Composing Multiple Concerns Using Composition Filters

Lodewijk Bergmans & Mehmet Aksit
TRESE group, Department of Computer Science, University of Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands.
email: {bergmans | aksit}@cs.utwente.nl
http://trese.cs.utwente.nl

1. Introduction

It has been claimed in a number of publications that the object abstraction may not be adequate in modeling certain concerns effectively, especially if these concerns incorporate complex semantics and have a crosscutting property [4, 12]. Various techniques have been proposed to manage such concerns such as Adaptive Programming [14], Hyperspaces [16], AspectJ [13] and Composition Filters [1].

This article first presents an example to illustrate the issue of composing and reusing multiple concerns in object-oriented programs when requirements evolve. As a solution to the identified problems, the Composition Filters (CF) model is presented.

Filters are used to express complex and crosscutting concerns. The CF model extends the object abstraction in a modular and orthogonal way. Modular extension means that filters can be attached to objects expressed in different languages without modifying the definition of objects in these languages. Orthogonal extension means that the semantics of a filter is independent of the semantics of other filters. The modular and orthogonal extension properties distinguish the CF model from most other aspect-oriented techniques. Modular extension makes filters independent of the implementation. Orthogonal extension makes filters composable. The previous version of the CF model was used to express concerns for a single object [6, 5]. This article explains how the CF model can be applied to express and compose concerns within and across multiple objects.

This article is organized as follows. The following section introduces an example application, which evolves in time due to changes in the requirements. Section 3 explains the CF model and shows how it can cope with the evolution problems. Finally, section 4 evaluates the CF model and gives conclusions.

2. An Example: An Administrative System For Social Security Services

We will now present a simple example to illustrate the issue of composing and reusing multiple concerns in object-oriented programs when the requirements evolve. This example is a simplified version of the pilot study [11]. Later, due to evolving business context, the initial software has undergone a series of modifications.

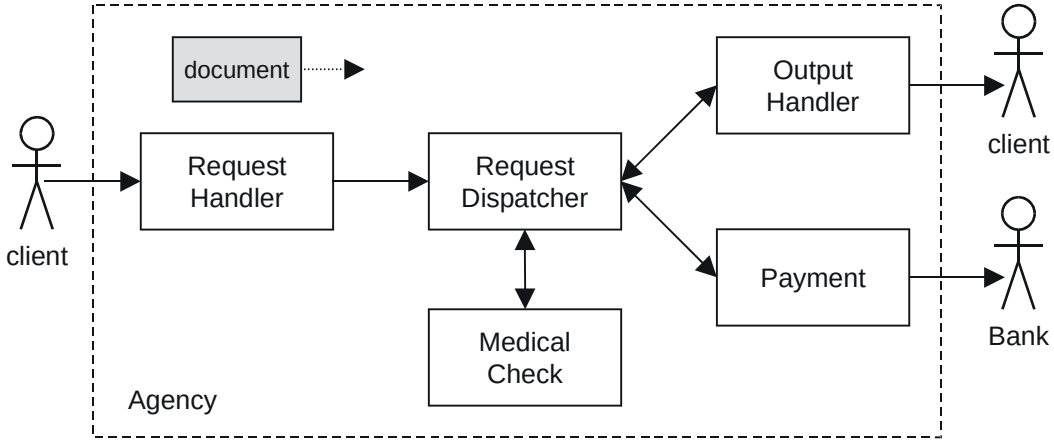

Figure 1. Tasks in the example system.

Assume that a government-funded agency is responsible for the implementation of disablement insurance laws. As shown in Figure 1, the agency implements five tasks. Task RequestHandler creates an entry for clients. Entries are represented as documents. RequestDispatcher implements the evaluation and distribution of the requests to the necessary tasks. A request can be dispatched to MedicalCheck, Payment or OutputHandler. MedicalCheck is responsible for evaluating client's disablement. Payment is responsible for issuing bank orders. ResponseHandler is responsible for communicating with the clients. A typical claim is subsequently processed by RequestHandler, RequestDispatcher, MedicalCheck, Payment and OutputHandler. Various other interaction scenarios are also possible.

2.1 The Software System

2.1.1 Modeling Client's Requests

The system has been implemented as a set of tasks. For each client's request, a document is created. Depending on the document type and client's data, the document is edited and sent to the appropriate tasks using a standard email system. Each relevant task processes the document accordingly. In this article, we neglect the agency specific user interfacing tools and concentrate more on the core application classes.

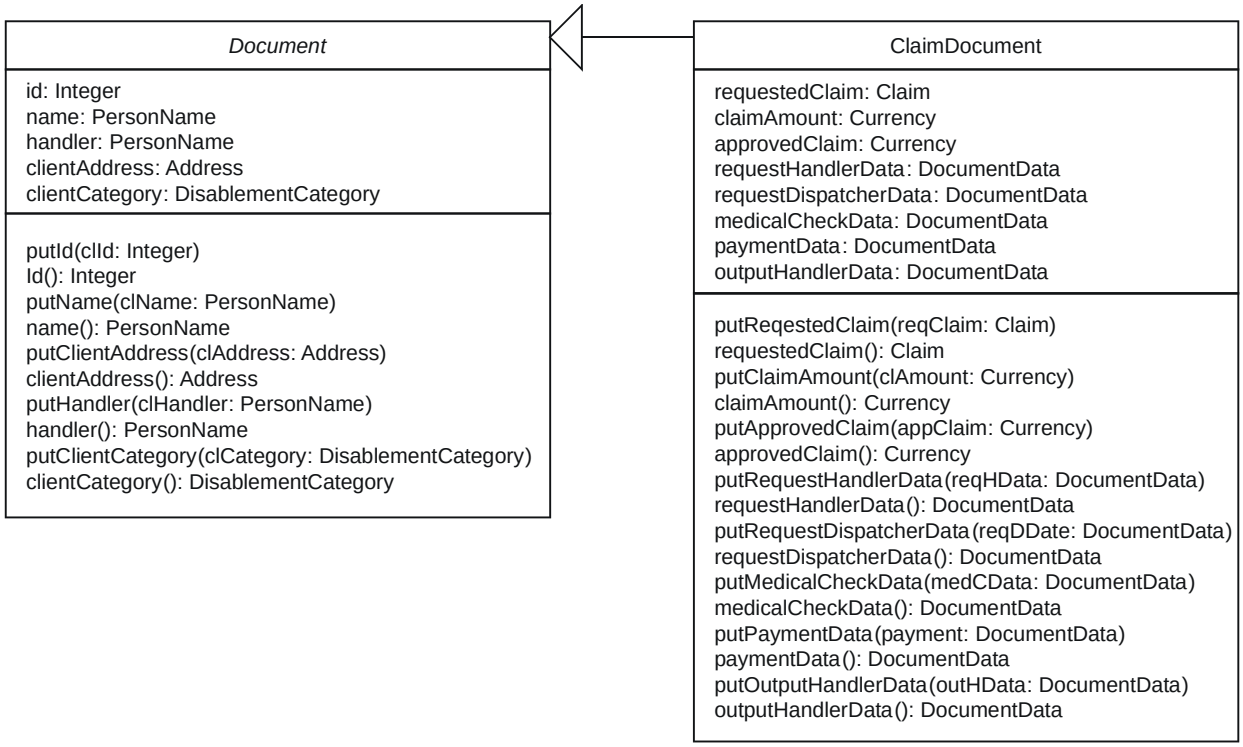

Figure 2. Part of the document class hierarchy, which is used for representing client requests.

As shown in Figure 2, class *Document* is the root class of all document types. Every document has 5 attributes. The attributes *id, name, clientAddress* are used for storing client's data. The attribute *handler* represents the clerk who is in charge of processing the request. The attribute *clientCategory* specifies the classification of the client with respect to the disablement laws. Class *Document* implements 10 operations, which are used to read and write these attributes.

Class *Document* has several subclasses. For example, *ClaimDocument* is used to represent the claims of clients. This class declares 8 attributes. The attribute *requestedClaim* represents the type of client's claim, such as *medicine, hospital costs*, etc. The attribute *claimAmount* is the claimed amount of money. The attribute *approvedClaim* is the amount approved by the agency. The remaining attributes are filled in by various tasks while the document is being processed.

2.1.2 Modeling The Tasks

As shown in Figure 3, *TaskProcessor* declares the basic operations for all tasks. The operation *processDocument* accepts a document as an argument and opens an editor for the document by calling on *startEditorWithDocument*. When the document is edited by a particular task, the operation *forwardDocument* is called. Both of these operations are overridden by the subclasses.

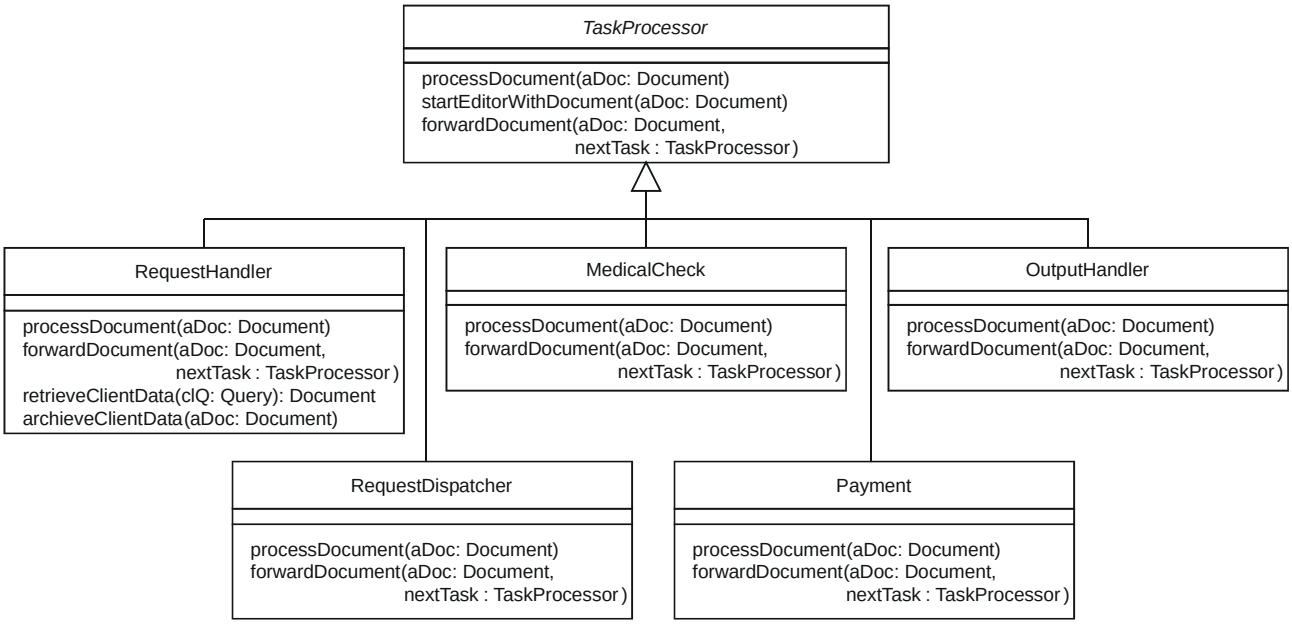

Figure 3. Class hierarchy for tasks.

RequestHandler implements the front-end of the office. For example, if a client wants to issue a claim, this task creates an object of *ClaimDocument*, retrieves the necessary client data and opens an editor for the document object. The responsible clerk should then enter the data on the field defined for *RequestHandler*. When the task is completed, the clerk selects the next task and calls *forwardDocument*. The operation *forwardDocument* prepares the document and passes it as an argument to the operation *processDocument* on the next task. Subsequently, each clerk in the process enters data into the appropriate data field and forwards the document according to the office procedure. In this system, creating a new process can be realized by creating a new structural document subclass[1].

2.2 Evolution 1: Protecting Documents

In the initial system, a clerk could edit any field in a document. A request dispatcher clerk could, for instance, accidentally edit the medical data field. Therefore, it was found necessary to protect the documents. We consider two alternatives for enhancing *ClaimDocument* for protection: to modify and recompile *ClaimDocument* or to introduce a new class and reuse class *ClaimDocument* through inheritance or aggregation.

Figure 4 shows an inheritance-based solution where class *ProtectedClaimDocument* inherits from *ClaimDocument*, declares a new operation called *activeTask*, and redefines 15 operations of class *ClaimDocument*.

[1] In the actual system, there were approximately 30 different document types.

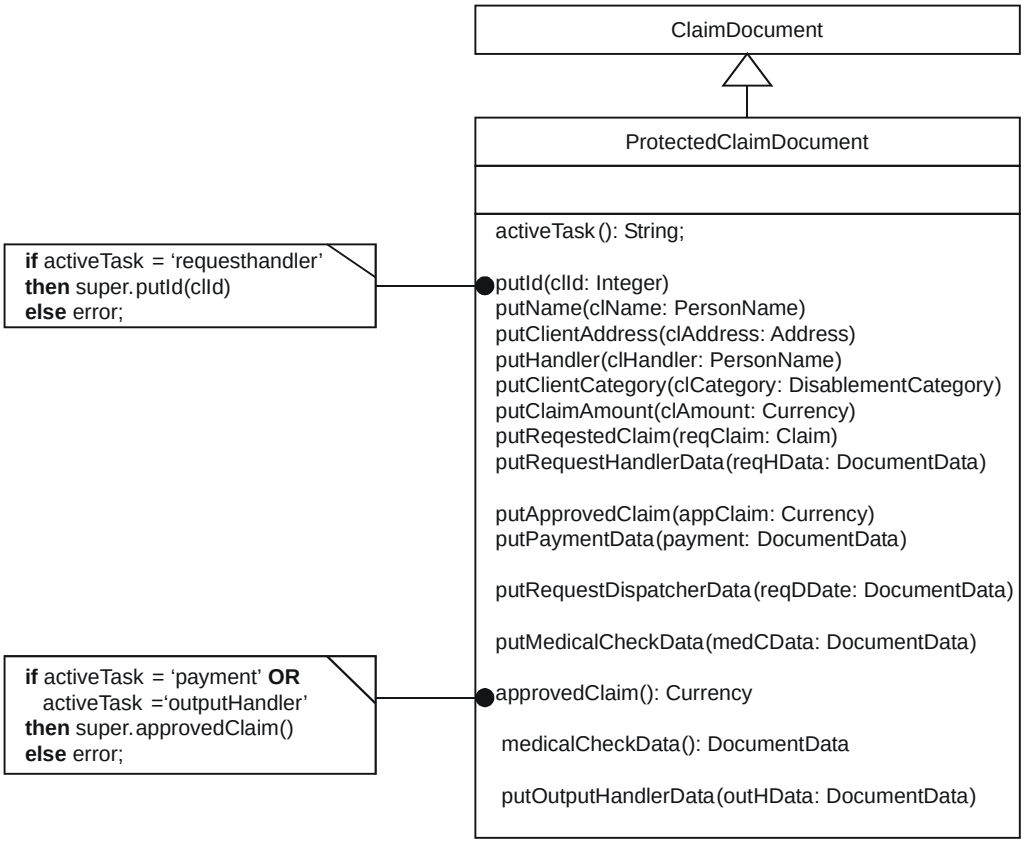

Figure 4. Class ProtectedClaimDocument *with the pseudo-code implementations of* putId *and* approvedClaim.

The operation *activeTask* returns the identity of the active task. The operations *putId*, *putName*, *putClientAddress*, *putHandler*, *putClientCategory*, *putClaimAmount*, *putRequestedClaim*, and *putRequestHandlerData* are redefined such that only *RequestHandler* can invoke on them. This is implemented by calling on the corresponding operation at the superclass only if the current task is RequestHandler. Otherwise an error message is generated. Similarly, *putApprovedClaim* and *putPaymentData* can only be invoked by Payment, *putRequestDispatcherData* by *Request Dispatcher*, *putMedicalCheckData* by MedicalCheck and *putOutputHandlerdata* by OutputHandler. The operations *approvedClaim* and *medicalCheckData* can be invoked by two different tasks. These are namely, Payment and OutputHandler, and RequestDispatcher and MedicalCheck, respectively. One of the advantages of using inheritance here is the transitive reuse of the remaining 11 operations, since they do not require any view enforcement.

Aggregation-based reuse has the advantage that the aggregated object can be replaced at run-time, for example to adapt the behavior of an object [9]. However, in this case, transitive reuse is not possible and the remaining 11 operations have to be forwarded to the aggregated object.

In the actual pilot project, the number of required operation redefinition was several order of magnitudes higher, since there were multiple document types used in the agency, which required a similar sort of protection.

2.3 Evolution 2: Adding Workflow

In the previous implementation, the clerks had to decide which task to be executed next. To enforce a process, class *WorkFlowEngine* is introduced. The attribute *workFlowSpec* of *WorkFlowEngine* represents the process to be enforced. This attribute can be set and read by the operations *putWorkFlowSpec* and *workflowSpec*, respectively. The operation *selectTask* accepts a document as an argument and based on the workflow specification and the state of the document, returns the identity of the next task.

Adding a workflow to the system requires redefinition of *forwardDocument* for all task classes. The *forwardDocument* first calls on *selectTask* of *WorkFlowEngine*, which returns the next task. The operation *forwardDocument* cannot be implemented at the superclass level, since every task implements this operation in a specific manner.

2.4 Evolution 3: Adding Document Queues

Each document in the process may require a different processing time. To improve the average throughput, a document queue is defined for every task. This requires implementing a buffer for every task class. The operation *processDocument* has to be *mutual exclusive*; every call must be queued until *processDocument* is ready with its current task. Also the operation *noActiveThreads* is introduced to determine if the task is idle.

2.5 Evolution 4: Adding Logging

One of the important concerns of the workflow system is to monitor the process, detect the bottlenecks and reschedule and/or reallocate the resources, if necessary.

To determine which operations to be monitored is difficult to determine a priori since it depends on the purpose of monitoring. All the interactions among objects are therefore registered.

Class *Logger* is introduced to register the interactions in the system. *Logger* has 7 operations. The operations *loggingEnabled* and *loggingDisabled* activates and deactivates the logging mechanism, respectively. The operation *log* accepts an object of class *Message* as an argument, extracts the necessary information from the message as desired, and returns. The operations *putOperationList* and *operationList* are used to set or read the list of operations to be monitored, respectively. The operation *reset* clears all the registered data. The operation *logdata* is used to read the registered calls. Adding a logging facility also requires redefinition of all methods of task and document classes. This is because before executing any call, the operation *log* of *Logger* must be called.

2.6 Evolution 5: Adding Locking

Soon after introducing the logging facility, it was found necessary to temporarily lock a *task* or *document*, for instance, for reallocating resources, debugging or for obtaining a snapshot of the system. For every class in the system, the operations *lock* and *unlock* are introduced. . The operation *lock* queues all the requests unless *unlock* is invoked. The invocation of *unlock* has no influence if

the object has been already *unlocked*. Note that if a semaphore-like mechanism is used to implement locking, every operation of a class has to be redefined.

3. The Composition Filters Approach

In this section we will introduce the CF approach to aspect-oriented programming and composition of multi-dimensional concerns. First we will briefly explain the CF object model, which modularly extends the 'conventional' object-oriented model with a filter specifications. The recent version of the CF model includes the so-called *superimposition* specification. The superimposition specification describes the places in the program –rather than just within the class– where concern behavior is to be added. This CF model supports crosscutting behavior across the set of methods supported by a single or multiple classes. For space reasons, we have left out details in several places.

3.1 A Conceptual Model of Composition Filters

The composition filters model adopts *declarative specifications*: these describe *what* should be done, rather than *how*. This means, for example, that many different implementations are possible for each specification. The semantics of composition filters are best explained in terms of a run-time model. Therefore we will adopt the run-time perspective to describe the conceptual model in this section[2].

3.1.1 Basic Structure of Composition Filters Objects

The composition filters model is a modular extension to the conventional *object-based* model [17] as adopted e.g. by programming languages such as Java, C++ and Smalltalk, and component models such as CORBA and Enterprise JavaBeans. Since in an object-based system, all behavior is implemented by sending messages between objects, the manipulation of incoming and outgoing messages of objects can express a large category of behavior changes. To support such manipulation, a layer called the *interface* wraps the *implementation* object. Any object-based model can define the implementation object. The composition filters model and its elements are shown in *Figure 5*.

[2] Note that this model is intended as a representation, which is independent of both the programming language syntax, and of the actual implementation in executable code.

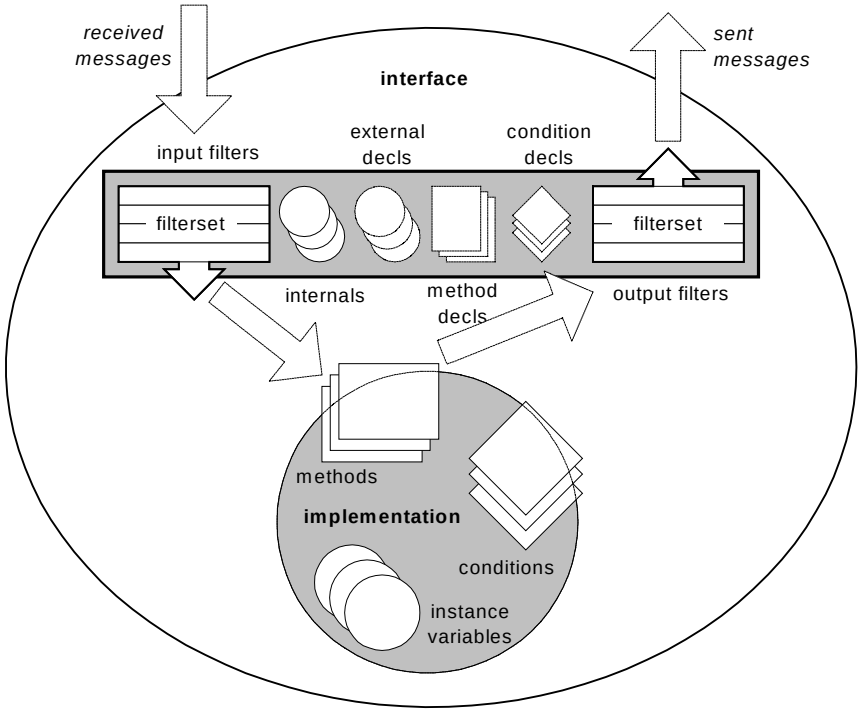

Figure 5. Simplified representation of the CF model.

The *filters* define the (observable) behavior of the object. Each filter specifies a particular inspection and manipulation of messages. Input and output filters can manipulate messages received by respectively sent from an object. These are declared in separate *filtersets*. Filters may refer to *internal objects* or *external objects*. Internal objects are instantiated and encapsulated within the CF object whereas external objects are references (by name) to objects outside the CF object, such as globals or shared objects.

Filters define the behavior of the object as a composition of the behavior of its implementation part, its internal and its external objects. The interface part is a modular, language-independent[3] extension to the implementation part. In Figure 6, an example is shown of the specification of a CF object with most of the elements we have mentioned so far. The specification of the filters will be explained in 3.1.3.

The implementation part can define two types of methods: *regular methods* and *condition methods* (*conditions* for short). The regular methods, which implement the functional behavior of the object, may be invoked through messages, if the filters of the object allow this. Conditions must implement side-effect free Boolean expressions that provide information about the state of the object[4]. Conditions support the filters to decide how to manipulate messages. In the interface, methods and conditions are declared to verify consistency and for type-checking purposes.

---

[3] We adopt the UML notation for declaring objects and methods.

[4] Although it would be preferable if the constraint of being side effect free could be statically enforced, this is not possible in general without sacrificing some of the expression power of the condition expressions.

```
concern ProtectedClaimDocument begin
   filterinterface documentWithViews begin
      internals
         document: ClaimDocument;
      externals   // no externals defined by this class
      conditions
         inactiveRH; inactiveRD; inactiveMC; inactiveP; inactiveOH;
      methods
         activeTask();
      inputfilters
         // introduced later in this paper
   end filterinterface DocumentWithViews;

   implementation in Java  // for example
   class ProtectedClaimDocument {
      boolean inactiveRH() { return this.activeTask().class()!=RequestHandler };
      boolean inactiveRD() { … };
      boolean inactiveMC() { … };
      boolean inactiveP()  { … };
      boolean inactiveOH() { … };
      String activeTask()  { … };
      }
   end implementation
end concern ProtectedClaimDocument;
```

Figure 6. Specification of ProtectedClaimDocument*, illustrating the elements of a CF object.*

3.1.2 The Principle of Message Filtering

We will explain the basic mechanism of message filtering with the aid of Figure 7. In the description we will assume input filters, but output filters work in exactly the same manner.

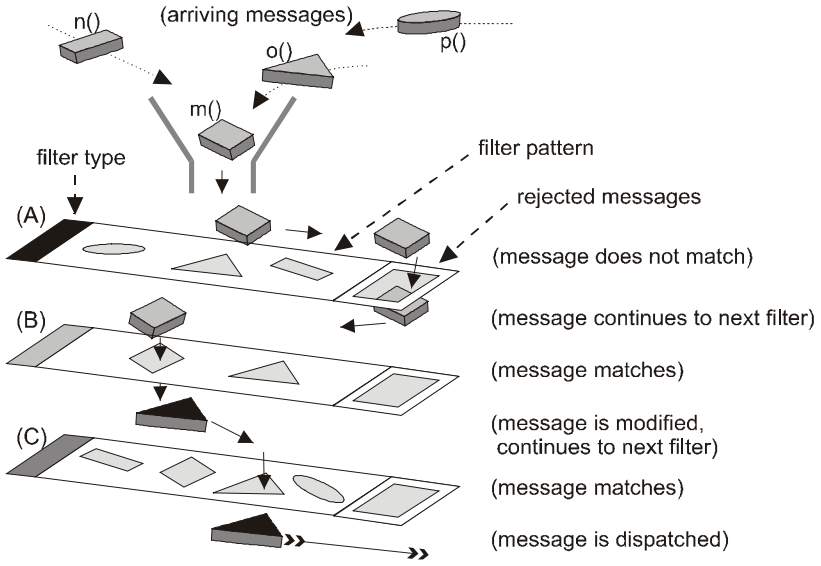

Figure 7. An intuitive schema of message filtering.

To properly understand Figure 7, the following should be kept in mind: filters are defined in an ordered set. Each message[5] has to pass the filters in the set, until it is discarded or can be dispatched. Dispatching means that the message is activated again, for example to start the execution of a local method, or to be delegated to another object. Each filter can either accept or reject a message. The

[5] Messages are first reified, i.e. a first-class representation is created. Composition filters thus –conceptually– apply a form of message reflection [**Error! Reference source not found.**], but note that actual implementations, may 'optimize away' the reification.

semantics associated with acceptance or rejection depend on the type of the filter. Examples of predefined filter types are:

Dispatch: if the message is accepted, it is dispatched to the current target of the message, otherwise the message continues to the subsequent filter (if there is none, an exception is raised) [1].

Error: if the filter rejects the message, it raises an exception, otherwise the message continues to the next filter in the set [1].

Wait: if the message is accepted, it continues to the next filter in the set. The message is queued as long as the evaluation of the filter expression results in a rejection [5, 6].

Meta: if the message is accepted, the reified message is sent as a parameter of another –meta message– to a named object, otherwise the message just continues to the next filter. The object that receives the meta message can observe and manipulate the message, than re-activate its execution [2].

Figure 7 visualizes the processing of messages. Each filter tries to match messages based on its own *filter expression*. All filters use a simple common –declarative– pattern matching language. The matching process is primarily based on the target object and the selector of the message, together with the conditions of the object.

3.1.3 Specifying Filters

We will explain filter specifications in more detail through class *ProtectedClaimDocument* that was introduced in section 2.2 and for which the skeleton code has been shown in Figure 6. *ProtectedClaimDocument* composes the view behavior with the existing *ClaimDocument* abstraction through inheritance. We will first illustrate how filters can express views in a modular way, and then show how filters can express inheritance.

ProtectedClaimDocument requires constraints on the execution of the methods that are available on the interface, based on the task that is currently processing the document. For example, the following filter specification can express these views:

```
protection: Error = { PaymentActive => {putApprovedClaim, approvedClaim},
                      MedicalCheckActive => {putMedicalCheckData, medicalCheckData},
                      …  // etc. for the other views
                      };
```

This specification declares a filter of type *Error* with the name *protection*, followed by the filter (initialization) expression. The filter expression consists of a number of *filter elements*, each of these is a pair `"<condition> => <message expressions>"`. The meaning of this pair is that the message expressions are only evaluated if the condition evaluates to *true*. For the first filter element this means that only if the condition *PaymentActive* is true *and* if the selector of the received message is either *putApprovedClaim* or *approvedClaim*, the message is accepted at the first element and can proceed to the next filter. Otherwise the subsequent filter element (in this case verifying

messages that apply to the *MedicalCheck* task) will be evaluated. If the message is not accepted by any of the filter elements –i.e. the filter rejects the message–, the *Error* filter will raise an exception.

An alternative filter specification of the same concern is shown in the following:

```
inputfilters
    viewP :Error = {inactiveP  ~> {putApprovedClaim, approvedClaim} };
    viewMC:Error = {inactiveMC ~> {putMedicalCheckData,medicalCheckData} };
        // etc. for the other views
```

In this case, the view is specified 'inversely': each *Error* filter rejects the methods that are only allowed for a particular task, if this task is not active. These filters use instead of the '=>' *enable* operator between the condition and the message expression, the '~>' *exclusion* operator, which specifies that if the condition is satisfied, all messages match except the ones that are specified on the right hand side. This is a more 'open' specification, since it is now possible to introduce additional constraints upon messages (similarly): either in separate filter specifications, or in separate (derived) classes.

The last input filter of *ProtectedClaimDocument* specifies inheritance from *ClaimDocument* as follows:

```
inh:Dispatch = { inner.* , document.* };
```

This filter of type *Dispatch* adopts the '*' wildcard which means that it matches the received message if the selector of the message is in the signature of the target object specified before the dot (in this case *inner* respectively *document*). The target *inner* is a pseudo-variable that refers to the 'bare' implementation object. The target *document* is declared as an *internal* instance of class *ClaimDocument* (see Figure 6). Evaluation of this filter will thus result in a dispatch to *inner* (i.e. the execution of a local method) if the message is in the signature of *inner*. Otherwise, if it is in the signature of *ClaimDocument*, the message will be delegated to *document*. This filter effectively simulates inheritance (including a dynamically bound notion of *self*).

3.2 Superimposition of Crosscutting Behavior

So far, we have shown only examples of behavior that crosscuts a number of methods within a single object. An important category of aspect-oriented programming problems are those where concerns crosscut multiple objects. The Composition Filters model provides *superimposition*, which means that one abstraction can enhance other abstractions with additional concerns by decorating ('superimposing') concern specifications.

Figure 8 illustrates the CF model including superimposition. Here, the declaration of a filterinterface has been separated from its instantiation and multiple declarations are allowed. In addition, multiple instantiations of filterinterfaces are possible as well. Incoming and outgoing messages have to pass

through all the instantiated filterinterfaces[6]. We added a *superimposition* element that specifies where filterinterfaces are superimposed: on the concern itself, or on other concerns[7].

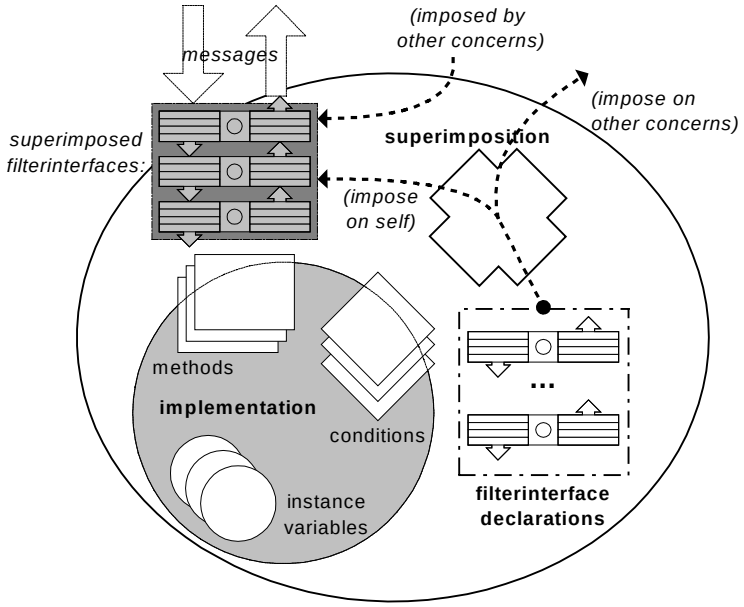

Figure 8. The CF model with superimposition.

As illustrated in the figure, it is possible to compose multiple, crosscutting concerns by superimposing filterinterfaces. The concerns are defined by filters, and thus work by manipulating incoming and outgoing messages of the implementation object only. This brings two important properties: first, the encapsulation of the implementation object is not violated, and second, composability of concerns is improved.

Consider, for example, the implementation of the workflow concern using the CF model. As shown in Figure 9, this concern consists of two parts: a (shared) part that implements the workflow engine itself, and a crosscutting part that ensures that all relevant concerns in the application will actually use the engine for selecting the next task.

---

[6] In this paper we will simply assume that the filterinterfaces are ordered sequentially in order of instantiation. We will not discuss (interesting) issues such as how to influence the ordering and alternatives to sequential evaluation.

[7] The superimposition element can also impose a binding of methods and conditions to other concerns, but this is not shown in the figure.

```
concern WorkFlowEngine begin // introduces centralized workflow control

   filterinterface useWorkFlowEngine begin // this part declares the crosscutting code
      externals
         wfEngine : WorkFlowEngine; // *declare* a shared instance of this concern
      inputfilters
         redirect : Meta = { [forwardDocument]wfEngine.selectTask };
   end filterinterface useWorkFlowEngine;

   filterinterface engine begin //defines the interface of the workflow engine object
      methods
         selectTask(Message);
         setWorkFlow(WorkFlow);
      inputfilters
         disp : Dispatch = { inner.* };  // accept all methods implemented by myself
   end filterinterface engine;

   superimposition begin
      selectors
         allTasks = { *=RequestHandler, *=RequestDispatcher, *=OutputHandler,
                      *=MedicalCheck, *=Payment};
      filterinterfaces
         self <- self::engine;
         allTasks <- self::useWorkFlowEngine;
   end superimposition;

   implementation in Java;
      class WorkFlowEngineClass{
         WorkFlow workFlowRepr;
         void selectTask(mess Message) { … };
         void setWorkFlow(WorkFlow wf) { … };
      }
   end implementation;

end concern WorkFlowEngine;
```

Figure 9. Specification of WorkflowEngine*, illustrating the elements of a (crosscutting) CF concern.*

The *useWorkFlowEngine* filterinterface defines a filter of type *Meta*, which intercepts *forwardDocument* messages and sends them in reified form, as the argument of message *selectTask*, to *wfEngine*. The *engine* filterinterface and the implementation part together implement the workflow engine. Next to some methods for accessing and manipulating the workflow representation (in this case only the method *setWorkFlow* is shown), it defines the method *selectTask*. This method determines the next task that should handle the document, modifies the corresponding argument of the message object, and then *fires* the message so that it continues its original execution–but with an updated argument.

The *superimposition* clause specifies how the concerns crosscut each other. The superimposition clause starts with a *selectors* part that specifies a number of *join point selectors*: a selector is an abstraction of all the locations that designate a specific crosscut. Concern *WorkFlowEngine* defines a single selector named *allTasks*. This selector repeatedly uses the "`*=<ConcernName>`" expression to specify all objects that are instances of the various classes that represent tasks. The selectors part is followed by a number of sections that can specify which objects, conditions, methods respectively filterinterfaces are superimposed upon locations as designated by one or more selectors. In this example the filterinterface *engine*, is superimposed upon *self*: this means that instances of *WorkFlowEngine* will include an instance of the *engine* filterinterface. In addition, the

useWorkFlowEngine filterinterface (which can be found in the same concern, as designated by "`self::`"[8]) is superimposed upon all the instances defined by the *allTasks* selector.

The Appendix shows the implementation using the CF model of the remaining evolution steps of the application as introduced in sections 2.4 to 2.6.

4. Evaluation

In this section we evaluate the work presented in this paper from two perspectives: first we discuss the design of the social security services system and its evolution, and then we discuss the main characteristics of the CF model.

In section 2.1, to implement the administrative system, 41 operations were defined in 8 classes. The evolution of the initial requirements introduced new concerns, such as multiple views on documents, re-directing calls to the workflow and logger objects and synchronization of tasks. These concerns could not be adequately separated from the implementation of the initial system. As a consequence, the corresponding operations had to be redefined each time a new concern was introduced. Moreover, certain concerns had a crosscutting characteristic. For example, enforcing views was repeated in multiple operations in *ProtectedClaimDocument*, buffering documents had to be implemented for every task, and logging and locking operations had to be repeated for every class.

In our pilot study, redefinition was designed in 4 ways: edit & recompile, inheritance, aggregation and using the CF model. The edit & recompile approach required approximately 2 new classes and (re)-compilation of 123 operations in 30 classes. Inheritance gave a similar result: 2 new classes and 123 operation definitions in 30 subclasses were implemented. The aggregation approach required about 166 operation definitions in 32 classes. The CF approach required approximately 6 filterinterface and 5 superimposition declarations, 8 filter condition implementations, and 13 operations in 5 new classes. The main reason for the relatively low number of definitions of the CF approach is that it exploits the ability to separate concerns and express their crosscutting: this avoids many repeated definitions that the other approaches are forced to do.

We will now discuss the important characteristics of the CF model and evaluate it with respect to the example and some of the related work.

- *Declarative*: concerns are specified declaratively in a simple pattern matching language allowing various implementation strategies. For example, by using a dedicated compiler [18], filters of *ProtectedClaimDocument* can be in-lined in operations as conditions. Filters can also be implemented as run-time meta-level objects [14]. Approaches such as Adaptive Programming

[8] Note that the prefix "`self::`" is optional, shown here for illustrative purposes mainly.

[14] and AspectJ [13] typically adopt a general-purpose language to express concerns[9]. The actual realization of concerns is therefore incorporated in the concern specifications.

- *High-level semantics*: filter specifications use a common pattern matching language, and adopt filter types to add concern semantics. The semantics of filter types are well defined and highly expressive in the concern domains [5, 6]. As shown in the example, Error, Dispatch, Meta and Wait filters could effectively express multiple views, delegation, message reflection and synchronization, respectively. Since most related approaches adopt general-purpose programming languages for specifying concerns, in general little or no reasoning about the semantics of the concerns is possible.
- *Open ended*: new kind of concern semantics can be introduced as new filter types; for example, in [3] we introduced a filter type that allows for expressing real-time constraints on message executions. HyperJ [16] also supports an open-ended set of composition operators.
- *Strong encapsulation*: The implementation part is a strongly encapsulated object; superimposition of filter-interfaces, objects, methods and conditions is restricted to the interface level. Therefore superimposed concerns do not rely on the details of the implementation (even the implementation language is encapsulated). Several other, more fine-grained approaches, such as AspectJ [13], allow the crosscutting concerns (aspects) to 'break encapsulation', which makes the aspects less reusable and more vulnerable to implementation changes.
- *Modular*: the CF model unifies traditional object behavior with crosscutting behavior. The workflow and logging concern specifications in the CF implementation illustrate this: these are both modules that contain both 'object-like' and crosscutting behavior. In addition, concerns can crosscut any other concern; for instance the *logging* concern is superimposed upon all other concerns in our example. Many approaches that adopt the aspect-base level distinction (such as in AspectJ [13]) are not able to express this while retaining aspect modularity.
- *Composable*: the CF model supports composability at two levels: first, composition is supported because all filters are based on the same underlying model of message manipulation; second, filter expressions support composition of signatures, for example through the conditional-OR operator "," that we have shown in this paper. This is an important topic for all related work in this area, since the original developer of an abstraction may be unaware of (later) superimpositions, and thus only at instantiation time one can reason whether the composition of superimposed concerns actually works and makes sense. This may be difficult, if possible at all, for approaches that adopt general-purpose languages for expressing concern behavior.

---

[9] In those cases where the predefined semantics are insufficiently expressive, application-specific semantics can be implemented as a CF concern and applied through the use of the *Meta* filter [2]. In this case obviously part of the behavior is no longer specified declaratively.

The CF model has been implemented in the past on top of several languages: Smalltalk [14], C++ [10], and Java [18], but these implementations only supported crosscutting within a class, not between classes. The implementation of the latter (as an extension of the Java-based ComposeJ implementation) has just started as of submitting this article. Future work includes development of compiler support for the generation of both static, in-lined, code as well as dynamic, run-time representations, and research into predicting the composability of crosscutting concerns.

Acknowledgement

This work has been supported by the European IST Project 1999-14191 EASYCOMP and the AMIDST project of the 'Telematica Instituut' (http://www.trc.nl).

References

1. M. Aksit, L. Bergmans & S. Vural. *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, Proceedings of ECOOP '92, LNCS 615, Springer-Verlag, 1992, pp. 372-395
2. M. Aksit, K. Wakita, J. Bosch, L. Bergmans & A. Yonezawa. *Abstracting Object-Interactions Using Composition-Filters*, In *Object-based Distributed Processing*, R. Guerraoui, O. Nierstrasz & M. Riveill (eds), LNCS 791, Springer-Verlag, 1993, pp 152-184
3. M. Aksit, J. Bosch, W. v.d. Sterren & L. Bergmans. *Real-Time Specification Inheritance Anomalies and Real-Time Filters*, Proceedings of ECOOP '94, LNCS 821, Springer Verlag, July 1994, pp. 386-407
4. M. Aksit and L. Bergmans, *Guidelines for Identifying Obstacles when Composing Distributed Systems from Component*s, to be published in *Software Architectures and Component Technology: The State of the Art in Research and Practice*, M. Aksit (Ed.), Kluwer Academic Publishers, 2001
5. L. Bergmans. *Composing Concurrent Objects*, Ph.D. thesis, University of Twente, The Netherlands, 1994
6. L. Bergmans & M. Aksit, *Composing Synchronization and Real-Time Constraints*, Journal of Parallel and Distributed Programming, September 1996
7. L. Bergmans & M. Aksit, *Constructing Reusable Components with Multiple Concerns Using Composition Filters*, to be published in *Software Architectures and Component Technology: The State of the Art in Research and Practice*, M. Aksit (Ed.), Kluwer Academic Publishers, 2001
8. J. Ferber, *Computational Reflection in Class Based Object-Oriented Languages*, OOPSLA'89 Conference Proceedings, 1989
9. E. Gamma, R. Helm, R. Johnson & J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

10. M. Glandrup, *Extending C++ using the concepts of Composition Filters*, MSc. thesis, Dept. of Computer Science, University of Twente, 1995
11. N. de Greef, *Object-Oriented System Development*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, 1991.
12. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, *Aspect-Oriented Programming*. In proceedings of ECOOP '97, Springer-Verlag LNCS 1241. June 1997.
13. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. Griswold, *An Overview of AspectJ*, whitepaper at http://aspectj.org, 2001
14. P. Koopmans, *On the Design and Realization of the Sina Compiler*, MSc. thesis, Dept. of Computer Science, University of Twente, 1995
15. M. Mezini & K. Lieberherr, *Adaptive Plug-and-Play Components for Evolutionary Software Development*, OOPSLA '98, October 1998
16. H. Ossher & P. Tarr, *Multi-Dimensional Separation Ofconcerns And The Hyperspace Approach*, to be published in *Software Architectures and Component Technology: The State of the Art in Research and Practice*, M. Aksit (Ed.), Kluwer Academic Publishers, 2001
17. P. Wegner, *Dimensions of Object-Based Language Design*, OOPSLA '87, pp. 168-182
18. J. C. Wichman, *The Development of a Preprocessor to Facilitate Composition Filters in the Java Language*, MSc. thesis, Dept. of Computer Science, University of Twente, 1999

Appendix: Modeling the Evolution Steps 3-5 with Composition Filters

This section shows the implementation of the evolution scenario using the CF model. For space reasons, we will explain only the interesting concepts..

Evolution 3: Adding Document Queues

The requirement was to avoid processing multiple documents simultaneously. This is specified by the concern *MutEx* in Figure 10. *MutEx* is an *abstract* concern because it has an empty superimposition part. This concern defines a filter of type *Wait* that buffers all messages if there is an active thread within the concern instance.

```
concern MutEx begin  // implements the mutual exclusion synchronization concern
   filterinterface mutExSync begin
      conditions
         NoActiveThreads;
      inputfilters
         buffer : Wait = { NoActiveThreads=>* };
   end filterinterface mutExSync;

   superimposition begin  // not imposed anywhere; abstract concern
   end superimposition;

   implementation in Java;
      class MutExSupport{
         Boolean NoActiveThreads() { … };
      }
   end implementation;
end concern MutEx ;

concern ConcurrentDocumentProcessing begin // allows concurrency without interference
   superimposition begin
      conditions
         WorkFlowEngine::allTasks <- MutEx::NoActiveThreads;
      filterinterfaces
         WorkFlowEngine::allTasks <- MutEx::mutExSync;
   end superimposition;
end concern ConcurrentDocumentProcessing;
```

Figure 10. Specification of the concerns MutEx and ConcurrentDocumentProcessing.

To apply this abstract concern, the concern *ConcurrentDocumentProcessing* defines the superimposition of the *MutEx* concern upon the task classes. The selector *allTasks* is reused from the *WorkFlowEngine* concern to avoid redundant selector definitions. The condition *NoActiveThreads* (this is declared by the filterinterface) is superimposed to the same tasks as well.

Evolution 4: Adding Logging

The *Logging* concern consists of *NotifyLogger*, which is superimposed upon all concerns except *Logging*. Logging is implemented by sending all received messages as objects to the global object *logger*, using a *Meta* filter. The *Logging* concern creates an internal Boolean object *logOn* for every instance, which is used to enable or disable the logging of messages. More details of the implementation are shown in Figure 11. Note that logging is also supported for the methods of the *WorkFlowEngine* concern.

```
concern Logging begin // introduces centralized logger
   filterinterface notifyLogger begin // this part declares the crosscutting code
      externals
         logger : Logging; // *declare* a shared instance of this concern
      internals
         logOn : boolean; // created when the filterinterface is imposed
      methods
         loggingOn();  // turn logging for this object on
         logginOff();  // turn logging for this object off
         log(Message);  // declared here for typing purposes only
      conditions
         LoggingEnabled;
      inputfilters
         logMessages : Meta = { LoggingEnabled=>[*]logger.log };
         dispLogMethods : Dispatch = { loggingOn, loggingOff };
   end filterinterface notifyLogger;

   filterinterface logger begin //defines the interface of the logger object itself
      methods
         log(Message);
         // various methods for information retrieval from the log
      inputfilters
         disp : Dispatch = { inner.* };  // accept all methods implemented by myself
   end filterinterface logger;

   superimposition begin
      selectors
         allConcerns = { *!=Logging };  //everything except instances of Logging
      conditions
         allConcerns <- LoggingEnabled;
      filterinterfaces
         allConcerns <- notifyLogger;
         self <- logger;
   end superimposition;

   implementation in Java;
      class LoggerClass {
         boolean LoggingEnabled() { return logOn };
         void loggingOn()  { logOn:=true; };
         void loggingOff()  { logOn:=false; };
         void log(Message mess) { … }; // get information from message and store
      }
   end implementation;
end concern Logging;
```

Figure 11. Specification of the Logging *concern.*

Evolution 5: Adding Locking

The final example shows (in Figure 12) how the locking concern can be added to other concerns. It defines a generic (and abstract) concern *Locking*, which can implement per-instance locking by superimposing a Boolean object *lockState*, two methods *lock* and *unlock*, the condition *UnLocked* and two inputfilters upon all concerns. The filters are respectively a *Wait* filter that blocks all messages except the *unlock* message when the concern is in the locked state, and a *Dispatch* filter that is required to make the methods *lock* and *unlock* available on the interface of the concerns.

```
concern Locking begin      // implements the locking synchronization concern
   filterinterface lockBehavior begin
      internals
         lockState : boolean;  // state is true when locked
      methods
         lock();
         unlock();
      conditions
         UnLocked;
      inputfilters
         lockAll : Wait = { unlock, UnLocked=>* };
         disp : Dispatch = { lock, unlock };
   end filterinterface lockBehavior;

   superimposition begin // this is an abstract concern
   end superimposition;

   implementation in Java;
      class LockingSupport{
         void lock() { lockstate:=true };
         void unlock() { lockstate:=false };
         boolean UnLocked() { return !lockstate; };
      }
   end implementation;
end concern Locking;

concern WorkflowLocking begin     // applies the locking concern to the workflow appl.
   superimposition begin
      selectors
         applObjects = {*:Document, *:TaskProcessor};  // the ':' means that the
         selector includes all instances of derived concerns of the named concern
      methods
         applObjects <- {Locking::lock, Locking::unlock};  // impose these methods
      conditions
         applObjects <- Locking::UnLocked;  // impose the UnLocked condition
      filterinterfaces
         applObjects <- Locking::lockBehavior;
   end superimposition;
end concern WorkflowLocking;
```

Figure 12. Specification of the generic Locking *and the specific* WorkFlowLocking *concerns.*