Computer Science and Information Systems 14(2):423–445 DOI: 10.2298/CSIS151101005B

Preserving Use Case Flows in Source Code: Approach, Context, and Challenges

Michal Bystrický and Valentino Vranić

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

Abstract. Being able to see use case steps directly in code would make the intent expressed by the code more comprehensible and easier to maintain. In this paper, an approach to preserving use case flows (of events) in source code called InFlow is presented within a broader context of its application in the software development process along with the issues that remain as challenges. The approach preserves individual steps of use case flows in the form of their counterpart statements, as well as their ordering. This is achieved by mimicking each use case flow by a sequence of method implementations with each step of the use case flow corresponding to one statement in one of these methods, enabled by the annotation named InFlow. A study of implementing an audio streaming service using InFlow, DCI, and aspect-oriented software development with use cases has been performed and assessed using several metrics in order to determine the level of preserving use case flows in source code. Although InFlow introduces a lot of indirect references to code, the overall results speak in favor of InFlow. DCI adds more complexity to following a use case flow in code than other two approaches. Complexity of making a change to a use case flow ended up with closely matching results. But InFlow guarantees traceability of use case implementation from the domain model implementation.

Keywords: use case, flow of events, intent, annotation, Python, aspect, reflection, DCI.

1. Introduction

Fixing bugs, as well as resolving change requests and feature requests, involves a lot of tracing. As a result, it is difficult to comprehend code and navigate in code. Consider, for example, an audio streaming service that consists of a front-end application and two back-end applications of 28 thousand lines of JavaScript and Python code located in 179 files within 60 folders. Though it might appear as simple from the user point of view, playing a stream is actually quite a complex functionality that involves a sort of replication dependent on the number of listeners. In the developer's context, treating a bug report of the audio stream not playing requires a number of steps. After ruling out that the bug is in the front-end by checking that a request was correctly sent to servers, we end up in the wilderness of the back-end with more than dozen of classes spread throughout several folders directly related to playing a stream and many more indirectly related classes to be inspected in our quest for the bug.

This example from our own experience illustrates the daily routine for countless software developers. In general, navigating and comprehending object-oriented code is difficult due to lots of inevitable tracing [16]. Implementing changes is particularly difficult and slow. One has to traverse many folders and files to find the places to which changes are to be applied. The situation is worsen by interleaving different concerns within one module.

Code comprehension can be improved with documentation or formally written tests, but connections between the executed code and these have to be maintained [20]. Notable are behavioral tests, which describe scenarios from the perspective of users. Gherkin and Cucumber [11] are certainly a reference point for test automation today. Roughly, test scenarios consist of the steps parameterized by the data they operate upon. Empirical evidence shows that test scenarios are long and confusing [21] because they are more low level [36], despite the existence of the appropriate tools (such as rubygem saki [21]) that employ aspect-oriented mechanisms creating an abstraction layer over test scenarios. Of course, it is possible to create higher level steps in test cases, which would be very similar to use case ones or even the same.

Since changes are expressed in terms of system usage just as use cases are, tracing would be easier in the code that preserves use cases. The idea of use cases—to capture user interaction with a system in the form of prose proved to be very powerful over time. Important in particular to highly interactive systems, use cases appear in a variety of notations out of which Jacobson's [19] and Cockburn's [6] are probably the most notable ones, even though in practice each organization usually develops its own particular style [6] often combining elements of different notations [35]. Use cases entered the agile and lean area of software development disguised—and somewhat relaxed—as user stories [7], but have been recognized as agile and lean in their original form making them a good basis for agile and lean development [8, 10].

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. The 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 [18] ever since he came up with the idea of use cases in 1960s [6] (though his seminal book came quite a bit later [17]), ending in his and Pan-Wei Ng's aspect-oriented approach to preserving use cases in code [19]. 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 *use case 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 get in 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 [2].

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 the coding starts, it takes the primacy over the documentation. 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 domain model objects as a stable part of the system from the roles they play in use cases [10, 29]. 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 a use case is captured in one use case method. Albeit this improves the comprehensibility of the corresponding use case flow, it still remains fragmented among the roles.

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 flows is mimicked by a sequence of method implementations with each step of the use case flow corresponding to one statement in one of these methods. The method implementations and the statements they consist of are arranged in the order of the steps they correspond to. 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 use case flow instead of the corresponding methods of the underlying domain object, which can be achieved by metaprogramming or aspect-oriented techniques.

The rest of the paper[1] is organized as follows. Section 2 presents the basic idea of preserving a use case flow in source code. Section 3 describes the implementation of attaching the classes that implement use cases to the classes that represent domain model objects in Python. Section 4 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 5 brings the results of the evaluation. Section 6 discusses the approach proposed in this paper in the context of related work. Section 7 identifies the challenges in preserving use cases in code. Section 8 concludes the paper and sketches directions for further work.

2. Making Use Case Flows Visible in Source Code

In their essence, use cases are a textual description of a user–system interaction. They are often mistaken for UML use case diagrams, which provide merely an overview of the use cases within a particular context [6, 35] without actually describing them, as can be seen in Figure 1. As has been mentioned in the introduction, use cases themselves commonly take the form of step sequences called *flows of events*, *use case flows*, or simply *flows*,[2] with the main sequence, called main or basic flow, separated from possible variations required to

[1] This paper is an extended version of the paper presented at ECBS-EERC 2015 [5].

[2] Sometimes, term *scenario* is used instead of flow. However, it is also used to denote a particular execution of steps).

resolve particular situations that may arise. These are, in turn, expressed as separate step sequences called *extensions*, *extension flows*, *alternative flows*, or *alternate flows*.

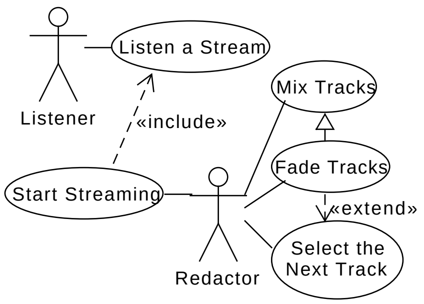


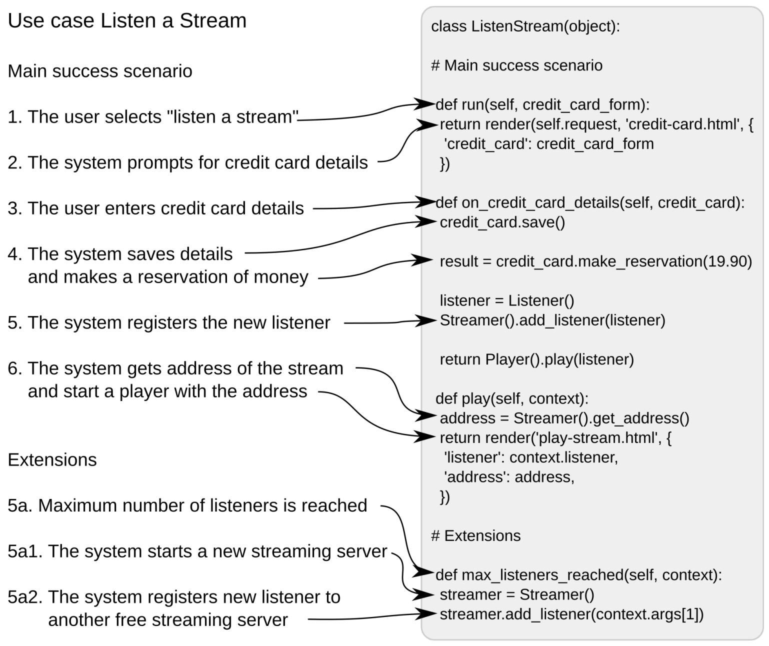
Fig. 1. The UML use case diagram of the audio streaming service.

The left side of Figure 2 shows an example use case of an audio streaming service called *Listen a Stream* (in Cockburn's notation [6]) that describes what does it mean to play an audio stream. It embraces an extension flow (activated in step 5) that resolves the problem of reaching the maximum number of listeners. Another example of a use case—*Start Streaming*—is shown in Figure 7 in Section 5.1.

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 [6]. Also, lower level interactions are commonly expressed as use cases, too, more specifically denoted as habits [10] or subfunctions [6], though—if not too low level—this distinction is not always clear (cf. different perception of the *Log In* use case [10,33]). 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 right part of Figure 2 shows a use case implementation in Python that preserves both flows in the *Listen a Stream* use case. Arrows connect the use case steps with the corresponding statements. Even a cursory look at Figure 2 reveals that this code preserves the ordering of steps in the use case. We explain how is this achieved in the rest of this section and in the next section. First of all, we propose to implement a use case as one class. In the example, this is the ListenStream 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 listen 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.


Use case Listen a Stream
Main success scenario
1. The user selects "listen a stream"
2. The system prompts for credit card details
3. The user enters credit card details
4. The system saves details and makes a reservation of money
5. The system registers the new listener
6. The system gets address of the stream and start a player with the address
Extensions
5a. Maximum number of listeners is reached
5a1. The system starts a new streaming server
5a2. The system registers new listener to another free streaming server

```
class ListenStream(object):

# Main success scenario

def run(self, credit_card_form):
 return render(self.request, 'credit-card.html', {
  'credit_card': credit_card_form
 })

def on_credit_card_details(self, credit_card):
 credit_card.save()

 result = credit_card.make_reservation(19.90)

 listener = Listener()
 Streamer().add_listener(listener)

 return Player().play(listener)

def play(self, context):
 address = Streamer().get_address()
 return render('play-stream.html', {
  'listener': context.listener,
  'address': address,
 })

# Extensions

def max_listeners_reached(self, context):
 streamer = Streamer()
 streamer.add_listener(context.args[1])
```

Fig. 2. Preserving use case flows in source code.

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 displayed at the very 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 signature of this method already makes the purpose of the method comprehensible. Again, the method name should be indicative (on_credit_card_details in our example). The method should be bound to the corresponding event handler.

Similarly, events, such as a validation that failed, 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 extension 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 Figure 3.

Consider the play method of the Player class. In our approach, this method is pulled out into two use cases implemented by the corresponding classes: StartStreaming and

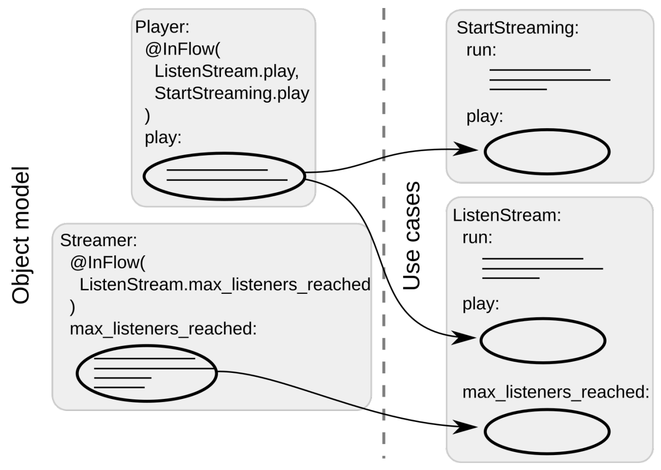


Fig. 3. Domain object methods appear as being pulled out into classes that implement use cases.

ListenStream. However, the method is still meant to be called upon the objects of the Player class. Depending on the context—i.e., use case—in which this call occurs, the actual execution is redirected to the StartStreaming . play or ListenStream . play method. This is arranged by the mechanism whose part is a particular annotation denoted as InFlow (see Figure 3; a detailed explanation of the annotation 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. In case of polymorphism, the context would not be explicit, which would worsen code comprehension.

Note that the approach involves manual mapping of natural language constructs to programming constructs: the beginning of the use case is mapped to the run method, while the user input is mapped to a method whose name corresponds to the event handler (e.g., on_credit_card_details). Other steps are mapped so that nouns are classes and verbs are their methods. For example, the step: "The system saves details of credit card and makes the reservation of money." is mapped to the save and make_reservation methods of the CreditCard class, while the step: "The system gets address of the stream." is mapped to the get_address method of the Streamer class. To help maintaining the consistency, we provided an extension described in Section 4.4.

3. 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 (discussed further in Sections 3.1 and 3.2). The approach proposed here is not limited to Python and we actually implemented the InFlow annotation in Ruby, AspectJ, and PHP (Section 3.3). We also discuss a hypothetical generic implementation of the InFlow approach valid for any object-oriented language (Section 3.4).

3.1. The *InFlow* Annotation Under the Hood

As we explained in the previous section, 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, a method has to be provided. The control flow of the method is redirected to the use case implementation. For example, consider the validation of a user input. In our implementation, it does not throw an exception as is common. Instead, it calls a method that reports an error. The control flow of this method is redirected to the use case implementation: to its extension flow, to be more precise. The method that was redirected can be called from the use case implementation by calling the proceed method.

If a domain model object method 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 control flow of this method is then redirected to the use case implementation to ensure the exact representation of use case steps in the classes that implement them.

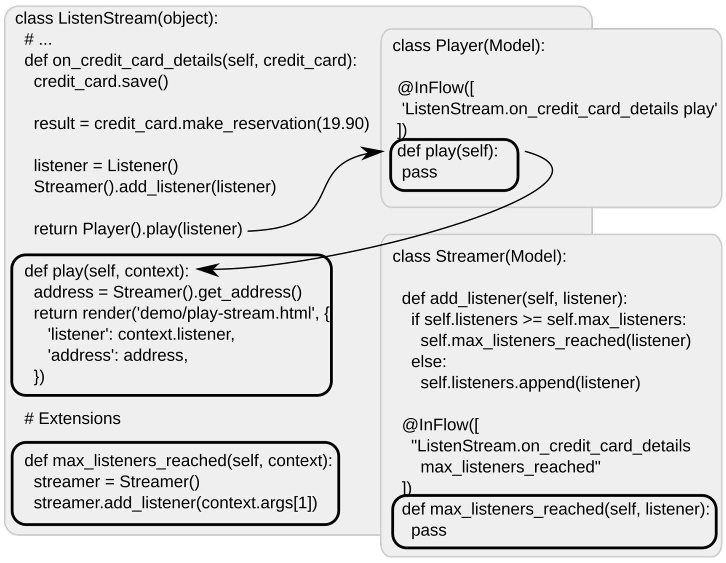
To redirect the control flow, we designed an annotation named InFlow by which a Python decorator wraps the decorated method in another, so-called wrapper method. Consider this example of decorating the play method:

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

The annotation wraps the decorated method in the InFlow method, The redirection rules are provided as an array of strings within this method only parameter. In our example, the characters up to the first dot define an object in the use case implementation. This is followed by the names of the two methods of this object separated by a space. The first method defines the source of the flow, while the second one defines the redirecting method, also called the extension or the extension method. In this case, the extension is the play method of the ListenStream class and the redirected method is the Play method of the Player class.

As implies from the example, the actual redirection is based on wrappers. Each domain model object method is replaced with a wrapper method that changes the control flow. This is similar to weaving in aspect-oriented languages. What actually happens is that the control flow is redirected by a wrapper object created by the wrapper method. The redirection is realized according to a so-called control flow list provided by the standard Python module called *inspect*. This list contains all the methods, along with the objects they belong to, from which the wrapping object was called. The control flow list is traced before a method call to see whether its source satisfies the redirection rules in the InFlow annotation. If this is so, the wrapping object calls the extension instead of proceeding with the execution.

Finally, consider again the last code sample containing the definition of the Player class and Figure 4. Whenever the Player class play method is called directly, the PlayStreaming class play method will be executed instead. Extensions are applied at runtime depending on their source in the flow. The frames in the left part of Figure 4 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 ListenStream type) is executed in the context of the object of the ListenStream type (use case), but also in the context of the object of the Player type that is accessible in the extension.



```
class ListenStream(object):
  # ...
  def on_credit_card_details(self, credit_card):
    credit_card.save()

    result = credit_card.make_reservation(19.90)

    listener = Listener()
    Streamer().add_listener(listener)

    return Player().play(listener)

  def play(self, context):
    address = Streamer().get_address()
    return render('demo/play-stream.html', {
      'listener': context.listener,
      'address': address,
    })

  # Extensions

  def max_listeners_reached(self, context):
    streamer = Streamer()
    streamer.add_listener(context.args[1])
```

```
class Player(Model):

  @InFlow([
   'ListenStream.on_credit_card_details play'
  ])
  def play(self):
   pass
```

```
class Streamer(Model):

  def add_listener(self, listener):
    if self.listeners >= self.max_listeners:
      self.max_listeners_reached(listener)
    else:
      self.listeners.append(listener)

  @InFlow([
    "ListenStream.on_credit_card_details
      max_listeners_reached"
  ])
  def max_listeners_reached(self, listener):
    pass
```

Fig. 4. The control flow (indicated by arrows).

3.2. Reusing Use Case Implementation Methods

Obviously, some methods of the classes that implement use cases would be repeated. This is avoided by introducing a new layer between these classes and domain model objects to initiate use cases and also provide the implementation of the methods that are to be reused. As is illustrated by the example in Figure 5, the InFlow annotation in domain model objects enforces using this reuse layer instead of having a repeated implementation. The CreditCardReuse reuse layer calls the *Listen a Stream* or *Start Streaming* use case. Because the reuse layer contains the make_reservation method implementation 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 make_reservation method implementation from the reuse layer is reused for both classes that implement use cases. This way, any use case can be called from the reuse layer and the implementation in the reuse layer is called according to the InFlow annotation. Multiple reuse layers can be arranged in a sequence in order to reuse methods of the classes that implement use cases or enhance them with implementation details that do not belong to these classes.

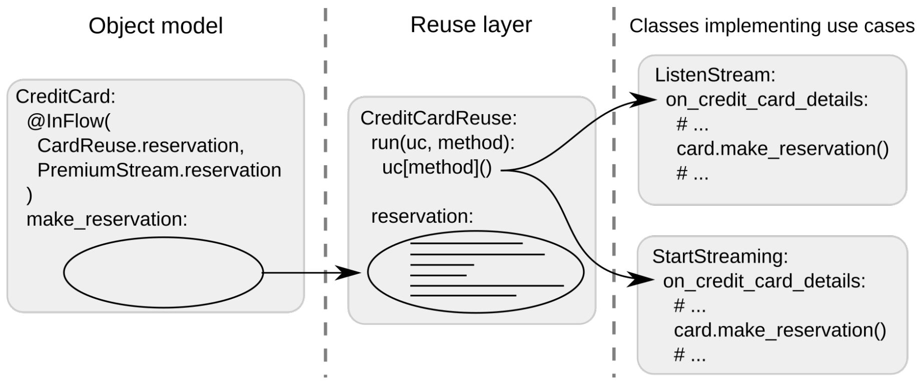


Fig. 5. Reusing the implementation of the credit card reservation.

3.3. InFlow in Other Languages

The approach proposed here is not limited to Python. We implemented it in Ruby, AspectJ, and PHP. As Ruby is a dynamic language like Python, a similar approach was used to implement the InFlow annotation. Ruby implementation involved using inheritance, annotations, and blocks, as well as the binding_of_caller module (a counterpart to the Python's inspect module) and an object lifecycle callback (self .method_added). The same results have been achieved in Ruby as in Python.

In AspectJ, we used the Wormhole design pattern [23] to support the InFlow annotation. Because AspectJ is based upon Java, which is a statically typed language, the Wormhole design pattern had to be applied for each occurrence of the InFlow annotation in order to influence the control flow (and also to pass the context). The AspectJ implementation also required using Java annotations.

The languages that only support object-oriented modularization, such as PHP, require applying the Proxy design pattern to influence the control flow. Proxy wraps the methods inside of the domain model objects. This also requires using dynamic function calls (call_user_func_array) and annotations. Here's a sample code in PHP:

```
class ListenStream {
    public function on_credit_card_details() {
        // ...
        return $this−>proxy−>call($player, "play");
    }
    public function play($context) {
        // ...
    }
}
class Player {
    /**
     * @InFlow: ["ListenStream.on_credit_card_details play"]
     */
    public function play($language) { }
}
```

Note that each call must go through the proxy to wrap the methods in order to apply the extension based on the InFlow annotation in the Player class. The proxy uses a trick: along with the arguments, it passes one parameter of the previously called object to the calling object. This replaces the Python's inspect module used to save the control flow list (recall Section 3.1), which is not available in PHP. The solution is actually applicable to any object-oriented language.

Whatever the underlying approach—be it object-oriented programming, aspect-oriented programming, or metaprogramming—method wrapping is required in order to influence the control flow to enable traversing the control flow list. Unfortunately, this significantly increases complexity and impairs performance. To some extent, performance issues could be alleviated by using appropriate data structures that could be traversed faster, but the essential burden of the runtime processing would remain.

3.4. InFlow: A Generic Preprocessor

The InFlow approach can be implemented as a generic preprocessor that would allow for adopting it in any object-oriented programming language. By using source code preprocessing, InFlow may be employed without performance drawbacks since preprocessor does its work before the code is executed.

A generic preprocessor would resolve the problem of influencing the control flow to execute additional code with the PHP implementation mentioned in the previous section. With a generic preprocessor, this would not be necessary because the preprocessor could copy the code directly there where it needs to be executed. Consider an example in Figure 6. Both classes that implement use cases provide different implementation of the Player.play method. The code from these classes would be copied to the Player class by the preprocessor, which is indicated by the dashed arrows. To differentiate the use case implementations of the play method, the preprocessor would add a use case reference as a parameter to the methods upon calling them. Based on the use case reference, the use case specific implementation of the Player class play method is executed, which is implemented by the switch command. Of course, classes implementing use cases and switch commands would cause delays, which is probably not suitable for the performance critical applications. The rectangles in Figure 6 indicate what would be added by the preprocessor. This code would improve the comprehensibility of the Player class because it would bring in crosscutting concerns of different use cases right into the Player class separated by the switch statement.

4. InFlow Within a Broader Context

There are several aspects to consider when applying the InFlow within a broader context that we discuss in the following sections.

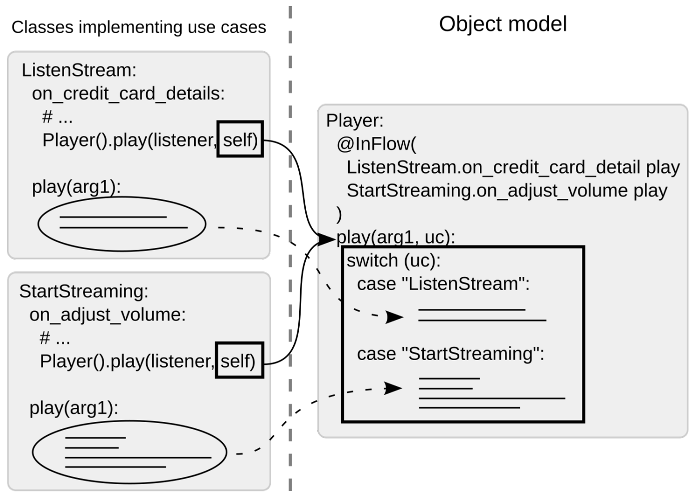


Fig. 6. A hypothetical InFlow preprocessor in action.

4.1. Implementing Relationships Between Use Cases

Three types of relationships are used to cope with the complexity within use cases: include, extend, and generalization.[3] The include relationship can be implemented simply as a method call. 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. 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.

The generalization relationship is usually 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 individual use case steps in a similar manner as method overriding, and this is exactly how use case generalization is implemented in InFlow.

4.2. Within the Software Development Process

The activities of the InFlow approach fit well within the typical activities of the software development process. If the process itself does not involve writing use cases, which is not very likely in modern software development, then it is necessary to additionally express the use cases before InFlow can be applied.

Modularizing the whole software system according to use cases affects also the activity of architectural design. Each component can be viewed as involving one or several use cases. Adding or removing components then means adding or removing use cases. An example could be the *Provide Metadata Information* use case of the API component, which provides a service for an external system. For example, the API component allows other systems to download metadata about streaming, such as track names and program.

[3] A more detailed explanation on implementing use case relationships is provided in the precursor paper [5].

Applications employing service-oriented architecture can benefit from InFlow. There, requests are dispatched based on the context of parameters included in particular request or authentication.

While InFlow does not involve expressing software by any kind of models other than code, it does not forbid graphical modeling and. In fact, this is a part of our further work.

With respect to testing, behavior driven testing is particularly in accord with a use case driven software development process [14].

InFlow particularly well fits into maintenance. A significant part of maintenance is devoted to resolving change requests that subsume changes in use cases. In InFlow, these are present directly at the code level. Apart from functional changes, there are also refactoring changes. These involve changes within a use case, such as embracing an extension flow in the main flow, adding steps, removing steps, and adjusting steps. Also, some changes happen outside a use case at the level of use case relationships. For example, we may decide to embrace track mixing within the *Fade Tracks* use case because in order to fade two tracks, they must be mixed first. But when we identify the *Loop Tracks* use case, which also assumes track mixing, it would be more appropriate to extract it into a separate use case that would be included by both use cases.

Within an iterative and incremental software development process, use cases are discovered one by one and often implemented in parts. For example, given the *Listen a Stream* use case, we could first focus on the financial issues and implement the steps that enable the payment for listening a stream. The actual playing of a stream could be a part of another increment implemented in one of the following iterations. The classes related to the classes that implement use cases have to be implemented accordingly. Iterative and incremental application of InFlow is strongly related to its application as refactoring, which has been addressed in the precursor paper [5].

Development teams can benefit from InFlow. The use case layer does not stand in the way of collaborative work. On the contrary, it is easier to identify contexts of changes and apply new changes or merge them when version control systems are used. Developers can even partition the work based on their experience with particular use case topics.

4.3. Model–View–Controller

InFlow can be implemented on top of the Model–View–Controller architectural pattern. As a matter of fact, our audio streaming service study is based on this pattern. The pattern is implemented so that the front controller delegates an event from a user to a use case, where another controller is initiated and particular actions or methods in the front controller are replaced by the class that implements the use case. Here is an example of such a controller:

```
class CreditCardController(Controller):
    def get(self, request):
        return ListenStream(request).run(CreditCardForm())
    def post(self, request):
        credit_card_form = CreditCardForm(request.POST)
        if credit_card_form.is_valid():
            return ListenStream(request).on_enter_credit_card_details(credit_card_form)
        else:
            return ListenStream(request).on_invalid_credit_card_details(credit_card_form)
```

Notice that the controller serves as a mapper of user events to a use case. In this case, the CreditCard controller calls the ListenStream class that implements the use case resulting in code indirection in the controller. Code indirection is avoiding to mention a value, variable, method, or object directly in code, but rather pointing to it with a roundabout. This is typical for wrappers, mappers, or middle layers. Indirection makes code less comprehensible as it forces developers to switch between modules when they read the code. In this, they have to remember what was in the previous module instead of being able to simply see it. Here, code indirection is caused by validations, which typically splits the main scenario and alternate flows.

In this case, we propose to omit the controllers that are directly related to use cases to prevent code indirection and to route the user events right to the classes that implement the use cases. It would be possible to make it default as convention over configuration paradigm urge, though it is against Python philosophy, which states that explicit is better than implicit. In these classes implementing the use cases, the first method of the corresponding object would validate the data. If data are invalid, the corresponding method of use case is called. The save method of the CreditCard class is an example of such a method (to be renamed to validateAndSave). Consider the following code:

```
class ListenStream(object):
  def on_enter_credit_card_details(self, request):
    credit_card = CreditCard(request.POST)
    if not credit_card.validateAndSave():
      return self.on_invalid_credit_card_details(credit_card)
    # ...
  def on_invalid_credit_card_details(self, credit_card):
    # ...
```

Other user actions are routed to the controllers. Not doing so would cause spreading the user interface implementation into the use cases layer. For example, assume there is a need to confirm the credit card details by the user, but such confirmation is not a part of the use case. In this case, the first submission of the credit card details to be confirmed by the user would be routed into the controller. The controller would display the details so that the user could confirm them. The second submission of the confirmed details would be routed right to the on_enter_credit_card_details method of the ListenStream class that implements the use case.

4.4. Traceability

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.

Several figures in this paper include a visual representation of such links as arrows between use case steps and their implementation. For this, we developed an extension in

JavaScript, which has to be imported into the HTML document containing use cases and code. With this extension, each use case step can be manually mapped to a line of code using regular expressions. If the line is not found in code, an error is reported to enforce consistency.

5. 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 throughout this paper to explain the approach.[4] The study was preceded by a conventional, MVC based implementation of the audio streaming service consisting of 28 thousand lines of JavaScript and Python code located in 179 files within 60 folders.[5]

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 [3, 4].

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."

Measuring the first two attributes involved counting the *number of context switches*. By a context switch we mean a necessity to look elsewhere than at the next statement in source code when following a particular thread of thought. Here, the thread of thought is bound to a particular use case flow that has to be followed step by step. When following a use case flow, if the next line of code is related to a use case step other than the current one or next one, the thread of thought is disturbed and the number of context switches increases by one. The same applies to counting context switches in making a change to a use case flow, with the difference of taking into account only the context switches related to the code being changed.

5.1. Complexity of Following a Use Case Flow in Source Code

The complexity of following a use case flow in source code is affected by the number of context switches and indirect references as both these phenomena distract the developer's attention. The results of measuring the number of context switches and indirect references in our audio streaming service study are summarized in Table 1 (AOSDwithUC stands for aspect-oriented development with use cases).

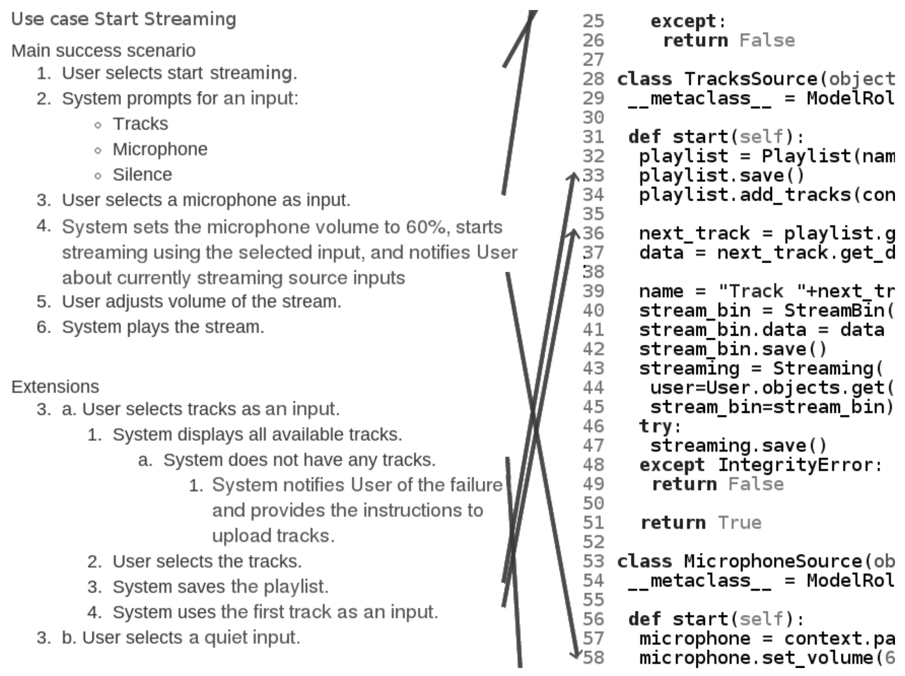
As an example of a context switch, consider the one within the *Start Streaming* use case in the audio streaming service DCI implementation captured in Figure 7. In the fourth

[4] The study is available at http://fiit.stuba.sk/~bystricky/InFlow/.
[5] The initial audio service implementation is available at https://github.com/soundslash/soundslash.

Table 1. Complexity of following a use case flow and code indirection.

	Complexity of following a use case flow [context switches]			Code indirection [indirect references]		
	InFlow	DCI	AOSDwithUC	InFlow	DCI	AOSDwithUC
Start Streaming	6	15	4	9	3	5
Listen a Stream	0	5	0	11	4	5
Select the Next Track	0	10	0	7	2	4
Fade Tracks	0	7	0	4	2	3
Average	1.5	9.25	1	7.75	2.75	4.25

step of the use case, the system adjusts the microphone volume, which is implemented in the MicrophoneSource role. However, in the previous step, the system prompts for an input, which is implemented in the class representing the use case. In order to follow the use case, a developer has to switch between these contexts in his mind, which we count as one context switch.



Use case Start Streaming

Main success scenario

1. User selects start streaming.
2. System prompts for an input:
   - Tracks
   - Microphone
   - Silence
3. User selects a microphone as input.
4. System sets the microphone volume to 60%, starts streaming using the selected input, and notifies User about currently streaming source inputs
5. User adjusts volume of the stream.
6. System plays the stream.

Extensions

3. a. User selects tracks as an input.
   1. System displays all available tracks.
      a. System does not have any tracks.
         1. System notifies User of the failure and provides the instructions to upload tracks.
   2. User selects the tracks.
   3. System saves the playlist.
   4. System uses the first track as an input.
3. b. User selects a quiet input.

```
   except:
    return False

class TracksSource(object
 __metaclass__ = ModelRol

 def start(self):
  playlist = Playlist(nam
  playlist.save()
  playlist.add_tracks(con

  next_track = playlist.g
  data = next_track.get_d

  name = "Track "+next_tr
  stream_bin = StreamBin(
  stream_bin.data = data
  stream_bin.save()
  streaming = Streaming(
   user=User.objects.get(
   stream_bin=stream_bin)
  try:
   streaming.save()
  except IntegrityError:
   return False

  return True

class MicrophoneSource(ob
 __metaclass__ = ModelRol

 def start(self):
  microphone = context.pa
  microphone.set_volume(6
```

Fig. 7. Preserving a use case in source code in DCI.

InFlow and aspect-oriented software development with use cases ended up with closely matching results as they both are able to gather the methods that express use case steps into a class or aspect. DCI fragments use case implementation according to roles each of which implements a part of some use case flow. This can be observed in Figure 7. 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.

The results of measuring code indirection are presented also in Tab. 1. We measured the number of indirect references to variables or methods of implementation of use cases. Indirect references add complexity to program because of tracing needed to locate the implementation they refer to. For example, the *Listen a Stream* use case implementation includes the following two indirect calls through the Model–View–Controller implementation (upon which our audio streaming service is based; recall Section 4.3) and the use case layer:

- controller $\rightarrow$ use case layer: two requests and one validation resulting in calling the use case layer three times from the controller, which we count as three indirect references
- use case layer $\rightarrow$ model $\rightarrow$ use case layer: the CreditCard, Streamer, and Player model classes are extended within the use case layer four times by the InFlow annotation, which we count as four indirect references)

In case of DCI, code indirection occurred only within the use case layer and model with references from the use case layer to model objects in order to connect the model objects with the use case layer. In case of aspect-oriented software development with use cases, indirection occurred only with the methods of the aspects extending controller and model classes. As can be seen from the results, the InFlow approach introduces a lot of code indirection compared to DCI and aspect-oriented software development with use cases.

5.2. Complexity of Making a Change to a Flow of Events in Source Code

Table 2 summarizes the results of evaluating the complexity of making a change to a flow of events in source code expressed by the number of context switches. We considered the following types of changes:

Integrate: integrating an extension flow with the main flow (e.g., integrating the video source input with the main flow along with the microphone source input in the *Start Streaming* use case)

Add-main: adding steps to the main flow (e.g., adding the steps that specify the volume adjustment based on the replay gain standard in the *Listen a Stream* use case)

Add-alt: adding steps to an extension flow (e.g., extending the *Listen a Stream* use case with alternate audio stream players, such as HTML 5, Flash, or Java, in the corresponding extension flows)

Remove-main: removing steps from the main flow (e.g., removing the steps that allowed a user to select a sound effect between the tracks in the *Mix Tracks* use case)

Adjust: adjusting steps in the main flow (e.g., a different algorithm has to be used to fade the tracks in the *Fade Tracks* use case)

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 [1, 34]. Handling DCI roles caused additional context switches.

Table 2. Complexity of making a change to a flow of events.

	InFlow	DCI	AOSDwithUC
Integrate	4	4	2
Add-main	2	3	2
Add-alt	2	2	2
Remove-main	2	2	2
Adjust	1	1	1
Average	2.2	2.4	1.8

5.3. Explicit Coverage of Use Case Steps in Source Code

As can be seen from the mapping in Figure 7, the DCI implementation explicitly covers use case steps only partially. InFlow moves the rendering of the view into the use case implementation (recall the render method from Section 2), 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.

5.4. Traceability of Use Cases from the Domain Model

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.

The traceability links from the domain model classes to the classes that implement use cases (in the form of InFlow annotations) that InFlow involves actually determine the control flow and hence must be correct at all times.

5.5. The Overall Impact on the Intentionality of Source Code

Table 3 summarizes average values of the attributes for each approach normalized by the standard score to make different metrics comparable. This gives negative values for the original values below the mean. The lower the number, the better at preserving use case flows approach is.

Table 3. The overall impact on the intentionality of source code.

	InFlow	DCI	AOSDwithUC
Complexity of following a use case flow	-0.522	1.153	-0.631
Complexity of making a change to a flow of events	0.218	0.873	-1.091
Explicit coverage of use case steps in source code	-0.577	1.155	-0.578
Traceability of use cases from the domain model	-1.155	0.577	0.577
Average	-0.51	0.94	-0.43

As can be seen from the table, it is easier to follow a use case flow in Inflow than in DCI and aspect-oriented software development with use cases. Also, the traceability of use cases from the domain model is better in InFlow than in DCI and aspect-oriented software development with use cases. aspect-oriented software development with use cases is slightly but not significantly better in the explicit coverage of use case steps in source code than InFlow. Aspect-oriented software development with use cases comes out as superior with respect to the complexity of making a change to a flow of events, but InFlow is also significantly better than DCI. Provided the attributes have the same significance, the average speaks in favor of InFlow with aspect-oriented software development with use cases being very close.

5.6. Threats to Validity

Measuring the complexity of following use case flows (internal validity). The complexity of following use case flows has been measured by the number of context switches. A different metric could have been used for this. For example, measuring what cyclomatic complexity has to be overcome when following a use case flow could also be relevant. However, it is reasonable to expect that InFlow would score better even with cyclomatic complexity because both DCI and aspect-oriented software development with use cases involve a larger number of constructs in their use case layers than InFlow.

Size of the system and domain (external validity). Evaluation has been performed on one application of a particular audio streaming service. It is unknown how the approach would fit into different or more complex situations.

Programming language (external validity). The study was implemented in Python programming language. Lower-level programming languages or special programming constructs may introduce additional code to the use case layer, i.e., low-level details, resulting in additional context switches while following use cases in code.

6. Related Work

As we already stressed, DCI strives at preserving use cases in source code, too [10, 29]. According to Coplien, DCI separates the changing parts from stable parts [9] by separating the model with local behavior from use case implementation [29]. 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 comprehensibility. 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 [19]—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 control flow. For example, the control 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. This allows for a transfer of the control flow list among threads, which is not possible in AspectJ and the developer is constrained to use the control flow within one thread only.

Literate programming allows to preserve use cases in code [22], too, even though this was not a concern of our approach. In literate programming, use cases would be represented by chunks of code that would be then woven into the rest of the code. Derivative of literate programming is theme-based literate programming [20], which enhances LP with themes. Themes allow to integrate text into chunks enabling to maintain links between use cases and code similar to our approach.

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.

Several approaches [15, 16, 24] enable monitoring of method execution sequences. Spy@Runtime [15] even creates a behavioral model for a component out of method execution sequences, which resembles a use case flow in InFlow. Albeit this enables mapping of natural language constructs of use cases to programming constructs as in InFlow, low-level method executions, which are not a part of a use case flow in InFlow, would be captured, too, cluttering use cases with unnecessary details.

Other approaches provide promising results in the broader area of preserving the intent in source code. These include creating programming abstractions with intentional programming [31], preserving domain information in domain driven design [12], employing annotations to record applied design patterns [30], dynamic code structuring [25–27], and code projection based on the intent expressed by structured comments [28]. 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.

7. Challenges in Preserving Use Cases

There are several points of concern we consider to be challenging with respect to preserving use cases. 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. It might be possible with programming languages employing natural languages like Python is attempting to do so, or tools able to generate a code based on a free text [13, 32]. Maybe an easier way would be a configuration of existing modules based on free text of use cases.

The classes that implement use cases are associated 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.

The classes that implement use cases cause code indirection that forces a developer to switch contexts more often when trying to understand a piece of code (recall Sections 4.3 and 5.1). To avoid code indirection in the absolute sense, one has to maintain all the code related to a particular use case in the same place. However, this will impair code comprehensibility overwhelming developers with details. Thus, a way of managing what parts of code are exposed is necessary. Moreover, code duplication will be unavoidable requiring some form of the synchronization of repeated chunks of code. The question is how to deal with these two effects while retaining the flexibility of having all the code of each use case focused in one place.

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.

Extra-functional requirements (also known as non-functional requirements) can be captured by use cases, too, as has been shown by Jacobson [19]. Jacobson denotes such use cases as infrastructure use cases. Examples include performing a transaction, handling authorization, tracking preferences, or providing cache access. The challenge is to fit infrastructure use case into the rest of use cases in code. Infrastructure use cases affect many other use cases: they represent cross-cutting concerns. Aspect-oriented programming, as proposed by Jacobson, is a possible solution, but it introduces code indirection (recall again Section 5.1).

8. Conclusions

In this paper, an approach to preserving use case flows of events in source code called InFlow is presented within a broader context of its application in the software development process along with the issues that remain as challenges. 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 the annotation named InFlow. The annotation is implemented in Python, Ruby, AspectJ, and PHP. The approach can also be implemented as a generic preprocessor that would allow for adopting it in any object-oriented programming language.

A study of implementing the audio streaming service using InFlow, DCI (Data, Context and Interaction) [10], and aspect-oriented software development with use cases [19] has been performed and assessed using several metrics in order to determine the level of preserving use case flows of events in source code in terms of 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. Although InFlow introduces a lot of indirect references to code, the overall results speak in favor of it. DCI adds more complexity to following a use case flow in code than other two approaches. The complexity of making a change to a use case flow ended up with closely matching results. However, InFlow guarantees traceability of use case implementation from the domain model implementation.

Apart from the problem of an increased number of indirect references caused by the classes that implement use cases, these classes extend the code base that has to be maintained. Although the current InFlow implementation is not optimized for performance, this is something that can be resolved by employing an appropriate preprocessor.

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 accord with the growing popularity of end-user software engineering [2].

Acknowledgments. The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under grant VG 1/0752/14. This contribution/publication is also a partial result of the Research & Development Operational Programme for the project Research of Methods for Acquisition, Analysis and Personalized Conveying of Information and Knowledge, ITMS 26240220039, co-funded by the ERDF.

References

1. Bebjak, M., Vranić, V., Dolog, P.: 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)
2. Burnett, M.M., Myers, B.A.: Future of end-user software engineering: Beyond the silos. In: Proceedings of Future of Software Engineering, FOSE 2014. ACM, Hyderabad, India (2014)
3. Bystrický, M.: AspectPy. bitbucket.org/bystricky/aspectpy (2014)
4. Bystrický, M.: Implementing the control flow pointcut in Python. In: Proceedings of 10th Student Research Conference in Informatics and Information Technologies, IIT.SRC 2014. Bratislava, Slovakia (2014)
5. Bystrický, M., Vranić, V.: 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. IEEE Computer Society, Brno, Czech Republic (2015)
6. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley (2000)
7. Cohn, M.: User Stories Applied: For Agile Software Development. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2004)
8. Cohn, M.: Succeeding with Agile: Software Development Using Scrum. Addison-Wesley, 1st edn. (2009)
9. Coplien, J.: The DCI architecture: Supporting the agile agenda (Nov 2009), Øredev Developer Conference
10. Coplien, J., Bjørnvig, G.: Lean Architecture for Agile Software Development. Wiley (2010)
11. Cucumber, Ltd.: Getting started with Cucumber. cucumber.io/docs
12. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley (2003)

13. Franců, J., Hnětynka, P.: Automated code generation from system requirements in natural language. e-Informatica Software Engineering Journal 3(1), 72–88 (2009)
14. Garg, S.: Cucumber Cookbook. Packt Publishing (2015)
15. Ghezzi, C., Mocci, A., Sangiorgio, M.: Runtime monitoring of component changes with Spy@Runtime. In: Proceedings of 34th International Conference on Software Engineering, ICSE 2012. pp. 1403–1406. Zurrich, Switzerland (2012)
16. Hirschfeld, R., Perscheid, M., Haupt, M.: Explicit use-case representation in object-oriented programming languages. In: Proceedings of 7th Symposium on Dynamic Languages. ACM, Portland, Oregon, USA (2011)
17. Jacobson, I.: Object Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley (1992)
18. Jacobson, I.: Use cases and aspects – working seamlessly together. Journal of Object Technology 2(4) (July–August 2003)
19. Jacobson, I., Ng, P.W.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley (2004)
20. Kacofegitis, A., Churcher, N.: Theme-based literate programming. In: Proceedings of 9th Asia-Pacific Software Engineering Conference, APSEC 2002. IEEE Computer Society, Gold Coast, Queensland, Australia (2002)
21. Kidwell, N.: Saki - For times when you can't swallow Cucumber. rubydoc.info/gems/saki/0.1.4
22. Knuth, D.E.: Literate programming. The Computer Journal pp. 97–111 (1984)
23. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning (2009)
24. Nobakht, B., Bonsangue, M.M., de Boer, F.S., de Gouw, S.: Monitoring method call sequences using annotations. In: Proceedings of 7th International Workshop on Formal Aspects of Component Software, FACS 2010, Revised Selected Papers. LNCS 6921, Springer, Guimarães, Portugal (2012)
25. Nosáľ, M., Porubän, J., Nosáľ, M.: Concern-oriented source code projections. In: Proceedings of 2013 Federated Conference on Computer Science and Information Systems, FedCSIS 2013. pp. 1541–1544. IEEE, Kraków, Poland (2013)
26. Nosáľ, M., Porubän, J.: Supporting multiple configuration sources using abstraction. Central European Journal of Computer Science 2(3), 283–299 (2012)
27. Nosáľ, M., Porubän, J.: Xml to annotations mapping definition with patterns. Computer Science and Information Systems Journal (ComSIS) 11(4), 1455–1477 (2014)
28. Porubän, J., Nosáľ, M.: Leveraging program comprehension with concern-oriented source code projections. In: Proceedings of Slate'14, 3rd Symposium on Languages, Applications and Technologies. pp. 35–50. Bragança, Portugal (2014)
29. Reenskaug, T., Coplien, J.O.: The DCI architecture: A new vision of object-oriented programming. Artima Developer (2009), http://www.artima.com/articles/dci_vision.html
30. Sabo, M., Porubän, J.: Preserving design patterns using source code annotations. Journal of Computer Science and Control Systems 2(1), 53–56 (2009)
31. Simonyi, C.: The death of computer languages, the birth of intentional programming. Tech. Rep. MSR-TR-95-52, Microsoft Research (1995)
32. Smialek, M., Jarzebowski, N., Nowakowski, W.: Translation of use case scenarios to Java code. Computer Science Journal 13(4), 35–52 (2012)
33. Övergaard, G., Palmkvist, K.: Use Cases: Patterns and Blueprints. Addison-Wesley (2004)
34. Vranić, V., Menkyna, R., Bebjak, M., Dolog, P.: Aspect-oriented change realizations and their interaction. e-Informatica Software Engineering Journal 3(1), 43–58 (2009)
35. Vranić, V., L'uboš Zelinka: A configurable use case modeling metamodel with superimposed variants. Innovations in Systems and Software Engineering: A NASA Journal 9(3) (2013)
36. Zielczynski, P.: Traceability from use cases to test cases. IBM developerWorks, Technical Library (2006), http://www.ibm.com/developerworks/rational/library/04/r-3217/

Michal Bystrický is a PhD student in the field of software engineering at the Slovak University of Technology in Bratislava and an active Python developer. He explores use case preserving in source code.

Valentino Vranić is an associate professor of software engineering at the Slovak University of Technology in Bratislava. He explores different aspects of software development. In particular, he is interested in preserving intent comprehensibility in code and models using advanced modularization, as well as in effective agile and lean organization of people in software development and its wider social connotations.

Received: November 1, 2015; Accepted: February 15, 2017.