JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 1, January-February 2004

On Getting Use Cases and Aspects to Work Together

Renaud Pawlak and Houman Younessi
Rensselaer Initiative in Software Engineering (RISE), Rensselaer Polytechnic Institute - Hartford Graduate Campus, USA

Abstract

Aspect orientation (AO) as an underlying logical model deduced from Aspect Oriented Programming (AOP) [Kiczales97] is attracting attention and gaining in popularity. A number of authors have recently written about how ideas of aspect orientation might be used in connection or in conjunction with existing modeling techniques or technologies in order to enhance the capabilities of the latter. For example, in a recent JOT article Ivar Jacobson writes about the relationship between use cases and AOP [Jacobson03] claiming essentially that the two can be viewed as equivalent. It is however critical for those of us proposing these ideas and connections to ensure that such claims are based on a foundation of adequate comparison and analysis. As investigators who have been working on similar ideas for some time, we would like to take the opportunity of this short paper to provide a constructive critique of what such comparisons need to entail and where the critical issues lie. The Jacobson paper would be used as an exemplar to raise and discuss some common issues and shortcomings. We will explain that although we agree with the essence of Jacobson's overall statement, we also believe that such assertion is – as it stands – neither new in its essence, nor is it complete and needs to be complemented by crucial improvements if we really want to bring new answers to software engineering.

1 USECASES AND ASPECTS

In a recent JOT paper [Jacobson03], Ivar Jacobson argues that there is a correlation between use cases and AOP [Kiczales97]. He explains that a use case crosscuts a set of components, which is intuitively true, because the essence of a use case is to describe a protocol between several components. He goes further to explain that a design using component-based techniques will achieve component modularity but will fail to achieve use case modularity. Indeed, since several use cases may work on the same set of components, the implementation of the components will eventually have the following properties:

1. a given component will contain the code coming from the implementation of several use cases (code tangling property)
2. a set of components will be required to implement a given use case so the use case implementation will not be well modularized (crosscutting property).

We agree with all these statements. Moreover, our own work in this area is based on the same interesting yet obvious logical relationship between use cases and aspects: that they both overlap the components with tangling and crosscutting. We therefore contend that in the future, aspect orientation and use cases will eventually work together to greatly simplify the development life-cycle. In other words, our goals for a methodology we are actively developing is based on this very basic and intuitive idea we share with Jacobson. In theory, developing an application will be done in two main steps:

1. find the use cases and specify each use case,
2. design, code, and test each use case.

As expressed here, a component design step will be completely removed from the design, making the applications and the process of creating them much more convenient, simple, and powerful. As Jacobson states, this is possible because AO will allow the modularization of use cases in the implementation, so that there will be a direct match between the analysis/design level and the implementation level. Separation of concerns will be optimal during the whole application development cycle.

Where is it incomplete?

After introducing this main statement, Jacobson then presents his vision of solving the modularity problem of use cases. He re-presents a proposal originally made in a different context in 1979. In doing so he introduces the notions of *existion*: an existing functional use case, and of *extension*: a use case that extends the existion. The extension mechanism is based on a very simple technique that consists of designating extension points in the existion, and to switch to possible extensions before and after these points [Jacobson79].

It is easy to note that this mechanism is obviously very similar to a core mechanism that we find in AOP, called *before* and *after* advising. In short, this mechanism consists of adding advice code before or after a joinpoint, a joinpoint being, for instance, the invocation or execution of a given method.

On the strength of this observation, Jacobson proceeds to set up a mapping between his use case extension technique, and AOP. Whilst we understand this mapping, we believe it to be a dangerous assertion as it could lead to the conclusion that the two techniques are similar, even equivalent, which is inherently fallacious. Indeed, even if we show that the extension mechanism is entirely covered by AOP (AOP includes Extensions) it does not mean that the reverse is also true.

In the next section, we will show why Extensions do not include AOP, and consequently demonstrate that AOP is different from Extensions. The reader should note here that we do not wish to contradict Jacobson's core idea, but only to complement it with a set of important concepts that are crucially needed to achieve our common goal: make ideas from AO and use cases work together.

2 WHY EXTENSIONS DO NOT CONTAIN AOP?

The essence of AO is modularization of crosscutting concerns. To this end, AO defines a set of concepts that can be used to handle the modularization, and subsequent re-composition of these concerns. The piece of software that performs the re-composition is called a weaver. Composing all the concerns together is a complex task. Uncovering techniques to do so remain one of the main challenges of AOP.

However, Jacobson's proposal mainly relies on the simple before/after mechanism, which is certainly a necessary mechanism of AOP, but obviously not a sufficient one to achieve the essence of AOP in a straightforward way. Besides, the before/after insertion of code is not new and has been widely recognized and used even before AOP. Flavors [Cannon82], New Flavors [Moon86] ,CommonLoops [Bobrow86] and CLOS [Steele90] all support the before, and after constructs. Reflective language, and especially behavioral reflection [Maes87, Ferber89, Foot89, Smith84], compile-time reflection [Chiba95], and metaobject protocols [Kiczales91], are well-known candidates to support the before and after mechanisms [Sullivan01]. Knowing that, we should ask ourselves (i) why does Jacobson not refer to these techniques as implementation candidates available much before AOP, and (ii) if AOP can be reduced to after and before mechanisms, why AOP is not simply called "behavioral reflection" or "interception" or "method composition"?

The answer is obvious: AOP and therefore AO is more than just introducing before/after mechanisms in an existing paradigm. In the following section, we will depict the important concepts that are needed in AO and that need to be handled at a use case level if we really want to achieve our goal.

AOP implies pointcuts

Having the ability to add behaviors before and after insertion points introduces an additional dimension: to define and combine behaviors. This new dimension is not easy to represent since it is a reverse dependency: the base program will not know explicitly about the added behavior. As a consequence, handling this complexity within a consistent methodology and providing the adequate abstractions is one of the main challenges of AOP. If an AOP solution does not manage this issue, then why not use metaobject protocols, or CLOS-like languages in place of AOP?

AOP proposes a concrete solution to this problem: the *pointcut* construct (which Jacobson only briefly mentions in passing). A pointcut is a construct which is held by an aspect and that literally defines a cut within the base program elements. By cutting the base-program elements, the pointcut defines a set of *joinpoints* (insertion points) that corresponds to the intersection of the cut and the base-program elements. Consequently, the pointcut is able to crosscut a set of base components thus making it possible to introduce a crosscutting concern in a simple way.

Without the pointcut mechanism, the programmer/designer, would have to define these joinpoints manually, thus making the before/after dimension complexity too difficult to control to be actually usable and scalable. Simple AOP-ized extensions as defined by Jacobson do not provide the pointcut facility. One may note also that Jacobson is not alone here; many works claiming to implement AOP have made the same mistake. For instance, most of the early AOP works targeted for the J2EE environment, such as JBoss-AOP, Spring, or Nanning would not provide pointcut definition features, or, at best, ones based on regular expressions. One should not be surprised then, if only the logging-like examples can be straightforwardly implemented in these environments.

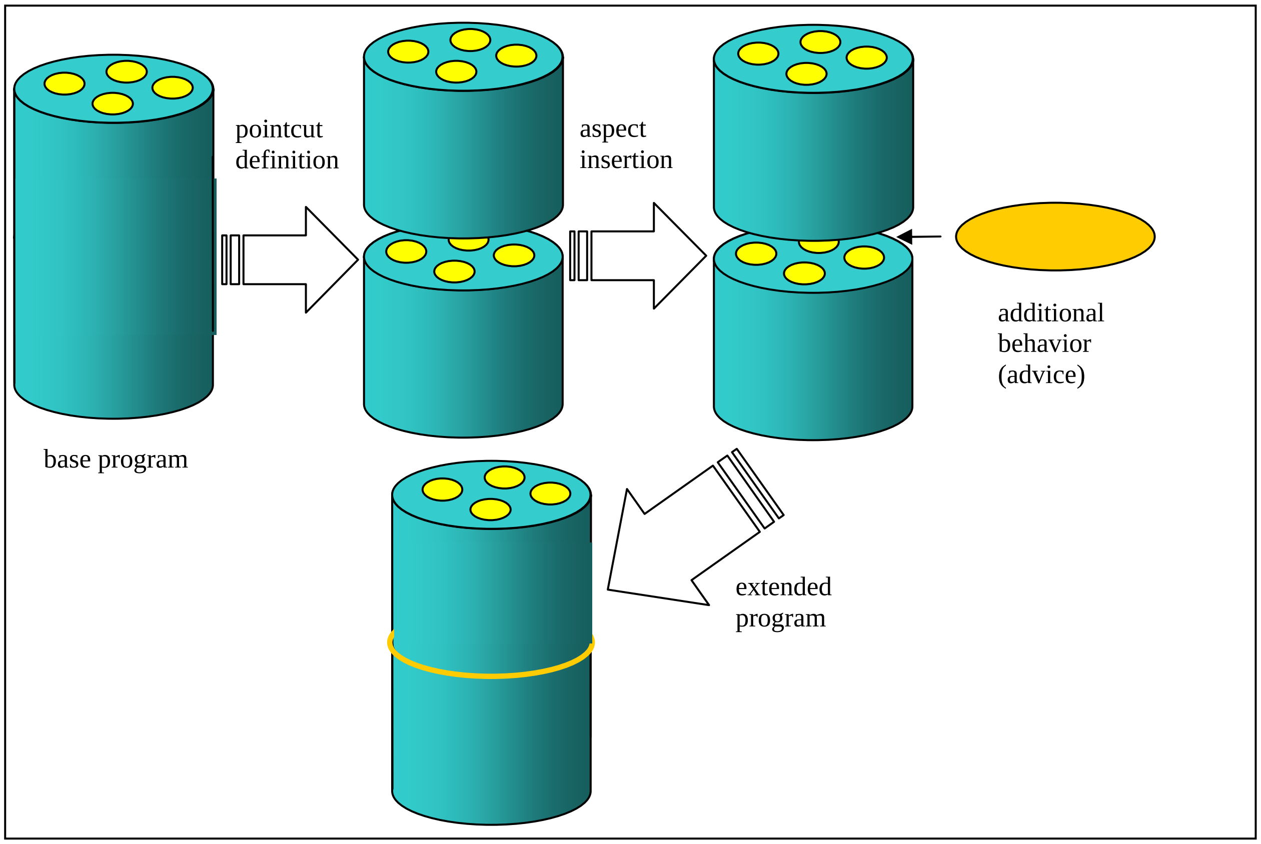
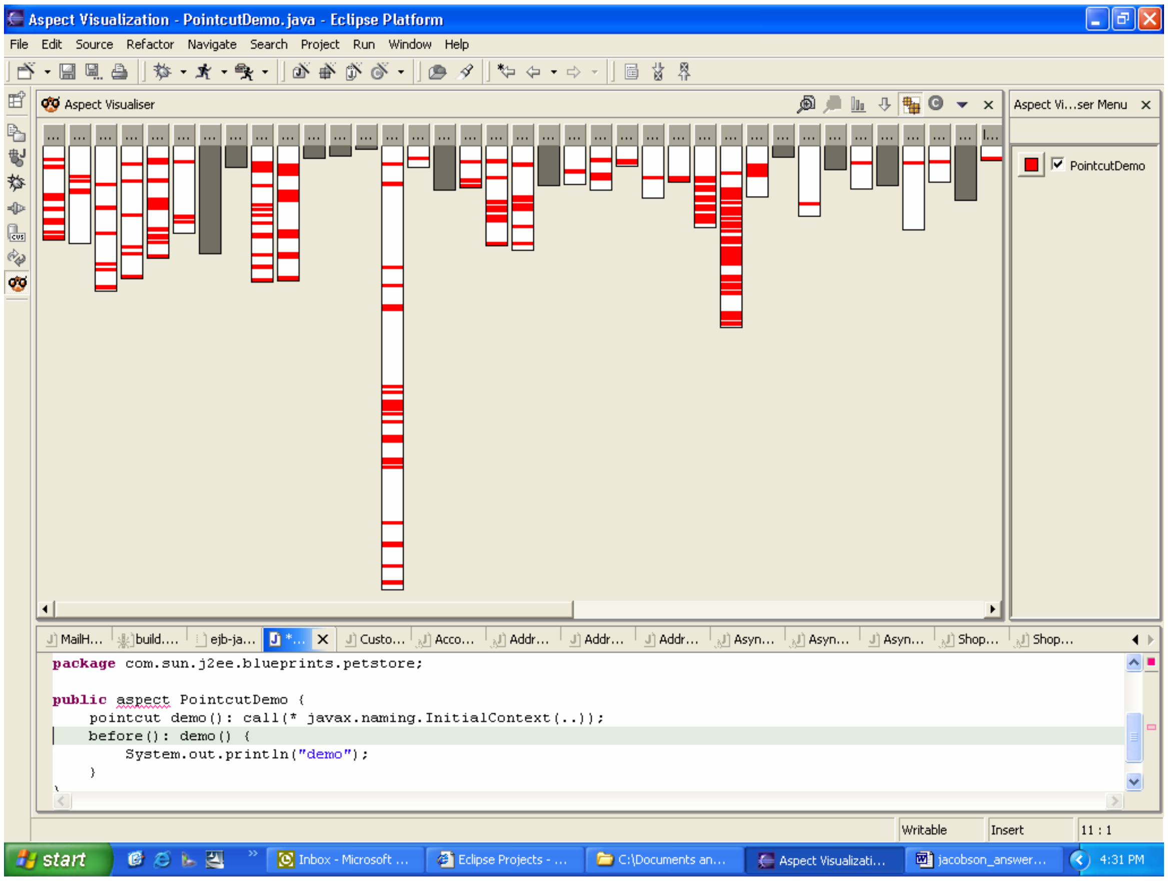


Figure 1 – AOP base program extension with the pointcut notion

Figure 1 shows the importance of the pointcut notion of AOP. In this figure, the base program is represented by the inner cylindrical components (yellow cylinders). A pointcut can be represented by a cut in the whole program cylinder, thus cutting all the components. The additional behavior, which is defined independently from the pointcut, by using the before/after mechanism can then be added in a consistent way to the base program in order to add a whole concern at once. One of the main challenges of AOP is to allow the programmer/designer to define meaningful pointcuts so **that he or she would** be able to easily define and manipulate them (this may imply, in an ideal AOP, the ability to define, abstract, and reuse pointcuts).

Figure 1 is only a simplistic representation of the pointcut notion. However, in real-world applications, the pointcuts can be very complicated. This is why we need powerful pointcut definitions capabilities. For instance, Figure 2 helps in visualization of a very simple pointcut within the AJDT (AspectJ Development Tool) [Kiczales01] of Eclipse. It corresponds to the points where the SUN Blueprint Petstore well-known J2EE example uses a part of the naming API. Each vertical bar corresponds to a class of the Petstore example, and each red line corresponds to a code location where the class actually uses the naming API to resolve an EJB component reference. As one can easily see here, it is impractical to do this manually even in a program of moderate size and complexity.


Aspect Visualization - PointcutDemo.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Aspect Visualiser
Aspect Vi...ser Menu
PointcutDemo
MailH... build.... ejb-ja... Custo... Acco... Addr... Addr... Addr... Asyn... Asyn... Asyn... Shop... Shop...
Writable
Insert
11 : 1
start
Inbox - Microsoft ...
Eclipse Projects - ...
C:\Documents an...
Aspect Visualizati...
jacobson_answer...
4:31 PM

```
package com.sun.j2ee.blueprints.petstore;

public aspect PointcutDemo {
    pointcut demo(): call(* javax.naming.InitialContext(..));
    before(): demo() {
        System.out.println("demo");
    }
}
```

Figure 2 – a simple pointcut visualization with the AJDT Eclipse plugin.

We need advanced pointcut definition and implementation mechanisms. As good examples of such advanced pointcut definition mechanisms, we can cite the *cflow* and the *semantics based* pointcuts. The *cflow* construct of AspectJ [Kiczales01], allows the programmer to define cuts that depend on the control flow of some methods. The semantics based pointcuts of the JAC framework [Pawlak01] allow the programmer to write pointcuts that will cut the methods based on what they are doing in the program (for instance, the programmer can include/exclude a whole method set depending on whether it modifies a given object state or not).

AOP implies composition

AOP is about (i) modularizing several crosscutting concerns, but it is also about (ii) reusing them and composing them to produce the final application. Jacobson completely ignores the second point in his paper (but again he is not the only one). In modern large-scale application development, the integration of existing components to address specific concerns is very important. It is not reasonable, nowadays, to build the applications from scratch, when so many middleware components can be reused to deal with technical concerns such as persistence, transaction, scalability, fault-detection and recovery, etc. and when several software components provide business-oriented capabilities that can be reused to solve business-traversal issues such as workflow control, profiled presentation, access rights, mobile and cooperative work.

One of the main goals of AOP consists of providing means to reuse and compose different concerns. As a consequence, a use case based methodology should provide a way to (i) make the different identified concerns explicit, (ii) reuse existing components when available, (iii) specify the composition of these different concerns so that the final application specification is complete.

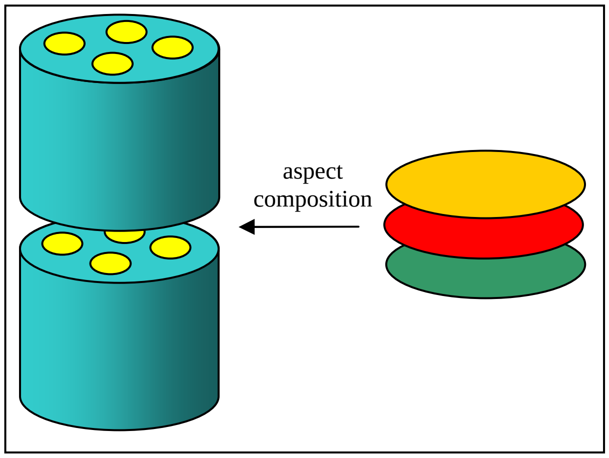


Figure 3 – the composition issue of AOP

Figure 3 shows a simple representation of the AOP composition issue. This issue is very similar to the general issue of integration in software engineering. The main idea consists of being able to check the correctness of the new formed system once the integration or composition step is performed. Of course, most of the difficulties arise from the fact that the composed components may have been developed independently and present incompatibilities when used together in a certain way.

Despite availability of simple composition support in AOP languages (e.g. the precedence notion in AspectJ), AOP still provides incomplete means to this end. Resolving this issue in general is still an open research question. However, there are several efforts in progress that are showing promise [Pawlak99, Sihman02].

Within a use case driven development process, some particular attention should also be given to composition. For instance, the composition of technical and functional concerns will certainly be a frequent task in such a process. Some methodological support would certainly be needed and have to be investigated.

3 ACTUALLY MAKE USE CASES AND AOP WORK TOGETHER: THE ELSA PROJECT

As mentioned in the previous sections, we agree that use cases are a promising basis for a methodology that would greatly simplify the development of large-scale applications. Indeed, by modularizing each concern in a use case model and by having a straightforward correspondence between the design level and implementation, we can improve the development process of large-scale systems.

To achieve this, we need to define an environment that relies on a methodology, formalisms, and tools that enhance the idea of usefulness of usecases as put forth. Enhancements need to be aligned with the two aforementioned crucial issues: (i) the ability to define meaningful pointcuts, and (ii) the ability to compose extensions not only in order to improve our control on the final application behavior, but also in order to be able to integrate existing components (e.g. technical middleware components).

These are the goals of the ELSA project (stands for Environment for Large Scale Applications). ELSA is an on-going project to develop and validate an innovative software process and its associated automation tool. ELSA integrates the strengths of the Object oriented/Component Based (OO/CB) paradigm, formal usecase modeling and Aspect Orientation (AO) in a novel and innovative fashion in order to provide a robust but quick to market process for development of software. ELSA is based on early, disciplined and automated integration of architectural concerns into the product model. Such an approach has enormous potential to positively impact budget, schedule and product quality of all and particularly large-scale software projects. As the project specifically targets architectural concerns, ELSA stands to significantly improve our ability to design for products attributes such as reliability, scalability, robustness, security and maintainability. Potential for technology transfer and commercialization would be significant.

In the next sections, we will give some indications on how the ELSA methodology provides a workable environment based on both AO ideas and usecases.

Use case sets: a basis for pointcut definition

As stated above, a use case based methodology would need a straightforward means to define pointcuts. To us, the most important thing about pointcut support at an analysis/design level is that they be easily represented (maybe graphically), and easily understood. Consequently, we do not think that an AspectJ-like syntax would be appropriate, even if its succinctness is well-suited at a programming level. Therefore, we are currently investigating other means to define pointcuts in ELSA. Our main approach consists of relying on the global use case set structure.

The ELSA methodology proposes a set of use case writing rules that helps the designer to write clean and unambiguous use cases. These rules induce organization of the use cases into hierarchical use case sets (each use case is described by a set of child use cases and each child use case is described likewise). Since our methodology is influenced by AO ideas, the original use case set will only describe the pure functional behavior of the application. Thanks to the hierarchical decomposition, the complexity of the system is easy to handle and separation of concerns is adequately provided.

The interesting point about this use case based decomposition of the system is that it is a behavioral decomposition rather than a component-based one. Even though the two **points of view** are dependent (we can automatically generate the component view from the use case view), the borders of the modules are completely distinct. The use case point of view thus opens up a new dimension that will allow the definition of pointcuts.

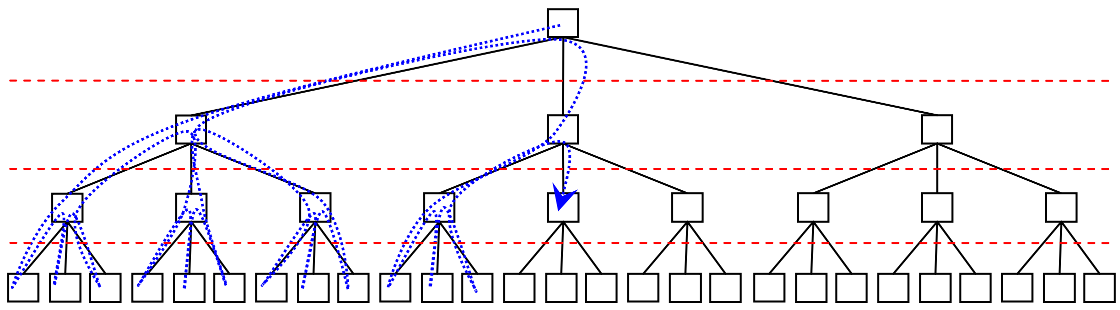


Figure 4 – flow of control in a hierarchical UC set

Figure 4 shows the flow of control (doted curve) in a hierarchical use case set (that follows the ELSA use case writing rules). As shown in the figure, the separation between a level and another (dashed horizontal lines) are crossed by the flow of control several times at multiple points that may correspond to various service requests in the system. Consequently, these separations seem to be a natural location to introduce aspects. Indeed, in the AO sense, level separation can be regarded equivalent to pointcut since they actually crosscut a set of heterogeneous base components.

Contrary to classical AOP, these separations crosscut the behavior of the system rather than its modularized structures. Therefore we can refer to this kind of AOP as *behavioral AOP*.

Note that the definition of behavioral AOP is a work in progress but it is very promising. Pointcuts will certainly be defined on the two points of view (the use case and the component ones) in an intuitive way. The combination of the two partial definitions will entirely define the pointcut. Once the pointcuts are defined, it will be easy to apply some extension mechanism to it.

Composition of aspects

Beyond pointcut definitions, the second important issue that ELSA deals with is the composition issue. Automatic composition is also an open research issue, but we think that most of the problems can be solved if the methodology provides suitable guidelines and the environment provides simple tools for system reorganization and reengineering. Therefore, we are currently working towards defining the aspect weaving mechanism as intermediate use case level insertion in the functional use case set hierarchy.

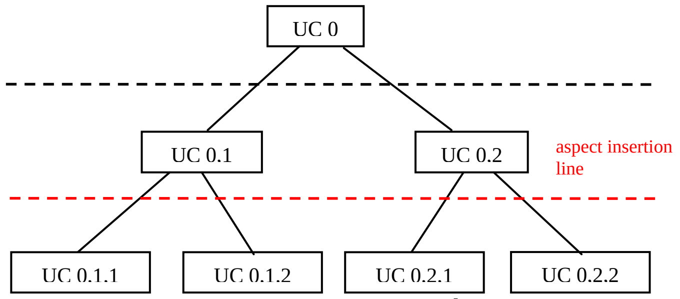


Figure 5 –aspect insertion line

Figure 5 shows a simple use case hierarchy. Separation between any two levels of the hierarchy (shown as a dotted lines) is a potential aspect insertion line. In other words, some intermediate use case level(s) can be inserted here in order to implement a crosscutting concern. Note that the use cases are numbered with a convention that allows the designer to easily know the place of the module in the use case set hierarchy.

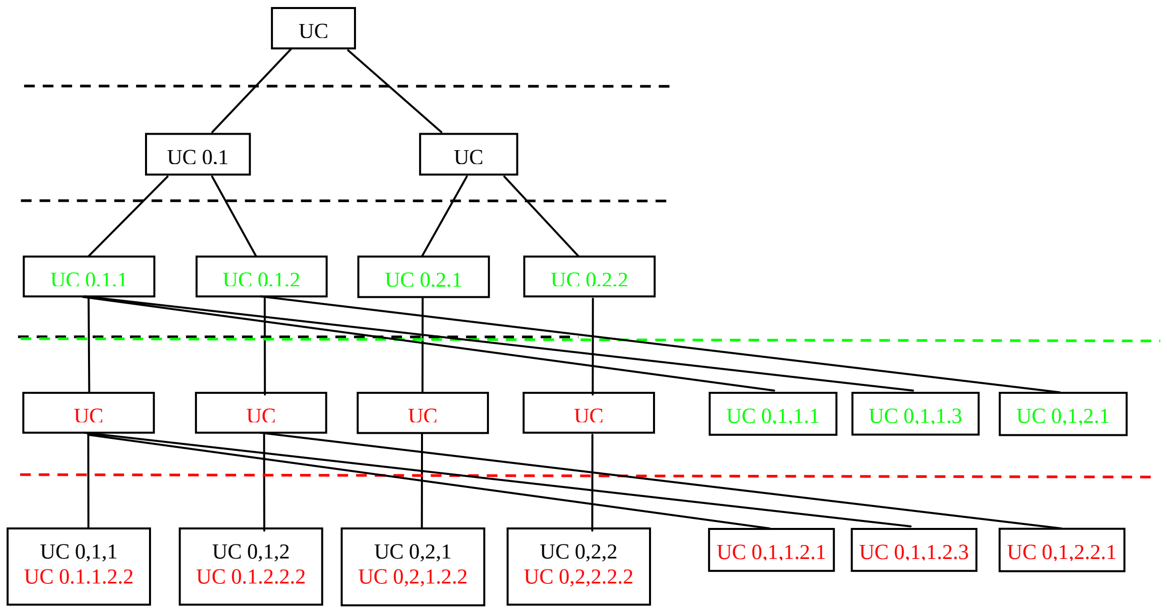


Figure 6 – composition behavioral aspects

Figure 6 show a composition example involving two behavioral aspects. On the left part, the use cases deal with the composition of the new inserted use cases. On the right hand side, one can see the actually inserted new use cases. By using an appropriate simple tool that would allow use case renumbering, it would be possible to control the composition process so that it is (i) automatic in the simple cases, (ii) configurable in the more complex or difficult cases – simply by defining specific composition use cases, and (iii) traceable. Traceability is very important, since an effective methodology should be able to support the easy adding or removing of an aspect in a controlled way. Additionally, the underlying tools must be able to perform the mapping between the design-level aspects and the implementation-level aspects.

4 CONCLUSION

In this short paper, we showed that the interesting and intuitive ideas put forth by Jacobson in his recently published paper need enhancement. Although at a fundamental level we agree with him, it is important to recognize in the larger context that ideas put forth in AO do go beyond the before/after notions used therein and that if we really wished to make use cases and AO work together then the pointcut definition issues and the composition issue are also to be considered.

In the last section of this paper, we give a short overview of our ongoing project called ELSA. We show how ELSA deals with the aforementioned issues. Whilst the project is still on-going, indications are there that such an approach has a good chance to succeed. We also agree with Jacobson that this approach would actually bring up an entirely new perspective to software engineering and would really have deep positive impacts on all the phases of developing large-scale applications.

REFERENCES

[Bobrow86] D. Bobrow et al.: “CommonLoops: Merging Common Lisp and object-oriented programming”, *ACM Conference on Object-Oriented Systems, Programming, Languages and Applications*, September 1986, pp. 17-29.

[Cannon82] H. I. Cannon. Flavors: “A non-hierarchical approach to Object Oriented Programming”, *Symbolics*, Inc., 1982.

[Chiba95] S. Chiba, “A Metaobject Protocol for C++”, *Proceedings of the 10th Annual Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'95)*, pages 285-299, 1995.

[Ferber89] J. Ferber, “Computational Reflection in Class Based Object Oriented Languages”, *Proceedings of the 4th Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'89)*, pages 317-326, 1989.

[Foot89] B. Foot and R. E. Johnson, “Reflective Facilities in Smalltalk-80”, Proceedings of ACM Conference on Object-Oriented Programming System, Languages and Applications, pages 327-335, 1989.

[Jacobson03] I. Jacobson: “Use Cases and Aspects – Working Seamlessly Together”, *Journal of Object Technology*, vol. 2, No. 4, July-August 2003, pp. 7-28. http://www.jot.fm/issues/issue_2003_07/column1.

[Jacobson79] I. Jacobson: “Use Case Modularity”. *Ericsson internal document X/Tg* 2618, 1979.

[Kiczales91] G. Kiczales, J. des Rivieres and D.G. Bobrow, “The Art of the Metaobject Protocol”, *MIT Press*, 1991.

[Kiczales97] G. Kiczales et al., “Aspect-Oriented Programming”, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.

[Kiczales01] G. Kiczales et al., “An Overview of AspectJ”, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 327-353, 2001.

[Maes87] P. Maes, “Concepts and Experiments in Computational Reflection”, *Proceedings of the 2nd Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'87)*, pages 147-155, 1987.

[Moon86] D. A. Moon: "Object-oriented programming with Flavors", *Norman Meyrowitz, editor, Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1-8, New York, NY, November 1986. ACM Press.

[Pawlak99] R. Pawlak, L. Duchien and G. Florin, "An Automatic Aspect Weaver with a Reflective Programming Language", *Proceedings of Reflection'99*, 1999. *Note that the main results of this research have been submitted to AOSD04.*

[Pawlak01] R. Pawlak et al., "JAC: A Flexible Solution for Aspect-Oriented Programming in Java", *Proceedings of Reflection 2001*, pages 1-21, 2001.

[Sihman02] M. Sihman and S. Katz, "A Calculus of Superimpositions for Distributed Systems", *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 28-40, 2002.

[Smith84] B. C. Smith.: "Reflection and semantics in Lisp", *Conf. Rec. 11th ACM Symposium on Principles of Programming Languages*, pages 23--35, 1984.

[Steele90] G. L. Steele, Jr.: *Common Lisp: The Language*, Second Edition, Bedford, Mass.: Digital Press, 1990.

[Sullivan01] G. T. Sullivan, "Aspect-Oriented Programming using Reflection and Metaobject Protocols", Communications of the ACM, pages 95-97, 2001.

About the authors

Renaud Pawlak (pawlakr@rh.edu) is a research fellow at the RPI Hartford graduate campus. He holds a Ph.D. from the CNAM of Paris, France. His main subjects of interest are Reflection and AOP. He is the founder of the JAC project (http://jac.objectweb.org), an AO framework in Java, and tries to focus the Java and J2EE community interest to AOP with the AOP Alliance project (http://aopalliance.sf.net).

Houman Younessi (houman@rh.edu) is professor of computer science at the RPI Hartford graduate campus and the director of Rensselaer Initiative in Software Engineering (RISE). He holds a PhD in computer science from Swinburne University of Technology in Melbourne Australia. His main areas of interest include object and component technologies, aspect orientation and software architecture, particularly integration and platform issues.