Data-Context-Interaction) architecture is a vision to capture system-wide actions in the object-oriented paradigm [9], [10]. Its promise is to align perspective of the programmer and the end user in source code. This is done by separation of data (domain classes) and roles that are played in a given context. DCI approaches the problem with the intention to discern between the model domain and the more complex interactions of objects but offers rather brief case study with only few examples of use cases retained in source code. It does not deal with managing added complexity when dozens of use case implementation will be present in one system. Annotations can map certain methods to use cases, but the whole project structure remains the same and is not implicitly organized by use case as in the approach proposed in this paper. In addition, no refactoring solution are offered as part of the approach.

Requirement traceability aims at ensuring that all requirements are tied to business objectives [11]. The goal is to link stakeholder's needs to the software product features, which are linked to use cases, which are in turn linked to test cases. A change in a requirement has an impact on the corresponding features, and, consequently, on the corresponding use cases. Respecting requirements is essential for a software product to meet the business objectives. Without maintaining traceability links, it is very difficult to achieve this effectively. The approach proposed in this paper has, to some degree, a traceability matrix directly in source code in a form of the `UseCase` directory, which contains classes with methods that correspond to use cases. These classes are distributed into directories based on actor names, which makes the connection to the corresponding use cases and their realization in the source code even more apparent.

Use cases are a particularly useful way of modularizing code that makes its intent more comprehensible, but other views might be also of interest. Dynamic code structuring [12], [13], [14] may be of help in providing multiple views without having to actually restructure the code.

VII. Conclusions

Retaining use cases in the source code is not a trivial problem. While aspect-oriented programming offers one solution to retaining use cases, there are also opportunities in the mainstream object-oriented programming languages that have no reliable support for aspect-oriented programming. This paper brings an approach to retaining use cases in source code by object-oriented means that employs three techniques: traits, the Event pattern (extracted from the Zend's framework EventManager component), and the Front Controller pattern. By these techniques, all possible kinds of use cases with the exception of specialization use cases are addressed: peer use cases, extension use cases, and inclusion use cases. Use cases are not fully modularized as in aspect-oriented software development with use cases, but the approach ensures their parts can be easily traced.

The evaluation of the approach has been performed on the online shop application demonstrating how the presented approache can help developers retain parts of the use cases at the source code level and how a use case can be added, removed, or altered.

Acknowledgment

The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under the grant No. VG 1/1221/12. 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.

References

[1] V. Vranić and Ľuboš Zelinka, "A configurable use case modeling metamodel with superimposed variants," *Innovations in Systems and Software Engineering: A NASA Journal*, vol. 9, no. 3, pp. 163–177, 2013.

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

[3] M. Achour *et al.*, "PHP manual," PHP Documentation Group, Jun. 2015, http://php.net/manual/.

[4] Zend Technologies Ltd., "Programmer's reference guide of Zend framework 2," Jan. 2014, release 2.3.4, http://framework.zend.com/manual/.

[5] J. Lang and J. Janík, "Reactive distributed system modeling supported by complex event processing," in *Proceedings of 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2013*. Budapest, Hungary: IEEE CS, Aug. 2013, pp. 163–164.

[6] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Wiley, 1996.

[8] 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: ACM, 2012.

[9] T. Reenskaug and J. O. Coplien, "The DCI architecture: A new vision of object-oriented programming," *OOPSLA '02 Workshop on Tool Support for Aspect Oriented Software Development*, 2009.

[10] J. O. Coplien and G. Bjoørnvig, *Lean Architecture: for Agile Software Development*. Addison-Wesley, 2010.

[11] D. Leffingwell and D. Widrig, "The role of requirements traceability in system development," The Rational Edge, Sep. 2002.

[12] M. Nosáľ and J. Porubän, "Supporting multiple configuration sources using abstraction," *Central European Journal of Computer Science*, vol. 2, no. 3, pp. 283–299, 2012.

[13] 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: IEEE, 2013, pp. 1541–1544.

[14] J. Porubän and M. Nosáľ, "Leveraging program comprehension with concern-oriented source code projections," in *Proceedings of Slate'14, 3rd Symposium on Languages, Applications and Technologies*, Bragança, Portugal, 2014, pp. 35–50.