CHAPTER 5

Principles and Design Rationale of Composition Filters

Lodewijk Bergmans and Mehmet Aksit

A wide range of aspect-oriented programming languages has appeared in the past years [7]. Current research on future generation AOP languages is addressing issues like flexibility, expressive power and safety. We think that it is important to understand the motivations and design decisions of the first generation AOP languages. The composition filters model [1, 7, 12] is one example of such a first-generation AOP language. The goal of this chapter is two-fold: first, it aims at explaining the principles of composition filters, in particular its aspect-oriented composition capabilities. Second, we aim to provide insight into the motivations and design rationale decisions behind the composition filters model.

5.1. INTRODUCTION

During the last several years, many aspect-oriented languages have been proposed, including such representative examples [17] as Adaptive Programming [27], Hyperspaces [30], AspectJ [24] and Composition Filters (CF's) [11]. The idea of CF's dates back to as early as 1986. As such, it is among the earliest, if not the first aspect-oriented language. Like other approaches, the CF model has evolved. This chapter presents the contemporary CF model, illustrating how it can address certain modeling problems and providing insight into its motivations and design rationale.

The structure of this chapter is as follows: in Section 5.1.1 we introduce the background and objectives of the CF model. Section 5.2 introduces an example, which is used to illustrate the issue of composing and reusing multiple concerns in object-oriented programs when requirements evolve. In Section 5.3, the CF model is presented as an approach to address the identified problems. This section introduces the CF model, focusing on its application to concerns that crosscut within an object. Section 5.4 extends this discussion by explaining how the CF model can address crosscutting over multiple objects. Finally, Section 5.5 evaluates the CF model and presents our conclusions.

5.1.1. Background and Aims of the Composition Filters Model: Finding the Right Abstractions.

The CF model has originated from the *Sina* language, which was first published in 1988 [1]. The concepts and ideas of Sina have since evolved, with the main objective to improve the composability characteristics of object-oriented and, ultimately, aspect-oriented programming languages.

We use the term *composability* to refer to the ability to define a new program entity—with a behavior as required—as the construction of two or more program entities. We distinguish two key elements of composability. The first element refers to the mechanisms (or *composition operators*) used to compose software units (objects, aspects, etc.). Typical composition operators are inheritance, aggregation and weaving mechanisms. The second key element refers to the properties or restrictions imposed on software units to make them safe for composition. For example, well-defined interfaces and declarative join point models may contribute to safe composition: these can be used for early detection of problems such as references to non-existent program elements and naming conflicts. The challenge is to find the right balance between the expressiveness of composition mechanisms and the restrictions imposed on software units. Addressing this challenge has been the main focus of the work on composition filters.

A fundamental design decision of the CF model is to distinguish two kinds of abstractions: (class-like) *concerns* and *filters*. Briefly, a concern is the unit for defining the primary behavior, while a filter are used to extend or enhance concerns so that (crosscutting) properties can be represented more effectively.

The main objectives of the composition filters model are summarized below. Later in this chapter, we discuss how these objectives are addressed.

Composability. Support composition (of the behavior) of modules into new modules with the desired behavior.

Evolvability. Extend existing (object-oriented) programming models in a modular way, instead of replacing or adapting them.

Robustness. Support the creation of correct programs through appropriate language abstractions that avoid common programming mistakes and enable verification of certain quality properties.

Implementation-independence. Allow multiple implementations of the same behavior, including both static and dynamic implementations, tradeoffs between time and space efficiencies, and execution on different platforms.

Dynamics. Support dynamic adaptation of structure and behavior to meet changes in requirements or context, without compromising the above objectives.

5.2. EXAMPLE: SOCIAL SECURITY SERVICES

We first present a simple example to illustrate the issue of composing and reusing multiple concerns in object-oriented programs when the requirements evolve. The example is a simplified version of the pilot study conducted in [11]. Due to the evolving business context, the initial software has undergone a series of modifications. We use a change scenario to explain the concepts and the application of the CF approach.

5.2.1. An Overview of the Application

The context of the example is a (Dutch) government-funded agency that is responsible for the implementation of disability insurance laws. As illustrated in Figure 5-1, the agency implements five tasks. Task `RequestHandler` creates a

case for clients. Cases are represented as documents. `RequestDispatcher` implements the initial evaluation and distribution of the requests to the necessary tasks. A request can be dispatched to `MedicalCheck`, `Payment` and/or `Approval`. `MedicalCheck` is responsible for evaluating a client's degree of disability. `Payment` is responsible for issuing bank orders. `Approval` is the (management) task of approving the proposed decisions, such as rewarding or rejecting a claim. Once a request is forwarded by `RequestDispatcher`, the further processing and routing of documents between tasks is a responsibility of the participating tasks. We would like to emphasize that Figure 5-1 only shows a small part of the actual system [11].

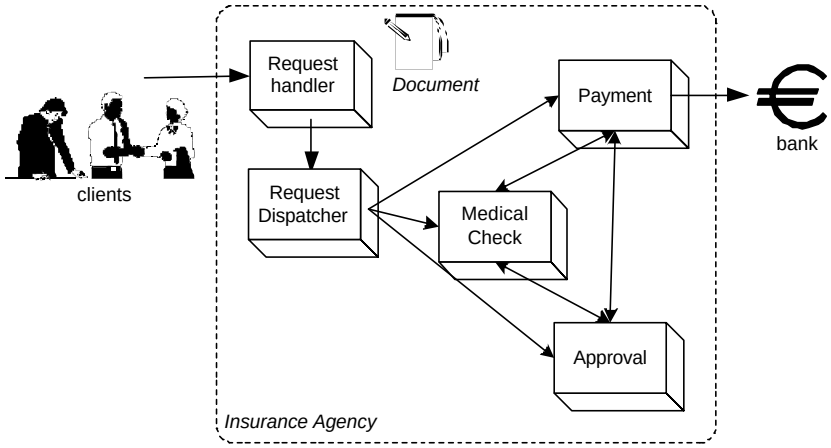

Figure 5-1. Tasks in the example system.

5.2.2. The Software System

The implementation of the system is based on a set of *tasks* and a number of *documents*. For now, assume that tasks and documents are implemented using classes. Each client request results in the creation of a document instance. De-

pending on the document type and input from the client, the document is edited and sent to the appropriate tasks (objects). Each involved task processes the document according to its specific needs.

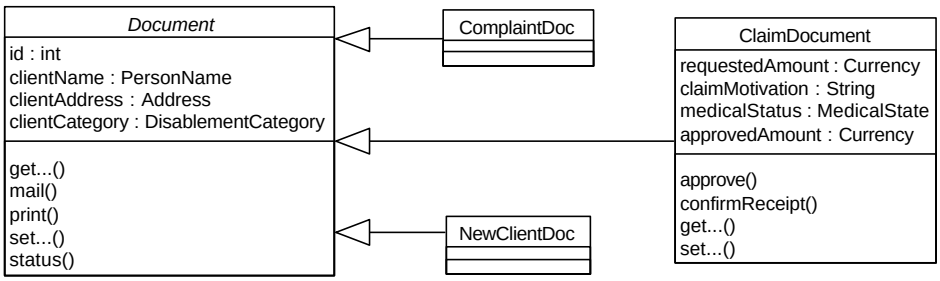

Figure 5-2. Part of the class hierarchy of documents that represent client requests.

As shown in Figure 5-2, class `Document` is the *abstract* root class of all document types. Every document inherits its attributes and methods (a number of getters and setters, and several additional methods, e.g. for accessing the status of the document, or for printing and mailing it). Class `Document` has several subclasses. For example, `ClaimDocument` is used to represent the clients' claims. Its attributes are typically written—through the appropriate methods—by various tasks as the document is processed.

As shown in Figure 5-3, the interface `TaskProcessor` declares the basic methods for all tasks. The method `process()` accepts a document as an argument and starts the task processing for the document; this might be a partial or fully automated process. In the cases where human interaction is required, an 'editor' UI tool is opened that presents the document and offers a task-specific interface to perform the manual part of processing. This is handled by the method `startEditor()`. Typically the last action of a task consists of forwarding the document to the next task(s); this is handled by the method `forward()`. The actual selection of all permitted tasks in the given state of the system has

been factored out into the method `selectTask()`, the method `forward()` has the responsibility of choosing one of the permitted tasks.

These methods must be defined by classes that implement this interface. Figure 5-3 shows some classes for different tasks. Each of these may inherit (task-specific) methods from different superclasses; this is not shown in the figure.

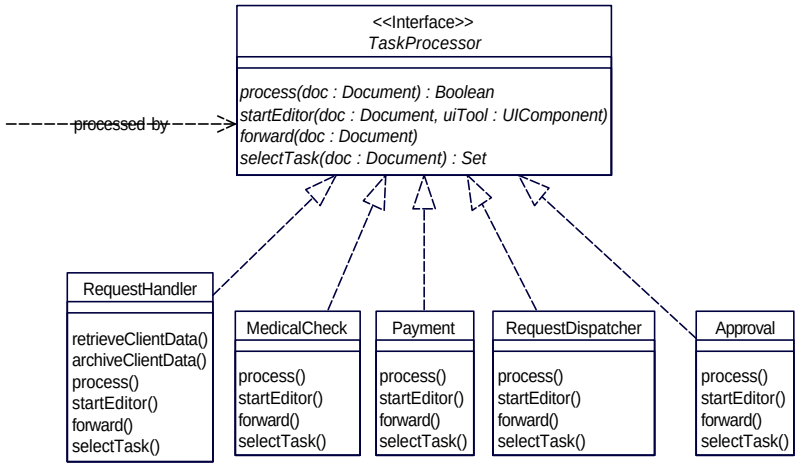

Figure 5-3. Class hierarchy for tasks.

`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 can then perform actions specific for `RequestHandler`. When the task is completed, the clerk selects a subsequent task—from the permitted ones—and forwards the document. This causes the invocation of method `process()` on the next task, with the document passed as an argument.

In this system, new activities are introduced by creating a new structural document class. (In the actual system from which this example has been derived, there were approximately 30 different document types.)

5.3. INTRA-OBJECT CROSSCUTTING WITH COMPOSITION FILTERS

In this section, we begin by briefly explaining the CF object model, focusing on concerns that crosscut *within* an object. We discuss the basic mechanisms of the CF model using the example that was introduced in Section 5.2. This chapter presents a conceptual model of composition filters, explained in an operational way. Actual implementations may vary substantially, as we discuss in Section 5.5.1.

5.3.1. Concern Instance = Object + Filters

The CF model is a modular extension to the conventional *object-based* model [34] used by programming languages such as Java, C++ and Smalltalk, as well as component models such as .NET, CORBA and Enterprise JavaBeans. The core concept of this extension is the enhancement of conventional objects by manipulating all sent and received messages. This allows expressing many different behavioral enhancements, since in an object-based system all externally visible behavior of an object is manifest in the messages it sends and receives. Figure 5-4 visualizes this extension by "abstracting" the *implemen-*

tation object with a layer[3] that contains filters for manipulating sent and received messages. These filters are grouped into subcomponents called *filter modules*. Filter modules are the units of reuse and instantiation of filter behavior. In addition to the specification of filters, the filter modules may provide some execution context for the filters.

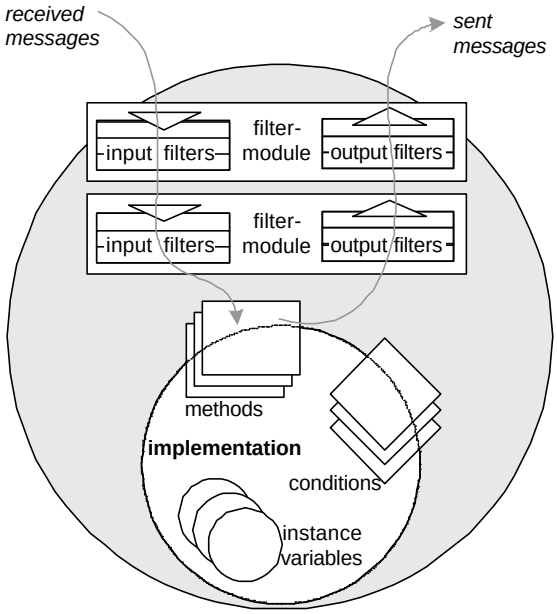

Figure 5-4. Simplified representation of concern instances with filters.

The *filters* define enhancements to the behavior of objects. Each filter specifies a particular inspection and manipulation of messages. *Input filters*

---

[3] Note that this is different from straightforward object wrapping; e.g. problems such as object schizophrenia [31] are avoided because in the CF case there is only one object with a single identity.

and *output filters* can manipulate messages that are respectively received and sent by an object. After the composition of filter modules and filters, received messages must pass through the input filters, and sent messages through the output filters[4].

The enhanced object, which we refer to as the *implementation object*, may be defined in any object-based language, given the availability of proper tool support for that language. The main requirement is that the object offers an interface of available methods. Two types of methods are distinguished: *regular methods* and *condition methods* (*conditions* for short). Regular methods implement the functional behavior of the object. They may be invoked through messages, if the filters of the object allow this. Conditions must implement side-effect free Boolean expressions that typically provide information about the state of the object.

Conditions serve three purposes:

1. They offer an abstraction of the state of the implementation object, allowing filters to consider only relevant states.
2. They allow filters to remain independent of the implementation details of the implementation object. This has the additional benefit of making the filters more reusable.
3. Conditions can be reused by multiple filter (modules) and concerns.

---

[4] More precisely, both filter modules and filters are composed together using dedicated composition operators. In this chapter we assume that both filter modules, and filters within those modules, are composed with a fixed order that depends on the declaration order of various elements.

In summary, conditions enforce the separation of the state abstraction and the message filtering concerns.

5.3.2. Evolution Step 1: Protecting Documents

In the initial system, each task could access all the attributes of a document. A request dispatcher clerk, for instance, could accidentally edit the medical data properties. To overcome this problem, the second release placed restrictions on the execution of messages. Within a given document-processing phase, only the appropriate tasks should have access to the interface methods of the corresponding document. A possible implementation for this is to test the role of the sender of the received message[5] before executing the corresponding method, throwing an exception if the method was invoked by an inappropriate object.

In the evolution steps that follow, we reuse all the functionality that has been implemented so far, incrementally introducing the new requirements through modular extensions.

With conventional objects, there are two primary extension alternatives: subclassing an existing document class or aggregating an instance of an existing document class, and reusing its behavior internally by sending ('forwarding') messages to it. Both of these patterns require overriding many methods of the reused class. These methods must implement the verification of the identity of the sender object, in addition to invoking the behavior of the original method. We refer to [11] for an extensive discussion on advantages and disadvantages of inheritance-based and aggregation-based composition.

---

[5] Depending on the implementation language, it may be non-trivial to identify the sender of a message and its role.

In the actual pilot project, the number of required method redefinitions was substantially high, due to the quantity of document types and the number of methods for each document type that had to be protected.

5.3.3. A Composition Filters Solution

We describe filter specifications in more detail using the class `ProtectedClaimDocument` example as described in the previous section. The code for `ProtectedClaimDocument` is shown in Figure 5-5. Using inheritance, `ProtectedClaimDocument` extends the existing class `ClaimDocument` with a protection mechanism to avoid inappropriate objects to invoke certain methods. We refer to this mechanism as *multiple views*.

Class `ProtectedClaimDocument` consists of a filter module, `DocumentWithViews` (lines 2-17), and an implementation part, which is defined as a Java class named `ProtectedClaimDocumentImpl` (lines 18-27). In principle, any object-based language can be used to realize the implementation part. In this example, the implementation defines the conditions and methods that are newly introduced by class `ProtectedClaimDocument`.

The filter module `DocumentWithViews` contains four parts: The first is the declaration of `internals` (lines 3-4); these are internal objects, encapsulated by the filter model. A new instance of an internal is created for each instantiation of the filter module that declares it. In this example, an instance of `ClaimDocument` named `document` is created for each instantiation of the filter module. A filter module can also declare `externals`; these are references to instances created outside the filter module and concern, and are used for representing shared state. Typically, the reference consists of a name that is bound according to lexical scoping rules.

The second part refers to the `conditions` (lines 5-6). Conditions serve to abstract the state of the implementation object. In this case, five conditions are

declared. It is possible to reuse conditions declared elsewhere by using the syntax "<instance name>.<condition name>", where <instance name> must be a valid internal or external object. The form "<instance name>.*" can be used to 'import'[6] all conditions from an instance. Possible name conflicts are resolved by selecting the first occurrence of a condition name based on the order of declaration.

The third part of the filter module refers to the lines 7-9, where the methods that are used within the filter expressions are declared, including the types of arguments and return values. Line 8 shows the declaration of method activeTask(), which takes no arguments and returns an instance of type String. For the purpose of language independence, the adopted notation follows the UML conventions [29] where appropriate. Line 9 shows the use of a wildcard '*' as a shorthand notation to declare all the methods in the signature of class *ClaimDocument*[7]. Again, possible name conflicts are resolved by selecting the first method according to the order of declaration.

The fourth and the last part of this example filter module is the specification of the *input filters* in lines 10-16. Two filters are declared in this part; *protection*, and *inh*. The first filter handles the multiple views and the second specifies the inheritance relation from the 'previous version', i.e. from *ClaimDocument*. We will go into more detail about the way filters work in the next subsection.

---

[6] This is semantically close to the notion of inheritance of conditions.

[7] This shorthand notation can be seen as a compromise between explicitly declaring all the methods on one hand, and convenience and open-endedness on the other hand. A particular property of the wildcard is that extensions to the interface of the reused class (in the current example ClaimDocument) will be automatically available to the reusing concern (ProtectedClaimDocument).

```
(1)  concern ProtectedClaimDocument begin
(2)  filtermodule DocumentWithViews begin
(3)    internals
(4)      document: ClaimDocument;
(5)    conditions
(6)      Payment; MedChck; ReqHndlr; ReqDisp; Approval;
(7)    methods
(8)      activeTask():String;
(9)      document.*;  //all methods on interface of ClaimDocument
(10)   inputfilters
(11)     protection: Error = {
(12)       Payment=>{setApprovedAmount, getApprovedAmount},
(13)       MedChck=>{setMedStatus, getMedStatus, getClaimMotivation},
(14)       …  //etc. for the other views
(15)       True=>{mail, print, status, getClientName, getCategory} };
(16)     inh : Dispatch = { inner.* , document.* };
(17) end filtermodule DocumentWithViews;

(18) implementation in Java  //for example
(19)   class ProtectedClaimDocumentImpl {
(20)     boolean Payment() { return this.activeTask()!="Payment" };
(21)     boolean MedChk() { … };
(22)     boolean ReqHndlr() { … };
(23)     boolean ReqDisp()  { … };
(24)     boolean Approval() { … };
(25)     String activeTask()  { … };
(26)     }
(27) end implementation
(28) end concern ProtectedClaimDocument;
```

Figure 5-5 Implementation of *ProtectedClaimDocument* in ConcernJ

All the named instances, conditions and methods within filter specifications must always be declared within the filter module. The resulting *declarative completeness* (as in HyperJ [30]) improves the ability to do modular and incremental reasoning about correctness of the program. In addition, when instantiating a filter module, it is straightforward to verify that all the declared

entities within the filter module can actually be bound to concrete implementations.

5.3.4. Message Processing

We explain the process of *message filtering* with the aid of Figure 5-6. The description focuses on input filters, but output filters work in exactly the same manner. In Figure 5-6, three filters, A, *B*, and *C*, are shown. We assume sequential composition of these three filters. Each filter has a *filter type*, and a *filter pattern*. The filter type determines how to handle the messages after they have been matched against the filter pattern. The filter pattern is a simple, declarative expression to match and modify messages. Typically, messages travel sequentially along the filters until they are dispatched. Dispatching here means either to start the execution of a local method, or to delegate the message to another object.

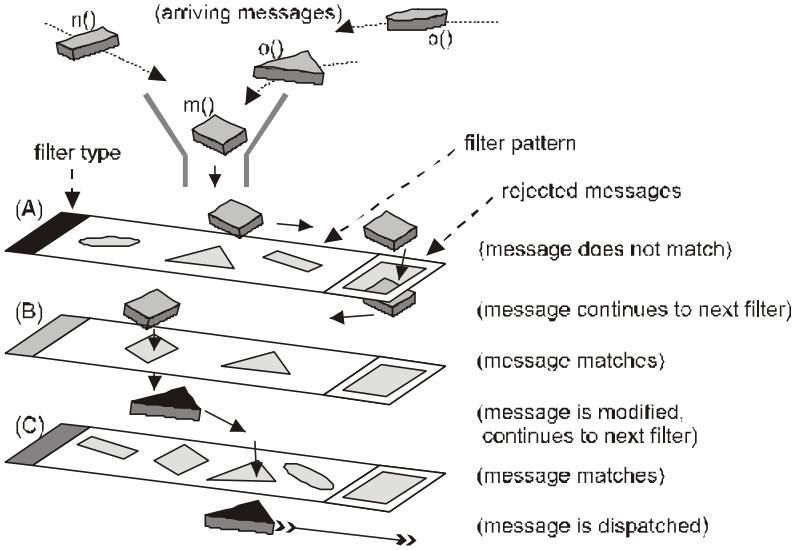

Figure 5-6 An intuitive schema of message filtering.

Figure 5-6 illustrates how a message[8] is rejected by the first filter (*A*), continues to the subsequent filter (*B*), where it immediately matches, is modified consequently, and then continues to the last filter (*C*). In the example, at the last filter, the message matches and is then subject to a dispatch.

Each filter can either accept or reject a message. The semantics associated with acceptance or rejection depend on the type of the filter. Typically, these are manipulation (modification) of the message, or execution of certain actions. 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) [3].

Substitute. Is used to modify (substitute) certain properties of messages explicitly.

Error. If the filter rejects the message, it raises an exception, otherwise the message continues to the next filter in the set [3].

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 [7, 9].

Meta. If the message is accepted, the message is reified, and sent as a parameter of a new message to a named object, otherwise the

---

[8] Messages are first reified, i.e. a first-class representation is created. Composition filters thus *conceptually* apply a form of message reflection [18], but note that actual implementations may 'optimize away' the reification.

message just continues to the next filter. The object that receives the message can observe and manipulate the reified message, and reactivate its execution [4].

We will see examples of the application of various filter types throughout the remainder of this chapter. Although filters can also be defined by the programmer, we do not discuss this in this chapter.

A composition filter specification corresponds to the creation of an instance of a filter type; for example, in Figure 5-5, *ProtectedClaimDocument* declares the following filter in line 15:

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

This expression declares a filter instance with name *inh* of filter type *Dispatch*, which is initialized with the filter pattern between the curly brackets. The filter pattern consists of a number of *filter elements*, connected with composition operators. In this particular case, the two filter elements are "`inner.*`" and "`document.*`". A filter element is mainly used for matching messages. In addition a filter element may modify certain parts of messages. Evaluation of a filter element always yields at least a Boolean result indicating whether the message actually matched the filter element.

We only discuss one filter element composition operator: the sequence operator ',' . The semantics of this operator are similar to a conditional OR; when the filter element on the left hand matches, the whole expression is satisfied, and no further filter elements should be considered. However, if the filter element on the left-hand side does not match, the filter element on the right-hand side will be evaluated, and so on, until either a filter element matches, or all filter elements have been evaluated. The total expression always yields a

Boolean result, i.e. *true* if the message did *match* one of the filter elements, *false* if it could not match any of them—a so-called *reject*)[9].

In the example shown above, the message matches the first filter element if the selector of the received message is within the signature of `inner`. This is the case when a corresponding method has been declared by the implementation object itself. If so, the `target` property of the message is replaced by a reference to the `inner` object. If the selector of the message is not in the signature of inner, the message is matched with the second filter element, "`document.*`". This is successful if the selector of the message is in the signature of class `ClaimDocument`. If the message does match, it has reached the end of the filter elements, yielding a `reject`.

After this matching process, an action is performed, based on the filter type, the result of the matching process (i.e. `accept` or `reject`), and the potentially modified message. In the above example, either this means the dispatch of the message to the new target, or the original, unchanged message simply continues to the subsequent filter. Dispatching by a `Dispatch` filter means one of three things:

1. If the target of the message is `inner`, this implies the execution of a method of the inner (implementation) object with the name that equals the selector of the message.
2. If the target of the message has been declared as an external object, dispatching is equivalent to (true) delegation [26]

---

[9] For an impression of the possible alternative composition operators, consider for example operators based on pure ORs or ANDs, where multiple matches might lead to multiple dispatch of the same received messages, either sequential or in parallel.

of the message to the target object. This means that the message is forwarded to the target, where the essential difference with message invocation is that the server pseudovariable (usually referred to as 'self', in C++ and Java 'this'), still refers to the original receiver of the message invocation, and not to the new target. In the literature, this property is also described as "allowing the ancestor to be part of the extended identity of the delegating object" [34]. A key feature of delegation is that it allows multiple instances to share (reuse) both the behavior (i.e. methods) *and* state (value) of an instance.

3. If the target of the message has been declared as an `internal object`, dispatching is again equivalent to true delegation, but the intuitive meaning is different: because each instance has its own copy of a declared internal object, delegating to an internal object is equivalent to inheriting from the class of the internal object. This is because both the behavior and the data structure (but not the actual values) of the superclass are reused.

This is exemplified by the dispatch filter in the above example. This filter accepts and executes all received messages that are declared and implemented by the inner (implementation) object (in lines 19-24 of Figure 5-5). All other messages that are in the signature of internal object `document` (i.e. available on its interface), as defined by concern `ClaimDocument`, match at the second part of the filter, and are thus dispatched (delegated) to the `document` object. Remaining messages that are in neither of the two signatures continue

to the next filter; if there is none, this yields a run-time error ("message not understood")[10].

We now consider the first filter in this example implementation, the filter named `protection` with filter type `Error`. The semantics of filter type `Error` are that it does nothing when a message is accepted, and raises an exception[11] when the message is rejected. The filter contains several elements, separated by the ',' sequence composition operator:

```
protection : Error = {
  Payment => {setApprovedAmount, getApprovedAmount},
  MedChck => {setMedStatus, getMedStatus,
              getClaimMotivation},
  …  //etc. for the other views
  True => {mail, print, status, getClientName, getCategory}
};
```

Each of these elements has the form "`<condition> => {<list of messages>}`". Its semantics are that the expression on the right hand side of the characters '`=>`' is only evaluated if the condition on the left hand side evaluates to `true`.

The aim of this example is to ensure that only the relevant messages are allowed to pass, and all others should result in an exception. To achieve this, the above `Error` filter uses conditions (`Payment`, `MedChk`, ...) that evaluate to true if the invocation was sent by respectively the tasks `Payment`, `MedicalCheck`, etc. In combination with the enable operator '`=>`', the `protection` filter only allows messages `setApprovedAmount()` and `getApprovedAmount()` if the condition `Payment`

---

[10] Run-time errors can be avoided through static type checking, with some reduction of flexibility.

[11] It is possible to add the type of exception as a parameter to the filter type, for example "`protection: Error(AuthorizationException)=...`".

is true. This means the sender of the message was a payment task. The filter expresses such constraints for all the different tasks, ending with a list of messages that are always acceptable, hence these are associated with the condition `true`.

5.3.5. Intra-Object Crosscutting

The above example of adding views exemplifies so-called *intra-object crosscutting*: each view constraint applies to a set of messages. For example, the `MedChk` condition is used to ensure authorized access for the messages `setMedStatus()`, `getMedStatus()`, and `getClaimMotivation()`. Instead of re-implementing the restriction in each of the corresponding methods, a filter defines this crosscutting constraint in a modular way.

However, the range of the crosscut specification is limited: although a filter expression may refer to messages that are inherited from, or are delegated to other concerns, only messages on the interface of a single concern are considered. In the next section, we show how to extend the application of composition filters in a broader (crosscutting) scope.

5.4. INTER-OBJECT CROSSCUTTING

From a software engineering perspective, the distinction between intra-object and inter-object crosscutting is more fundamental than just the expressiveness of the pointcut mechanism. Typically, software engineers consider classes or concerns as the unit of development and change. In addition, all specifications that are part of a single concern specification are assumed potentially interdependent and likely to be developed and evolve mutually, typically by one or a few closely co-operating software engineers. In accordance with this view,

software engineers should design filters as an integral part of the concerns[12]. This is a sharp contrast with inter-object crosscutting concerns, which are assumed to be developed independently, typically at a different time and potentially by different software engineers. As a result, the interfaces between the crosscutting concerns and the concerns that are affected by them is substantially more critical. The CF model reflects this distinction by introducing a different ('higher-level') mechanism for inter-object crosscutting. Its basic concept is to use groups of filters and related definitions, called *filter modules*, that are composed with concerns through the so-called *superimposition*[13] mechanism. The *superimposition* specifications describe the locations within the program where concern behavior is to be added in the form of filter modules.

5.4.1. Evolution Step 2: Adding Workflow Management

To explain the mechanism of superimposition, we introduce a second extension to our running example. In the design of the example so far, each clerk (i.e. human user) has to choose which task is to be executed next for a given

---

[12] This may seem in contradiction with the idea that filters are a modular extension to the object-oriented model, but as we explain in the side bar "Common Misconceptions about Composition Filters," the fact that the language model is an extension does not mean that applications should be designed by first designing objects and adding the interface part afterwards.

[13] Please note that our notion of superimposition bears resemblance to, but is truly distinct from the technique of superimposition as proposed by Bougé and Francez [14] and Katz [22]. Conceptually, superimposition as proposed by Bosch [13], is very close, but it refers to instantiation-time composition and does not include any support for crosscutting.

document. Accordingly, the document is forwarded to the appropriate task object. To enforce a better-managed business process, we introduce a new concern, called `WorkFlowEngine`.

Based on a workflow specification, the concern `WorkFlowEngine` is responsible for implementing the task selection process. In the current version, task selection is implemented within the method `forward()`. This method further calls on the method `selectTask()`, which returns a task selected among a set of alternatives. The method `selectTask` is implemented specifically for each `TaskProcessor` concern.

The concern `WorkFlowEngine` declares the attribute `workFlowSpec`, which represents the process to be enforced. This concern also implements the method `choose()`, which returns the task to be forwarded based on the workflow specification and the set of alternatives available.

The method `forward()` first calls `selectTask()` of `WorkFlowEngine`, followed by an invocation on `choose()`. Adding workflow management to the system in an object-oriented implementation would require redefinition of method `forward()` for all task classes. The method `forward()` cannot be implemented by a superclass, since every task implements this method in a specific manner. An aspect-oriented implementation is a preferable solution.

We use the above example to illustrate how crosscutting can be expressed using the CF model. As shown in Figure 5-7, this concern consists of four parts: (1) a (crosscutting) filtermodule named `UseWorkFlowEngine` that ensures that all relevant concerns in the application actually use the engine for determining the next tasks, (2) a (shared) filtermodule named `Engine` that im-

plements the behavior of the workflow engine itself, (3) a `superimposition` specification, and (4) the implementation of the necessary functionality.

```
(1)  concern WorkFlowEngine begin        //introduces global workflow control
(2)    filtermodule UseWorkFlowEngine begin      //declares crosscutting code
(3)      externals
(4)        wfEngine : WorkFlowEngine;          //*declare* a shared instance
(5)      methods                          //declare the –intercepted- messages
(6)        Set selectTask(Document);
(7)        workflow(Message);
(8)      inputfilters
(9)        redirect : Meta = { [selectTask]wfEngine.workflow };
(10)   end filtermodule useWorkFlowEngine;

(11)   filtermodule Engine begin       //defines interface of workflow engine
(12)     methods
(13)       workflow(Message);
(14)       choose(Document, TaskProcessor, Set) : TaskProcessor;
(15)       setWorkFlow(WorkFlow);
(16)     inputfilters
(17)       disp : Dispatch = { inner.* };            //accept all my methods
(18)   end filtermodule engine;

(19)   superimposition begin        //defines actual crosscutting composition
(20)     selectors               //queries set of all instances in the system
(21)       allTasks = {*=RequestHandler, *=MedicalCheck, *=Payment,
                       *=RequestDispatcher, *=Approval };
(22)     filtermodules
(23)       self <- Engine;
(24)       allTasks <- UseWorkFlowEngine;
(25)   end superimposition;
```

```
(26)  implementation in Java;
(27)    class WorkFlowEngine {
(28)      WorkFlow workFlowRepr;
(29)      void workflow(Message mess {
(30)        Document doc = mess.getArg(1);
(31)        TaskProcessor curTask = mess.target();
(32)        Set alternatives = mess.send();
(33)        mess.return( choose(doc, curTask, alternatives) );
(34)     };
(35)     TaskProcessor choose(Document, TaskProcessor, Set) { … };
(36)      void setWorkFlow(WorkFlow wf) { … };
(37)    }
(38)  end implementation;
(39) end concern WorkFlowEngine;
```

Figure 5-7 Specification of `WorkflowEngine`, illustrating a crosscutting CF concern.

The filtermodule `UseWorkFlowEngine` defines a filter of type `Meta`, which intercepts the calls on the method `selectTask()` and sends them in reified[15] form to the external object `wfEngine` as the argument of a message `workflow()`. This filtermodule represents the crosscutting behavior that must be superimposed upon all the `TaskProcessor` concern instances. In this case, the crosscutting behavior consists mostly of connecting the various task instances to the central workflow engine.

The filtermodule `Engine` and the implementation part together implement the workflow engine. In addition to some methods for accessing and manipulating the workflow representation (here only `setWorkFlow()` is shown), this filter module defines the method `workflow()`. This method `selectTask()` first determines the next task that should handle the document, and then modifies the

---

[15] A reified form is a representation of the message as an object (an instance of the *Message* concern), from which information about the message such as target, selector, arguments and sender can be retrieved and modified.

corresponding argument of the message object, and finally *fires* the message so that the message 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*, abstractions of all the locations that designate a specific crosscut. Selectors are defined as queries over the instance space, expressed using OCL (Object Constraint Language), which is part of the UML specification [29]. The concern `WorkFlowEngine` defines a single selector named `allTasks`. This selector repeatedly uses the expression "`*=<ConcernName>`" to specify all objects that are instances of the various classes that represent tasks. This is in fact an abbreviated form, as explained in the side bar "Using OCL to Select Join Points". The `selectors` part can be followed by a number of sections that can specify respectively which objects, conditions, methods, and filter modules are superimposed on locations designated by selectors. In this example the filter module `Engine` is superimposed upon `self`. This means that instances of `WorkFlowEngine` include an instance of the filter module `Engine`. In addition, the filtermodule `useWorkFlowEngine` [16] is superimposed on all the instances defined by the selector `allTasks`.

Using OCL to select join points

There have been several motivations for using OCL for selecting join points, i.e. as a pointcut language:

Firstly, we were looking for a pointcut language that was language-independent. Secondly, we assumed that the threshold of using a standardized

[16] Names on the right hand side can be prefixed with the concern name followed by a double semi-colon. Otherwise the prefix "`self::`" is assumed.

language would be much lower, in particular with many software engineers already familiar with it. Thirdly, although we had a number of concrete usage cases that we wanted to express, we were concerned with the evolution of the pointcut language: we expected that as we would apply the language to different domains and larger applications, new needs for certain selection expressions and operators might appear. Introducing those as incremental changes to the language would be non-trivial and would likely lead to an ill-formed language. Relying on a well thought-out language is likely to offer a smoother evolution path. Finally, OCL has some attractive properties: it has been designed to be both formal and readable. OCL expressions are free of side-effects, which is an important property for our purposes as well. In addition, the intention of OCL is to express statements about programs and the state of programs, which is very close to our goal of expressing a query upon (objects of) a program.

On the other side, it should be noted that adopting OCL for specifying selectors is also a compromise of sorts; it uses OCL as a query language, whereas it has been designed to express constraints. Alternatively, we could have adopted a dedicated query language such as SQL. One further drawback of OCL is that it can easily lead to lengthy and fairly complex expressions, even for relatively straightforward and frequent usage. However, by applying standard OCL shorthand notations and by using a little bit of syntactic sugar, complex OCL expressions can be simplified considerably.

For example, consider the selection of all instances of a number of explicitly listed concerns (for brevity, assume three concerns named respectively A, B and C). This can be expressed in OCL as follows:

```
System->select(ins|ins.oclIsTypeOf(A))
->union( System->select(ins|ins.oclIsTypeOf(B)) )
   ->union( System->select(ins|ins.oclIsTypeOf(C) )
```

This expression is the union of three different sets: it selects from *System* (i.e. the set of all instances in the system) all those instances that are respectively of type A, B, and C, and results in the union of those three sets. We can rewrite this expression as follows, still in pure OCL:

```
⇔ { System->select(oclIsTypeOf(A)),
      System->select(oclIsTypeOf(B)),
      System->select(oclIsTypeOf(C))  }
```

As a further simplification, we introduce some syntactic sugar; we replace *System* with '*', since this pseudo variable is used very frequently, and we introduce an additional shorthand notation: the common expression "`->select(oclIsTypeOf(<expr>))`" can be replaced with "`=<expr>`", hence our expression above can be rewritten as:

```
⇔ { *=A, *=B, *=C }
```

Similarly, "`->select(oclIsKindOf(<expr>))`" can be replaced by "`:<expr>`". The expression "`X.oclIsKindOf(Y)`" is true if *X* inherits directly, or indirectly, from *Y*.

The side bar "Unification of Implementation Class, Filter Modules and Superimposition into Concerns" discusses the effects of combining filter modules, the superimposition clause and the implementation clause within a single concern and the ramifications of omitting some of these parts.

Unification of classes, filter modules and superimposition into concerns

The composition filters model adopts a single abstraction as the major module concept; the *concern* abstraction. In this side bar, we discuss some of the ramifications of this design decision.

A concern abstraction consists of three optional parts: (1) the filter module specifications, (2) the superimposition specification, and (3) the implementation of the behavior. This is illustrated in the Figure 5-8, especially by the concern on the left side of the picture:

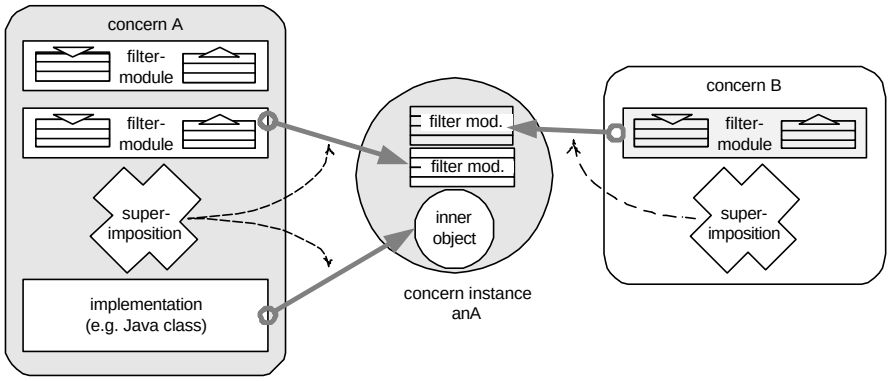

Figure 5-8 The elements of concerns and their mapping to concern instances

The figure shows on the left side a *concern specification* for concern *A*, consisting of two filter modules, a superimposition specification and an implementation. On the right hand side of the figure another concern specification, *B*, is depicted that consists (only) of a filter module and a superimposition specification. In the middle an example *concern instance*, *anA* is shown, which is an instance of concern *A*, including the implementation defined by *A* and one filter module superimposed by *A*, as well as an additional filter module that has been superimposed by *B*.

As a result of the unification into concerns, the model does not distinguish between 'aspect' and 'base' module abstractions, which has several advantages: (1) It makes the model 'cleaner': no different syntax and/or semantic rules need to be defined for the various possible configurations. (2) This struc-

ture allows for modeling base level functionality and state together with crosscutting behavior into a single module abstraction (rather than two separate modules for respectively the 'aspect' and the 'base' level abstractions).

This offers a degree of symmetry [21] between concerns that avoids design compromises and allows for more stable designs where the decisions what to model as base and what as aspect abstractions can be avoided. However, depending on the particular implementation, the (structure of the) run-time model may be different; e.g. when the superimposition specifications are resolved statically, there are no equivalent abstraction at run time.

Not all three parts of the concern specification are obligatory. Table 5-1 outlines the various possible combinations of leaving out one or more parts, and summarizes their intuitive meaning:

Table 5-1 The parts of a concern and their mapping to concern instances

filter module(s)	*super-imposition*	*implemen-tation*	*explanation*
×	×	✓	c.f. conventional class
✓	only to self	✓	c.f. conventional composition filters class
✓	✓	✓	crosscutting concern with implementation
✓	✓	×	'pure' crosscutting concern, no implementation
✓	×	×	c.f. abstract advice without crosscutting definition
×	✓	×	superimposition only (of reused filter specs.)
×	✓	✓	CF class or aspect with only reused filter specs.

5.4.2. Evolution Step 3: Adding Logging

One important concern of the workflow system is monitoring the process, detecting the bottlenecks and rescheduling and/or reallocating resources when necessary. Which methods need to be monitored is difficult to determine a

priori (i.e. at compile time) since it depends on the purpose of monitoring. Therefore, all interactions among objects may potentially need logging. More precisely, logging the reception of messages by an instance is sufficient. For each instance, actual logging of the message receptions can be turned on or off at run time.

The definition of concern `Logging` in Figure 5-10 starts with the filtermodule `NotifyLogger`, which defines the crosscutting behavior needed to collect the information to be logged from all over the application. 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 5-10. Note that logging is also supported for the methods of the `WorkFlowEngine` concern, but that we exclude the instances of `Logging`, especially to avoid recursive logging of the `log()` messages.

The shared part of the logging functionality is defined by filtermodule `Logger`. This declares the method `log()`, which takes an instance of `Message` as an argument, and logs the information about the message. Typically, a range of methods for retrieving and/or displaying the logged information should be declared by `Logger` as well; we omit these for brevity. The filtermodule also contains a filter definition `disp` that ensures that these method(s) are made available on the interface of the `Logging` concern.

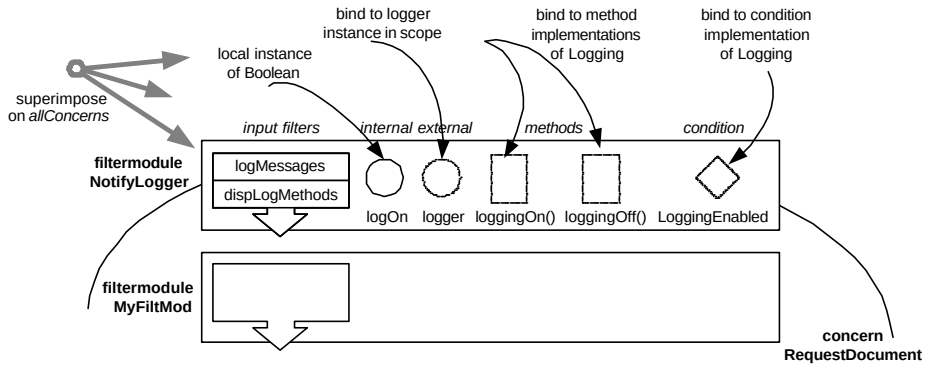

Figure 5-9 Illustration of superimposition and binding of filter module `NotifyLogger`.

The superimposition part of this example is interesting since it involves a slightly more complicated binding process. this is illustrated by Figure 5-9. The set of join points where logging must be added is defined by `allConcerns`. This selector uses an OCL expression to select all instances in the system, except for the instances of concern `Logging`. The last part of the superimposition clause defines the superimposition of the `Logger` filtermodule to the `Logging` concern itself, and the superimposition of the `NotifyLogger` to the join points defined by the `allConcerns` selector. The latter filtermodule declares an external global instance of `Logging` named `logger`, an internal `logOn`, which is created for each superimposed filtermodule, and several methods. The `loggingOn()` and `loggingOff()` methods must be available for execution in the context of the superimposed instance (because of the dispatch filter `dispLogMethods`), but they are declared by concern `Logging`. Therefore, the superimposition specification must explicitly bind the declared methods to the implementation within concern `Logging`, which is done in line 28. Similarly, the condition `LoggingEnabled` is declared in the filtermodule `NotifyLogger` (line 12). When this condition is evaluated in the `logMessages` filter in line 14, normally the condition would be

searched in the local context. Therefore, in line 30, the implementation of the `LoggingEnabled` condition from concern `Logging` is bound to all other concerns.

```
(1)  concern Logging begin                    //introduces centralized logger
(2)    filtermodule NotifyLogger begin       //declares the crosscutting code
(3)      externals
(4)        logger : Logging;      //declare a shared instance of this concern
(5)      internals
(6)        logOn : boolean;        //created when the filtermodule is imposed
(7)      methods
(8)        loggingOn();                     //turn logging for this object on
(9)        logginOff();                    //turn logging for this object off
(10)       log(Message);             //declared here for typing purposes only
(11)     conditions
(12)       LoggingEnabled;
(13)     inputfilters
(14)       logMessages : Meta = { LoggingEnabled=>[*]logger.log };
(15)       dispLogMethods : Dispatch = { loggingOn, loggingOff };
(16)   end filtermodule NotifyLogger;
(17)   filtermodule Logger begin         //defines interface of logger object
(18)     methods
(19)       log(Message);     //and methods for information retrieval from log
(20)     inputfilters
(21)       disp : Dispatch = { inner.* };         //accept all my own methods
(22)   end filtermodule Logger;
(23)   superimposition begin
(24)     selectors
(25)       allConcerns = { *->reject(oclIsTypeOf(Logging)) };
(26)              //selects all concern instances except instances of Logging
(27)     methods
(28)       allConcerns <- { loggingOn(), loggingOff() };       //bind methods
(29)     conditions
(30)        allConcerns <- LoggingEnabled;                  //bind condition
(31)     filtermodules
(32)       allConcerns <- NotifyLogger;     //superimpose NotifyLogger on all
(33)       self <- Logger;          //superimpose Logger filtermodule on self
(34)   end superimposition;
(35)   implementation in Java;
(36)     class LoggerClass {
(37)       boolean LoggingEnabled() { return logOn };
```

```
(38)       void loggingOn()  { logOn:=true; };
(39)       void loggingOff()  { logOn:=false; };
(40)       void log(Message mess) { … }; // collect information & store
(41)     }
(42)   end implementation;
(43) end concern Logging;
```

Figure 5-10 Specification of the *Logging* concern.

5.5. EVALUATION

We begin this section with a discussion of a variety of prototype implementations. In Section 5.5.2 we address the technique of inlining, since this is an important implementation technique that also demonstrates the ability to generate efficient code for (at least for a subset) composition filters. Finally, in Section 5.5.3 we evaluate the CF model based on the objectives that were presented in Section 5.1 and the example case of this paper, and we relate these issues to some other work in this area. In the side bar "Common Misconceptions about Composition Filters," we have listed a number of examples of what composition filters are *not*, or *do not*.

5.5.1. Implementations of the CF model

Throughout the years, a wide range of implementations of the composition filters model has been realized. These differ from each other in several respects:

The adopted base language. That is, the language that expresses the implementation part.

The level of integration with the base language. For example, whether the syntax of the interface part is adapted to the syntax of the base language.

Translation time. Ranging from a pure interpreter to a byte code interpreter to a compiler.

These implementations have been proof-of-concept research prototypes. We now present various implementations, focusing on their distinguishing characteristics. The aim of this presentation is to give an impression of the wide range of possible implementation strategies.

Sina. The *Sina* language [34] was the first implementation of composition filters, which were at that time referred to as interface predicates [1]. It translated the Sina language into a byte code representation that was interpreted by a dedicated virtual machine. The latter was built on top of Smalltalk.

Sina/st. *Sina/st* [25] was another implementation of the Sina language on top of Smalltalk, but this version supported the full composition filters mechanism. The implementation was based on an interpreted, reflective, run-time, first-class model of filters and messages.

CFIST. *CFIST* [16] aimed at the integration of Smalltalk programming and composition filters, both at the language level and at the tool level. A key concept of this prototype was that it kept the Smalltalk class definitions completely separate and independent of the composition filters part.

C++/CF. The *C++/CF* language [19] investigated the integration of composition filters with C++. It was based on the explicit reification of message invocations (implemented through macros).

ComposeJ. *ComposeJ* [36] is an inlining compiler for an integrated Java/CF language.

ConcernJ. *ConcernJ* [32] is the implementation of composition filters including the superimposition mechanism, as a front-end to ComposeJ. It resolves the crosscutting and generates corresponding filter specifications for ComposeJ.

JCFF. An alternative, more pragmatic, approach has been taken in JCFF (*Java Composition Filters Framework*) [33], where composition filters are implemented in Java as a library. This has the advantage of allowing software engineers to remain completely in a familiar programming language.

Compose*.NET. Recently, we have initiated the design and development of Compose*.NET [29], which adds the capability of adding composition filters to .NET assemblies. Since the latter are a universal, object-oriented, representation of programs in any (one of many) programming language. This gives us the ability to add composition filters to any programming language that has been implemented by .NET.

Further, we would like to point out that composition filters seem to be well-suited to extend (other) component models such as CORBA. This has been investigated for example in [15].

5.5.2. Inlining Composition Filters

As a another example of the benefits of the implementation-independent model and to address the issue of performance, we present an example of *inlining* of filters as a compilation technique to achieve efficient execution of composition filters. The principle of filter inlining is to generate a custom version of

the filter code for a specific message selector, thereby typically eliding the bulk of the filter code. This custom version is then inserted at the start of the method with the corresponding selector.

To demonstrate the effect of inlining, we show how the example filter code in Figure 5-11 is translated by the ComposeJ tool:

```
inputfilters
  verifyUserView : Error = {
    userView()=>{putOriginator, putReceiver,
                 putContent, getContent,
                 send, reply },
    true~>{putOriginator, putReceiver,
           putContent, getContent, send,
           reply }   };
  verifySystemView : Error = {
    systemView()=>{approve, putRoute, deliver},
    true~>{approve, putRoute, deliver}   };
  dis: Dispatch = { inner.*, mail.* };
```

Figure 5-11 Example filter code from an email application [44].

This example consists of three filters that every message should pass. It uses two conditions that must be evaluated at run-time. Further, the last line involves signature matching. Obviously, literally processing the full filter specifications for every message is, even when generating efficient code, quite expensive.

However, a look at the generated code that is inlined in `USVMail::reply()`, after a clean-up reveals that the amount of code to be executed for this message is quite limited:

```
if (!userView())
  throw new FilterException("Error filter exception");
return mail.reply();
```

For message `reply()`, this is the full code that is needed to execute the filters in Figure 5-11.

The ability to inline filters is simplified by the fact that the filter specifications are declarative and easy to analyze statically.

There are a few considerations, however:

- Not all the messages that an object receives may correspond to the execution of an inlined filter implementation. In those cases a potentially more elaborate execution of the whole set of filters is required.
- The code above does not show some overhead in our implementation, e.g. for the purpose of bookkeeping pseudovariables.
- Code inlining generally results in increased code size.

An important lesson to be learned from this example, as well as the range of examples in the previous subsection, is that general conclusions about the efficiency of the implementation of filters must be drawn with great care.

Common Misconceptions about Composition Filters

Composition Filters have been around for a substantial amount of time. Among the various feedback we have received, we have discovered some common misconceptions. By explicitly addressing these, we hope to achieve that the readers of this chapter gain an improved understanding of these specific issues.

Composition filters can only 'filter out' messages. The term filtering may suggest that the only purpose of filters is to selectively allow certain messages. In general, filters can observe and *manipulate* messages, as well as trigger certain actions.

Composition filters objects are equivalent to 'wrappers. Traditional object wrapping is similar to the composition filters approach, but it differs in at

least two important ways: first, the behavior of object wrappers is implemented by regular methods, whereas composition filters are tailor-made abstractions for composing the behavior of objects. Second, object wrapping suffers from some modeling problems such as object schizophrenia [31]. Composition filters, e.g. by the merit of the dispatching mechanism, do not suffer from these problems.

Composition filters are a tool for adapting existing classes. The fact that the implementation object is separate and can be an already implemented class, does not mean that this is the actual purpose. Although there are certainly useful applications in this area, it is important to realize that the filters are depending on the offered interface of methods and conditions. This interface must in general be designed before constructing the implementation and the filters. Hence, the approach of choice is to identify the filters along with other properties of classes during the design phase.

The filtering mechanism must be very inefficient. The declarative style of composition filters allows for various optimizations, in particular avoiding overhead that is irrelevant for specific messages. We refer to section 5.5.2 for more details about this topic. It should be noted that some mechanisms that are offered by filters, such as synchronization and message reflection, have inherent performance penalties that cannot be avoided.

There are only a few composition filter types. Most publications about composition filters, including this chapter, discuss only a limited set of filter types. The presented filter types are the result of our attempts to define filter types that are canonical models for some common aspects of software systems. However, there are no inherent restrictions on the number or behavior of filter types.

Composition Filters implementations must be multi-language. We have stressed that the CF model is language-independent; the filtering mechanism can be made to work with implementation parts expressed a variety lan-

guages. However, we make no claims about the support multi-language systems or for heterogeneous platforms: this is strictly depending on a particular implementation. Typically, existing implementations work with a single language/platform [16, 19, 25, 36]. Clearly, a multi-language implementation is possible, in particular by relying on a multi-language platform such as CORBA [15] or .NET [28].

Composition filters are strictly reactive. This would mean that filters can only be active when triggered by incoming messages. Although this is an important category (e.g. *Error*, *Substitute* and *Dispatch* filters), some other filters can be active without being triggered by incoming messages. For example, the *Wait* filter must repeatedly reconsider whether some of the blocked messages in the queue can be activated again. Similarly, the *RealTime* filter is responsible for continuously rescheduling threads such that deadlines are met.

5.5.3. Conclusion

In Section 5.1.1 we presented the objectives for the CF model: composability, evolability, robustness, implementation-independence and dynamics. In this section, we briefly discuss each of these objectives; to what extent they are met with the present model, relate to the presented example whenever applicable, and in some cases, we refer to related or future work.

Composability. The CF model supports composability at two levels: composition of filters and composition of concerns.
Composition of filters is supported because all filters are based on the same underlying model of message manipulation. Only specific

semantic dependencies between filters can cause interference problems. Second, filter expressions support the composition of behavior of concerns, most notably the signatures of concerns. The example evolution scenario that we used in this paper illustrates the stepwise composition of the final application from several concerns. For example, an instance of `ProtectedClaimDocument` after the final evolution is a composition of the `ClaimDocument` and `ProtectedClaimDocument` concerns, with the (superimposed) filtermodules `DocWithViews` and `NotifyLogger`. As a result, such an instance composes the redirect `Meta` filter (line 9 of Figure 5-7), the `protection` filter of type `Error` (lines 11-15 of Figure 5-5), and the `inh` filter of type `Dispatch` (in line 16 of Figure 5-5).

Crosscutting concerns require the composition of filter modules; this is essentially similar to the composition of (multiple) filters. Further, declarative completeness of filter modules makes it easier to check composability at compile time or instantiation time. Superimposition among crosscutting concerns is also possible, as demonstrated by the workflow and logging concern in our example.

Evolability. The CF model extends traditional object models with possibly crosscutting behavior. This is achieved by extending objects with filters (filter modules). This is possible for virtually any object or component model. The only possible issue is that inheritance of the implementation language and inheritance of the CF model are orthogonal: developers must take care to avoid implementation language inheritance relations between CF objects.

Robustness. A number of properties of the CF model contribute to robustness, including the overall language design, the chosen abstractions and their syntax and semantics. Verifying or even discuss-

ing robustness in general is quite hard; hence, we focus on the following three properties:

(1) *Encapsulation*; the implementation of a CF concern is strongly encapsulated; 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 approaches, such as AspectJ [24], allow the crosscutting concerns (aspects) to refer to, and depend on, implementation details of the base level abstractions. This makes the aspects less reusable and more vulnerable to implementation changes. Although researchers do not all agree whether the benefits of respecting encapsulation outweigh the limitations, it seems fair to state that encapsulation is beneficial for robustness.

(2) *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 their specific concern domains [7, 9]. As shown in the example, `Error`, `Dispatch` and `Meta` filters could effectively express multiple views, delegation and message reflection, respectively. Given the availability of suitable filter types, this enables the programmer to express his intents clearly and concisely.

(3) *Analyzable aspect description language*; most aspect-oriented approaches adopt general-purpose programming languages for specifying concern. In general, reasoning about the semantics of such concerns is hard to do (e.g. consider the undecidability problems of Turing-complete languages). The restricted (pattern match-

ing) language used to define filters has much more opportunities for automated analysis of correctness.

Implementation-independence. The CF model is largely independent of specific implementation techniques, programming language and platforms. This was demonstrated in Section 5.1 through the discussion of eight implementations of the composition filters model that are all substantially different in one or more respects. In particular, this is made possible by the declarativeness of filter specifications.

Dynamics. To a large extent, the implementation independence supports the dynamic adaptation of structure and behavior. Again, declarative specifications of filters and superimposition are helpful, but in particular, it must be possible to explain and implement the language through an operational model.

The CF model addresses the quality objectives as depicted in Section 5.2; for each objective, one or more specific properties of the CF model contributes to meeting the objective. Our future work will mainly focus on new verification techniques, new filter types, and robust CF development tools. In addition, we continue to explore ways to improve composability.

References

1. M. Aksit and A. Tripathi, "Data abstraction mechanisms in SINA/ST", Proceedings of OOPSLA'88, *ACM SIGPLAN Notices*, v.23 n.11, p.267-275, Nov. 1988.

2. M. Aksit, J.W. Dijkstra and A. Tripathi, "Atomic Delegation: Object-Oriented Transactions", *IEEE Software*, Vol. 8, No. 2, pp. 84-92, March 1991.

3. M. Aksit, L. Bergmans and 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.

4. M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa. "Abstracting Object-Interactions Using Composition-Filters", In *Object-based Distributed Processing*, R. Guerraoui, O. Nierstrasz and M. Riveill (eds), *LNCS* 791, Springer-Verlag, 1993, pp 152-184.

5. M. Aksit, J. Bosch, W. v.d. Sterren and L. Bergmans. "Real-Time Specification Inheritance Anomalies and Real-Time Filters", Proceedings of ECOOP '94, *LNCS* 821, Springer Verlag, July 1994, pp. 386-407.

6. M. Aksit and L. Bergmans, "Guidelines for Identifying Obstacles when Composing Distributed Systems from Components", in M. Aksit (Ed.), *Software Architectures and Component Technology: The State of the Art in Research and Practice*, Kluwer Academic Publishers, 2001.

7. *Aspect-Oriented Software Development–Tools* and *Languages*, http://aosd.net/tools.htm.

8. L. Bergmans, *Composing Concurrent Objects*, Ph.D. thesis, University of Twente, The Netherlands, 1994.

9. L. Bergmans and M. Aksit, "Composing Synchronization and Real-Time Constraints", *Journal of Parallel and Distributed Programming*, September 1996.

10. L. Bergmans and P. Cointe (eds.), "Workshop report of the ECOOP'96 workshop on Composability Issues in Object-Orientation", in M. Mühlhaüser (ed.), *Special Issues in Object-Oriented Programming*, pp. 53-124, dpunkt, Heidelberg, 1997.

11. L. Bergmans and M. Aksit, "Constructing Reusable Components with Multiple Concerns Using Composition Filters",in M. Aksit (Ed.), *Software Architectures and Component Technology: The State of the Art in Research and Practice*, Kluwer Academic Publishers, 2001 .

12. L. Bergmans and M. Aksit, "Composing Crosscutting Concerns Using Composition Filters", *Communications of the ACM*, Vol. 44, No. 10, pp. 51-57, October 2001.

13. J. Bosch, "Composition through Superimposition", in [10], pp. 94-101.

14. L. Bougé and N. Francez. "A Compositional Approach to Superimposition". In *ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 240–249, Jan 1988.

15. A. Burggraaf, *Solving Modelling Problems of CORBA using Composition Filters*, M.Sc. thesis, Department of Computer Science, University of Twente, The Netherlands, 1997.

16. W. van Dijk and J. Mordhorst, *CFIST—Composition Filters In SmallTalk*, graduation thesis, HIO Enschede, The Netherlands, 1995.

17. T. Elrad, R. Filman and A. Bader, "Aspect-Oriented Programming", special theme, *Communications of the ACM*, Vol. 44, No. 10, pp. 29-97, October 2001.

18. J. Ferber, "Computational Reflection in Class Based Object-Oriented Languages", *OOPSLA'89 Conference Proceedings*, 1989.

19. M. Glandrup, *Extending C++ using the Concepts of Composition Filters*, MSc. thesis, Dept. of Computer Science, University of Twente, 1995.

20. N. de Greef, *Object-Oriented System Development*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, 1991.

21. W. Harrisson, H. Ossher and P. Tarr, "Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition", position paper for the *AOSD'03 workshop on Software-Engineering Properties of Languages for Aspect Technologies*, (`www.daimi.au.dk/~eernst/splat03`), 2003.

22. S. Katz. "A Superimposition Control Construct for Distributed Systems". *ACM Trans. on Programming Languages and Systems*, 15:337–356, April 1993.

23. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, "Aspect-Oriented Programming". In *Proceedings of ECOOP '97*, LNCS 1241, Springer-Verlag, June 1997.

24. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, "An Overview of AspectJ", *Proceedings of ECOOP 2001*, LNCS 2072, Springer Verlag, 2001

25. P. Koopmans, *On the Design and Realization of the Sina Compiler*, MSc. thesis, Dept. of Computer Science, University of Twente, 1995.

26. H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems", In *Proceedings of OOPSLA '86*, SIGPLAN Notices 21(11) Nov. 1986, pp. 214-223.

27. M. Mezini and K. Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development", *Proceedings of OOPSLA '98*, October 1998.

28. C. Noguera, *Compose*; A Run-time for the .NET platform*, EMOOSE MSc. thesis, Vrije Universiteit Brussel, September 2003.

29. OMG, *OMG Unified Modeling Language Specification, version 1.5*, formal/03-03-01, March 2003.

30. H. Ossher and P. Tarr, "Multi-Dimensional Separation of Concerns and the Hyperspace Approach", in *Software Architectures and Component Technology: The State of the Art in Research and Practice*, M. Aksit (Ed.), Kluwer Academic Publishers, 2001.

31. M. Sakkinen. "Disciplined Inheritance". In *Proceedings ECOOP '89*, pages 39–56. Cambridge University Press, 1989.

32. P. Salinas, *Adding Systemic Crosscutting and Super-Imposition to Composition Filters*, EMOOSE MSc. thesis, Vrije Universiteit Brussel, August 2001.

33. L. Savarese, Aspect Oriented Software Development applied to Track Modeling for Command and Control, M.Sc. thesis, Dept. of Computer Science, University of Twente, expected, 2003.

34. A. Tripathi, E. Berge and M. Aksit, "An Implementation of the Object-Oriented Concurrent Programming Language Sina", *Software: Practice and Experience*, Vol. 19(3), pp 235-256, March 1989 .

35. P. Wegner, "Dimensions of Object-Based Language Design", *Proceedings of OOPSLA '87*, pp. 168-182.

36. 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.