



ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in
Theoretical Computer
Science

Electronic Notes in Theoretical Computer Science 163 (2007) 45–64

www.elsevier.com/locate/entcs

Impact of Evolution of Concerns in the Model-Driven Architecture Design Approach

Bedir Tekinerdogan ¹ Mehmet Aksit ² Francis Henninger ³

*Department of Computer Science, Software Engineering Group
University of Twente
Enschede, The Netherlands*

Abstract

Separation of concerns is an important principle for designing high quality software systems and is both applied in the Model-Driven Architecture (MDA) and Aspect-Oriented Software Development (AOSD). The AOSD and MDA techniques seem to be complementary to each other; historically AOSD has focused on modeling crosscutting concerns whereas MDA has focused on the explicit separation of platform independent concerns from platform specific concerns and the model-driven generation processes. In order to assess the benefits of AOSD for MDA we provide a systematic analysis on crosscutting concerns within the MDA context. The analysis consists of three steps. First, we define an abstract model of MDA transformation with respect to concerns. Second, we define a number of evolution scenarios that correspond to a selected list of crosscutting concerns. Third, we analyze the model transformations in MDA with respect to the abstract model, the evolution scenarios and the related crosscutting concerns. This analysis results in the definition of a number of key problems related to the integration and evolution of crosscutting concerns in the MDA approach. Based on this analysis we provide a set of recommendations for the language and the process that is used in the MDA approach.

Keywords: Crosscutting concern, Evolution, MDA

1 Introduction

One of the most important principles to cope with the complexity in software engineering is the separation of concerns principle. This principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability, maintainability, and reusability.

In this context, Model Driven Architecture (MDA) [5,7] is a framework defined by the OMG [10] that separates the platform specific concerns from platform independent concerns to improve the reusability, portability and interoperability of

¹ Email: bedir@cs.utwente.nl

² Email: aksit@cs.utwente.nl

³ Email: henninger@cs.utwente.nl

software systems. To this end MDA separates Platform Independent Models (PIMs) from Platform Specific Models (PSMs). The PIM is a model that abstracts from any implementation technology or platform. The PIM is transformed into one or more PSMs which include the platform specific details. Finally the PSM is transformed to code providing the implementation details. Obviously by separating the platform specific concerns and providing mechanisms to compose these concerns afterwards in the code MDA provides a clean separation of concerns and as such the systems are better reusable easier to port to different platforms and have increased interoperability.

However, current software systems also have to cope with other concerns than platform specific concerns. Very often software systems also need to deal with other important concerns such as distribution, persistence, synchronization, and error detection. These concerns tend to crosscut various components of the software architecture and as such increase the complexity and decrease the maintenance of software systems.

Aspect-Oriented Software Development (AOSD) [3] aims to cope with these crosscutting concerns by providing explicit abstractions called aspects. By separating the crosscutting concerns in aspects and providing the composition of aspects with the components the impact of crosscutting concerns is better managed. Both AOSD and MDA provide in essence useful techniques for separating the concerns and in that sense AOSD and MDA techniques are complementary to each other; historically AOSD has focused on modeling crosscutting concerns whereas MDA has focused on the explicit separation of platform independent concerns from platform specific concerns and the model-driven generation processes. As such we think that both AOSD and MDA can benefit from each other to even further tackle the challenges of current large and complex software systems.

Since AOSD is primarily focused at solving the problems related to crosscutting concerns we will provide a systematic analysis of crosscutting concerns within the MDA context. The analysis consists of three steps. First, we define an abstract model of MDA transformations as defined by so-called concern transformation patterns (CTP). CTPs characterize the corresponding transformation and help to pinpoint the key problems in the transformation. Second, we define a number of evolution scenarios that correspond to a selected list of crosscutting concerns. The evolution scenarios are applied to a Concurrent Versioning System which is developed using an MDA-based approach. Third, we analyze the model transformations in MDA with respect to CTPs, the evolution scenarios and the related crosscutting concerns. This analysis results in the definition of a number of key problems related to the integration and evolution of crosscutting concerns in the MDA approach. Based on this analysis we provide a set of recommendations for the language and the process that is used in the MDA approach.

In section 2 we will present the concern transformation patterns as an abstraction of different potential transformations. Section 3 explains the case on concurrent versioning systems. This case will be used to explain the notion of crosscutting concerns and to analyze the impact of crosscutting concerns in the MDA process.

Section 4 will define the number of scenarios including the concerns that will be applied to the CVS in the MDA life cycle. Section 5 defines the lessons learned and provides recommendations for coping with crosscutting concerns in the MDA-approach. Finally section 6 will provide the conclusions.

2 Concern Transformation Patterns

To analyze the impact of crosscutting concerns in MDA, we will first consider the major modeling concepts and transformations as defined within the MDA context. As shown in Figure 1, a typical MDA development process consists of model building and transformation activities, starting with a computation independent model (CIM), which is subsequently transformed to a platform independent model (PIM), platform specific model (PSM) and finally to an executable code. It is also possible to transform models at the same abstraction level. Figure 3 shows the following set of transformations: CIM-to-CIM, CIM-to-PIM, PIM-to-PIM, PIM-to-PSM, PSM-to-PSM, and PSM-to-code.

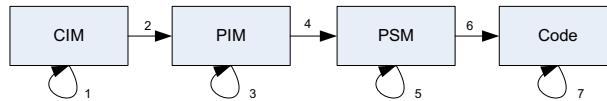


Fig. 1. Simplified View of the MDA Process

Obviously, each model building and transformation process considers a set of possibly new concerns that are relevant to the model being considered. Since AOSD aims to model crosscutting concerns explicitly, it would be logical to analyze the impact of aspects to MDA (or vice versa) from a concern modeling and transformation perspective. Our analysis, therefore, will be based on the following three assumptions:

- A model is a representation of concerns. Each model defined within the MDA context consists of two kinds of concerns: the concerns that are distinguished based on their platform dependency and the concerns that are derived from the requirements and the corresponding problem/solution domain.
- Model transformations are concern transformations. Since every model consists of concerns, naturally every transformation is also a concern transformation.

Based on these assumptions we define eight concern transformation patterns as described in Table 1. In the table A , B and C represent models of concerns, the arrow labeled with T represents a transformation. The brackets $\langle \rangle$ in both the models and transformations include the concerns that are addressed by these. In addition two composition operators are defined \bullet and \otimes . The null-composition operator \bullet represents the required conceptual composition and is defined in the input to the transformation. The operator \otimes defines a composition in which the models can be separately identified. If the models can not be separated after the composition then we use the $\langle \rangle$ symbols. For example, $A_{\langle B, C \rangle}$ refers to a merged

composition of concerns B and C in model A , whereby B and C can not be localized. Note that patterns 1, 3, 5, 6 and 7 lead to such a merged composition in which one or more concerns are not explicitly localized. Patterns 5 to 8 can be considered as composite concern transformation patterns because each of them could be in essence defined as a composition of two other primitive transformation patterns. Pattern 5 could be seen as a composition of pattern 1 and 3, pattern 6 as a composition of patterns 1 and 4, pattern 7 as a composition of patterns 2 and 3, and finally pattern 8 as a composition of patterns 2 and 4.

In the first two patterns the transformations do not include an explicit concern but are mainly used to compose the provided concerns (in this case A and C). In the first pattern the composition result is a merged output in which concern C is not separated anymore. In the second pattern, the concern C is kept separate indicating a more composable design.

Patterns 3 to 8 include transformations that add concerns themselves. In pattern 3 and 4 include situations in which no additional concerns are provided to the transformations. The third pattern represents a transformation in which the model A is transformed to the target model $A_{}$. Here, the concern B is introduced by the transformation process $T_{}$ and it is not separable from the concerns that are originally defined in A . Assume for example that the model A represents a PIM which is transformed to a Java PSM. In case of pattern 3, it is assumed that the Java specific concerns introduced in $A_{}$ cannot be separated from the original concerns of A . In pattern 4, the model A is transformed to the composition of the models A and B as represented by the composition operator \otimes . Similar to pattern 3, model B represents the newly introduced concern by the transformation process. The difference is that the result of transformation $T_{}$ is a modular composition in which model A and B can be separately identified.

The patterns from 5 to 8 assume that the source model consists of two modularly composed modules A and C as represented by the composition operator \bullet . In pattern 5, the original concerns A and C together with the concern as defined in the transformation $T_{}$ are merged in the target model $A_{<B,C>}$ and therefore they cannot be treated separately anymore. In pattern 6, concern C is merged but concern B from the transformation is provided as a separate model. In pattern 7 concern B from the transformation is merged while concern C is represented as a separate model. Finally pattern 8 shows the case in which concerns B and C are separated. In fact this can be considered as the most maintainable alternative because the concerns are fully separated, both in the source and the target.

3 Example: Concurrent Versioning System

To analyze the evolution of concerns in the MDA approach we will utilize the Software Configuration Management (SCM) case. The SCM deals with control of software changes, proper documentation of changes, the issuing of new software versions and releases, the registration and recording of approved software versions. An important functionality in SCM forms the concurrent version control system (CVS)

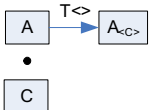
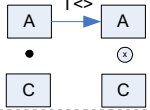
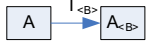
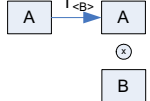
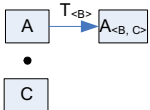
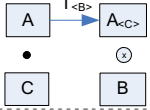
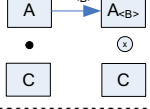
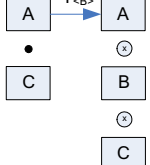


Id	Concern Transformation Pattern	Explanation
1.		<p>A is transformed with concern C to a model $A_{\langle C \rangle}$.</p> <p>A is the dominant decomposition, concern C cannot be separately identified.</p>
2.		<p>A is transformed along with concern C to a model A and model C. Concern C is separately modeled in model C.</p>
3.		<p>A is transformed to $A_{\langle B \rangle}$ via $T_{\langle B \rangle}$. A includes new concern B that is inherent in the transformation but which is not separable.</p>
4.		<p>A is transformed to a composable model of A and B, which are separable in the final result.</p>
Composite Concern Transformation Patterns		
5.		<p>A is transformed along with concern C to a single model $A_{\langle B, C \rangle}$. In the resulting model concern B and C can not be identified any more as separate models.</p>
6.		<p>A is transformed along with concern C to a single model $A_{\langle C \rangle}$ and model B. In the resulting model $A_{\langle C \rangle}$ the concern C cannot be separated. Concern B is separately modeled in model B.</p>
7.		<p>A is transformed along with concern C to a model $A_{\langle B \rangle}$, and model C. In the resulting model $A_{\langle B \rangle}$ concern B from the transformation cannot be separated. Concern C is separately modeled in model C.</p>
8.		<p>A is transformed along with concern C to a model A, B and C. In the resulting model all the three concerns A, B and C can be separated.</p>
<div style="display: flex; align-items: center;"> <div style="margin-right: 20px;"> <p>LEGEND</p> <div style="display: flex; align-items: center;"> <div style="border: 1px solid black; padding: 2px; margin-right: 5px;">A</div> <div>Model</div> </div> <div style="display: flex; align-items: center;"> <div style="display: flex; align-items: center; margin-right: 5px;"> <div style="border: 1px solid black; padding: 2px; margin-right: 2px;">A</div> <div style="border: 1px solid black; padding: 2px; margin-right: 2px;">C</div> </div> <div>Resulted Composition of Models</div> </div> <div style="display: flex; align-items: center;"> <div style="display: flex; align-items: center; margin-right: 5px;"> <div style="border: 1px solid black; padding: 2px; margin-right: 2px;">A</div> <div style="display: flex; align-items: center; margin-right: 2px;">•</div> <div style="border: 1px solid black; padding: 2px; margin-right: 2px;">C</div> </div> <div>Required Composition of Models</div> </div> </div> <div style="border: 1px solid black; padding: 5px; flex-grow: 1;"> <div style="display: flex; justify-content: space-between; align-items: center;"> <div style="display: flex; align-items: center;"> <div style="margin-right: 10px;"> $T_{\langle \rangle}$  </div> <div>Transformation (no implicit concern)</div> </div> <div style="display: flex; align-items: center;"> <div style="margin-right: 10px;"> $T_{\langle C \rangle}$  </div> <div>Transformation with explicit concern</div> </div> <div style="display: flex; align-items: center; margin-top: 10px;"> <div style="border: 1px solid black; padding: 2px; margin-right: 10px;">A<C></div> <div>Model A with non-localized concern C</div> </div> </div> </div> </div>		

Table 1
Concern Transformation Patterns

[2], which keeps a history of the changes made to a set of files that can be concurrently accessed. In Figure 2, the conceptual architecture of a CVS system is

shown. This architecture consists of four major sub-systems: programmer's environment, administrator's environment, session management system and repository management system.

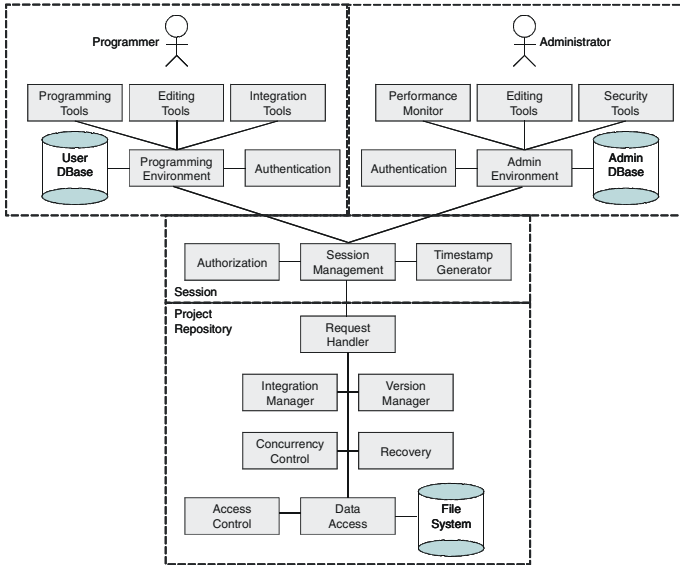


Fig. 2. Conceptual Architecture of the Concurrent Versioning System (CVS)

The programmer's environment provides a set of programming tools, such as compilers, interpreters and debuggers, editors and tools for integrating program modules into a consistent program. Programmers have to be authenticated before they start using the system. When a programmer wants to edit a file which is stored in the project repository, a request is made to the session manager. The session manager authorizes the request, associates a timestamp with it, and initiates an editing session by calling on the request handler of the project repository. The request handler checks out the requested file and passes it to the programmer's environment. When files are checked out they can be edited and compiled and check in the modifications to the file. Checking out a file does not give a programmer exclusive rights to that file. Other programmers can also check it out, make their own modifications, and check it back in. The concurrency control module administers all the simultaneous accesses to the same file, together with the identity of the users, access time and the version numbers. This module is also responsible in identifying the read/write conflicts in accessing the files. If a conflict is detected, the integration manager is called. The integration manager provides a set of functions to resolve the conflicting accesses. For example, the integration manager may notify the programmers, or may ask assistance from the authorized person to resolve the conflict. The recovery manager makes a copy of all the modified files so that the original files can be restored if necessary. The version manager generates version ID's, compares versions of files and notifies if there are inconsistent versions in the same configuration. The access control unit provides low-level access control functions such as protecting read only files from updates, etc.

The administrator's environment provides a set of management tools. For example, the performance monitor is used to generate reports on the average time of accesses, the affect of data size and simultaneous accesses to performance, number of aborts, etc. The administrator is also responsible for installing the session workflow attributes, i.e. priorities, responsibilities in resolving conflicts. Authentication and authorization rights are also determined by the administrator.

4 Evolution Scenarios

Given the models and the transformation of MDA we can now focus on the analysis of crosscutting concerns in MDA. To illustrate the impact of crosscutting on model transformations we will add the concerns security, logging, versioning strategy and persistence. Obviously, these concerns might be introduced at the CIM, PIM, PSM or even the code level. If we also consider the various orderings in which these concerns might be introduced in the model transformation process, it becomes clear that the number of model transformations increase dramatically. An exhaustive analysis of all the possible alternative model transformations with these specific concerns could provide us a complete overview of the problems that we might encounter. However, it is from a practical point of view not possible to analyze all these possibilities. Further, our goal in this paper is not to provide a complete analysis for the specific set of concerns but rather to pinpoint practical recurring problems that might appear in MDA while applying crosscutting concerns. Secondly, in principle it also does not seem to be necessary to provide an exhaustive analysis to identify the key problems. This is because we can easily group the transformations and reason about crosscutting concerns in the model transformations in a more abstract manner.

Transformation	Scenario	Order of scenarios
CIM to CIM	S1. Adding new concern security to use case model	
CIM to PIM	S2. Transforming use case model to PIM	
PIM to PIM	S3. Add Security at the PIM level	
	S4. Add/Update Logging at the PIM level	
PIM to PSM	S5. Transform to Relational DB Platform model	
	S6. Transform PIM to Java platform model	
PSM to PSM	S7. Adding Versioning Strategy	
	S8. Adding Persistency	
	S9. Upgrade to Remote Security invocation	
PSM to Code	S10. Transformation to Java Code	
	S11. Transformation to Relational Tables	

Table 2
List of Scenarios including Crosscutting Concerns applied in the MDA Transformations

For our analysis we will define a set of evolution scenarios to the given example,

in which crosscutting concerns will be added. The set of scenarios that we apply is depicted in Table 2. Note that we have included each model abstraction level (CIM, PIM, PSM and code) as well as the transformations among these. In addition, for the horizontal transformations we have included one or two concerns. Later in the discussions we will also analyze the various possible orderings of these scenarios. The ordering of the scenarios that we will discuss in this paper is presented in the scenario transition diagram as depicted in the right column of Table 2. Hereby, the labeled circles present the scenarios on the left, whereas the arrows represent the transition between scenarios.

4.1 CIM to CIM Transformations

The first transformation (scenario S1) that we consider is the CIM to CIM transformation (Table 2). The Computation Independent Model (CIM) focuses on the environment and the requirements of the system; it does not concern with any structural or processing details of the system. We assume that the CIM is expressed as a set of use case models. We consider in this example the addition of the concern security to the CIM model. For the given scenario S1 we can apply concern transformation pattern 1 and 2. It appears that we can model and compose the use case security and as such transformation pattern 2 is applied, as depicted in Figure 3. The left part of the figure shows the null-composition of the source model with the security use-case. The right part shows the result after the transformation, in which the use cases have been composed using the "uses" relation to the new concern security. Although the concern security could be separated at this level, the number of relationships already denotes that there is possibly crosscutting in the later models that will be derived from this model. We could also imagine a case in which a concern can not be easily localized in the use case model. This could be for example a concern like, optimize time performance in accessing CVS. Obviously it is very hard to specify this in a separate use case, and typically all the time performance optimization must be performed within each use case itself. As such we could characterize this transformation as an application of pattern 1 specified as $CVS_{<timeperformance>}$.

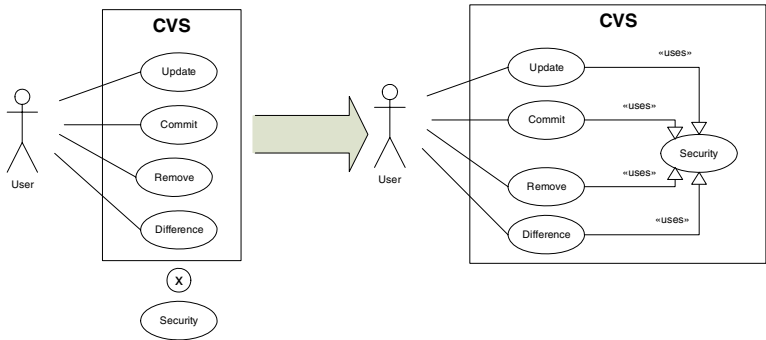


Fig. 3. Adding Security Concern in a CIM to CIM Transformation

4.2 CIM to PIM Transformations

Figure 4 presents the result of the transformation to a Platform Independent Model (PIM), as defined by scenario S2 (transformation to PIM) in Table 2. In Figure 5, Repository is used to group the different versions of the system and can be used for tracking changes in a configuration management system. The class Branch includes class Directory that can store File. Class Version defines a particular version and multiple versions of a file may exist. The attributes deleted in the classes File and Directory denote whether the file or directory has been deleted or is still used. Files might be labeled using class Tag. The transformation from a computation independent model to a platform independent model can be considered as a transformation in which the concern is computation concern itself. Naturally, it is very hard to separate the computation concerns in the PIM and as such we can state that pattern 3 is applied.

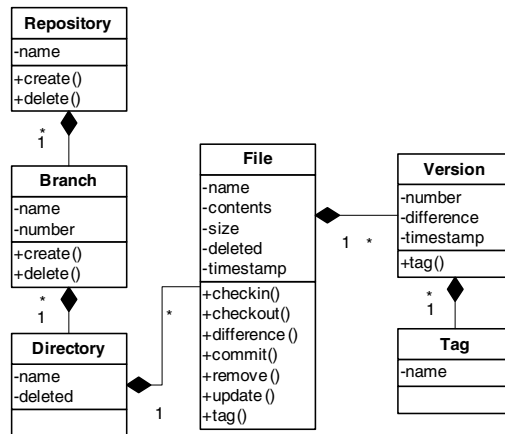


Fig. 4. CIM to PIM Transformation for the CVS

4.3 PIM to PIM Transformations

For the PIM to PIM transformation we apply two scenarios S3 (security) and S4 (logging) of Table 2. Figure 5 shows this transformation of the initial PIM in Figure 4 to a target PIM in which the security and logging concerns are added. In principle both concerns could be applied together, or sequentially. In case, the concerns are applied sequentially then the ordering of the scenarios will need to be considered explicitly. We will first discuss these separately and then analyze the ordering of these scenarios.

Adding Security

To apply the security concern to the CVS, the system should store the different users and their data access permissions. For this, in Figure 5 the classes User and Permission represent the users and their permissions respectively. Class User

can be either a normal user (programmer) or an administrator depending on the value of the type attribute. Class Permission includes the attributes read and write indicating whether a user can read or write from or to a specific object in the system. The attributes InstanceType and InstanceName identify the object on which permission for a user has been placed. To uniquely identify a specific object, a unique identifier has to be chosen for the InstanceName attribute. This can be for example an object reference or the complete name inside the structure, such as `'/branch/directory/filename'`. Class SecurityManager checks whether a particular user with the related permissions can execute the requested operation. Ideally we would like to separate the security concern in the target model to be able to enhance it if necessary. In fact, at a first glance, the security concern also seems to be nicely separated. There are three separate classes and one interface that implement the security protocol. Unfortunately, the classes Repository, Branch, Directory, and File that implement the interface also require implementing the permission lookup. The arrows marked with s3 indicate which methods need to check the requested permission with the actual permissions stored by the security manager. This means that the security concern is spread over the different classes and is tangled with the concerns that are implemented in the corresponding classes. As such, the transformation of the initial PIM to a PIM with security concerns is realized by applying pattern 1 again. In order to apply pattern 2 so that security concern is separated in the end-result, we might need explicit abstractions to modularize this crosscutting behavior. Typically, aspect-oriented implementation techniques could be useful for this purpose.

Adding Logging

Figure 5 also includes the application of the logging concern as defined by scenario S4. Logging of the system is required to identify bottlenecks within the system and to get statistical information. The classes Log and LogManager implement the logging concern. Class LogManager includes the log-operation for updating the log. Class Log implements the log containing the entries timestamp of the operation, the host and the user, the requested operation, the branch, the status, and the result. Although the logging concern seems to be localized in the two classes Log and LogManager, a close analysis shows that the concern crosscuts over the methods of various classes. In Figure 5 the methods that are affected by adding the logging concern are denoted by S4. As such, similar to adding the security concern we can state that we have applied pattern 1 for this case.

Adapting Logging

Assume now that the logging functionality needs to be enhanced with verbosity levels to the log messages. For this the following levels need to be added: severe, warning, config, and debug. During the transformation this requirement can be accomplished by adding an attribute level to class Log. Further attributes severityLevel, warningLevel, configLevel and debugLevel must be added to class LogManager (these changes are not shown in Figure 5). The changes to the classes

Log and LogManager can be automatically applied by a transformation. However, a change of logging concern also requires changing the methods that are logged. Again, all the methods indicated by the arrows labeled with S3 in Figure 5 need to be adapted to meet this concern. To resolve this issue typically the transformation itself must be aware of the crosscutting nature of the concern.

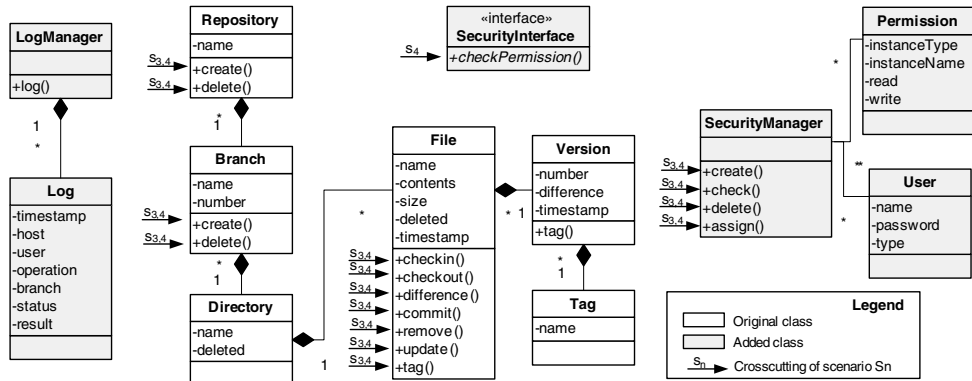


Fig. 5. PIM with Security and Logging

Impact of Sequential Transformation and Ordering

Interestingly, the logging concern seems also to crosscut the security concern. This has an impact on the ordering of the transformation. In case the logging concern transformation would take place before the security concern transformation, then this would imply that the methods of the security concern classes can not be logged. For consistency, a retransformation would then be required. Equally, security concern might crosscut the logging concern, in case logging is not allowed freely. This shows that the transformation must also be aware of the ordering of the concerns.

In the above cases we have assumed that the concerns were transformed one by one. We could imagine a case in which both concerns are transformed together in the model. If we assume that security and logging concerns would be again crosscutting in the end-result then we could specify this as follows:

$$CVS \bullet Security \bullet Logging \rightarrow CVS_{\langle Security, Logging \rangle}$$

To provide modularity of both concerns typically the transformation must be aware of the crosscutting nature, and the semantic conflicts (ordering) of the concerns.

4.4 PIM to PSM Transformations

For the PIM to PSM transformation we have defined scenarios S5 and S6, which transform the PIM to a relational DB Platform model, and a Java Platform model, respectively. Both scenarios can be applied after scenario S4. Below we discuss both alternatives.

PIM to Relational PSM

Figure 6 shows the result of the transformation from PIM to Relational PSM. Here the classes have been mapped to tables and operations of classes are implicitly mapped to database operations. It is hard to spot any crosscutting in Figure 6, so one might assume that this model is crosscutting free. However, the PSM shown in the figure is only the static structure of the system. If we consider the transformation of the behavior to SQL statements then we might observe several problems. One major restriction is that queries can only do relatively simple operations on the data within the database, as SQL is never meant to be a generic programming language. Although this is not directly visible in Figure 6 the concerns security and logging seem both to be crosscutting. For example, concern security needs to check for each data access whether the permission constraints are met. This could be implemented by some value checking, before data is inserted or updated in the database via trigger or stored procedures. In this case, some of the checking behavior of the system could be moved to the database. In any case the transformation itself does not highlight the crosscutting. We could state that the crosscutting nature has changed. Compared to a Java implementation (next), for example, some crosscutting concerns are more localized in database operations. In any case we can state that the transformation pattern 1 has been applied to realize scenario S5 (transform to relational PSM).

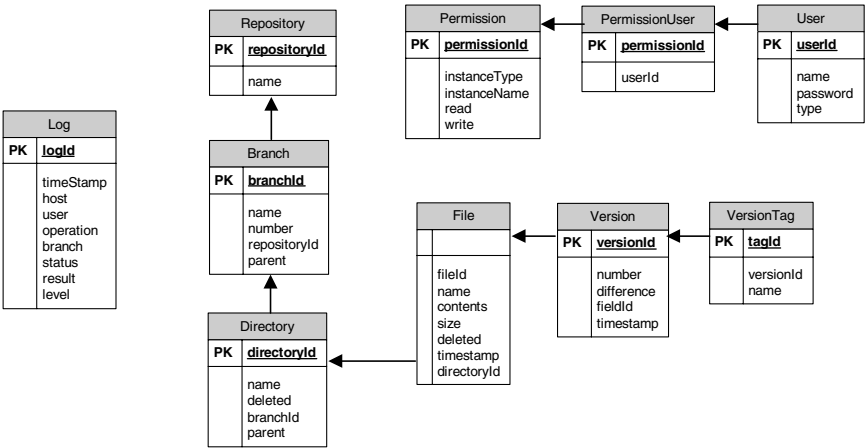


Fig. 6. Relational PSM

PIM to Java PSM

Figure 8 defines the Platform Specific Model of the CVS for a Java platform. Contrary to the relational PSM the operations of the classes are explicit and crosscutting becomes more transparent. Like in the previous case, it is clear that the selection of the platform has a direct impact on the whole model and as such can be considered as a crosscutting concern. Similarly we can state that also for scenario S5 pattern 1 is applied. We also observe that the ordering of the transformations

is important. The scenario S4 (update logging with verbosity levels), for example was applied at the PIM level as it has been explained in the previous section. This scenario could equally be applied during the PIM to PSM transformation. In that case the crosscutting occurs at the PSM level and as such becomes specific to the platform. For the relational PSM this would mean that the logging is mainly adapted in the database operations, whereas in the Java PSM the methods of the classes that call the `log()` operation need to be adapted. The bottom line is that the time at which a concern is introduced has a clear impact on the crosscutting nature.

4.5 PSM to PSM Transformations

For the PSM to PSM transformations the scenarios S7 (versioning strategy), S8 (adding persistency) and s9 (upgrading to remote security checking) are applied. These scenarios are applied either after scenario S5 (relational PSM) or scenario S6 (Java PSM). It appears that both transitions have different implications.

Adding Versioning Strategy

The result of scenario S7 (add versioning) of Table 2 is shown in Figure 7. Here we assume that we have chosen for the Java PSM. Note that only a partial view of the complete class model is given, because the other classes remain practically unmodified. There appears to be no (crosscutting) problems, because the effect of scenario S7 is localized to only the class `Version`. It should be noted that the interface of the class `Version` has changed and that these changes may need to be processed in some other classes, such as the user interface, so from that perspective, the effect is not completely localized. In that case pattern 1 is applied.

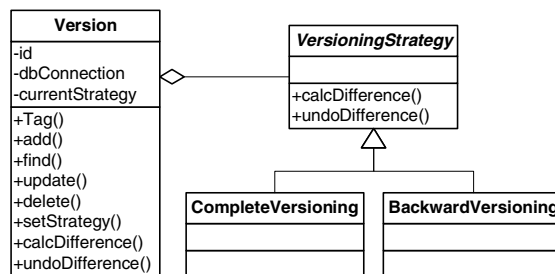


Fig. 7. Addition of Versioning Strategy to PSM

This case also shows that adding a new concern can have an impact on previous transformations. In this scenario we should be aware that the new methods introduced by the versioning concern may need to make calls to the `SecurityManager` and `LogManager` for checking privileges and logging messages. That is to say that adding the new concern might easily require rechecking of conflicts and retransformations. If we would have chosen the relational PSM, that is, from S5 to S7, then we had to make changes there too, because the relational PSM has now to reflect

the used versioning strategy per row. This is required to allow correct retrievals of the previous versions of a file. Further, also in this case a total recheck of the applied concerns and the retranformations become necessary for consistency.

Adding Persistency

Figure 8 defines the Platform Specific Model of the CVS for a Java platform including the persistency concern. To add persistency to the Java PSM (scenario S8 of Table 2) every class has private attribute called dbConnection. This shows that the persistency concern also leads to crosscutting in the PSM transformation. As such we are dealing again with pattern 1. In case we would have adopted aspect-oriented techniques then we would have been able to separate these concerns better and as such could characterize the transformation as an instance of pattern 2. In case of the relational PSM the persistency will be put in the database operations which are not directly visible.

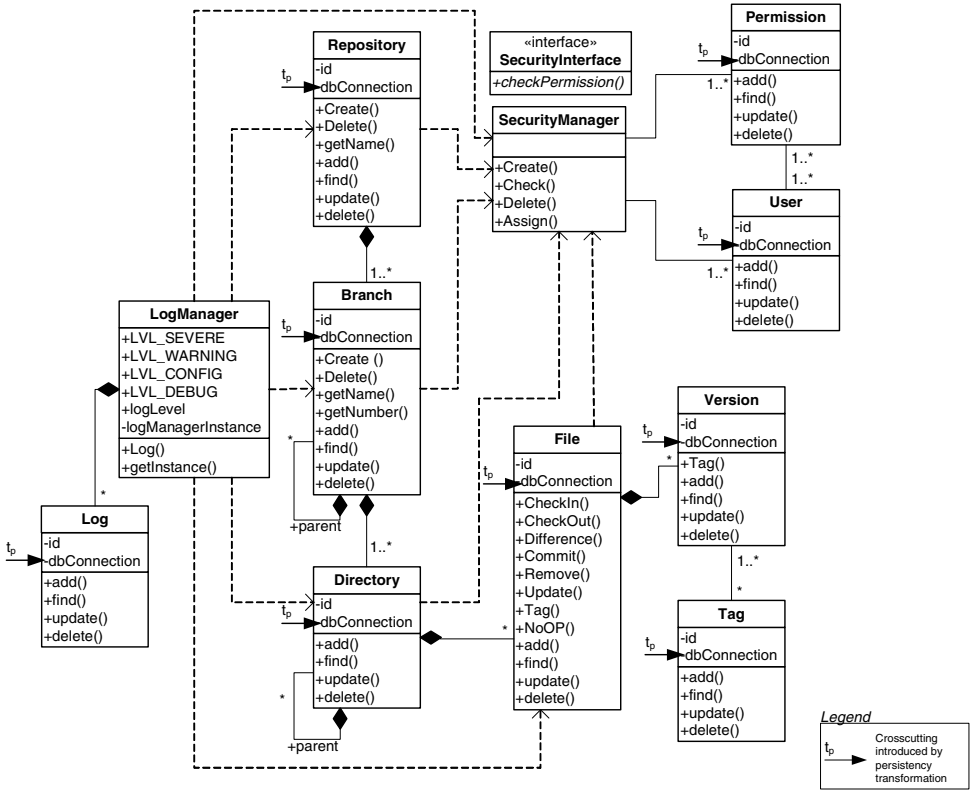


Fig. 8. Java PSM with Persistency Concern

Updating Security

Scenario S9 allows the moving of the security concern to another, perhaps a remote centralized, com-puter. The impact of crosscutting for this scenario is two-

fold. First of all, the classes that make use of the `SecurityManager` require special initialization code to obtain a reference to the `SecurityManager`. That code will be present in every class after the transformation. Although the code could be generated, it is scattered code and a replacement of the already existing crosscutting code. Furthermore, remote method invocation might need more work such as in the case of exception handling to handle for example failures of the network connections. The logging concern has also to be adapted to allow remote operation invocation, because both concerns, logging and security, can be on separate computers and the security concern still needs to perform logging. This is again an example in which a newly added concern has a direct impact on the previous transformations.

4.6 PSM to Code Transformations

Scenarios S10 and S11 define the transformation to Java Code, and to relational database, respectively.

Transformation to Java Code

For the transformation to Java Code model sufficient information is required to provide a complete and executable code. Unfortunately this is not always the case. For example for scenario S9 (update security) for the transformation of the Java PSM to the Java code model, the PIM and PSM do not have enough information to generate the code automatically. Normally the following example code would suffice to implement security for scenario 1 (adding security):

```
SecurityManager secMan = SecurityManager.getInstance();
```

However in case of scenario S9 in which security is handled remotely then additional information is required in the code. For example, in case we assume that the reference for security access is obtained via a file and the remote invocation is done via a CORBA ORB, then we would have to insert the corresponding Java code. Obviously this complicates the automatic generation and a large part of the code must still be written by hand. In fact this is also a specific kind of crosscutting, which is actually introduced due to lack of information in the transformation. In this case pattern 3 is applied.

Transformation to Relational Database

The transformation of the relational PSM to 'code' (scripts) consists of generating 'create table'-scripts and possibly the generation of update-queries. We can consider the transformation as an instantiation of pattern 1. Here the crosscutting remains hidden, as well as the part that still needs to be adapted by hand.

5 Discussions and Recommendations

So far we have defined transformation patterns and illustrated their application to the CVS case using a predefined set of selected scenarios. During the discussion of the case we have already seen several interesting issues related to concerns, concern

evolutions and model transformations in the MDA process. In the following we discuss the main issues that were identified in the analysis and based on this we derive some recommendations for coping with concern evolution in the MDA process.

- *Crosscutting is introduced in the target model because of lack of expression of the language*

If the target model is not expressive enough this might lead to scattered concerns. This could occur even in case the concern was separated at the source model. In the given scenarios the concerns security, logging, versioning and persistency were crosscutting due to lack of explicit abstraction mechanisms in Java and the relational database platform. The lack of a modeling language for aspects, leads to the problem that aspects cannot be neatly separated from the other concerns. This means that a solution has to include a modeling language for aspects for the different modeling stages of the MDA process.

- *Crosscutting in source model is inherited in target model due to lack of expressiveness*

In the scenarios that we have applied we can derive that crosscutting can be inherited from higher abstraction models. The problem starts at the CIM where some possible crosscutting is introduced and with the transformation to the PIM, to the PSM and eventually to the code model, it became clear that the crosscutting problems propagated, thus inherited, throughout the entire process and different models. Furthermore, the inheritance of crosscutting also applies to crosscutting that was later introduced than at the CIM, that is, at any phase in the model-driven engineering process crosscutting might be introduced. An example is the persistency concern that was introduced by scenario S8. The crosscutting problems of this concern were also inherited into the code model. The main reason for the inheritance of crosscutting is obviously the lack of expression power of the target language. To avoid crosscutting in each phase from CIM to code, we could delay the introduction of the crosscutting concern until the code. However, to provide a complete solution naturally it is required to apply aspect-oriented modeling techniques in which crosscutting concerns are represented using explicit language abstractions.

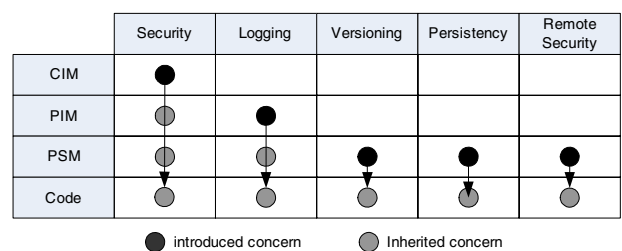


Fig. 9. Introducing concerns early: Inherited Crosscutting

- *Introduced crosscutting in later phases might conflict crosscutting concerns introduced in earlier models*

Earlier introduced concerns might propagate through the lower (concrete) model abstractions, and therefore it is worthwhile to consider introducing concerns later in the process. However, later introduced concerns on the other hand could have the problem that existing transformations need to be redone. We have seen this in introducing versioning (scenario S7), persistency (scenario S8) and remote security (scenario S9). All of these scenarios required the retransformations of the previous concerns. An example of a concern that has an impact on previous transformations is shown in Figure 10. Here we see that introducing versioning at the PSM level has an impact on the transformations of the concerns security and logging in earlier phases.

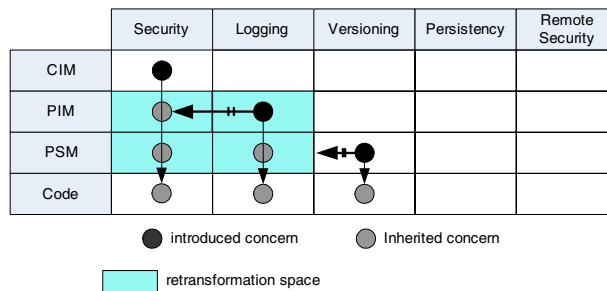


Fig. 10. Introducing concerns late: Inconsistencies with earlier transformations

- *Transformation order can result in different crosscutting*

The ordering of the transformation plays an important role in the quantity of crosscutting that can or will occur. The previous two points have already showed the impact of introducing concerns earlier or later in the MDA process. Moreover, the ordering is also important within the same abstraction level. For example the security and logging concerns both impact each other as it has been discussed in section 4.3. In that case we have to explicitly reason about the semantic conflicts and the impact of the ordering on the model transformations. It would be worthwhile if the transformation would be aware of the ordering semantics [11]. For this the domain of the concerns should be well-understood [4].

- *Crosscutting in the target model introduced by the transformation*

Obviously all the vertical model transformations including CIM to PIM, PIM to PSM and PSM to code introduce some crosscutting behavior. This crosscutting behavior is the platform concern itself. Unfortunately, it is very hard to separate the platform concern. At least we have not been able to show this in our analysis. The transformations are easier in case the models are closer to each other. For example, in the given analysis we have chosen for transforming a UML-based [1] PIM to a Java based PSM. This is naturally different than transforming a UML-based PIM to a relational PSM, which results in different kind of crosscutting (next).

- *Transformation might lead to non-transparent crosscutting*

In section 4.4.1 in which we have mapped a PIM to a relational database PSM we

have seen that crosscutting is not visible in the model. Compared to a Java implementation some crosscutting concerns are more localized in database operations. This shows that the selection of the platform does not only impacts how much the concern is crosscutting, but also how much this crosscutting is visible. In our relational PSM the crosscutting is not directly visible and as such, was compared to the Java PSM more difficult to address. The visibility of the crosscutting is of course also important for the next transformation (code).

- *Crosscutting might disappear after transformation*

Crosscutting in the source model might disappear in the target model if the transformation can map the crosscutting to a more modular concern in the lower model. As discussed before this requires that the target model includes notations to express aspects, and the transformation must be aware of the crosscutting nature of the concerns and map this to aspects in the target model. We have not shown an aspect-oriented model for this but refer to existing aspect-oriented modeling techniques.

6 Conclusions

We have provided a systematic analysis of crosscutting concerns in the Model-Driven Architecture (MDA) approach. We have expressed a case example, concurrent versioning system (CVS), in the MDA approach and defined the transformations to the Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) and code. By applying selected set of scenarios that represent crosscutting concerns in the case, we have analyzed the various problems related to crosscutting concerns in the MDA process. Hereby we have abstracted from the various names of the models and basically focused on concerns that are realized. For this we have assumed that each model represents particular concerns and this has led to the assumption that model transformation are in essence concern transformations. We have identified four primary CTPs and four composite CTPs. During our analysis we have indicated the type of CTP that could be applied to express the required concern, and this has led to several important observations. First of all, whether a concern is crosscutting is largely dependent on whether the source model, the target model, and the transformation can express the concern in modular units. If the source language, the transformation language or the target language does not support the modularization of crosscutting concerns then this could lead to various problems as we have shown in our analysis. If the source language does not support crosscutting then the crosscutting is easily inherited by the subsequent model transformations. If the target language does not support the modularization of crosscutting concerns then the model transformation at any level in the MDA process will in general result in crosscutting. To cope with crosscutting concerns it is thus required to have explicit language abstraction mechanisms. This could be realized by adopting an existing AOP language [8] or enhancing current transformation language for crosscutting concerns [11].

Secondly we have seen that the nature of crosscutting can also be different with

respect to the language of the target. For example, in case of mapping the PIM to the PSM with the concerns security and logging to a relational PSM, results in a different type of crosscutting than in case the target language is UML [1]. In the first case the crosscutting is not directly visible and hidden over many database operations, while in the latter case the crosscutting concerns is scattered over different UML classes. A crosscutting concerns is usually defined as the scattering of the concern over multiple units of implementation and tangling of multiple concerns in local implementation units. Obviously from our study it seems that the notion of crosscutting should be further specialized to distinguish between various types of crosscutting and as such provide a deeper insight in the problems of crosscutting concerns.

Another important conclusion from our study is the direct impact of the ordering of the introduction of concerns throughout the MDA process lifecycle. The moment of introduction of crosscutting concerns in the MDA process appears a key decision. Regarding the MDA process we can in essence identify two different moments in which the crosscutting concerns can be introduced. On the one hand crosscutting concerns might be introduced very early at the CIM or PIM level. If the source, target or transformation language does not support modularization of crosscutting concerns then this might lead to crosscutting. This crosscutting is then generally inherited throughout the MDA life cycle. To avoid the inheritance of crosscutting one might think to introduce it as late as possible. However, as we have shown in the previous sections this is also problematic because the introduction of crosscutting concerns might be conflicting with earlier introduced concerns (e.g. security and logging). In that case all the transformations must be redefined. In fact this problem can be categorized as an instance of inheritance anomaly problems as introduced by Matsuoka and Yonezawa [9]. In the current MDA approach as proposed by OMG very little attention has been paid to the process related issues. However, as our analysis has shown, the process related issues directly impact model transformations and as such need to be explicitly managed.

By abstracting from the various models and model transformations in MDA our analysis could also be considered from the perspective of Model-Driven Engineering (MDE) [12]. In our future work we will broaden our analysis for other MDE approaches than MDA and consider also approaches such as Software Factories [6], which adopts a different process for model transformations.

Acknowledgements

This research has been carried out in the Aspect-Oriented Software Architecture Design project which is funded by the Dutch Scientific Organisation in the Jacquard Software Engineering Program.

References

- [1] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*, Addison-Wesley, 2002.

- [2] "Concurrent Versions System - The Open Standard for Version Control", online resource <http://www.cvshome.org>, February 2004
- [3] T. Elrad, R. Fillman, A. Bader. Aspect-Oriented Programming. Communication of the ACM, Vol. 44, No. 10, October 2001.
- [4] E. Evans. Domain-Driven Design-Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2003.
- [5] D.S. Frankel. Model-Driven Architecture, Wiley Publishing Inc., 2003.
- [6] J. Greenfield et al. Software Factories - Assembling Applications with Patterns, Models, Frameworks and Tools, Wiley Publishing Inc., 2004.
- [7] A. Kleppe, J. Warmer, W. Bast. MDA Explained, The Model-Driven Architecture: Practice and Promise, Addison-Wesley, 2003.
- [8] I. Krechetov, B. Tekinerdogan. Integrated Aspect-Oriented Software Architecture Specification Approach, University of Twente, European Network of Excellence AOSD project deliverable, 2005
- [9] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming language. In Research Directions in Concurrent Object-Oriented Programming, pages 107150, 1993.
- [10] OMG, MDA Guide Version 1.0, Eds. J. Miller and J. Mukerji. May, 2003.
- [11] OMG, MOF 2.0 Query/View/Transformation Final Adopted Specification, OMG Document number ptc/05-11-01, November 2005, <http://www.omg.org/docs/ptc/05-11-01.pdf>
- [12] T. Stahl, M. Vlter. Model-Driven Software Development, Wiley, 2006.