A case study on making the transition from functional to fine-grained decomposition

Constantinos A. Constantinides

School of Computer Science and Information Systems
Birkbeck College, University of London, Malet Street, London WC1E 7HX, UK
Email: cc@dcs.bbk.ac.uk, WWW: http://www.dcs.bbk.ac.uk/~cc

Abstract: In this paper we argue that the explicit capture of crosscutting concerns in code should be the natural consequence of good and clean modularity, based on fine-grained decomposition, and not the result of a corrective measure of tangled implementation. The main theme of this paper is how to adapt established analysis and design techniques for object-oriented systems to an aspect-oriented context, thus making the transition from functional to fine-grained decomposition.

1. Introduction

Aspect-Oriented Programming (AOP) has come to existence as a proposed solution after the observation of two symptoms during program construction: (1) code scattering, and (2) code tangling. Both symptoms appear as a result of crosscutting code. Even though providing the linguistic means to explicitly capture crosscutting concerns during implementation is necessary, it is not sufficient in order to fully support advanced separation of concerns in software engineering. This is because between a nice idea and a working software that implements it there is much more than programming.

It is imperative that development includes analysis and design activities, which involve investigating the problem domain and devising logical solutions, while being documented by software "blueprints", illustrated by some modeling language, like the Unified Modeling Language (UML) [12]. These blueprints serve as a tool for thought and as a form of communication.

Functional (traditional) decomposition in object-oriented systems focuses on the decomposition of systems in terms of classes which, primarily, target functional requirements. Functional decomposition cannot support the representation of crosscutting concerns as modular units. As a result, the implementation of crosscutting concerns ends up being tangled inside classes.

Implementation is the mapping from design to code and in some cases it can be performed automatically using a number of available tools. However, in order to produce a clean (tangled-free) implementation, the design artifacts themselves (such as UML interaction diagrams and the class diagram) must in turn explicitly address crosscutting concerns. Developers have realized that, in order to fully grasp the nature of crosscutting concerns as well as in order to produce aspectual software, they must visit the root of the problem. As a result, they must study the nature of crosscutting concerns and provide the means to identify and model them from the early stages of the software life cycle. To this end, Aspect-Oriented Software Development (AOSD) has extended AOP to include aspect-oriented activities throughout software development.

In order to have a clean mapping from requirements to analysis, and subsequently from analysis to design and from design to implementation, developers have to revisit their traditional decomposition strategies and treat crosscutting concerns on an equal basis as components. Fine-grained decomposition implies that analysis and design artifacts are built with no dominance of components over crosscutting concerns. As a result, the explicit capture of crosscutting concerns in code would be the natural consequence of good and clean modularity and not the result of a corrective measure of tangled implementation.

Naturally, this does not guarantee that fine-grained decomposition can never produce some level of tangling, especially as requirements might change during development and new concerns may show up during implementation. However, with the aid of an iterative software development process, developers should be able to control changing requirements and produce quality software.

In this paper we will present a case study to investigate the identification and nature of crosscutting concerns throughout the development process as well as to see how crosscutting requirements can propagate and remain visible in requirements, analysis and design artifacts. The main theme of this study is how to adapt established analysis and design techniques for object-oriented systems to an aspect-oriented context.

2. Case study: Building a news service

Consider an online news service. Potential users and other stakeholders can state their needs (goals) and expectations as follows: The server should allow multiple clients to concurrently access information through collections of news items stored in a database. Access to the server requires client registration, thus establishing an account for logging in. There must also be a maximum allowable number of registered users that can be concurrently logged in. Furthermore, the server should allow certain designated clients to post information. However, only one client (writer) may post information at a time. The

In Proceedings of the ECOOP Workshop on Analysis of Aspect-Oriented Software, Darmstadt, July 21, 2003.
Last updated: July 16, 2003.

server assigns priority to writers over clients who read news (readers). One can easily identify the set of requirements to constitute a variation of the readers-writers protocol. Another requirement dictates that the server must provide certain real-time capabilities by holding constantly updated information of some data, where client software must be notified every time there is a change of state in the server in order to request the new state and update itself.

3. Adopting an iterative software development process

An iterative development process such as the one defined by the Unified Software Development Process (UP) [8] is organized into a series of short fixed-length mini-projects called iterations, where each iteration represents a complete development cycle: it includes its own requirements, analysis, design, implementation and testing activities. The outcome of each iteration is a tested, integrated, and executable system. Stakeholders usually have changing requirements. An iterative process embraces change by dictating that each iteration should involve tackling new requirements (by choosing a small subset of the requirements, or perhaps revisiting previously addressed requirements for improvement) and quickly designing, implementing and testing them. As a result, the system grows incrementally over time, iteration by iteration. This leads to rapid feedback, and provides an opportunity to modify or adapt understanding of the requirements or design.

We can argue that the ability for the system to provide concurrent access and to observe the correct semantics of the readers-writers protocol constitutes the high-value and high-risk requirements of the project, that will have to be addressed in early iterations.

3.1 Capturing requirements

There is no consistent manner in which the term “requirement” is used in the software industry as the term is expressed in a number of ways ranging from high-level abstract statements to mathematically formal definitions of system functions [11]. However, there seems to be a consistency followed by a number of authors about the fact that requirements constitute system descriptions [11], or system capabilities [10] or features that systems must have [2] as well as constraints that systems must satisfy to be accepted by clients. In the literature, requirements are widely placed in two groups: functional and nonfunctional, even though some authors [1] dislike this generalization. Functional requirements refer to the functionality or services that the system is expected to provide [11], describing interactions between the system and its environment, while not focusing on implementation details [2]. Functional requirements are observable during software execution. Nonfunctional requirements on the other hand is a collective term adopted to describe user-visible aspects of the system that are not directly related with the functional requirements [2] while attaching constraints to these services, thus affecting the performance and semantics of the system. Many nonfunctional requirements would relate to the system as a whole rather than to individual system features. Nonfunctional requirements include usability, reliability, performance and supportability. Some of these are collectively called the “ilities” of a system or quality requirements (or attributes).

Requirements elicitation is a process that provides a specification that clients can understand, and requirements analysis provides a model that developers can unambiguously interpret [2]. In [11] the author provides a separation of terms by adopting the term “user requirements” to refer to high-level abstract descriptions, and “system requirements”, that precisely describes the system services and constraints.

Use cases are stories of using the system to meet goals and can be used to bridge the gap between the two [2]. Use cases[1] are generally used to capture functional requirements [7]. In the subsequent subsections we will discuss the use cases for posting and reading news. We shall focus on posting news and use it as a sole requirement in order to complete a single development iteration.

Consider the partial definition of use cases Post News and Read News in fully dressed format in Figures 1, and 2 respectively.

Use Case	Post News
Primary Actor	Client
Stakeholders and interests	…
Preconditions	1. Client is authenticated. (authentication) 2. Server has not reached its maximum allowable number of clients logged in. (authentication) 3. No other client is currently reading news. (synchronization) 4. No other client is currently posting news. (synchronization) 5. No other client is currently waiting to post news. (scheduling)
Postconditions	…
Main success scenario	A writer client logs on to the server in order to start a posting session and may post a number of news items. Each news item receives an individual confirmation and at the end of posting, the system issues a general confirmation. News items are stored in a database.
…	

Figure 1. Use case Post News

We can observe that certain requirements (like authentication, synchronization and scheduling), expressed as preconditions, tend to cut across these two use cases. Even though more use cases can be identified (Figure 3), they would fall into the general categories of write (Post News, Delete News) or read

[1] It is perhaps important to note that use cases are not necessarily object-oriented but they can be nicely blended in object-oriented analysis and design.

(Read News, Search News) and where applicable we will refer to clients as writers and readers.

Use Case	Read News
Primary Actor	Client
Stakeholders and interests	…
Preconditions	1. Client is authenticated. (authentication) 2. Server has not reached its maximum allowable number of clients logged in. (authentication) 3. No other client is currently posting news. (synchronization) 4. No other client is currently waiting to post news. (synchronization) 5. Multiple clients may concurrently read news. (scheduling)
Postconditions	…
…	

Figure 2. Use case Read News

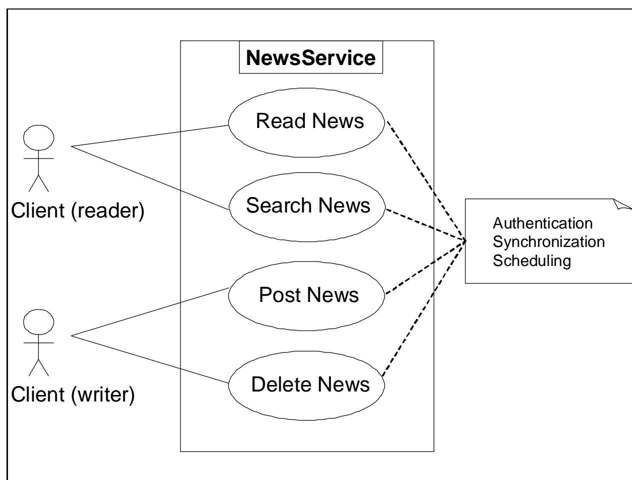

Figure 3. Use case diagram

3.2 From requirements to analysis

Object-oriented analysis investigates the problem domain and its requirements, by placing emphasis on finding and describing the concepts of the problem domain [10]. A domain model is a static structure diagram that illustrates (real-world) concepts in the problem domain. In [10] the author discusses two mechanisms by which developers can identify conceptual classes: (1) linguistic analysis (noun-phrase identification) based on the scenarios of the use cases, and (2) a candidate conceptual class category list. In this example we extend linguistic analysis to apply to preconditions of use cases in order to identify crosscutting concerns.

From the description of the Post News use case we can create a partial domain model as shown in Figure 4. Authentication, synchronization and scheduling constitute concerns that have been identified as crosscutting during the description of use cases and they are expressed as conceptual classes that define policies associated with `NewsServer`.

During analysis, it is useful to investigate and define the behavior of the software as a "black box", that is to provide a description of what the system does (without an explanation of how it does it). Use cases describe how external actors interact with the software system. During this interaction, an actor generates events that initiate operations upon the system. A system sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate (including their order) [10]. An SSD treats a system as a black box, placing emphasis on events that cross the system boundary from actors to systems.

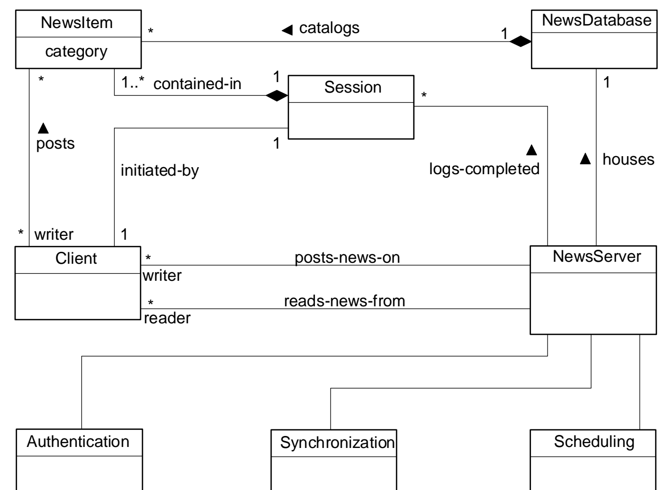

Figure 4. Partial domain model

The SSD for Post News use case is illustrated in Figure 5. The order of activation of authentication, synchronization and scheduling is dictated by the semantics of the readers-writers protocol, which is implemented by the system. Consider all guards in Figure 5: Authentication must be evaluated first, followed by the check of synchronization constraints, which define the eligibility of an entity for execution. Scheduling constraints refer to what must be done between eligible clients, and they are evaluated just before `addNewsItem(item)` is invoked.

Overall, client authentication would have to be performed before the server would try to determine the eligibility of a client for using a service (addressed by the synchronization aspect) and before the determination of what should be the race condition between eligible clients (addressed by the scheduling aspect). The above define the "before" behavior of a service. The "after" behavior of a service dictates that notifying waiting clients (scheduling) must be performed before updating synchronization counters, which in turn must be performed before updating authentication counters (shown in Figure 6 for Post News). In Figure 5, the "after" behavior for Post News is initiated after the termination of `addNewsItem(item)` (sched-

uling and synchronization) and upon reception of `endPostNews()` (authentication).

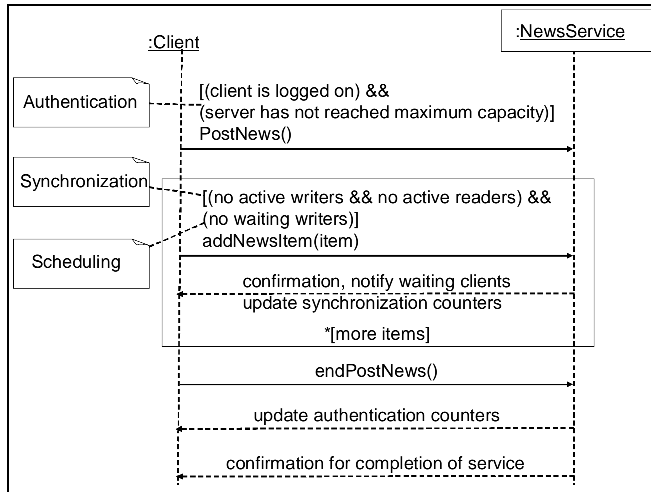

Figure 5. System sequence diagram for use case Post News

The order in which use case preconditions are defined, brings up the notion of non-orthogonality between concerns. In general, it is highly unlikely that crosscutting concerns are completely independent of each other (we refer to this as orthogonality of concerns). Experience has shown that concurrent systems contain a number of non-orthogonal aspects such as synchronization-scheduling, security-performance, and security-logging. In fact security and performance tend to be non-orthogonal to almost every other aspect of a system. The issue of non-orthogonality is important, as it addresses the overall semantics of the system. Close to this is the issue of the order of activation of aspects.

```
// "before" behavior
preactivation(Post News):
<authentication, synchronization, scheduling>

// "after" behavior
postactivation(Post News):
<scheduling, synchronization, authentication>
```

Figure 6. Semantics for Post News

The set of all required system operations within a SSD is determined by identifying the system events. From the SSD of Figure 5, we identify a number of system operations shown in Figure 7.

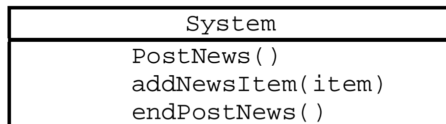

Figure 7. System operations

The set of system operations of Figure 7 constitutes a subset of the overall behavior of the system, as the entire set of system operations (across all use cases) defines the complete public interface of the system [10]. In order to investigate the system behavior with respect to a system operation, we can build an operation contract. We would generally choose to build operation contracts for those system operations that pose complex behavior (especially those that include complex preconditions and postconditions). In this example, we choose to create a contract for the operation `addNewsItem(item)` shown in Figure 8.

Operation	`addNewsItem(item)`
References	Use case Post News
Preconditions	1. No other client is currently posting news. (synchronization) 2. No other client is currently reading news. (synchronization) 3. No other client is waiting to post news. (scheduling)
Postconditions	1. A `NewsItem` instance `ni` was created. 2. `ni` was associated with `Session` 3. `ni.category` was set to `category`.

Figure 8. Operation contract for `addNewsItem(item)`

3.3 From analysis to design

Object-oriented design emphasizes a conceptual solution by defining software objects and how they collaborate in order to fulfill the requirements [10]. The next step during development would be to create an interaction diagram for each operation contract. Interaction diagrams illustrate how objects interact via messages. The interaction diagram for `addNewsItem(item)` is shown in Figure 9.

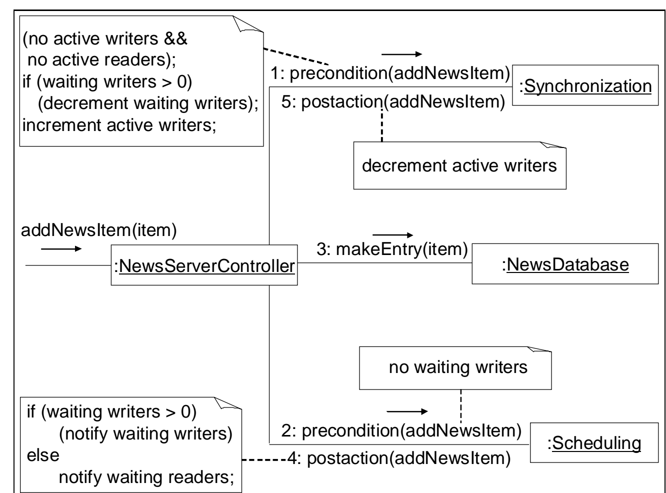

Figure 9. Interaction diagram for `addNewsItem(item)`

The semantics of synchronization constraints pose a precondition that there should be no active readers and no active writers. Furthermore, scheduling constraints pose a precondition that no other client is currently waiting to post. Once both precondi-

tions evaluate to true, the system must obtain a lock to the shared resource (instance of `NewsDatabase`) before write permission is granted. Once the operation is complete, the lock must be released after the system has updated scheduling and synchronization counters, an action defined in this example by what we refer to as a postaction ("after") behavior. More specifically, the postaction of scheduling will notify waiting writers, if there are any, (thus giving priority to a write operation over read) followed by a notification of waiting readers if there are any. The postaction of synchronization involves decrementing the count of active writers. Once the interaction diagrams have been completed it is possible to identify the specification for the software classes and interfaces and illustrating them in a class diagram (Figure 10). A class diagram depends upon the domain model and interaction diagrams. Ideally a design should be language (implementation) independent. The implementation of this example should be made by a system which can support aspect definitions and which provides a mechanism for the coordination of component and aspect code. There are plenty of mechanisms that one can adopt for implementing this system including (but not limited to) purely linguistic like AspectJ [9], or aspect-oriented frameworks[2]. The pseudocode of the scheduling aspect definition of method `addNewsItem(item)` is shown in Figure 11. This example illustrates that the coding of aspect definitions comes as a natural consequence of design. The example further illustrates that crosscutting concerns are modeled as first-class abstractions.

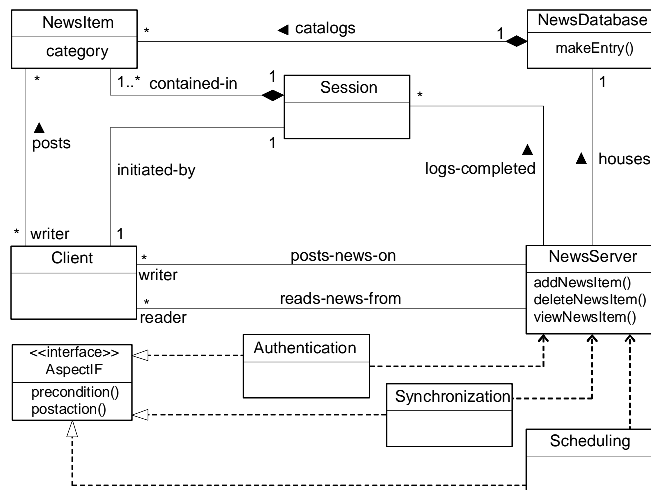

Figure 10. Class diagram

```
precondition (addNewsItem):
   if  (waiting writers == 0) then RESUME
   else wait;
postaction (addNewsItem):
   if  (waiting  writers  >  0)  then  (notify
   them)
   else (notify waiting readers);
```

Figure 11. Scheduling constraints for `addNewsItem(item)`

3.4 Addressing further requirements

The previous iteration addressed posting of news. In general initial iterations will deal with high-risk and high-value requirements. Other requirements of the same importance (e.g. reading news, searching for news, deleting news) could have been addressed in the same iteration, or can be handled in subsequent iterations in a similar way. Further iterations will address low-risk requirements. In this example, we assume that Post News, Read News, Search News and Delete News have been addressed by a number of iterations along the same lines as Post News in sections 3.1 – 3.3.

In a later iteration we can address the requirement which dictates that the server must provide certain real-time capabilities by holding constantly updated information of some data, as it could be the case with stock exchange data. Clients who register for this service hold a local copy of the data that must be maintained in synchronicity with the state of the server. Client software must be notified every time there is a change of state in the server in order to request the new state and update itself. This requirement can be easily achieved by the Observer design pattern [4]. This particular design pattern poses aspectual characteristics, as it is clear that a traditional implementation of the protocol would be scattered across a number of classes in the system. The Observer design pattern can be treated as an aspect in the system that can be added in a non-invasive way. Aspectual implementation of design patterns has been discussed in the literature in [5]. An updated (partial) class diagram of the system is illustrated in Figure 12.

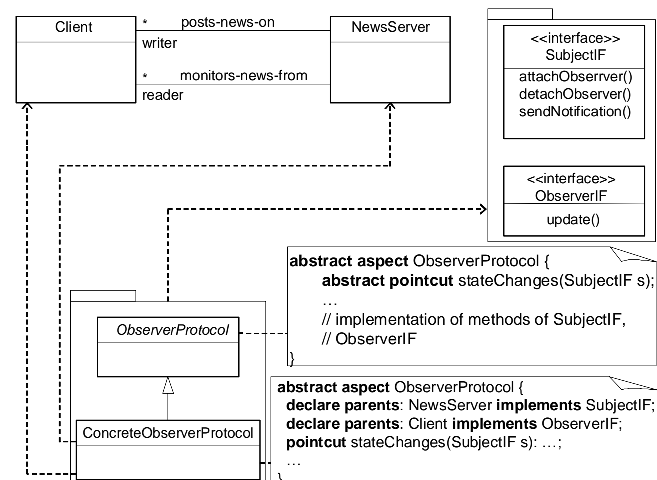

Figure 12. Updated class diagram (partial)

4. Obliviousness and visibility considerations

In a paper [3] that seems to have received a wide acceptance by the research community, Filman and Friedman argue that the provision of obliviousness is a necessary condition for aspect systems. This principle implies that component development should be oblivious from the fact that the semantics of the system may be modified at some later stage through the introduction of aspect code. Obliviousness brings up the notion of visi-

[2] Current AOSD technologies are indexed at `http://aosd.net`

bility between components and aspects. Visibility implies dependency and consequently coupling between components (aspects and system core concerns). In general, coupling can be viewed as restricting the scope of application of the components to their direct environment. In particular, strong coupling decreases component reusability. In [10] the author defines visibility as the ability of one object to see or have a reference to another object. There are four ways that visibility can be achieved from object A to object B: (1) attribute visibility (2) parameter visibility, (3) locally declared visibility, and (4) global visibility.

In this case study, the development of components has been carried out independently of the existence of aspects. The converse, however, is not true as aspects depend on the components whose semantics and performance they affect.

Of particular interest to the implementation discipline are general-purpose aspect languages, like AspectJ[3], a general-purpose aspect-oriented extension to Java[4], that currently seems to be among the most notable AOP technologies and probably the most widely used aspect-oriented language. In [6] we argued that aspect definitions in AspectJ may pose various visibility types based on the constructs the language has introduced[5] in conjunction to all types of visibility relationships already known in object-oriented programming. More specifically, visibility over components can be posed in AspectJ by pointcuts, introductions and advice.

Even though the adoption of AOP results in a good separation of concerns, restricting aspect definitions to match class and method names of system core concerns leads to strong binding between aspects and system core concerns. In such cases aspect definitions are not reusable, but they are restricted to be only applicable in one specific application context. In order to support reuse of aspect definitions, a level of genericity is supported by AspectJ through the provision of abstract pointcuts[6] and abstract aspect definitions (the latter provision is along the lines of abstract classes in Java). One approach in this direction is to provide abstract aspects with abstract pointcuts as library modules, while programmers would need to define concrete pointcuts that refer to application components. This approach, however, will not always provide a high degree of reuse. More specifically, in this example (Figure 12) there seems to be a visibility from the abstract aspect `ConcreteObserverProtocol` to `NewsServer` and `Client` through the introductions of interfaces[7].

[3] AspectJ is a trademark of PARC.

[4] Java is a registered trademark of Sun Microsystems.

[5] It should be noted that these issues are not exclusive to the AspectJ language, but they can be found in other general-purpose aspect-oriented languages, at least the ones whose design dimensions tend to be along the lines of AspectJ.

[6] Some familiarity with AspectJ is assumed. A pointcut is a collection of well-defined points of execution within the code, referred to as joinpoints.

[7] Even though there is no UML notation for this, we choose to illustrate this notion as a dependency relationship (dotted line).

5. Dynamic reconfigurability

An essential requirement for complex systems is that they shall have an open software architecture by which non-invasive adaptability can be performed, i.e. concerns can be dropped, reconfigured or deployed and be connected with newly installed concerns, while the system remains operational and without in each case forcing reengineering of either the system itself or the client code. Existing clients should be able to use the new objects because the new objects must support the operations that the clients request. Dynamic reconfigurability is particularly advantageous in e-commerce and on-line systems such as reservation systems and auctions as well as in mission-critical systems such as air-traffic control. Unavailability of these systems might result either in an increase in non-productive time or raise safety concerns. Essential for the provision of this dynamic behavior in AOSD systems, is the survival of aspects at run-time. Systems should also have to make sure that the provision of this dynamic behavior cannot be misused and thus violate the semantics of the system. If two (or more) crosscutting concerns are non-orthogonal, any action that is directed at one might affect the overall group. As a result, the system must be capable of preventing any possible misuse.

In the case of AspectJ, aspect definitions are first-class abstractions, though they cannot be easily manipulated as regular Java classes.

6. Conclusion

In this paper we presented a case study to investigate the identification and nature of crosscutting concerns throughout the development process, while trying to adapt established analysis and design techniques for object-oriented systems to an aspect-oriented context.

We believe that having access to a language, that can support an aspect-oriented implementation, is necessary but not a sufficient requirement in order to build aspect systems. This is particularly true when reusability and adaptability of both component and aspect code are essential requirements. As a result, we believe that developers should make the transition from functional to fine-grained decomposition, thus treating aspects as equal entities to components throughout the development process. This implies that analysis and design artifacts (such as the ones supported by UML) should pose no dominance of components over aspects. As a result, the explicit capture of crosscutting concerns in code should be the natural consequence of good and clean modularity and not the result of a corrective measure of tangled implementation.

Crosscutting concerns tend to appear during requirements analysis, and they may take different forms during development. During the requirements analysis stage, needs and expectations of stakeholders will be captured in use cases, where preconditions (and postconditions) may provide crosscutting requirements.

Crosscutting concerns appear in system sequence diagrams as constraints. In the domain model crosscutting concerns can be

visualized as conceptual classes. The facilitation of crosscutting concerns as software classes (and thus as first-class abstractions) during design allows for a higher level of reuse and adaptability. Furthermore, systematic composition can be easily achieved through the provision of aspect interfaces.

Fine-grained decomposition and the explicit capture of crosscutting concerns throughout all disciplines of development can also enable developers to trace crosscutting concerns from requirements to code artifacts.

References

1. Bass L., Clements P., and Kazman R., *"Software Architecture in Practice"*, Addison-Wesley, 1998.
2. Bruegge B., and Dutoit A. H., *"Object-Oriented Software Engineering; Conquering Complex and Changing Systems"*, Prentice Hall, 2000.
3. Filman R. E., and Friedman D. P., *"Aspect-Oriented Programming is quantification and obliviousness"*, In Proceedings of the OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Minneapolis, Minnesota, USA, October 16, 2000.
4. Gamma, E., Helm, R., Johnson, R., and Vlissides J., *"Design Patterns; Elements of Reusable Object-Oriented Software"*, Addison-Wesley Longman, Inc., 1995.
5. Hannemann J., and Kiczales G., *"Design Pattern Implementations in Java and AspectJ"*, In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2002), Seattle, WA, 2002.
6. Hassoun Y., and Constantinides C. A., *"Considerations on component visibility and code reusability in AspectJ"*, In Proceedings of Third Workshop on Aspect-Oriented Software Development, Essen (Germany), March 4-5, 2003. University of Duisburg-Essen, Institute of Computer Science and Business Information Systems Technical Report.
7. Jacobson I., *"Object-Oriented Software Engineering: A Use Case Driven Approach"*, Addison-Wesley, 1992.
8. Jacobson I., Booch G., and Rumbaugh J., *"The Unified Software Development Process"*, Addison-Wesley, 1999.
9. Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Griswold W. G., *"An Overview of AspectJ"*, In Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2001), LNCS, Vol. 2072, pp. 327-355.
10. Larman C., *"Applying UML and Patterns; An Introduction to Object-Oriented Analysis and Design and the Unified Process"*, 2nd edition, Prentice Hall PTR, 2002.
11. Sommerville I., *"Software Engineering"*, 6th Edition, Addison-Wesley, 2001.
12. UML Resource Page located at `www.omg.org/uml`