

Generic Reusable Concern Compositions (GReCCo): Description and Case Study

Aram Housepyan *Stefan Van Baelen*
Yolande Berbers *Wouter Joosen*

Report CW 508, January 2008, revised April 2008



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Generic Reusable Concern Compositions (GReCCo): Description and Case Study

Aram Housepyan Stefan Van Baelen
Yolande Berbers Wouter Joosen

Report CW 508, January 2008, revised April 2008

Department of Computer Science, K.U.Leuven

Abstract

This report presents the GReCCo approach to Aspect Oriented Modeling (AOM) using Generic Reusable Concern Compositions. GReCCo offers an AOM-based framework to promote and enhance the reuse of concern models. We focus on software design patterns, which represent complete solutions to recurring concern-specific problems. We have developed a prototype generic transformation engine written in ATL that can be used to compose two concern models specified in UML.

We first describe the GReCCo approach and the offered composition types. In the second part, we illustrate the GReCCo approach on a case study in the domain of Electronic Health Information and Privacy (EHIP). We start from a description of the base part of the application. On top of this application, we apply several reusable concerns using the GReCCo methodology.

Keywords : AOM, obliviousness, model compositions, reusable concerns.
CR Subject Classification : I.2.11, I.2.8, I.2.1.

Generic Reusable Concern Compositions (GReCCo): Description and Case Study

Aram Hovsepian, Stefan Van Baelen, Yolande Berbers, Wouter Joosen

Katholieke Universiteit Leuven, Departement Computerwetenschappen, Celestijnenlaan 200A,
B-3001 Leuven, Belgium

{Aram.Hovsepian, Stefan.VanBaelen, Yolande.Berbers,
Wouter.Joosen}@cs.kuleuven.be

Abstract. This report presents the GReCCo approach to Aspect Oriented Modeling (AOM) using Generic Reusable Concern Compositions. GReCCo offers an AOM-based framework to promote and enhance the reuse of oblivious concern models. We focus on software design patterns, which represent complete solutions to recurring concern-specific problems. We have developed a prototype generic transformation engine written in ATL that can be used to compose two concern models specified in UML. We first describe the GReCCo approach and the offered composition types. In the second part, we illustrate the GReCCo approach on a case study in the domain of Electronic Health Information and Privacy (EHIP). We start from a description of the base part of the application. On top of this application, we apply several reusable concerns using the GReCCo methodology.

1 Introduction

Aspect-Oriented Modeling (AOM) is a recent development paradigm that aims at providing support for separation of concerns at higher levels of abstractions [1]. Following an AOM approach, one can model parts of a complete solution separately, and compose them afterwards using model composition techniques.

In order for AOM to bring an increase of efficiency in software development, it must be possible to model concerns once and reuse them in different contexts. If one has to create a concern every time from scratch, the gains of AOM will diminish substantially. Also, there is a need to define criteria that evaluate whether a given concern is reusable.

In this paper we define and characterize reusable concern models. We provide several key criteria that are crucial in improving concern model reusability. We have developed a graphical framework that can be used to model the structure and the behavior of certain types of concerns, and we specify their composition in a composition model. We have implemented a generic composition engine in ATL [2], which can compose UML concern models specified in our framework .

In section 2, we define the key requirements that are necessary to improve concern reusability. We also comment why current AOM approaches come short in achieving some of these requirements. In section 3.2, we present our approach in details. We then describe the different composition mechanisms of the Generic Reusable Concern Compositions (GReCCo) approach. In section 4, we present a case study and apply our

approach to it. In section 6, we evaluate how GReCCo improves reusability by tackling each requirement presented earlier. In section 7, we present some related work. In the last section, we conclude and outline future work.

2 Concern Reuse

Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts [3]. We use the term *concern* to uniformly refer to what AOSD practitioners often call an *aspect concern* and a *base* or *component concern* [4]. The *component concern* usually represents the functional core of a given application, whereas different *aspect concerns* represent functional and non-functional parts that extend the core.

To our knowledge, there is no clear notion of reuse for concern models in AOM. We therefore define and characterize what makes a given concern model more or less reusable.

2.1 Requirements for reuse

We define a reusable concern model as a known solution for a certain problem, which can be used in several contexts to produce required assets. Different contexts mean for instance different applications, projects, companies, application domains, etc. However, using this definition we cannot measure how reusable a concern model is. That is why we introduce several more concrete qualities of a concern model and an AOM approach as a whole that are important in increasing the reusability of concerns.

Composition obliviousness: Concerns, represented by their models, should ideally be modeled independently from a concrete context in which they are going to be applied. Reusable concerns usually represent generic solutions to known software problems, and therefore such concerns cannot be completely oblivious [5] of the context in which they will be applied. However, a given concern is more oblivious of the composition if it allows more variations in the composition. Consider the following simplified security concern in Figure 1 that represents a solution to realize secure logging [6]. The concern designer provides a complete generic solution, declaring that the *Client* class is a template entity that should be instantiated by a concrete element whenever this concern is used. However, if the concern needs to be composed with an application that already implements a *LogFactory*, it would be impossible to reuse this concern as it is. Figure 2 shows a different version of the same concern, which is relatively more oblivious [5] as it allows any element to be instantiated by a concrete element. Of course, a template entity may have a full implementation, which will be used in case it is not instantiated during a composition.

Composition symmetry: All concerns should be treated uniformly, i.e., we should be able to compose any two given concerns. A non-symmetric (asymmetric) composition approach allows only *base-aspect* compositions. However, this will constrain the concern reuse as we will not be able to use concerns within other concerns.

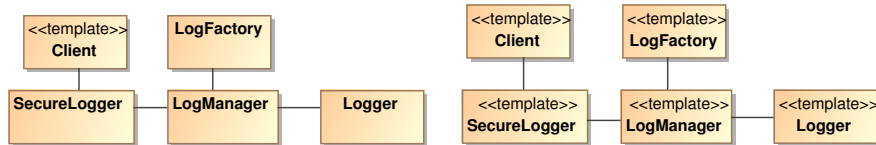


Fig. 1. Secure Logger Concern

Fig. 2. Secure Logger Concern with Increased Obliviousness

Interdependency management: There are many (often hidden) interactions between the different concerns, since they are not always completely orthogonal to each other. Such concern interactions can be classified to one of the following five categories: dependency, mutual exclusiveness, alternatives, conflict and mutual influence [7]. Hence, it is essential to be able to declare such a potential interdependency explicitly, and take it into account when composing different concerns. Otherwise a reused model may create an invalid composition by, e.g., introducing a dependency that is never resolved or adding a conflicting set of concerns.

In this paper we introduce a conceptual framework for representing concerns and specifying compositions with other concerns that improves support for reuse by tackling each of these requirements.

3 Generic Reusable Concern Compositions (GReCCo)

In this section, we first describe the general principle behind the GReCCo approach. Then we present the specifics of the composition of concern models. In addition, we discuss the problem of concern interactions and show how an existing solution can be plugged into our approach.

3.1 General Description

The GReCCo approach is used to compose concern models. As presented on Figure 3, we represent each composition step as the Greek letter Υ . The left and the right branches of the Υ contain two concern models. Our approach is symmetric (**composition symmetry**) in the sense that we treat *component* and *aspect* concerns uniformly. In order to combine the concern models, we provide a composition model that instructs the model transformation engine how the two models should be composed.

Concern Models 1 and *2* (fig. 3) describe the structure and the behavior of the concerns using respectively UML class and sequence diagrams. The *Composition Model*, which also consists of UML class and sequence diagrams using the GReCCo profile, specifies how the concern models are composed by defining all composition-specific parameters and their bindings. This assures a higher degree of reusability of the two concerns as they can be used in different contexts (**composition obliviousness**). The *Composition Model* describes how the source concern models should be composed,

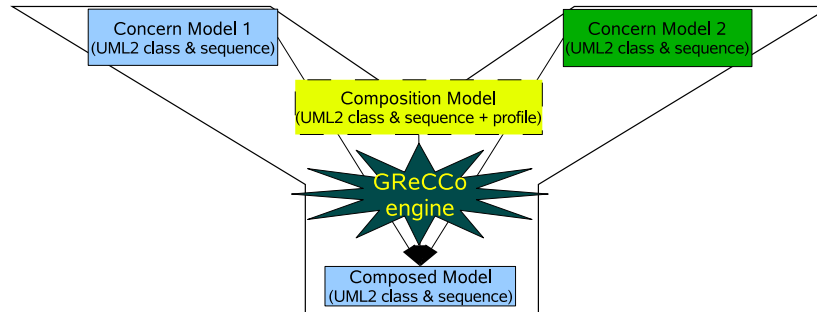


Fig. 3. General Approach

and is therefore by definition depending on these models. Using a generic composition engine, we generate the output *Composed Model* from the input composition and concern models.

As mentioned previously, our approach is **symmetric** as it does not require one of the input concerns to be a primary model. It can be used in two scenarios, even though it does not make any difference for usage of GReCCo in practice. Figures 4 and 5 represent two composition scenarios that can be realized using GReCCo. The rectangle boxes represent the UML models of five concerns (M1 through M5). The ovals represent the composition models.

The first scenario realizes a symmetric composition chain where any two concern models can be composed (fig. 4). M1 is composed with M2, while M3 is composed with M4. The combined models M12 and M34 are composed in the next step. Finally, M5 is added to the result of the previous compositions to obtain M12345.

The second case is used to realize an asymmetric composition chain where an *aspect* model, represented by *Model 2*, is composed with a given *base* model, represented by *Model 1* (fig. 5). In this scenario we consider M1 as a *base* model and M2 through M5 as concerns, which are added to the *base* model to obtain the augmented *base* model. In the end, we obtain the same M12345 as in the symmetric case.

3.2 Composition Specification

In order to compose two concern models, we need to specify the composition by defining the *Composition Model*. We consider the structural and behavioral concerns of the composition separately. Elements that are not directly referenced in the *Composition Model* are copied to the *Composed Model*. Other elements are modified by the composition engine according to the composition specification. The next subsections describe the composition specifics in detail and illustrate them with examples.

Structure We distinguish five structural composition directives in total. From the point of view of a single concern model, we distinguish the following directives that involve one element: (1) we can *add* a new element, (2) we can *modify* the properties of an

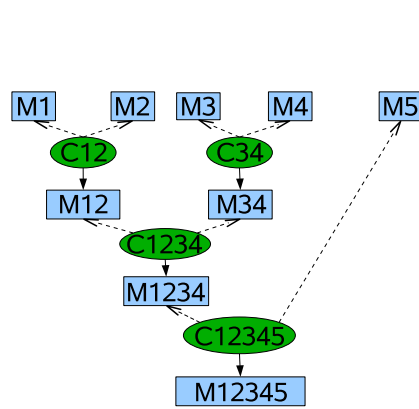


Fig. 4. Symmetric Composition

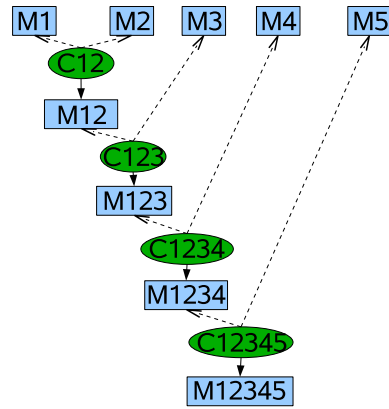


Fig. 5. Asymmetric Composition

existing element, and (3) we can *remove* an existing element. When two concern models are composed, there are some additional composition directives that we can specify for the input elements. We can (4) *merge* two elements to obtain a single entity with the combined properties. Some concerns introduce roles and/or template parameters as semantic variation points, which we can (5) *instantiate* by using concrete UML elements. As we are dealing with structure, we distinguish between four main types of UML elements: class, property, operation and association.

Add The composition of two concerns can add new elements to the composed model. For instance, we may need to link a class from one of the input concern models with a class from the other one. In order to do so, we have to add the element to the composition model and tag it with the UML stereotype `<<add>>`, which will indicate that this is a new element that must be added to the composed model.

Consider two input concerns containing the classes *A* and *B* respectively (fig. 6 and 7). Note that the class *A* is abstract. Figure 8 illustrates a composition model that shows how we can link *A* and *B* with an association. Figure 9 shows the composed output model.



Fig. 6. Concern Model 1



Fig. 7. Concern Model 2

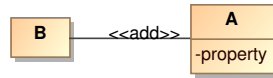


Fig. 8. Composition Model



Fig. 9. Composed Model

Modify During the composition, we may need to modify some properties of the existing elements. We can modify any UML property of a structural element. We specify this by marking the to be modified element with the `<< modify >>` stereotype in the composition model. In addition, we indicate the UML meta-property that needs to be modified and its new value using a stereotype attribute (called a tag in UML 1.x). Each stereotype attribute will have a *name*, indicating the UML property that should be modified, a *type*, which is the same as the UML type of the property to be modified, and a *value*, indicating the new value.

Given the two input concerns from figures 6 and 7, consider the case when we need to modify the name of A as well as make it concrete. In order to do this, we place the `<< modify >>` stereotype on A and specify the new values for the UML properties *name* and *abstract*. The new values of these are *ANew* and *false* and the types are *uml:String* and *uml:Boolean* (fig. 10). Note that *false* is shown with quotes, however, this is tool representation related issue. Figure 11 shows the composed output model, which contains two elements, the modified *ANew* and the old *B* classes.

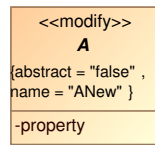


Fig. 10. Composition Model



Fig. 11. Composed Model

Remove Due to the composition of concerns, certain entities may become unnecessary in the composed model. For instance, a concern may introduce an indirect association between two entities that are already connected directly in another concern. We realize the removal of elements by putting the `<< remove >>` stereotype on the element that needs to be removed.

Consider the case when we need to remove the *property* attribute from Concern Model 1 (fig. 6). We need to place the `<< remove >>` stereotype on *property* on the composition model (fig. 12). The output model produces the same element A without the attribute *property* (fig. 13).

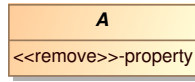


Fig. 12. Composition Model



Fig. 13. Composed Model

Merge When two concern models are involved, we sometimes have to deal with elements that represent a different view on the same entity. We need to merge these elements to obtain a single entity with combined properties. In order to merge two elements, we need to add a UML association between the elements and mark it with the `<<merge>>` stereotype. The merge results in a new element that is composed from the properties of the original two elements. The name of the composed element is set to the concatenation of the names of the original elements. Conflicts such as name clashes, mutually exclusive properties, etc. should be resolved explicitly by using the modify strategy. In the same manner, elements with the same name that are appearing in two models and that should be merged, must be explicitly labeled using the merge strategy.

Consider another set of input concern models (fig. 14 and 15), which we need to combine. Figure 16 shows how this can be specified on the composition model by placing an association between *D* and *E* and marking it with the `<<merge>>` stereotype. The composed output model shows the combined *DE* class (fig. 17).

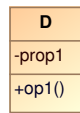


Fig. 14. Concern Model 3

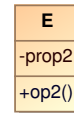


Fig. 15. Concern Model 4

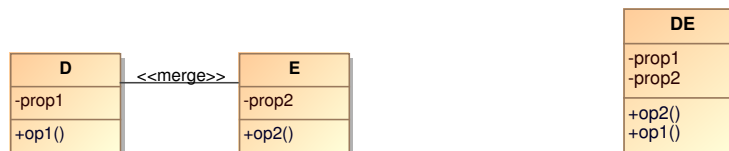


Fig. 16. Composition Model

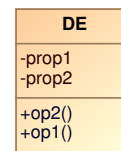


Fig. 17. Composed Model

Instantiate A concern can contain a number of template elements (e.g., roles) that must be bound to concrete elements when the concern is composed. As we aim for oblivious

compositions, we model all concern elements as concrete ones. On the composition model, however, we use the instantiation directive, which tells the composition engine that a given concern element is considered to be a template that must be instantiated by a concrete element from the other model. In practice, instantiation is similar to merge with the only difference that all conflicts are resolved by taking the properties of the concrete element. In addition, the name of the composed element is kept the same as the name of the concrete element. To specify an instantiation, we need to place an `<< instantiate >>` dependency link from the entity that must be considered to be a template to the concrete element.

Consider a third set of input concern models (fig. 18 and 19). We would like to compose the element *G* from Concern Model 6 as a template element with a template attribute (*template*), a template operation (*template_operation*) and a concrete operation (*concrete_operation*). In order to realize this, we place an `<< instantiate >>` dependency link from *G* to *F* in the composition model. In addition, we also specify that *template* should be instantiated by *property1* and *template_operation* by *operation1* (fig. 20). Figure 21 shows the composed output model with the element *F*, which is instantiated by the element *G*.

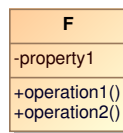


Fig. 18. Concern Model 5

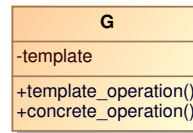


Fig. 19. Concern Model 6

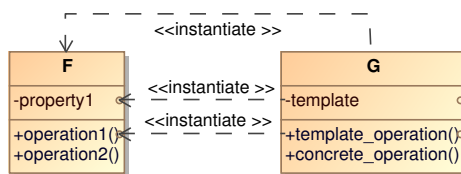


Fig. 20. Composition Model

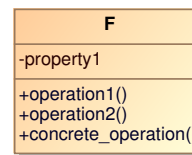


Fig. 21. Composed Model

Behavior We use UML sequence diagrams to describe the behavior of concerns, which must be in correspondence with their structural counterpart specified by UML class diagrams. A concern model may contain several sequence diagrams. Each sequence diagram represents a certain scenario. Scenarios from the input concern models that are completely independent of each other, are simply copied to the composed model. We

define two scenario's as independent of each other if they can be executed in parallel, meaning that the messages can be freely interleaved. For overlapping behavior scenarios we need to be able to address the following use-cases: (1) specify the sequence of messages between the input behavior scenarios, (2) indicate that a call from one input scenario is the same as a call from the other input scenario, and (3) replace a call or a set of calls from one input scenario by a call or a set of calls from the other input scenario.

General Ordering In order to realize the first use-case, we introduce a notion similar to that of general ordering from the UML 2.0 specification [8]. We introduce the general ordering as a directed binary relation between an event or a set of events enclosed in a fragment and another event or set of events. The resulting scenario defines a partial ordering of the input events where the direction of the relation indicates which events should come first. We use a dependency between the event(s) that should precede another event and mark it with the `<< genordering >>` stereotype. The interpretation by the composition engine is that the dependency client should immediately precede the dependency supplier. Events that are not involved in any general ordering relation are put in parallel combined fragments blocks.

Merge In order to specify the second use-case we use a dependency marked with the `<< merge >>` stereotype between the two calls. If we do not use this dependency the calls will appear in duplicate on the composed sequence diagram. Note that event call merging is only possible when the call receiver classes are the same or if they have also been merged on the structural composition diagram.

Replace Finally, to realize the last use-case, we group the event(s) that are to be replaced as well as the replacing events in interaction fragments. We place a `<< replace >>` dependency from the replacing to the to-be-replaced fragments. Grouping in interaction fragments may be omitted in case of a single event or if the set of events is already grouped in an interaction fragment.

We introduce a simple authentication concern, which consists of a *Client* and *Server* entities (fig. 22). The *Client* tries to login on the *Server*. In case of a successful login the *Server* replies with an *OK* message. If the authentication fails, the *Server* logs the failed attempt and replies with a *try again* message (fig. 23).

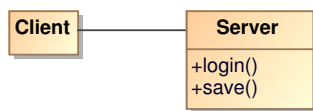


Fig. 22. Authentication Structure

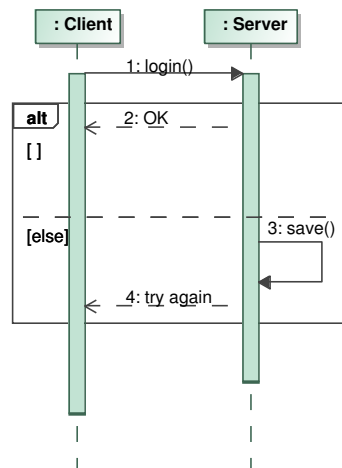


Fig. 23. Authentication Behavior

In addition, we introduce a simple logging concern, which consists of a *Client*, *Server* and *DBExchanger* entities (fig. 24). Every time the *Client* tries to login the *Server* asks the *DBExchanger* to log the event (fig. 25).

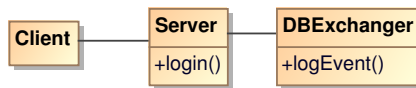


Fig. 24. Logging Structure

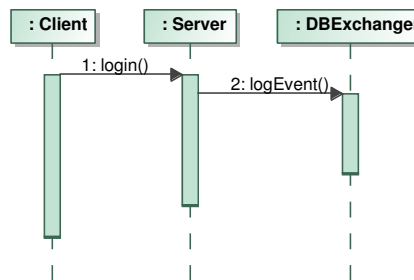


Fig. 25. Logging Behavior

Firstly, we specify the structural composition. We instantiate the *Client* as well as the *Server* entities from the two concerns. In addition we specify that the *login()* event from the authentication should be instantiated as a *login()* even from the logging concern (fig. 26).

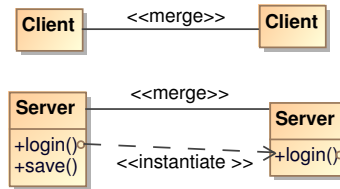


Fig. 26. Structural Composition

Then we specify the behavioral composition. We would like to obtain a composition that performs authorization and in case of a successful authentication writes it to a log. If the authorization fails, the authorization concern itself is in charge of saving the failed attempt. The *login()* events from the two concerns are the same and we place a *<<merge>>* dependency link between them. In addition, we need to put the *logEvent()* event after the *OK* message. We specify this by placing a *<<genordering>>* dependency from *OK* to *logEvent()*. As a result we obtain a combined sequence scenario that performs both authentication and logging in case of a successful authentication. Note that in case we need to use the logging scenario after the authentication independent from the outcome we need to place the *<<genordering>>* dependency from the *alt* fragment to the *logEvent()* (fig. 27).

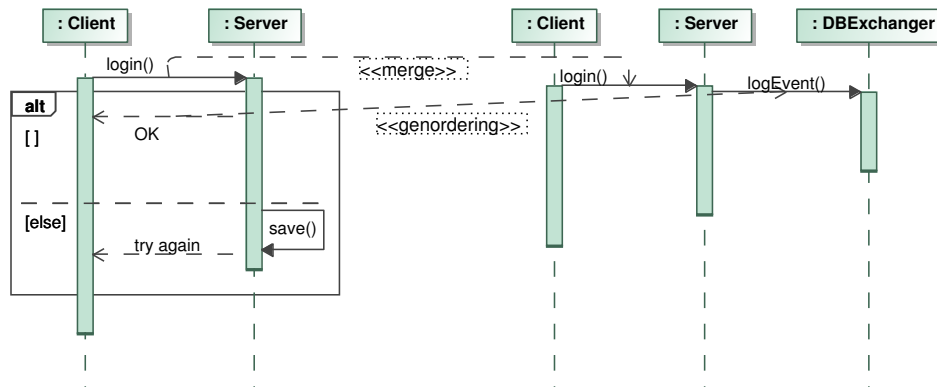


Fig. 27. Behavior Composition

3.3 Concern Interactions

One of the goals of our approach is the ability to build a system by reusing a number of existing concerns. Fig. 4 and 5 are examples of sample trees that can be obtained if we combine more than one concern model.

Concern models are rarely completely orthogonal to each other, but can relate to each other in a variety of different ways. A concern can be involved in an arbitrary number of interactions with one or more other concerns. Sanen et al. [7] distinguish between five different classes of concern interactions: dependency, conflict, choice, mutex and assistance. They also provide a conceptual Concern Interaction Acquisition (CIA) expert system for describing the relevant information about interactions between concerns that need to be captured. Figure 28 shows a simplified overview of the CIA framework. Domain experts add expertise about interactions between concerns into the CIA system. In order to use the CIA system for the investigation of concern interactions in GReCCo, the GReCCo tool has to provide the concern composition specification, which provides the CIA system with the information on concern selection. The list of the selected concerns is analyzed by the CIA system and the list of interactions is presented back to the GReCCo tool.

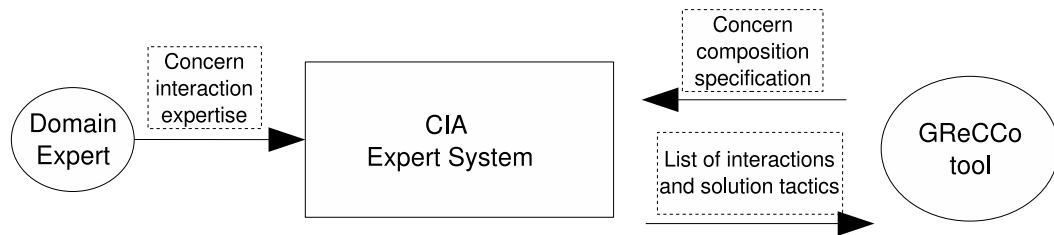


Fig. 28. Architecture of the CIA expert system

Our approach is currently constrained to the composition of only two concerns at a time. However, if we keep a composition history we could query about interactions with concerns that are already composed during the previous steps. An existing prototype of the CIA expert system is not yet incorporated in the GReCCo engine. However, its integration is conceptually straightforward and will be realized in the near future.

4 Case-Study

In this section, we present an application from the domain of Electronic Health Information and Privacy (EHIP). We start from a description of the base part of the application. On top of this application, we apply several reusable concerns using the GReCCo methodology described in the previous section.

4.1 Screening Application

Screening application represents an information system of a screening lab. Patients (*ScreeningSubject*) make an appointment to their radiographic pictures (*Screening*) taken by a *Radiographer*. Two different *Radiologists* perform a *Reading* of the radiographic

screening. In case the reading results are the, same an automatic *Conclusion* is generated. Otherwise, a third reading takes place, whereafter the third radiologist creates a final conclusion. In addition to the system itself, we have realized an additional client-server mechanism so that patients can consult their own data at home from, e.g., a web browser (*Client*). *RegistryService* offers a set of *ScreeningServices*, each having its own *id*. Fig. 29 presents a UML class diagram for the screening lab application. Fig. 30 represents the behavior of a service execution.

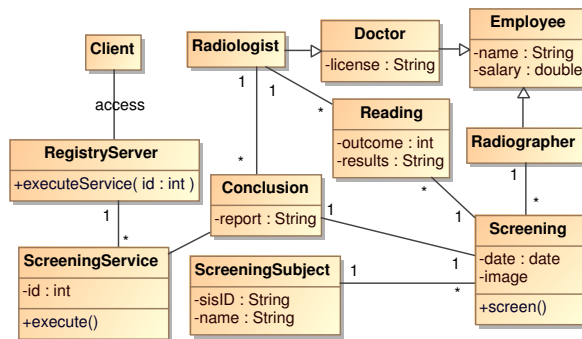


Fig. 29. Screening Lab Application Model

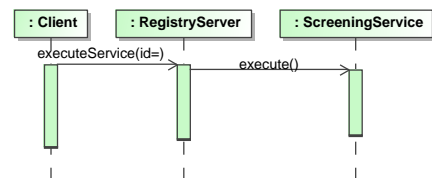


Fig. 30. Execute service

4.2 Application Firewall Pattern

The registry server, introduced in the previous section, must ensure that only authenticated and authorized clients may use a given service. A sound solution in this case would be to interpose an application-level firewall that can analyze incoming requests for the registry services and check them for authorization. A client can access a service of the registry server only if a specific policy authorizes it. Policies for each application are centralized within the *ApplicationFirewall* and they are accessed through a policy authorization point (*PAP*) (fig. 31 and 32).

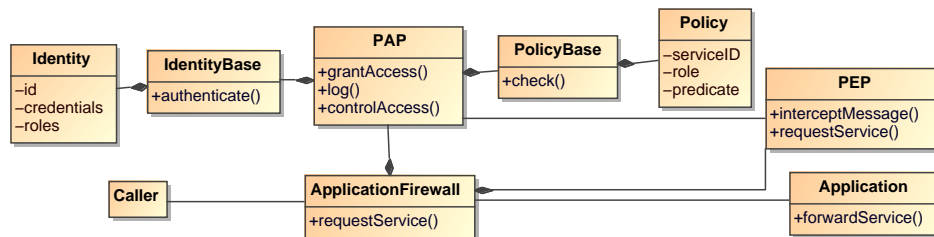


Fig. 31. Application Firewall Pattern

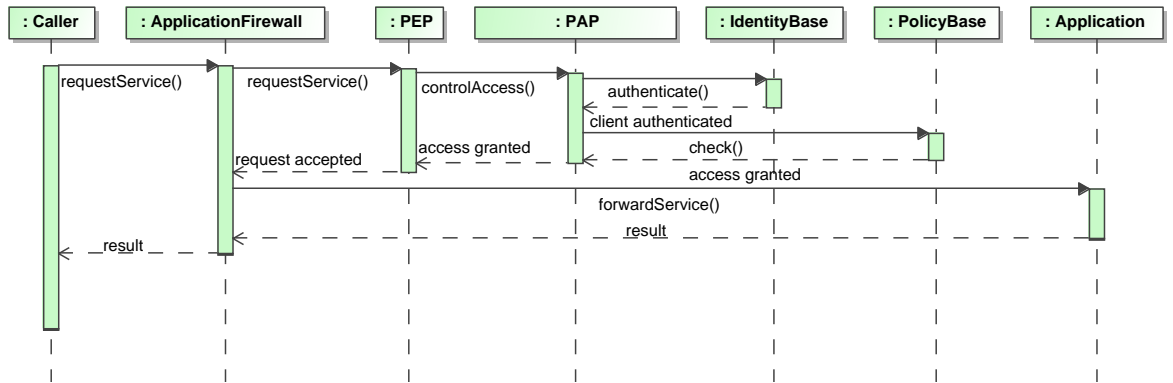


Fig. 32. Request Service

We want to compose the application firewall concern with the base. We follow the framework described in the previous section and define the composition model for the structure (fig. 33). We specify that *Application*, *Caller* and *forwardService()* are template elements that should be instantiated by *RegistryServer*, *Client* and *executeService()* elements respectively. In addition, we need to remove the direct association between *RegistryServer* and *Client* as the application firewall concern will introduce an indirect link between the two. For illustration purposes, we rename the *RegistryServer* to *Server* by placing a `<<modify>>` stereotype on the class, using a tag *name* to indicate the new name.

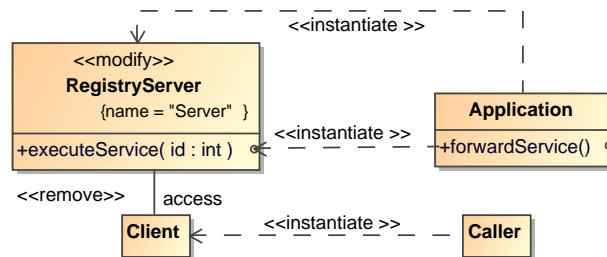
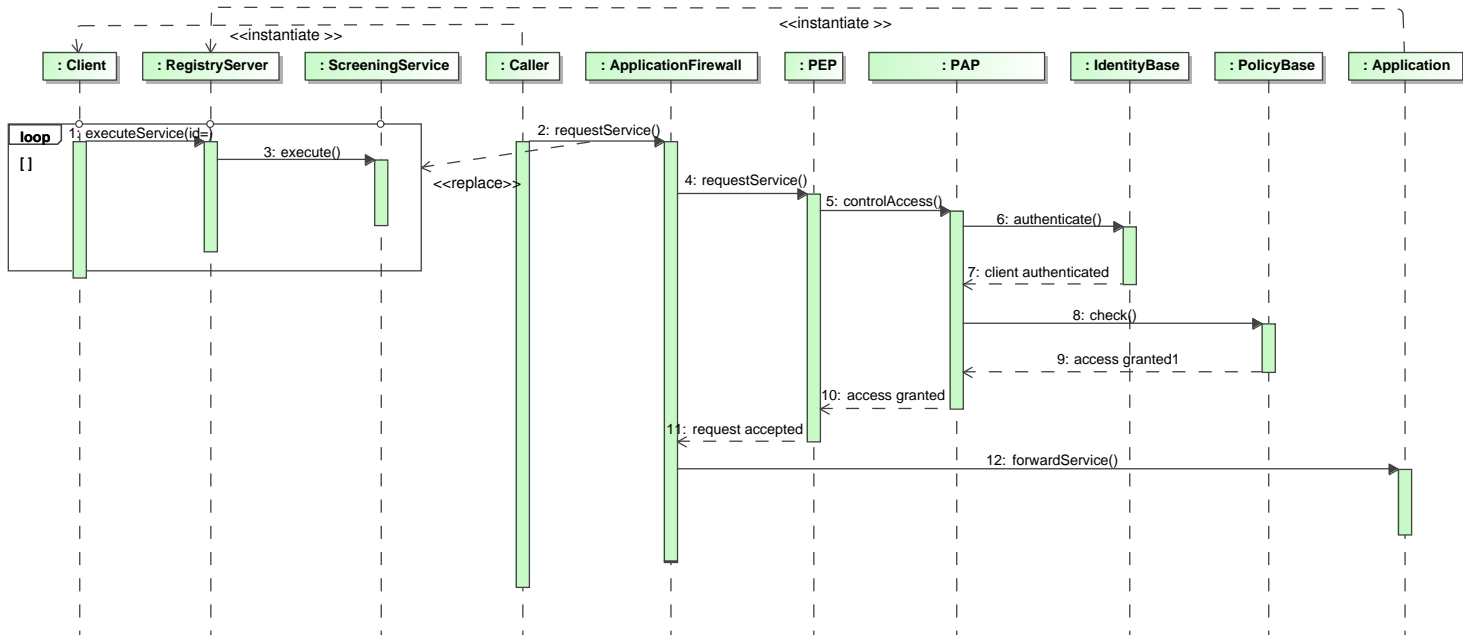


Fig. 33. Structural Composition Model

We want to obtain a composed behavior that actually changes the direct *executeService* call from *Client* to *RegistryServer* by an indirect *requestService* event. To specify this we place a `<<replace>>` dependency link from *requestService* to the *executeService* event (fig. 34).

Fig. 34. Behavior Composition Model



The structure and behavior of the resulting composed model are shown on fig. 35 and 36.

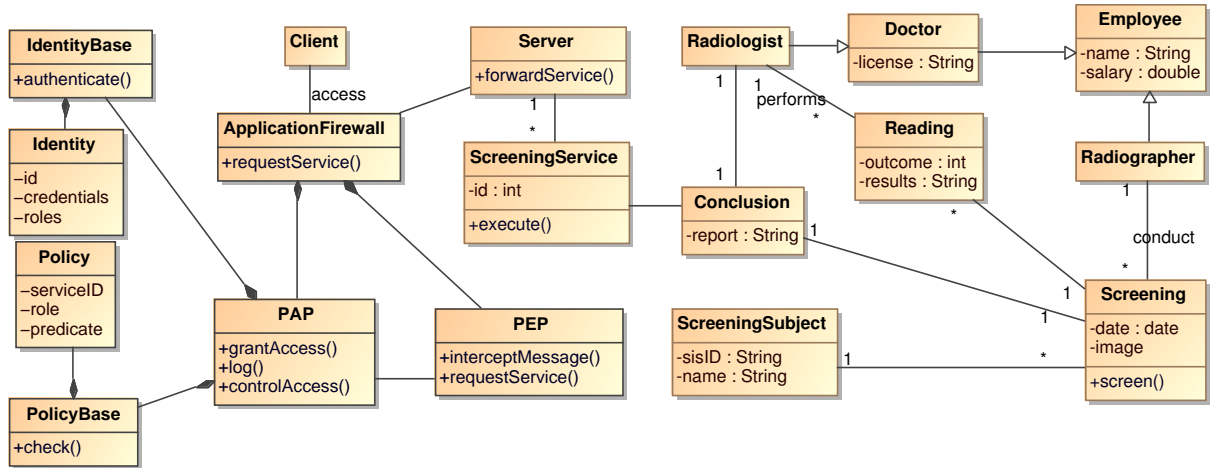


Fig. 35. Screening Application with Application Firewall Structure

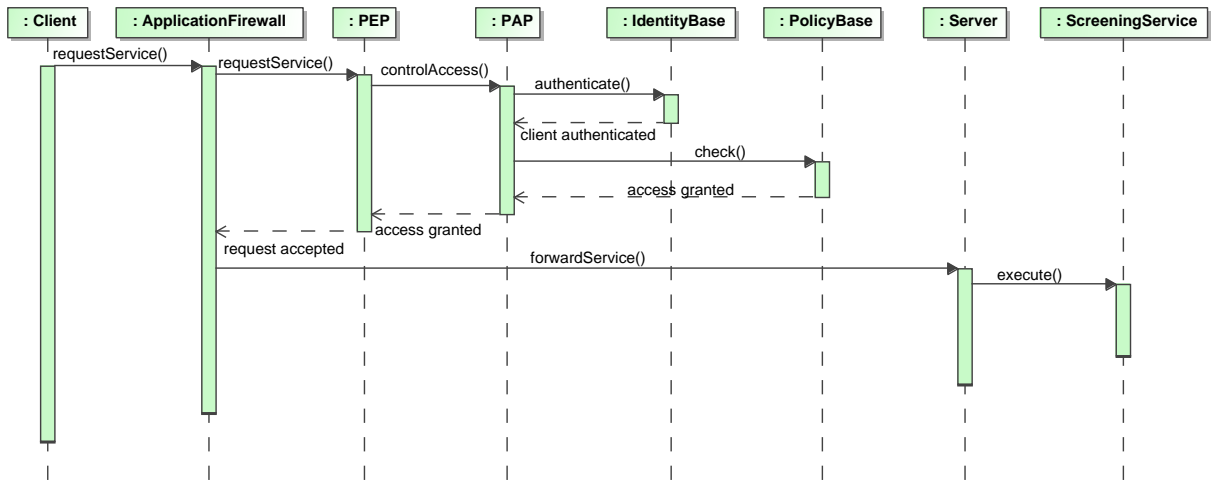


Fig. 36. Screening Application with Application Firewall Behavior

4.3 Audit and Secure Logger Patterns

In the next step of our case study we would like to add auditing support to our design. We have selected the Audit Interceptor pattern (fig. 37 and 38) to centralize auditing functionality. An Audit Interceptor intercepts business tier requests and responses and creates audit events. Audit Interceptor depends on some secure logging facility without which it is impossible to guarantee the integrity of the audit trails. This is why we introduce also the Secure Logger pattern (fig. 39 and 40) which will ensure this additional requirement.

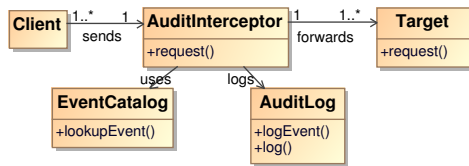


Fig. 37. Audit Interceptor Structure

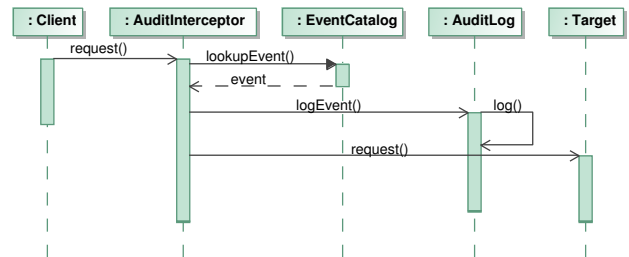


Fig. 38. Audit Interceptor Behavior

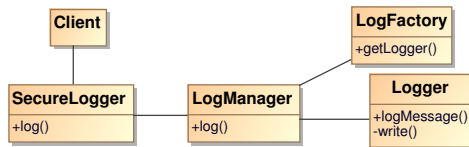


Fig. 39. Secure Logger Structure

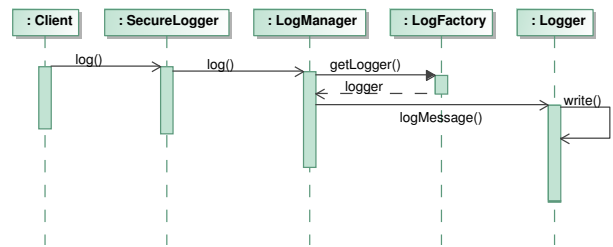


Fig. 40. Secure Logger Behavior

It is possible to apply these patterns to the application model (fig. 35) one by one. However, we chose to combine the two patterns using GReCCo, and apply the combined pattern on the application model. The *AuditLog* entity in the Audit Interceptor pattern represents the *Client* entity in the Secure Logger pattern. Hence, in order to combine the two patterns, we have to merge the two entities by relating them with a `<<merge>>` marked association (fig. 41). We will also place a general ordering relation

between the *log* event from the *AuditLog* object and the *log* event on the *SecureLogger* (fig. 42).

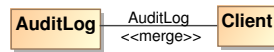


Fig. 41. Structure Composition Model

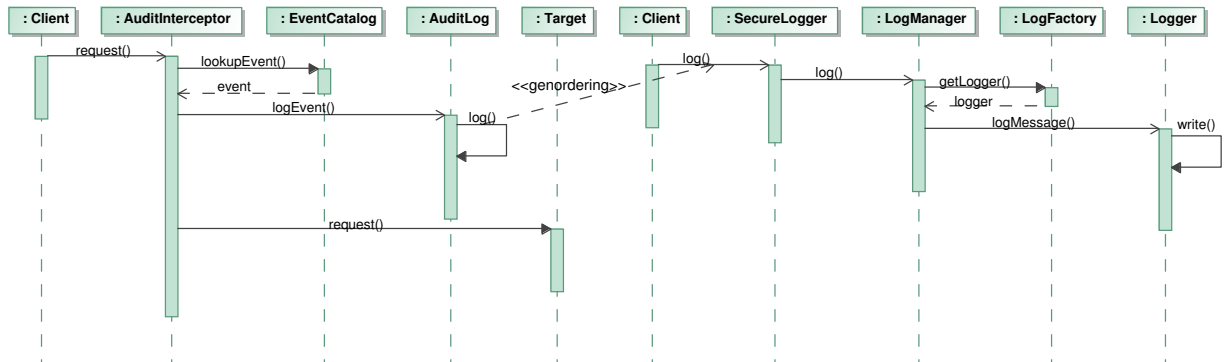


Fig. 42. Behavior Composition Model

The structure and the behavior of the combined pattern are shown on fig. 43 and 44.

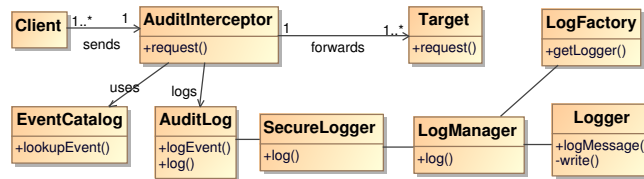


Fig. 43. Combined Structural Model

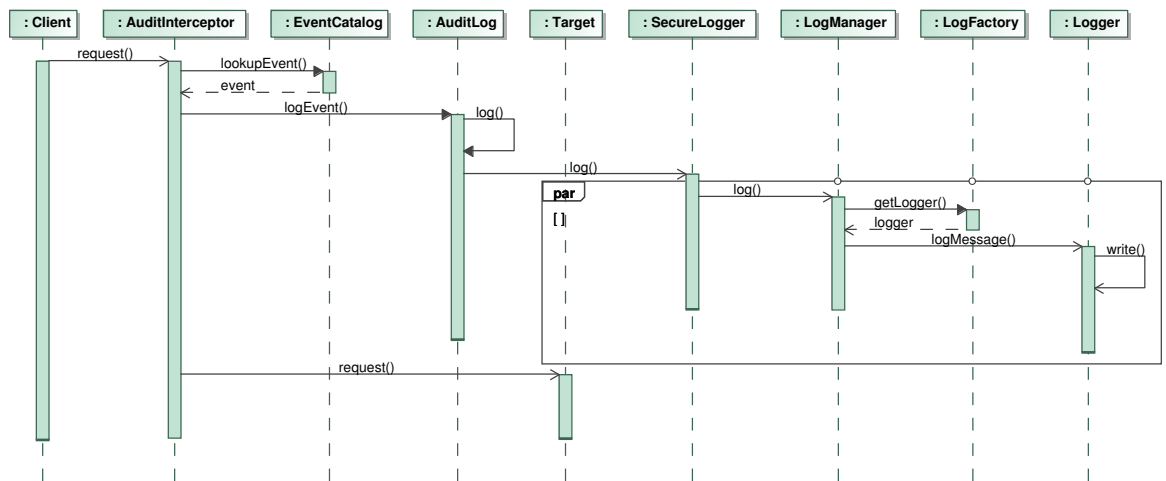


Fig. 44. Combined Behavior Model

4.4 Final Refined Application

In the final step we will compose the combined Audit and Secure Logging concerns (fig. 43 and 44) with the combined base and application firewall pattern (fig. 35 and 36). In order to realize the structural composition, we specify that *Client* and *Target* classes from the combined pattern should be instantiated by the *ApplicationFirewall* and *Server* classes from the main application respectively.

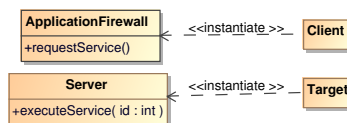


Fig. 45. Structure Composition Model

In order to compose the behavior we first notice that the auditing (using a secure logger) should be performed before the actual request is passed to the application firewall. This is why we use a loop combined fragment around the whole logging sequence and place a $\ll generalordering \gg$ dependency to the *requestService()* event. In addition, as *request()* event is mapped to *forwardService()* event we place an additional $\ll merge \gg$ dependency to denote that the two calls are the same (fig. 46).

Fig. 46. Behavior Composition Model

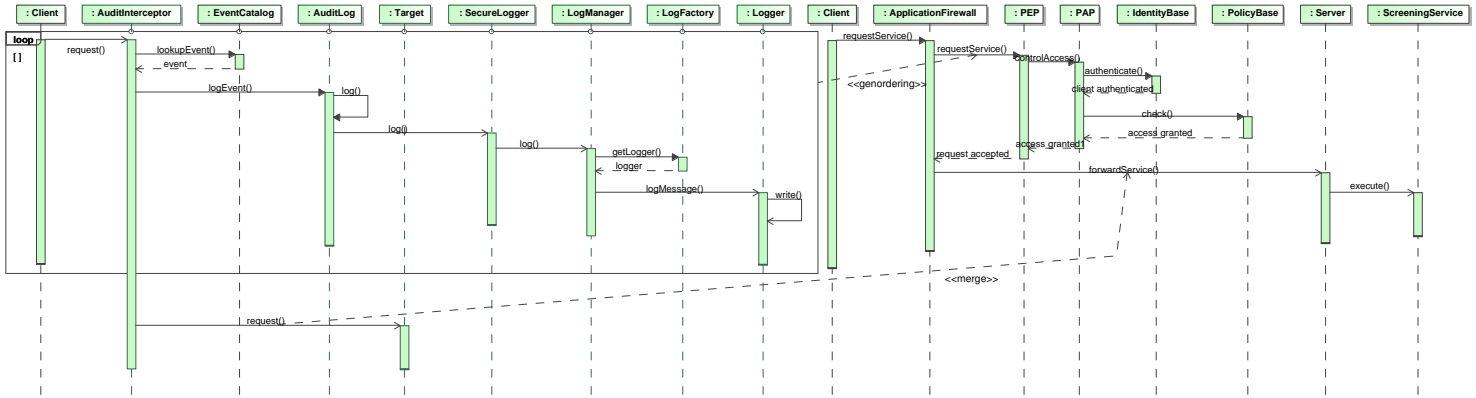


Fig. 47 and 48 show the structure and the behavior of the final application model with Application Firewall, Audit Interceptor and Secure Logger concerns composed into it.

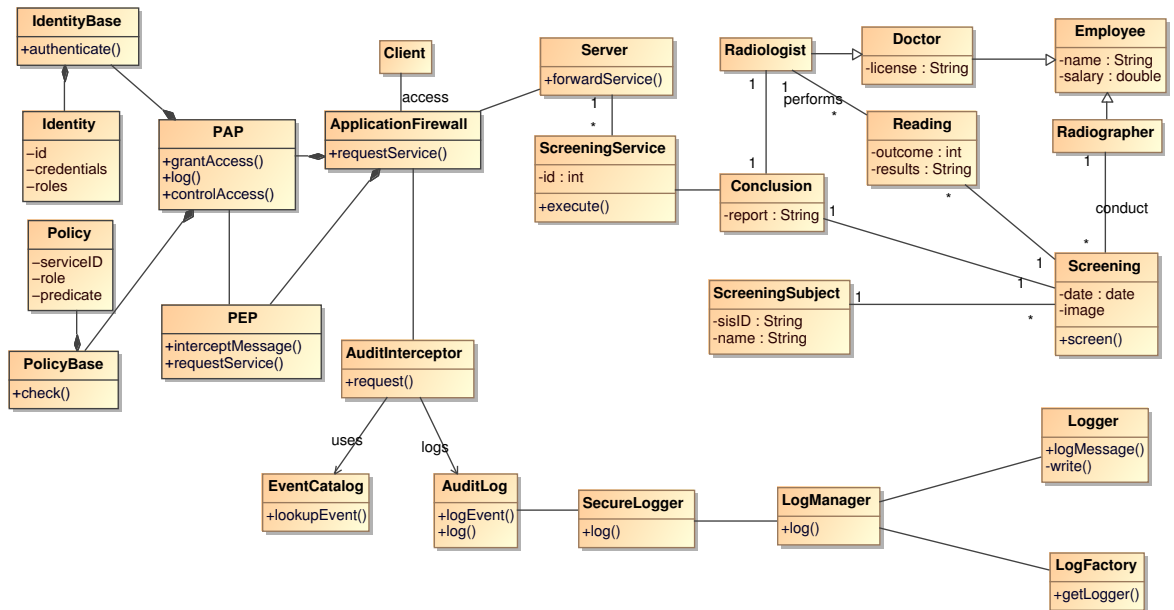


Fig. 47. Final Application Model Structure

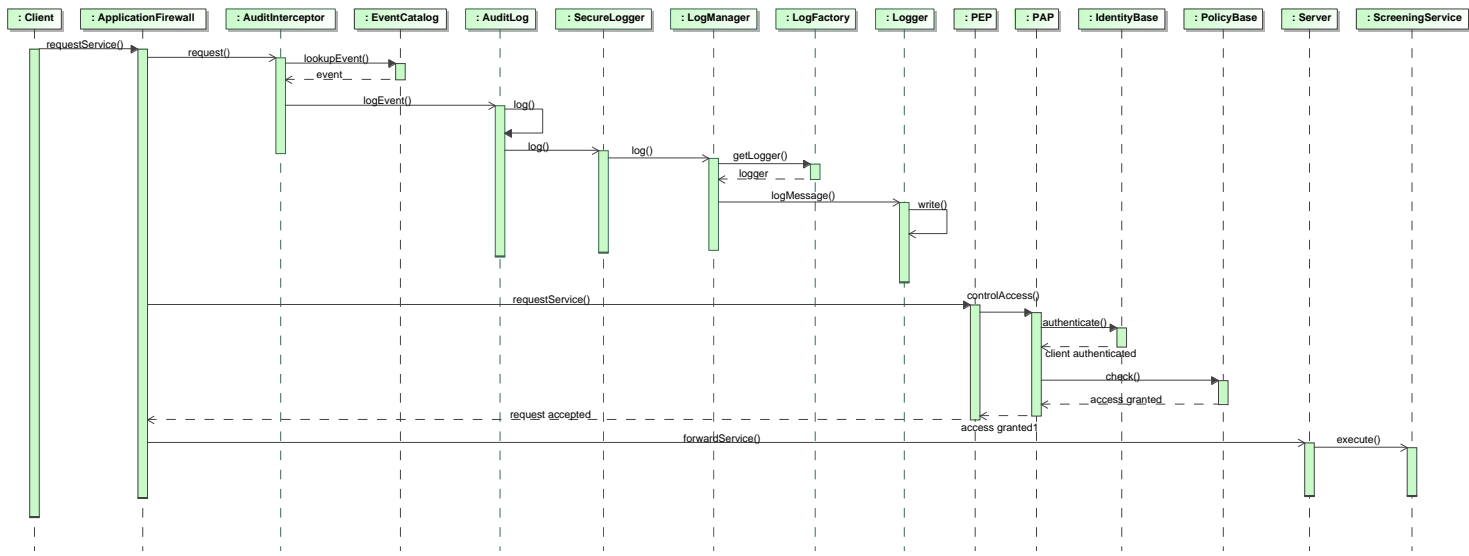
5 Evaluation

Our approach represents a framework for AOM using reusable concerns, in which each concern is modeled using UML class and sequence diagrams. The approach comes with a prototype generic composition engine written in ATL, that can compose two concern models based on the composition directives defined in a dedicated composition model. In this section, we evaluate how our approach tackles the reusability requirements presented in section 2.

5.1 Key requirements for reuse

We have illustrated in our case study how we can use the GRCCo approach to reuse modularized concerns. Each of the concerns used in our case-study can be reused in another application simply by defining an appropriate composition model.

Fig. 48. Final Application Model Behavior



Composition Obliviousness In our approach, each concern is modeled independently from a concrete context in which it is going to be applied. All composition specifics, such as template parameters, structural and behavioral modifications, etc., are specified in a separate composition model, which is unique per composition. We are aware that the maximized reuse of concerns comes at the cost of a potentially larger and more complex composition model. However, we believe that we can minimize the effort required to build a composition model by introducing reusable composition models. We will investigate this in our future work.

Composition symmetry Our framework supports a generic approach to concern composition, i.e., symmetrical composition. We use the term *concern* to uniformly refer to *base* and *aspect* concerns and we allow compositions between any two *concerns*.

Interdependency management We have tackled this requirement by reusing an existing generic framework that helps us to detect and manage the different interactions when composing different concerns. The implementation of the interdependency management framework integration in GReCCo will be available in the near future.

5.2 Proof-of-concept implementation

We have developed a generic composition engine that realizes our approach on top of ATL. A current working version of GReCCo can be found on the GReCCo website [9]. The behavioral composition is not yet fully supported.

6 Related Research

There are many AOM approaches, each pursuing a distinguished goal and providing different concepts as well as notations. We consider them categorized by the alignment to phases criteria introduced by Op de beeck et al. [10]:

6.1 AOM approaches aligned to implementation

Some AOM approaches offer no high level modeling concept that can map the design to the concerns identified during requirements phase. Typically they focus on modeling AOP concepts such as join points, pointcuts, advices, etc. [11,12,13,14]. Even though this allows the modeling of a given concern in a certain manner, these approaches remain too close to the implementation level. All these approaches typically come with a relatively rich tool support. However, these approaches do not score high given our reusability criteria. Almost all implementation centric approaches support composition asymmetry, which only supports *base-aspect* compositions. The composition context is very much pre-defined by the *aspect concerns* (composition obliviousness). In addition, as far as we know, these approaches provide little means to declare and detect concern interdependencies.

6.2 AOM approaches independent from implementation

Several other approaches are implementation independent and provide higher-level mechanisms for concern modeling and composition. These approaches by default score better concerning reusability as they allow a more abstract concern representation and composition. Our approach belongs to this group.

The approach of Jacobson et al. [15] represents a use case driven software development method. AOSD with use cases comes with a systematic process that focuses on the separation of concerns throughout the software development life cycle. The approach provides different means of modularization that allows one to define application-generic modules. However, Jacobson's approach does not focus on reuse, nor does it provide any guidelines on how to reuse artifacts throughout different applications and projects. Moreover, there is no tool support available.

The Theme approach of Clarke et al. [16] provides means for AOM in the analysis phase with *Theme/Doc* and in the design phase with *Theme/UML*. A *Theme* represents a modularized view of a concern in the system. It allows a developer to model features and aspects of a system, and specify how they should be combined. Theme supports composition symmetry and allows any given themes to be composed.

The Aspect-Oriented Architecture Models (AAM) approach of France et al. [17] presents an approach for composing aspect-oriented design models, where each aspect model describes a feature that crosscuts elements in the primary model. Aspect and primary models are composed to obtain an integrated design view. This approach is asymmetric and allows only aspect models to be composed with the primary model.

Klein et al. [18] present an approach for specifying reusable aspect models that define structure and behavior. The approach allows expressing aspect dependencies and weaving them in a dependency-consistent manner. The behavioral composition is realized using a semantic composition algorithm. Klein's approach is symmetric as well and allows does not differentiate between aspect and base models what concerns the composition.

Our approach is similar to these approaches, however, there are several key differences that result in a better concern reuse. All three approaches use a template-based mechanism for crosscutting concern compositions. Each crosscutting concern comes with a set of template parameters that are instantiated during the composition by concrete elements from the other concern. Template parameters already pre-define the composition context as it is not easy to reuse a given concern with a different set of parameters (composition obliviousness). Our approach allows any element of a reusable concern to be parametrized. This provides a more flexible composition mechanism as the same concern can be reused in more contexts using a different set of instantiated parameters. Many concerns should (or must) be used with an a priori given set of parameters. However, there are concerns where it makes sense to allow a more flexible selection of parameters. For example, even though the Application Firewall concern (fig. 31 and 32) comes with a generic policy authorization point (PAP), we would like to allow the possibility of overriding it with a more specific PAP.

Klein's approach is the only one that allows the definition and detection of concern interdependencies. However, only one sort of interdependency, namely dependency, is

supported. The CIA framework, on the other hand, is a systematic approach that supports the complete set of possible interdependencies.

6.3 Other approaches

An approach conceptually similar to GReCCo, is the hyperspace approach [3]. The hyperspace approach is more generic as it allows separation of concerns along multiple dimensions called *hyperspaces*. GReCCo separates them only along class and concern dimensions. The concepts of class and concern in GReCCo can be seen as *hyperslices*, which represent concerns in a given hyperspace. The *declarative completeness* criteria, which requires hyperslices to declare everything to which they refer, is similar to the *composition obliviousness* criteria that requires concerns not to refer to any specific composition. *Declarative completeness* also implies the *composition symmetry* criteria. Finally, the *composition model* in GReCCo is close to a *hypermodule*, which integrates hyperslices. Hyper/J and Theme are the most prominent implementations of the hyperspace approach. However, Hyper/J is a code-level solution and Theme's crosscutting templates do not quite conform to the declarative completeness criteria. Notice that both implementations support only two dimensions/hyperspaces, similar to GReCCo.

7 Conclusions and Future Work

In this paper we have listed and discussed what in our view are key characteristics for the enhancement of concern reuse: composition obliviousness, composition symmetry and concern interdependency management. We have described a new approach for specifying concerns and their compositions and illustrated it on a case-study from the Electronic Health Information and Privacy (EHIP) domain. We have evaluated how the GReCCo approach can help us tackle each of the key qualities for improving reuse.

In the future we plan to investigate the possibility to reuse the composition models. We will also extend the GReCCo engine to support the behavioral composition and integrate it with the Concern Interdependency Acquisition (CIA) framework. We are also investigating the possibility to use domain-specific modeling languages for certain concerns. In addition, we plan to formalize the composition mechanisms and evaluate the scalability of our approach.

References

1. R. France, B. Rumpe: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering'07. (2007)
2. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Satellite Events at the MoDELS 2005 Conference, LNCS 3844, Springer (2006) 128–138 ISBN=0302-9743.
3. Ossher, H., Tarr, P.: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In: Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. (2000)
4. Kiczales, G.: Aspect-oriented programming. (ACM Comput. Surv.) 154
5. Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis. (2000)

6. Steel, C., Nagappan, R., Lai, R.: Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management. (2005)
7. Sanen, F., Truyen, E., Joosen, W.: Managing concern interactions in middleware. In: DAIS. (2007) 267–283
8. OMG: UML superstructure, v2.0. OMG Document number formal/05-07-04 (2005)
9. Generic Reusable Concern Composition Engine: (<http://www.cs.kuleuven.be/~aram/implementation.html>)
10. Opdebeeck, S., Truyen, E., Boucké, N., Sanen, F., Bynens, M., Joosen, W.: A study of aspect-oriented design approaches. In: Technical Report CW435, Department of Computer Science, Katholieke Universiteit Leuven. (2006)
11. Hanenberg, S., Stein, D., Unland, R.: From aspect-oriented design to aspect-oriented programs: tool-supported translation of jpdds into code. In: AOSD. (2007) 49–62
12. Pawlak, R., Seinturier, L., Duchien, L., Martelli, L., Legond-Aubry, F., Florin, G.: Aspect-oriented software development with java aspect components. In: Aspect-Oriented Software Development. (2004) 343–369
13. Fuentes, L., Sánchez, P.: Execution of aspect oriented uml models. In: ECMDA-FA. (2007)
14. Cottenier, T., van den Berg, A., Elrad, T.: Joinpoint inference from behavioral specification to implementation. In: ECOOP. (2007) 476–500
15. Jacobson, I., Ng, P.W.: Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series). Addison-Wesley Professional (2004)
16. Baniassad, E., Clarke, S.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley (2005)
17. Reddy, Y.R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., McEachen, N., Song, E., Georg, G.: Directives for composing aspect-oriented design class models. (2006) 75–105
18. Klein, J., Kienzle, J.: Reusable aspect models. In: Proc. of the 11th Int. Workshop on AOM. (2007)