Preserving Use Case Flows in Source Code

Michal Bystrický and Valentino Vranić
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 2, Bratislava, Slovakia
E-mail: {michal.bystricky,vranic}@stuba.sk

***Abstract*—In this paper an approach to preserving use case flows of events in source code called InFlow is proposed. The approach preserves individual steps of use case flows of events in the form of their counterpart statements, as well as their ordering. This is achieved by mimicking each flow of events by a sequence of method implementations with each step of the flow of events corresponding to one statement in one of these methods, enabled by a metaprogramming mechanism based around the annotation named InFlow, first implemented in Python. The approach is not limited to Python and actually has been implemented in Ruby, AspectJ, and PHP, too. We evaluated the InFlow approach by conducting a study of the audio streaming service. The overall results indicate that the InFlow approach is better at preserving use case flows of events in source code than DCI and aspect-oriented software development with use cases. Having use case steps directly visible in code makes the intent expressed by the code more comprehensible and easier to maintain.**

***Keywords*-use case; flow of events; intent; annotation; Python; aspect; reflection; DCI**

I. Introduction

Modern software development is hardly imaginable without use cases. The simple idea of capturing user interaction with a system in the form of prose proved to be so powerful over time. Important in particular to highly interactive systems, use cases appear in a variety of notations out of which Jacobson's and Cockburn's are probably the most notable, even though probably each organization develops its particular style [1]. Use cases entered the agile and lean area of software development disguised—and somewhat relaxed—as user stories, but have been recognized as agile and lean in their original form [2].

What makes use cases so attractive is their ability to modularize the behavior to be available to users in the application. Having the behavior partitioned this way, one may transparently reason over what functionality needs or needs not to be supported in a particular version of the application. This is the key issue in managing configurability and it lies at heart of software product line development.

Unfortunately, the common use case realization in code does not follow the partitioning defined by them. In fact, use cases dissolve almost completely in code and it is hard to even trace the corresponding parts of the code without explicitly maintaining traceability links. Domain structure typically expressed in the form of classes serves as a common ground for use cases to contribute with their specific attributes and methods, or even affecting methods derived from other use case.

This was a long term concern of Ivar Jacobson [3] ever since he came up with the idea of use cases in 1960s [4] (though his seminal book came quite a bit later [5]), ending in his and Pan-Wei Ng's aspect-oriented approach to preserving use cases in code [6]. However, what this approach actually achieves is establishing a module for each use case (modularizing each use case as an aspect). It does not preserve the *flows of events*—denoted as *use case flows* or simply *flows*—i.e., the actual steps of use cases and their ordering.

Being able to see use case steps (i.e., steps of their flows of events) 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. This is of enormous help to programmers and even more to other stakeholders that come into touch with code. Such code would be potentially readable by end users and with some training hopefully even maintainable by them in line with current trend of end-user software engineering [7].

The actual use cases are not a part of the running software and as such are condemned to the faith of any other documentation: to become obsolete. Once coding is started, it takes over the primate. No one will ever run the text of use cases, so it rarely pays off to keep it up to date with code. Even with the best efforts, who can guarantee the consistency between the use cases and code?

DCI (Data, Context and Interaction) attacks this problem by decoupling the domain model objects as a stable part of the system from the roles they play in use cases [2], [8]. This distinction is conceptual; depending on the programming language, both domain objects and roles can be implemented by classes. The methods of the classes that implement roles become bearers of the user-goal level interaction. While there may be several helper methods, the overall interaction of roles in use case is captured in one use case method. Albeit this improves the readability of the corresponding use case flow, it still remains fragmented among the roles. This

© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

fragmentation seems to be inherent to the object-oriented programming family of languages in the broadest sense of this notion (consequently including most aspect-oriented programming languages as they usually have object-oriented foundations).

We propose an approach to preserving use case flows in source code in one piece while retaining the benefits of object-oriented decomposition. We do this by a simple trick: each use case flow is mimicked by a sequence of method implementations. Each step of the flow corresponds to one statement in one of these methods. The method implementations are arranged so their lines altogether copy the ordering of the steps in a given flow. As with all simple tricks, this one, too, needs a sophisticated support under the hood in order to work: the methods of the classes that participate in use case flows, are executed in the context of a given flow instead of the corresponding methods of the underlying domain object, which can be achieved by meta-programming or aspect-oriented techniques.

The rest of the paper is organized as follows. Section II presents the basic idea of preserving a use case flow in source code. Section III describes the implementation of attaching the classes that implement use cases to the classes that represent domain model objects in Python. Section IV deals with the details of implementing relationships between use cases. Section V presents the overall process of implementing use cases so that their flows are preserved in source code and explains how this can be applied in common software development situations. Section VI brings the results of the evaluation. Section VII discusses the approach proposed in this paper in the context of related work. Section VIII concludes the paper and sketches directions for further work.

II. Making Use Case Flows Visible in Source Code

The original meaning of *use case* is restricted to an interaction to achieve a clear user goal such as *Transfer Money* or *Place Order*. In a boarder sense, use cases may involve business processes outside the scope of the application being developed (such as handing paper documents between persons). The same technique may be used to capture high-level or overview interactions aiming at summary goals (*Product Ordering*), as denoted by Cockburn [4]. Also, lower level interactions are commonly expressed as use cases, too, more specifically denoted as habbits [2] or subfunctions [4], though—if not too low level—this distinction is not always clear (cf. different perception of the *Log In* use case [2], [9]). In this paper, a use case denotes a user goal use case or lower level interaction as distinguishing the two would make no difference in implementation.

The actual list of steps in a use case is called *flow of events*, *use case flow*, or simply *flow*. Sometimes, term *scenario* is used instead (though it is also used to denote a particular execution of steps). A use case may contain several flows. Consider the use case in the left part of Fig. 1 expressed in Cockburn's notation [4]. It contains two flows: the main one and an alternative flow (extension flow in Cockburn's terminology). This particular use case—*Play a Stream*—is a part of the audio streaming service that is going to be used throughout this paper.

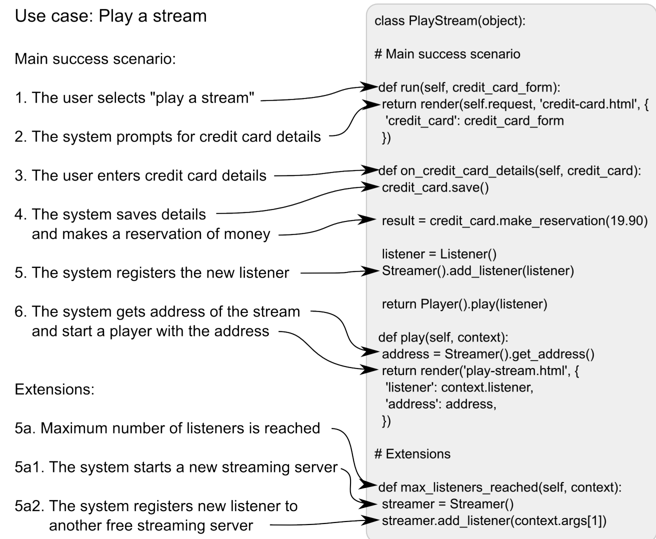

Figure 1. Preserving use case flows in source code.

The right part of Fig. 1 shows a use case implementation in Python that preserves both flows in the *Play a Stream* use case. An arrow connects each step with the statement that implements it. Even a cursory look at Fig. 1 reveals that this code preserves the ordering of steps in the use case. Let's go through the implementation and see how is this achieved with respect to the use case. There is further code behind this view that makes things work; this is explained in Sect. III.

We propose to implement a use case as one class. In the example, this is the PlayStream class (explicitly derived from object as required by Python 2; in Python 3, this is implicit). A class in Python, like in other object-oriented languages, contains method definitions ordering of which is insignificant with respect to the resulting behavior. However, we purposely order method definitions to make the ordering of their statements correspond to the ordering of the steps in the use case.

Step 1 simply indicates how is the use case activated: by having a user select to play a stream. Therefore, it is implemented by the method that initiates the whole interaction. We propose to call it run and to reserve this name for this purpose. The run method is to be called as a response to the corresponding user action upon the user interface that activates the use case.

In step 2, the system requests credit card details (the audio streaming service is paid), i.e., prompts for user input. This usually requires to render (prepare and activate) the

corresponding form (out of input controls such as text areas, drop-down list, etc.) to capture the user input, so the render method along with an indicative form name should be sufficient to make this intent comprehensible (the render method is a part of Django, a popular web application framework written in Python). In our example, the form to enter credit card details is necessary right at the beginning of the use case, so it is convenient to place the form rendering into the initiating method (run).

In step 3, the user enters the credit card details. In general, such user input is usually routed for further processing to the corresponding method. The very signature of this method stands for this type of user action. Again, the method name should be indicative; in our example, it is on_credit_card_details . The method should be bound to the corresponding event handler.

Similarly, events, such as failed validation, occur on the part of the system, too. Therefore, method signatures can also indicate system events as can be seen in step 5a. This also shows how alternative flows can be embraced in the class that implements the use case.

In a common object-oriented program, all these methods would be spread throughout a number of classes that implement domain objects. In our case, these could be a credit card, audio stream, user, etc. From this perspective, the classes that implement use cases seemingly “pull out” these methods and gather them in one place. This is sketched in Fig. 2.

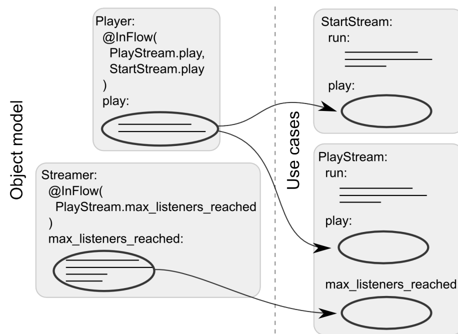

Figure 2. Domain object methods “pulled out” into classes that implement use cases.

Consider the method play in the class Player. In our approach, this method is “pulled out” into two use cases implemented by the corresponding classes: StartStream and PlayStream. However, it is still meant to be called upon Player objects. Depending on the context—i.e., use case—in which this call occures, the actual execution is redirected to the StartStream.play or PlayStream.play method. This is arranged by the mechanism whose part is the InFlow annotation as displayed in Fig. 2 (a detailed explanation follows in the next section). In a straightforward object-oriented implementation, polymorphism or an explicit conditional statement would have to be used to alter the method behavior according to the context in which it is being executed.

III. Attaching Use Cases to the Domain Model

In order to make the idea of preserving use case flows in source code using the InFlow annotation work in its Python implementation, the Python metaprogramming capabilities have to be employed. The approach proposed here is not limited to Python and we actually implemented the InFlow annotation in Ruby, AspectJ, and PHP.

A. The InFlow Annotation Under the Hood

As we explained in Sect. II, methods of domain model objects are seemingly “pulled out” into the classes that implement use cases. To ensure the exact representation of use case steps in the classes that implement them, we create a method or split the method in the domain model into two or more methods or even write a wrapper method. One of these methods is then extended with a method in the class that implements the use case. An example of such method creation is validation that does not throw an exception as is common, but instead it calls a method that indicates an error has occurred. This typically represents an alternative flow in the class that implements the use case because unexpected situations are handled by alternative flows.

If a method in domain model object contains other statements beside those corresponding to use case steps, the statements corresponding to use case steps have to be extracted into a separate method. The new method is then extended with the method from the class that implements the use case.

Now, we have prepared methods that are about to be extended. Yet, the information about a place of the extension needs to be provided before the actual extending takes place. We propose to store the metadata about the extension in the annotation that we named InFlow:

```
class Player(Model):
    @InFlow([
        ”PlayStream.on_credit_card_details play”
    ])
    def play(self):
        pass
```

The Python decorator allows to wrap the decorated play method into a wrapper method. In this case the wrapper method is an InFlow method that just saves the metadata to the decorated play method.

The metadata in the InFlow annotation consist of multiple strings that enable extensions. In these strings, the characters up to the first dot define an object out of which extension may be applied. This is followed by the names of two methods located in this object separated by a space. The first method defines the *origin* of the flow, while the second one defines the extension.

Subsequently, we need to take control over the execution flow (control flow) in order to inspect the InFlow metadata saved in the decorated methods, and then call the corresponding method based on the InFlow metadata (i.e., proceed with the execution or call a method in the class that implements the use case). The methods in the domain model objects are wrapped in order to take control over the execution flow. These methods are wrapped similarly as in the decorator:

```
class Player(Model):
    def play(self):
        # ...
def wrap(method):
    def wrapper(self):
        print("Before the method execution")
        returned = method(self)
    return wrapper
Player.play = wrap(Player.play)
```

As can be seen in the example, the methods in domain model objects are replaced with wrappers that can control the execution flow; this is similar to weaving in aspect-oriented languages. Python allows for calling an object that can store the state. Therefore, in the wrapper we instantiate the wrapping object that controls the execution flow according to a flow list. The flow list is a list from which the wrapping object was called, which is provided by the standard Python module called *inspect*. The flow list is traced before a method call to see whether its origin matches with the metadata in the InFlow annotation. If this is so, the wrapping object calls the extension instead of proceeding with execution.

Finally, consider again the last code sample containing the definition of the Player class and Fig. 3. If the play method of an object of the Player type is called from the origin (the on_credit_card_details method of the object of the PlayStream type), the play method of the object of the PlayStream type will be executed instead. Extensions are applied at runtime depending on their origin in the flow.

The frames in the left part of Fig. 3 indicate extensions in the classes that implement use cases. The frames in the right part of the figure indicate the methods of the domain model objects being extended (they are annotated with the InFlow annotations). These extensions are executed in the context of a given use case instead of the corresponding methods of the underlying domain object. In the example, the extension (the play method of the object of the PlayStream type) is executed in the context of the object of the PlayStream type (use case), but as well as in the context of the object of the Player type that is accessible in the extension.

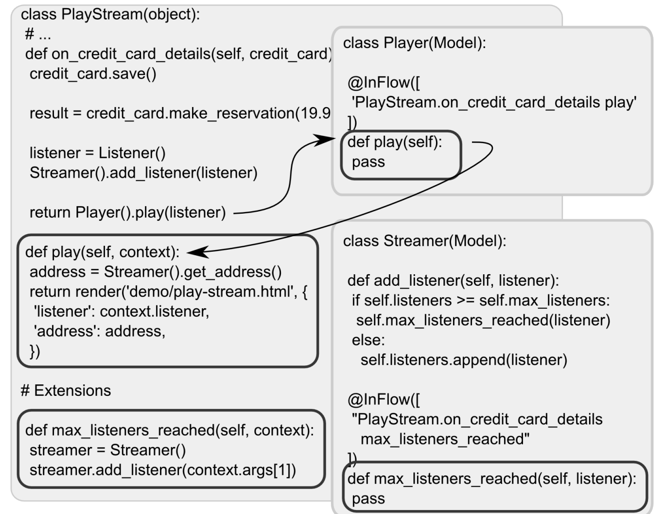

Figure 3. The execution flow indicated by arrows.

B. Reusing Methods in Use Case Implementation

Some methods of classes that implement use cases are repeated. This can be avoided by introducing a new reuse layer between the classes that implement use cases and domain model objects. This reuse layer initiates use cases and also contains implementation of the methods that are to be reused. The InFlow annotation in domain model objects enforces using the reuse layer implementation of these methods.

For example, the CreditCardReuse reuse layer calls the *Play a Stream* or *Start a Stream* use case. Because the reuse layer contains implementation of the make_reservation method of a credit card and the InFlow annotation is attached to the make_reservation method of the CreditCard object, according to the annotation, when the make_reservation method of the CreditCard object is called, the implementation of the make_reservation method from the reuse layer is reused for both classes that implement use cases.

IV. Implementing Relationships Between Use Cases

There are three types of relationships between use cases: include, extend, and generalization. In this section, we explain how to implement them in the context of our approach to preserving use case flows in source code.

A. Extend Relationship

To realize the extend relationship, the class that implements the use case to be extended has to enable this by embracing an array of modules. These modules are called, if they are present. By inserting the class that implements the extending use case into the array of modules, a call to this class will occur in the corresponding method of the class that implements the extending use case.

However, the extend relationship implemented in the form of a slot does not exactly reflect the nature of the extend relationship between use cases. Because the implementation of the use case that is being extended has to contain an implementation to support this extension, whereas a use case being extended is oblivious of this relationship. The extend

relationship actually corresponds to an asymmetric aspect-oriented implementation in which an aspect affects (extends) a class [6], [10]. On the other hand, the extend relationship in the form of an aspect is hard to spot in source code because join points are not visible explicitly in the class that implements the use case to be extended. Tracing such source code requires a tool capable of identifying join points. Therefore, our approach intentionally exposes join points in the form of the InFlow annotation.

B. Generalization Relationship

Most often, the generalization relationship between use cases is used to specify different variants of an abstract use case. However, it may occur between two concrete use cases as well, in which case it involves overriding the individual steps in a similar manner as method overriding. Therefore, in our approach we use inheritance and method overriding to implement the generalization relationship.

In Python, it is also possible to employ aspect-oriented programming to implement generalization. This can be achieved by so-called inter-type declarations.

C. Include Relationship

The include relationship can be implemented simply as a method call. In fact, the extend relationship implemented in the form of a slot is actually a call, too, assuming the slot is not empty. Thus, in both include and extend, one use case implementation calls the other one. In case that both use case involved in such a relationship extend the same domain model using the InFlow annotation, both these use cases are in the same execution flow and both can provide the necessary extension. One approach to solve this would be to call both extensions. For example, multiple reuse layers in a chain can implement the same extension method. Current implementation of the InFlow approach in Python extends the first found (the nearest) in the flow list that matches the InFlow annotation.

V. The Overall Process

The InFlow approach can be easily employed within a common software development process. If applied from the beginning, the process is straightforward. If the InFlow approach is applied to the existing code in a refactoring manner, the following steps apply:

1) Identify the parts or statements of source code that are related to use cases
2) If the parts or statements of source code related to use cases cannot be found or do not reflect use case steps, partition methods in order to be extended by the methods of the classes that implement use cases as follows:
    a) Create a method (in case it does not exists)
    b) Split the method into two or more methods (in case the method consists of multiple statements some of which have no counterpart in the steps of the use case)
    c) Write a wrapper method (in case of difference in abstractions; the method consists of low-level implementation details and it needs to be wrapped to correspond to the use case)
3) Create one class for each use case
4) Extend the methods in the domain model objects with the methods of the classes that implement use cases using the InFlow annotation
5) Move the use case related implementations into the methods of the classes that implement use cases
6) Order the methods in each class that implements a use case in order to statements reflect order of the use case steps
7) Substitute the calls to the methods of domain model objects with the calls to the methods of the classes that implement use cases

Regarding step 1, the parts of source code that are going to be extended into classes that implement use cases have to be prepared for this (statements reflecting use case steps wrapped into methods). One statement should correspond to one use case step. The statements are located in the domain model, but also they can be located in the controller if the MVC pattern is applied.

Regarding step 2, if the parts or statements of source code related to use cases cannot be found or do not reflect use case steps, the methods appearing there should be partitioned in order to reflect use case steps (see Sect. III-A). In case a statement or a method does not exists, it has to be created. In case a method consists of multiple statements and some of them have no counterparts in the use case steps, it has to be split into two or more methods. In case a method is too low-level for a use case, it has to be wrapped by a wrapper method.

Regarding step 4, the methods from the domain model are annotated with the InFlow annotation where a place of extension must be provided. A place of extension can be a method in a class that implements a use case or a method in the reuse layer (see Sect. III-B).

To be consistent with lean practices, use cases should be thrown away after they serve their purpose in analysis [2]. Manually maintaining the consistency between use cases and their implementation takes precious time and effort, and use cases so easily become obsolete struggling to keep pace with code. However, there may be need to keep use cases and the question of how to effectively keep them consistent with code arises.

After several changes, classes that implement use cases can become obfuscated due to new features being added without appropriate refactoring. The solution to this problem is to add mandatory links between use cases and classes that implement them, so that applying a change to one of

them would enforce a change to the other one. For this, we developed an extension in JavaScript.

VI. Evaluation

The approach to preserving use case flows proposed in this paper—denoted here as InFlow for brevity—is expected to raise the intentionality of source code. Simply stated, this means how well the intent is readable and maintainable with respect to the corresponding use cases as a referent expression of intent.

To evaluate the InFlow approach, we conducted a study based on the audio streaming service parts of which have been used in this paper to explain the approach.[1] The study involved twelve use cases. These were implemented using our InFlow approach and two other approaches that aspire at preserving use cases in source code: DCI and aspect-oriented software development with use cases [11], [12].

We consider the following attributes to be of significance when judging how well are use cases preserved in source code: complexity of following a use case flow, complexity of making a change to a use case flow, explicit coverage of all steps of a use case flow, and traceability of use case implementation from the domain model implementation. We assume that use cases are either explicitly recorded in a written form or there is an extensive awareness of use cases among developers as if they are "thinking in use cases".

We measured two first attributes by a *number of context switches*. By context switch we mean the necessity to look elsewhere than at the next statement in source code when following a particular thread of thoughts. Here, the thread of thoughts is represented by a particular use case flow that has to be followed step by step.

Complexity of following a use case flow in source code.: Table I summarizes the results of evaluating the complexity of following a use case flow in source code by the number of context switches. InFlow and aspect-oriented software development with use cases ended up with closely matching results as they both are able to focus the methods that express use case step into a class or aspect. DCI fragments use case implementation according to roles each of which implements a part of some use case flow. As a consequence, to follow a particular flow of events one must often switch among the roles. The *Start a Stream* use case implemented in DCI is presented in Fig. 4. Note how the use case implementation on the right side cannot be arranged to reflect the use case steps on the left side making the arrows linking the use case steps and corresponding statements crossed.

Complexity of making a change to a flow of events in source code.: Table II summarizes the results of evaluating the complexity of making a change to a flow of events in source code. We considered the following types of changes:

[1]The whole study is available at http://fiit.stuba.sk/˜bystricky/InFlow/.

Table I
Complexity of following a use case flow.

	InFlow	DCI	AOP
Start a Stream	6	15	4
Play a Stream	0	5	0
Select the Next Track	0	10	0
Fade Tracks	0	7	0

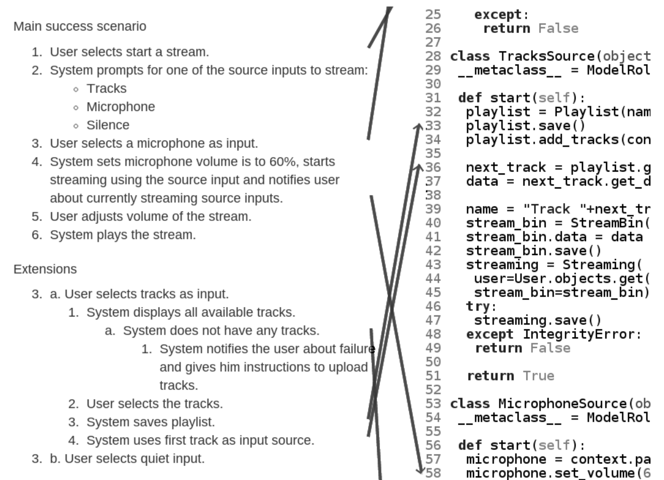

Figure 4. Preserving a use case in source code: DCI.

Integrate:	Integrating an alternative flow into the main flow
Add-main:	Adding steps to the main flow
Add-alt:	Adding steps to an alternative flow
Remove-main:	Removing steps from the main flow
Adjust:	Adjusting steps in the main flow

Table II
Complexity of making a change to a flow of events.

	InFlow	DCI	AOP
Integrate	4	4	2
Add-main	2	3	2
Add-alt	2	2	2
Remove-main	2	2	2
Adjust	1	1	1

Implementing changes in InFlow required searching for annotations. In aspect-oriented software development with use cases, the classes in the domain model contain no links to aspects that affect them, so in most cases only changes to aspects were needed. This might have been expected as aspects have been identified as a way to modularize changes [13], [14]. Handling DCI roles caused additional context switches.

Explicit coverage of all steps of a use case flow in source code.: Explicit coverage of all steps of a use case flow in source code in DCI implementation is only partial as can be seen in Fig. 4. DCI separates the controller from the context, which makes some steps to be missing in source code (steps are located in the controller instead in the use

case implementation). Note that some steps are not mapped to source code of the use case. InFlow moves the generation of the view into the context (recall the render method from Sect. II), which can be also achieved in aspect-oriented software development with use cases. By this, in both Inflow and aspect-oriented software development with use cases, all steps of a use case flow are observable in source code.

Traceability of use case implementation from the domain model implementation.: There are no traceability links included in the source code resulting from DCI and aspect-oriented software development with use cases. In DCI, role implementation is not traceable from the classes that constitute the domain model. Of course, one could easily add traceability links in the form of comments. However, consistency of such informal expressing of traceability is not guaranteed and actually the links would quickly become outdated. In InFlow, the traceability links from the domain model classes to the classes that implement use cases (in the form of InFlow annotations) are a part of the approach as such and hence must be correct at all times.

VII. Related Work

As we already stressed, DCI strives at preserving use cases in source code, too [2], [8]. According to Coplien, DCI separates the changing parts from stable parts [15] by separating the model with local behavior from use case implementation [8]. Our approach does not separate the model with local behavior because this would cause proliferation of implementation details in the classes that implement use cases. Therefore, in case of changing the implementation details, in DCI, developers apply the change only to the use case implementation, whereas in our approach they have to make changes to the domain model, too. We gave up these implementation details in use case implementation in favour of use case implementation readability. This is not a limitation of the InFlow annotation as it allows for pulling out any source code to classes that implement use cases, but a feature of the approach.

DCI modularizes source code into use cases, but keeps it fragmented according to roles. Due to this, the roles the domain objects play in use cases break the sequences of steps of use case flows in source code. Consequently, to get a complete picture of a use case, one has to trace source code over the roles.

The separation of methods related to use cases into aspects—as proposed in aspect-oriented software development with use cases [6]—is not sufficient to fully reflect use cases in source code. However, with appropriate partitioning of methods, this approach allows for similar effects as our approach. On the other hand, aspect-oriented hides away the traceability links to extensions making hard to identify what is actually being extended.

The InFlow declaration is somewhat similar to the control flow pointcut in aspect-oriented programming, as both operate upon the execution flow. For example, the flow list in AspectJ is kept in the thread-local storage, as opposed to the InFlow annotation with which it is saved in the wrapping object.

Hirschfeld et al. [16] employed Python annotating capabilities similarly as we did. They are using annotations to mark which method in the domain model semantically belongs to which use case. However, this approach is limited to tracing methods at runtime and testing whether domain model executes methods that belong to use cases.

Other approaches provide promising results in the broader area of preserving the intent in source code. These include intentional programming [17], literate programming [18], domain driven design [19], employing annotations to record applied design patterns [20], dynamic code structuring [21], [22], and code projection based on the intent expressed by structured comments [23]. Use cases can be seen as an embodiment of the end user intent and thereof these results are principally related to our work. However, none of these approaches aims explicitly at preserving use cases at any level.

VIII. Conclusions and Challenges

In this paper an approach to preserving use case flows of events in source code is proposed. The approach preserves the individual steps of the flows and their ordering by mimicking each flow by a sequence of method implementations, with each step of the flow (use case step) corresponding to one statement in one of these methods. To enable this, we developed a metaprogramming mechanism based around the annotation we named InFlow (that gave the name to the whole approach) first implemented in Python. The InFlow approach is not limited to Python and we actually implemented the InFlow annotation in Ruby, AspectJ, and PHP.

We evaluated the InFlow approach with respect to the complexity of following a use case flow in source code, complexity of making a change to a use case flow in source code, explicit coverage of all steps of a use case flow in source code, and traceability of use case implementation from the domain model implementation by conducting a study of the audio streaming service. The audio streaming service was implemented with InFlow, DCI (Data, Context and Interaction) [2], and aspect-oriented software development with use cases [6]. The overall results indicate that the InFlow approach is better at preserving use case flows in source code.

Having use case steps directly visible in code makes the intent expressed by the code more comprehensible and easier to maintain. This increase of intentionality brings end-user developers closer to the level of professional developers, which is in accordance with the growing popularity of end-user software engineering [7].

There are several points of concern we consider to be challenging. First, although the InFlow approach brings real software closer to end-user developers, they still have to cope with programming language constructs which they are typically not comfortable with. The question is how to shield away this complexity while at the same time retaining the flexibility of programming languages.

Second, classes that implement use cases are associated directly to domain model objects by the InFlow annotation. This causes mixing different levels of abstraction: high-level methods with implementation details of domain model objects. The question is how to depart them without losing benefits of the ability to trace use case implementation from domain model objects.

Third, despite the extension we developed in JavaScript enables to enforce consistency between use cases and their implementation, this is not the same as a native support. The question remains how to achieve this while not forcing developers away from established programming languages.

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, 2013.

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

[3] I. Jacobson, "Use cases and aspects – working seamlessly together," *Journal of Object Technology*, vol. 2, no. 4, July–August 2003.

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

[5] I. Jacobson, *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

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

[7] 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: ACM, 2014, pp. 201–211.

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

[9] G. Övergaard and K. Palmkvist, *Use Cases: Patterns and Blueprints*. Addison-Wesley, 2004.

[10] 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.

[11] M. Bystrický, "Implementing the control flow pointcut in python," in *Proceedings of 10th Student Research Conference in Informatics and Information Technologies, IIT.SRC 2014*, 2014, pp. 463–467.

[12] ——, "Aspectpy," bitbucket.org/bystricky/aspectpy, 2014.

[13] M. Bebjak, V. Vranić, and P. Dolog, "Evolution of web applications with aspect-oriented design patterns," in *Proceedings of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with ICWE 2007*, Como, Italy, Jul. 2007, pp. 80–86.

[14] V. Vranić, R. Menkyna, M. Bebjak, and P. Dolog, "Aspect-oriented change realizations and their interaction," *e-Informatica Software Engineering Journal*, vol. 3, no. 1, pp. 43–58, 2009.

[15] J. Coplien, "The DCI architecture: Supporting the agile agenda," Nov. 2009, Øredev Developer Conference.

[16] R. Hirschfeld, M. Perscheid, and M. Haupt, "Explicit use-case representation in object-oriented programming languages," in *Proceedings of 7th Symposium on Dynamic Languages*. Portland, Oregon, USA: ACM, 2011, pp. 51–60.

[17] C. Simonyi, "The death of computer languages, the birth of intentional programming," Microsoft Research, Tech. Rep. MSR-TR-95-52, 1995.

[18] D. E. Knuth, "Literate programming," *The Computer Journal*, pp. 97–111, 1984.

[19] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, 2003.

[20] M. Sabo and J. Porubän, "Preserving design patterns using source code annotations," *Journal of Computer Science and Control Systems*, vol. 2, no. 1, pp. 53–56, 2009.

[21] 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.

[22] 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.

[23] 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.