Evolving Pointcut Definition to Get Software Evolution

Walter Cazzola[1], Sonia Pini[2], and Massimo Ancona[2]

[1] Department of Informatics and Communication,
Università degli Studi di Milano, Italy
cazzola@dico.unimi.it
[2] Department of Informatics and Computer Science
Università degli Studi di Genova, Italy
{pini|ancona}@disi.unige.it

Abstract. In this paper, we have briefly analyzed the aspect-oriented approach with respect to the software evolution topic. The aim of this analysis is to highlight the aspect-oriented potentiality for software evolution and its limits. From our analysis, we can state that actual pointcut definition mechanisms are not enough expressive to pick out from design information where software evolution should be applied. We will also give some suggestions about how to improve the pointcut definition mechanism.

Keywords: AOP, Software Evolution, Design Information, UML, Pointcut Definition

1 Software Evolution: What is it?

Nowadays a topical issue in the software engineering research area consists of producing software systems able to adapt themselves to environment changes by adding new and/or modifying existing functionality. This characteristic is called *software evolution*.

The term *evolution* may, generally, be interpreted and studied from several distinct points of view. In general software evolution implies to reengineering the design and the code of software systems. Software evolution and maintenance can be categorized into [9]: *corrective*, *adaptive*, *perfective*, and *preventative*. The criteria that govern this taxonomy are well identified by the motivations that render necessary the evolution, e.g., adaptive software evolution is necessary when new functionality are required.

Nonstopping applications with long life span are typical applications that have to be able to dynamically adapt themselves to sudden and unexpected changes to their environment. Therefore, the support for run-time adaptive software evolution is a key aspect of these systems.

Design information provides all the necessary data for governing software evolution and is often used for manually evolving systems that can be stopped. Object oriented methodologies for software development, as UML [1], describe the system's behavior, architecture and components; all functions in the system are captured by a use-case model and the dynamic behavior of each use-case is described by scenarios and interaction diagrams. Therefore, the automatic reengineering of the design information of

a non-stopping system should represent the perfect tool for dynamically adapting such kind of of systems.

Unfortunately, design data are difficult to manage automatically but especially it is difficult to automatically generate working code from the design and inject it in the running system. In this case, the evolution can be carry out by defining some mechanisms that face the occurred events, manipulate the UML diagrams and then inject such a changes directly and automatically in the code. As discussed in [2, 3] the diagram manipulation is feasible by working on their XMI representation and by using a set of reconfigurable rules for planning the adaptation but the code injection is still far from being achieved.

Software evolution that involves a generic system is usually carried out stopping the system and manually, or with the aid of specific tools, changing the system code according with the required evolution. On the other hand, a similar approach is not feasible when the system subjects to the evolution cannot be stopped, e.g., because provides a critical service as a monitoring system.

Independently of the mechanism adopted for planning the evolution, the evolution of a nonstopping system requires a mechanism that permits of concreting the evolution on the running system. In particular this mechanism should be able of i) picking the code interested by the evolution out of the whole system code, ii) carrying out the patches required by the planned evolution on the located code.

Both computational reflection [8] and aspect-oriented programming [5] provide mechanisms (introspection and intercession the former and aspect weaving the latter) that allow of modifying the behavior and the structure of an application, also of a non-stopping application. Reflective mechanisms mainly focus their efforts on dynamically modifying the system on a per object-basis whereas the AOP mechanisms better address functionality that crosscut the whole implementation of the application. Evolution is a typical functionality that may crosscut the code of many objects in the system.

2 Why could AOP be useful for Software Evolution?

Aspect oriented programming (AOP) [5] is a designing and programming technique that takes another step towards increasing the kinds of design concerns that can be cleanly captured within source code. Its main goal consists of providing systematic means for the identification, modularization, representation and composition of crosscutting concerns such as security, mobility and real-time constraints. Moreover, the captured aspects (both functional and nonfunctional) are separated into well-defined modules that can be successively composed in the original or in a new application.

Where the tools of OOP are inheritance, encapsulation, and polymorphism, the components of AOP are *join points*, *pointcut*, and *advice*. Join points represent well-defined points in a program's execution, such as method calls, field get and set methods. Pointcut is a construct that picks out a set of join points based on defined criteria, such as method names and so on. Pointcuts serve to define an advice. An advice picks out additional code to be executed before, after, or around join points. Typical implementations of object-based AOP frameworks insert hooks at the join points. An advice is executed when the execution reach the corresponding (i.e., picked out by a pointcut)

join point. AspectJ [7] is one of the most relevant frameworks supporting the AOP methodology.

AOP can be classified in *static* and *dynamic* AOP. The systems, as AspectJ, compliant to the static approach permit of weaving aspects at compile or load-time. On the other hand, the dynamic AOP approach allows of dynamically plugging and unplugging aspects at run-time widening the applicability spectrum of the AOP methodology. The dynamic AOP approach requires a support middleware at run-time, called *execution monitor*; the system raises a callback to that middleware to notifies that a hooks has been encountered. The middleware takes also care of executing the advice.

Many frameworks support or may be used for implementing dynamic AOP, e.g., JAC [13], DJ [12], Prose [14], Wool [15] and JMangler [6].

From AOP characteristics, it is fairly evident that AOP is on the way of providing the necessary tools for instrumenting the code of a nonstopping system, especially when advices can be run at run-time. Pointcut should be used to pick out a region of the code involved by the evolution, whereas the advice should define the code evolution at the corresponding pointcut. Weaving such an aspect on the running system should either inject new code or manipulate the existing code, allowing the system dynamic evolution.

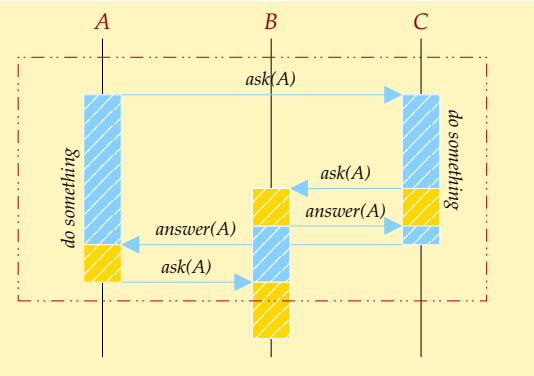
Unfortunately, in the evolution case, pointcut definition is an hard job because the portion of code interested by the adaptation can be scattered around in the code and not confined in a well-defined area that can be taken back to a method call. As pointed out by Tom Tourwé et al. in [17], this problem is due to the poor expressiveness of the pointcut definition languages provided by the actual AOP frameworks.

The developer has to identify and to specify in the correct way the pointcut. To pick out the pointcuts, the developer can use, what we call *linguistic pattern matching*. Nowadays, the pointcut definition languages permit to locate where an advice should be applied by describing the pointcut as a mix of references to linguistic constructs, such as method call or access to variables, and of generic references to the position, such as before or after; the result, for example, looks something like `after the call of method m`. Therefore, it is difficult to define generic, reusable and comprehensible pointcuts that are not tailored on a specific application. Moreover a similar approach is not feasible when the pointcut should involve code that spans among several classes, as in the case of a pointcut describing a collaboration among objects.

3 Towards a Pointcut Definition Driven by UML Diagrams

Several mechanisms for achieving software adaptability have been proposed [18, 11]. The approach that we believe the most promising consists of integrating a reflective architecture as the one proposed in [3] with an AOP framework. The reflective middleware has to take care of deciding the extent of the evolution and which code is affected by such an evolution. Whereas the AOP framework has to dynamically weaving the planned evolution on the join points picked out by the reflective architecture. Both frameworks should perform their duty manipulating the design information of the system prone to be adapted.

It is relatively simple to plan the system evolution by manipulating its UML diagrams and similarly it is quite simple to detect the extent of the code modification and which objects are affected by them from the design information. On the other hand, to use design information to pick out a set of pointcut is not so simple because what it is concisely described by a diagram or a portion of a diagram might be implemented by many instructions disseminated in several part of the code or, analogously, what it is abstracted in several entities by the design diagrams can be implemented as a single entity by the system code. Besides, as discussed in [17], computational patterns, easily recognizable in a sequence or in a collaboration diagram, are not trappable by actual pointcut definition languages.



The above reported portion of sequence diagram describes a quite frequent collaboration among three entities, `A`, `B` and `C`; `A` asks to `C` to intercede with `B` on its behalf, then, after the successful intercession, `A` will interact with `B`. This schema could be used to describe a control access protocol with an external authenticator. Notwithstanding that, it is quite simple to understand the computational pattern described by this sequence diagram is not so simple to pick out which code realize it, especially on a pattern matching bases. In fact, the diagram just describe the order and which operation MUST be done, but nothing is said about which code do that and about what `A` is doing while it is waiting an answer from `C`. Therefore, the computational pattern expressed in the squared portion of the reported sequence diagram cannot be picked out by the actual pointcut definition languages. As stated in [17] something can be done refactoring the code in order to localize and to encapsulate the code of each entity in a method, but still nothing can be done to pick out the collaboration among three entities.

Problems related to pointcut definition have been raised by several researchers [17, 4], in all their works they propose to use a more expressive pointcut definition language mainly based on logic deduction and pattern matching. Notwithstanding the powerfulness of their proposals, they cannot deal with the straightforwardness and the abstraction provided by a UML diagram. A pointcut defined in term of UML diagrams picks out portion of code otherwise not identifiable, as explained above. Sillito et al., in [16], highlighted the importance of using *use case* diagrams in the pointcut definition, our idea is quite similar but we do not want to define a novel pointcut definition language, as AspectU, that needs a special interpreter or to be mapped on an existing AOP lan-

guage, as AspectJ. Rather we would like to extend an existing pointcut language and act on the weaving mechanism to support UML-based join points.

Our proposal consists of using UML diagrams or portion of UML diagrams to describe where the advice should be woven. In this way, pointcuts are not tailored on the program to adapt but they are more general and represent patterns applicable to several computational flows. Of course, we will use a textual representation instead of a graphical one based on the XMI standard [10] to define our pointcuts. Moreover, we will exploit meta-data code annotations as supported by .NET or Java (version 1.5) to introduce XMI code, that will play the role of the hooks, in the code to be adapted. Annotations have the benefit to be supported by standard programming environments and to be skipped during the normal execution, i.e., in this case, when no aspect is woven on that annotation; therefore they should not add extra penalties during the execution.

4 Conclusions and Future Work

In this position paper, we have analyzed aspect-oriented development techniques in relation with the software evolution problem. In particular, we have focused our analysis on the approach to software evolution that we believe the most promising: software evolution driven by design information. From our examination results that with actual mechanism for pointcut definition is hard to pick out the code described by UML diagrams that with a higher abstraction level. Similar issues related to pointcut definition have also been raised by other researchers [17, 4]. Our proposal consists of marking the code with the corresponding UML diagrams (hooks for the weaving mechanism) and of using such diagrams in the definition of the pointcuts and therefore in helping to pick out where evolution should take place. In the next, we will extent the pointcut definition language of an existing AOP framework, as AspectJ, with the possibility of using UML diagrams as pointcut as well. Analogously, we will enable the weaving mechanism to act in conjunction with join points specified by such a kind of pointcuts.

References

1. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, third edition, February 1999.
2. Walter Cazzola, James O. Coplien, Ahmed Ghoneim, and Gunter Saake. Framework Patterns for the Evolution of Nonstoppable Software Systems. In Pavel Hruby and Kristian Elof Søresen, editors, *Proceedings of the 1st Nordic Conference on Pattern Languages of Programs (VikingPLoP'02)*, pages 35–54, Højstrupgård, Helsingør, Denmark, on 20th-22nd of September 2002. Microsoft Business Solutions.
3. Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Software Evolution through Dynamic Adaptation of Its OO Design. In Hans-Dieter Ehrich, John-Jules Meyer, and Mark D. Ryan, editors, *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software*, Lecture Notes in Computer Science, pages 69–84. Springer-Verlag, Heidelberg, Germany, February 2004.
4. Kris Gybels and Johan Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *Proceedings of the 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD'03)*, pages 60–69, Boston, Massachusetts, April 2003.

5. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *11th European Conference on Object Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science 1241, pages 220–242, Helsinki, Finland, June 1997. Springer-Verlag.
6. Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler - A Powerful Back-End for Aspect-Oriented Programming. In Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-oriented Software Development*, chapter 9. Prentice Hall, 2004.
7. Ramnivas Laddad. *AspectJ in Action: Pratical Aspect-Oriented Programming*. Manning Pubblications Company, 2003.
8. Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.
9. Tom Mens, Jim Buckley, Matthias Zenger, and Awais Rashid. Towards a Taxonomy of Software Evolution. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution (USE 2003)*, Warsaw, Poland, April 2003.
10. OMG. OMG-XML Metadata Interchange (XMI) Specification, v1.2. OMG Modeling and Metadata Specifications available at `http://www.omg.org`, January 2002.
11. Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the International Conference on Software Engineering (ICSE'98)*, pages 177–186, Kyoto, Japan, April 1998.
12. Doug Orleans and Karl Lieberherr. DJ: Dynamic Adaptive Programming in Java. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, LNCS 2192, pages 73–80, Kyoto, Japan, September 2001. Springer.
13. Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, LNCS 2192, pages 1–24, Kyoto, Japan, September 2001. Springer.
14. Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just in Time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of the 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD'03)*, pages 100–109, Boston, Massachusetts, April 2003.
15. Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A Selective, Just-in-Time Aspect Weaver. In *Proceedings of the 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE'03)*, LNCS 2830, pages 189–208, Erfurt, Germany, September 2003. Springer.
16. Jonathan Sillito, Christopher Dutchyn, Andrew D. Eisenberg, and Kris De Volder. Use Case Level Pointcuts. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*, Oslo, Norway, June 2004.
17. Tom Tourwé, Andy Kellens, Wim Vanderperren, and Frederik Vannieuwenhuyse. Inductively Generated Pointcuts to Support Refactoring to Aspects. In *Proceedings of Software engineering Properties of Languages for Aspect Technologies (SPLAT'04)*, Lancaster, UK, March 2004.
18. Emiliano Tramontana. Managing Evolution Using Cooperative Designs and a Reflective Architecture. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 59–78. Springer-Verlag, Heidelberg, Germany, June 2000.