C F I S T
Composition Filters in Smalltalk

by
Wietze van Dijk
John Mordhorst

Graduation thesis for the HIO Enschede

mentors:
Dr. ir. M. Aksit from University of Twente
P. Koopmans from University of Twente
Drs. J.H. Kerstholt from HIO Enschede
Ir. J. Meuleman from HIO Enschede

supervised by:
Drs. Th.F. de Ridder

University of Twente
Dept. of Computer Science
P.O. Box 217
7500 AE Enschede
The Netherlands

This document is prepared with TEX. The final output is printed on a QMS 1725 SLS PostScript laserprinter. All figures are created with xfig 3.1. The software of this graduation project is developed with Objectworks\Smalltalk in a UNIX environment.

UNIX is a trademark of Unix System Laboratories.
TEX is a trademark of American Mathematical Society.
Objectworks\Smalltalk is a trademark of ParcPlace Systems.
Smalltalk-80 is a trademark of Xerox Corporation.
xfig is originally copyrighted by Supoj Sutanthavibul
PostScript is a trademark of Adobe Systems Incorporated.

First printing
Copyright © 1995 by University of Twente

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the University of Twente.

Dedicated to our parents and to Janet,
who supported us during the whole study,
in storm or shine.

Contents

1 Problem statement and Background work **1**
1.1 Problem statement . . . 1
1.2 Approach . . . 2
1.3 Background and related work . . . 2
1.4 Thesis outline . . . 3

2 The Composition Filters Object Model **5**
2.1 Introduction to the composition filters object model . . . 6
2.1.1 Wait filter . . . 8
2.1.2 Meta filter . . . 8
2.2 The parts of the Composition Filters Object Model . . . 9
2.2.1 The implementation part . . . 9
2.2.2 The interface part . . . 10
2.3 The principles of Composition Filters . . . 11

3 Filters **13**
3.1 The Composition Filters Definition Language . . . 13
3.1.1 Class declaration and comment . . . 14
3.1.2 Internals declaration . . . 15
3.1.3 Externals declaration . . . 15
3.1.4 Conditions declaration . . . 16
3.1.5 Method declaration . . . 16
3.1.6 Inputfilters declaration . . . 17
3.1.7 Outputfilters declaration . . . 18
3.2 The syntax of filter elements . . . 19
3.2.1 Condition part . . . 19
3.2.2 Matching part . . . 20
3.2.3 Parameter part . . . 22
3.2.4 Example of a filter element . . . 22
3.3 Filters . . . 23
3.3.1 Dispatch filter . . . 24
3.3.2 Error filter . . . 24
3.3.3 Wait filter . . . 25
3.3.4 Meta filter . . . 25
3.3.5 Send filter . . . 27
3.3.6 Substitution filter . . . 27
3.4 Filter example . . . 28

4 Integrating composition filters in Smalltalk 29
4.1 Smalltalk introduction 29
4.1.1 Objects 30
4.1.2 Inheritance 30
4.1.3 Pseudo variables 31
4.1.4 First class message representation 31
4.1.5 Describing classes 31
4.1.6 Smalltalk messages handling 32
4.2 An example of using composition filters in Smalltalk 33
4.2.1 Multiple inheritance 33
4.3 Classes and objects with composition filters 36
4.3.1 Inheritance between Smalltalk and FIST classes 37
4.3.2 Message communication between objects 38
4.4 The Smalltalk extension 39
4.4.1 Message Manager 39
4.4.2 Input filters 40
4.4.3 Output filters 40
4.4.4 Messages sent to self and super 41
4.4.5 Object creation 41

5 Compiler 43
5.1 Semantic analysis 45
5.2 Code generation 46
5.3 Generating a condition control block 46
5.3.1 Condition part of a filter expression 48
5.3.2 Matching part of a filter expression 49
5.3.3 Parameter part of a filter expression 49

A Filter syntax diagrams 53

B Redefinition of the filter syntax 57

Preface

In a forest a fox bumps into a little rabbit, and says,
”Hi, junior, what are you up to?”
”I’m writing a dissertation on how rabbits eat foxes,” said the rabbit.
”Come now, friend rabbit, you know that’s impossible!”
”Well, follow me and I’ll show you.”
They both go into the rabbit’s dwelling and after a while
the rabbit emerges with a satisfied expression on his face.
Comes along a wolf. ”Hello, what are we doing these days?”
”I’m writing the second chapter of my thesis, on how rabbits devour wolves.”
”Are you crazy? Where is your academic honesty?”
”Come with me and I’ll show you.” As before, the rabbit comes
out with a satisfied look on his face and a diploma in his paw.
Finally, the camera pans into the rabbit’s cave and, as everybody
should have guessed by now, we see a mean-looking, huge lion sitting
next to some bloody and furry remnants of the wolf and the fox.

The moral: It’s not the contents of your thesis that are important –
it’s your PhD advisor that really counts.

Object Oriented Methods are a great improvement to the conventional methods for developing software. However, these object oriented methods do still have a number of lacks. Therefore the University of Twente developed the composition filters object model , a method that increases the power of the conventional object oriented methods. Composition filters manipulate messages sent to or sent by an object. The messages can –for example– be delayed, modified or redirected.

During the thirteen weeks of our graduation assignment for the HIO Enschede we worked on integrating composition filters in Smalltalk. This project is done in the TRESE — Twente Research & Education on Software Engineering — project of the University of Twente. In this document we give a description of the composition filters object model and the way to use composition filters within Smalltalk. It also describes how the model is integrated in Smalltalk.

Acknowledgements

We wish to thank the people who have been very helpful, with comments and hints about the composition filters object model in Smalltalk, especially dr. ir. Mehmet Aksit and Piet Koopmans from the University of Twente. We would also like to thank Lodewijk Bergmans, Maurice Glandrup, Coen Stuurman and the other members of the TRESE project at the University of Twente for their constructive comments during our discussions and their support. Furthermore thanks to Bonne van Dijk, Albert Hofkamp and Rick van Rein for their comment, help and ideas. We are greatful to drs. John H. Kerstholt and ir. Jan Meuleman from the HIO for guiding the whole project. Finally thanks to Drs. Theo F. de Ridder for supervising our graduation work.

Chapter 1

Problem statement and Background work

Hoare's Law of Large Problems:

Inside every large problem is a small problem struggling to get out.

In this chapter we describe the aim of our assignment, that is integrating the composition filters object model into a Smalltalk environment. Section 1.2 describes how we approach the problem. Work that is related to our assignment is described in section 1.3.

1.1 Problem statement

Programs are becoming larger and more complex. This makes it difficult to design and maintain the programs. Object Oriented methods and programming languages have been introduced to overcome these problems. However, there are still a number of important deficiencies. In a number of pilot projects, the TRESE group identified, conceptualized and defined an important category of modeling problems (see [Aksit and Bergmans, 1992]). The object oriented model as adopted in the current methods and languages fail in short in expressing some aspects of the application. Some of these aspects are:

- inheritance and delegation
- dynamic inheritance and delegation
- coordinated behavior
- reuse of synchronization
- reuse of real time specification

To overcome these deficiencies the composition filters concept has been introduced. The composition filters concept is a modular extension of the object oriented model.

The programming language Sina [1] — developed at the University of Twente — is a non commercial language based on Composition Filters, which has been introduced to demonstrate the composition filters concept. As stated above popular object oriented languages like Smalltalk [Goldberg and Robson, 1983] and C++ [Stroustrup, 1986] still have some problems. Extending those languages with the composition filteri concept would be a great benefit.

Therefore we put together Smalltalk and the composition filters concept. The result of this combination is called Smalltalk with filters. This "Smalltalk with filters" will be able to deal with an important class of modeling problems.

1.2 Approach

To extend Smalltalk with the composition filters concept, a syntax for the composition filters has to be defined in such a way that it is acceptable for Smalltalk. The language Sina already has a syntax for the specification of composition filters, called the interface part of a class. Our objective is to create a new syntax, using much of the Sina syntax. However, since the syntax has to be (re-)defined, we also could reconsider the composition filters concept for improvement, like unformity.

Our work must be seen as a pilot project. It is an illustration of the use of composition filters within Smalltalk. The prototype should convince the Smalltalk manufacturers of the benefit of the composition filters concept.

1.3 Background and related work

CFOM

This section describes work in related areas of our assignment. Advanced topics of the composition filters object model can be found in [Bergmans, 1994], [Aksit *et al.*, 1992], [Aksit *et al.*, 1993] and [Aksit *et al.*, 1994].

The Sina Language

Sina

Sina [2] is an object-oriented language and is the first language to adopt the composition filters object model. Early versions of the Sina language are described in [Aksit and Tripathi, 1988], [Aksit, 1989] and [Tripathi *et al.*, 1989]. Those versions implement a simple version of composition filters object model with one filter without conditions.

Many aspects from the Sina language can be used to implement the composition filters object model .

Sina in Smalltalk (Sina/st) *(Piet Koopmans)*

Sina/st

Piet Koopmans designed a Sina[2] version for Smalltalk. The Sina compiler is programmed in Smalltalk. This package, called *Sina/st*, was developed using Objectworks\Smalltalk and is placed on top of the Smalltalk system. It is an almost full implementation of Sina and comes with a compiler which translates Sina source code

[1] Named after the medieval philosopher, scientist and physician Ibni Sina (also known under the Latin name Avicenna). See also "The Sina Language" and "Sina in Smalltalk" in section 1.3, "Related work".

[2] More information about the Sina language can be found at the World Wide Web: *http://wwwtrese.cs.utwente.nl/sina/*

into Smalltalk, and a kernel which implements the various filter types and provides run-time support. Sina/st integrates with the Smalltalk system, in the respect that:

- one can use Smalltalk objects and classes in a Sina application, for example as an instance variable of a Sina class;
- one can look at the compiled Sina code, since every Sina class will be translated to a Smalltalk class;
- one can use Smalltalk's snapshot mechanism to save the image (environment) in which the Sina application is developed and continue with it at a later time.

The Sina/st sources are useful as an example of Smalltalk programming. Some of the ideas Koopmans used in Sina/st can be reused in our implementation.

C++ with Composition Filters *(Maurice Glandrup)*
At the same time we are extending Smalltalk with composition filters Maurice Glandrup is working on extending C++ using the concepts of composition filters.

Message manipulators *(Coen Stuurman)*
Coen Stuurman is involved with a composable message manipulator framework. This framework will be used to describe and develop filter types. The message manipulators will be capable of describing filters and filter expressions as a whole.

1.4 Thesis outline

This thesis describes the background of composition filters and the extension of Smalltalk with composition filtes. The thesis is divided in the following chapters.

First in chapter 2 the composition filters object model is explained. In chapter 3 we will show the syntax and semantic of a composition filter definition language. This language is based on the Sina syntax, but has some important changes making it more uniform. Chapter 4 will give a introduction to Smalltalk, then a example of how to use composition filters in Smalltalk and finally a description of the extension of Smalltalk with composition filters. Finally chapter 6 shows how to generate Smalltalk code for a composition filter definition.

Chapter 2

The Composition Filters Object Model

> The unique operations of the (human) brain are the result of natural selection operating through the filter of culture. They have suspended us between the two antipodal ideals of nature and machine, forest and city, the natural and the artifactual, relentlessly seeking, in the words of geographer Yi-Fu Tuan, an equilibrium not of this world.
> *Edward O. Wilson, "Biophilia"*

In this chapter we give a description of the composition filters object model. This was originally implemented in the Sina language. The composition filters object model extends the conventional object-oriented model by introducing input and output filters. When composition filters are defined for an object they intercept all messages received and sent by this object.

Section 2.1 gives an introduction to composition filters object model, comparing the model with photography. The composition filters object model consists of two major parts: the implementation part and the interface part. Each part will be described extensively in section 2.2. The implementation part, described in section 2.2.1, looks a lot like the conventional object oriented model. The next section shows the interface part, which is the extension of the composition filters object model. Finally section 2.3 shows the principles of the composition filters mechanism.

2.1 Introduction to the composition filters object model

It is desirable to keep the conventional object oriented model, so that programs written without composition filters do not need modification. The aim is to enhance the expressive power of the model for a given set of problems (like inheritance and delegation, coordinated behavior, real time aspects). The new model must be able to compose several enhancements together. In our case, the implementation environment for the conventional model is Smalltalk. The enhancement of the composition filters object model is applicable to other object oriented environments (like C++) as wel.

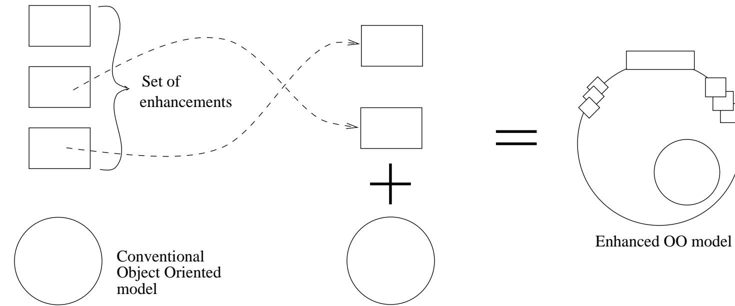


Figure 2.1: Composition of the enhanced model.

The extension of the model is comparable with a photo camera. Whenever the camera is not sufficient, an extension can be used to overcome the problem. The extensions on the camera can also be combined (see figure 2.2): A zoomlens is used to zoom in to the subject, the color filter is expected to correct the colors and the image intensifier makes it possible to take pictures at night. The composition filters object model works in the same way: objects (like the camera) can be extended with composition filters (such as the image intensifier for a camera). In this way we do not have to redesign objects (or the whole camera).

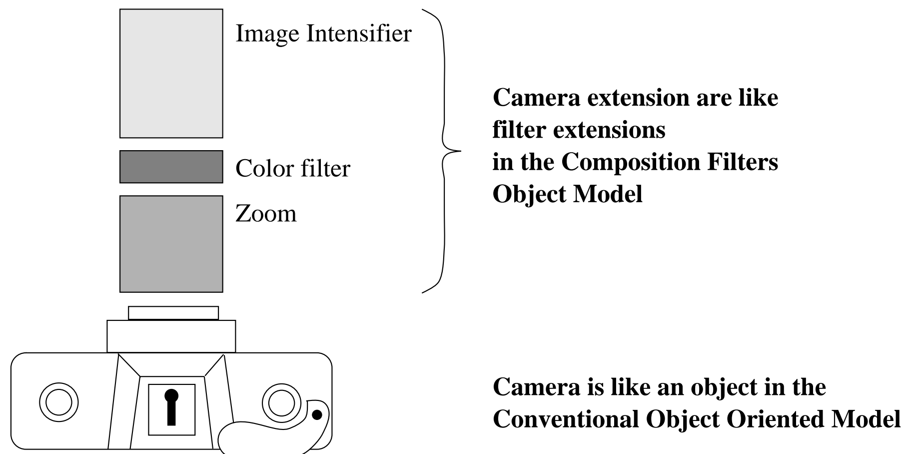
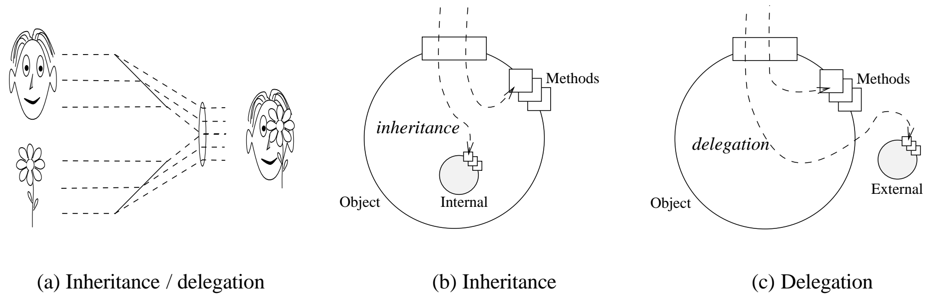


Figure 2.2: A photo camera with (combined) extensions.

The paper [Aksit and Bergmans, 1992] mentions the importancy of supporting inheritance and delegation. Inheritance and delegation can be seen in figure 2.3a as the objects woman and flower that are composed in one picture.

inheritance
delegation

Inheritance is a mechanism where a class inherits operations[1] from its superclass. In other words, it can use methods from that superclass. Delegation is a technique where an object can delegate requests to other objects, which are not part of the internal structure of the delegating object. In figure 2.3b-c is shown how inheritance and delegation are realized in the composition filters object model.

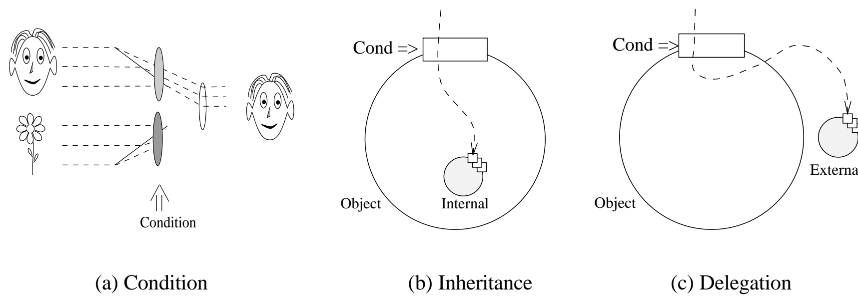


(a) Inheritance / delegation (b) Inheritance (c) Delegation

Figure 2.3: Inheritance and delegation

A photographer has to do with a lot of conditions (too much light, a flower in the foreground or not). Figure 2.4a shows how a condition can be used to make a picture of the person only.

In the object oriented models also a lot is dependent on conditions. Logical conditions can be used for dynamic inheritance and delegation. In figure 2.4b-c we can see how a condition affects the path of a message.



(a) Condition (b) Inheritance (c) Delegation

Figure 2.4: Dynamic inheritance and delegation

[1] In Smalltalk operations are called methods

2.1.1 Wait filter

When a picture has been recorded, at playback time the picture can be taken. In that way the photographer can delay taking the picture.

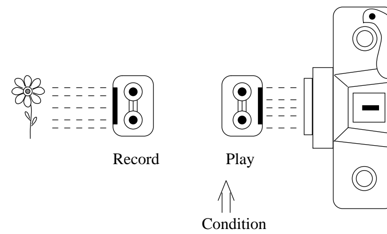


Figure 2.5: Take a delayed picture.

Messages between objects in an object oriented model have to wait too, this is called synchronisation. When a message has to wait due to a condition, the wait filter places the message in a wait queue.

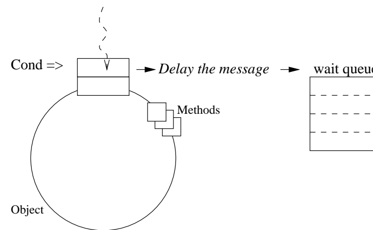
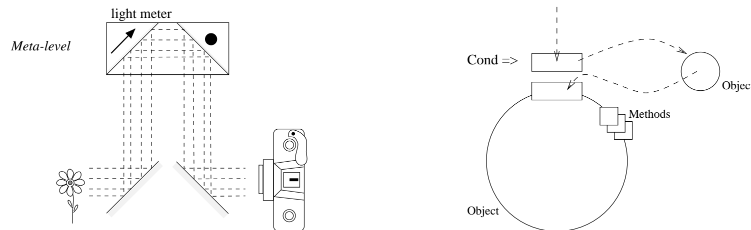


Figure 2.6: Delay a message.

2.1.2 Meta filter

One should be able to express meta level architecture such as constraints and coordinated behavior. Messages are converted to objects and whenever necessary converted back to executions. This is called message reification.

A photographer first checks the light intensity (with a lightmeter) before he takes a picture. This means that the data of the subject are first lifted to meta level (see figure 2.7a).



(a) A picture at two levels. (b) Meta filter.

Figure 2.7: Meta behavior.

In the composition filters object model it works in the same way as can be seen in figure 2.7b.

2.2 The parts of the Composition Filters Object Model

As is exemplified by figure 2.8. the composition filters object model can be divided into two parts. The first part, called the implementation part, holds the details of an object and can be seen as the –almost– conventional object model. Secondly the interface part includes the extensions made by the composition filters object model. We will now explain these parts in more detail.

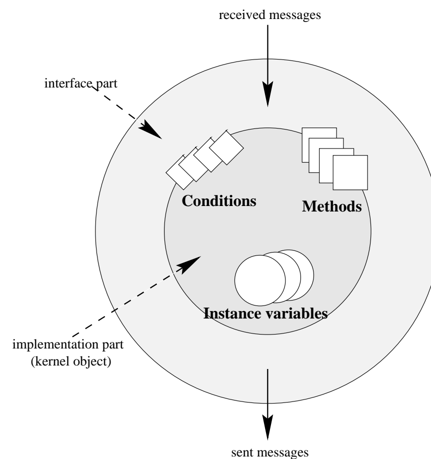


Figure 2.8: The two parts of the composition filters object model.

2.2.1 The implementation part

The implementation part, also known as the kernel object, refers to the state and the behavioral aspects of an object.
As stated in [Bergmans, 1994] the implementation part can be replaced without severe consequences by virtually any conventional object-based or object-oriented language, such as Smalltalk [Goldberg and Robson, 1983], C++ [Stroustrup, 1986], CLOS [DeMichiel and Gabriel, 1987], Self [Ungar and Smith, 1987] or actor languages [Agha, 1986].

We can see in figure 2.8 that the implementation part, also known as the kernel object, has three components, namely *instance variables*, *methods* and *conditions*. An instance variable is a local variable declaration that comprises the state of an object. *Instance variables* For example: *Color, Weight* and *Model-year* are instance variables of *Car* objects. A method is the implementation of an operation for a class The methods or operation *Methods* definitions of an object define the behavior of an object. *Accelerate* and *slowdown* are examples of methods of *Car* objects. When a method is invocated, a series of defined actions is performed. Conditions are defined to provide information about the current *Conditions* state of an object. They are actually a specific kind of method that will have no parameters and return a boolean value. Conditions however should not introduce side effects, they solely test states.

The actual definition of methods and conditions, together with instance variables are fully encapsulated within the kernel object. The method and condition declarations shown on the interface layer are the only connection between the object state and the world outside the object. In case of the composition filters object model this means the connection between the implementation part and the interface part.

✎ **Example 2.1** Boolean object

Figure 2.9 shows the implementation part of a boolean object. This object has an instance variable *truthValue* to hold the boolean state of the object. Furthermore there are two methods *beTrue* and *beFalse* that change the state of the object to true and false respectively. Finally, the two conditions *isTrue* and *isFalse* test the state of the boolean object.

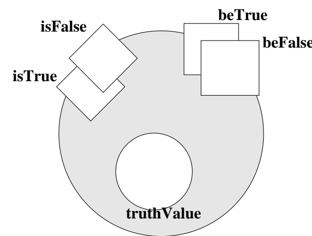


Figure 2.9: The kernel object of a Boolean .

2.2.2 The interface part

In this section we explain the interface part of the composition filters object model, which encloses to the implementation part.

The interface part introduces filters that deal with the messages that are received or sent by an object. Each message received or sent needs to pass a set of composition filters. These filters can change, delay, bounce or dispatch the message. For a detailed description of filters we refer to chapter 3, in which we describe their syntax and semantic. Note that messages sent to a conventional object –that is an object with no interface part– results in a normal execution of that method.

In the interface part, shown in figure 2.10, we can recognize the following components: *internals*, *externals*, *input filters* and *output filters*.

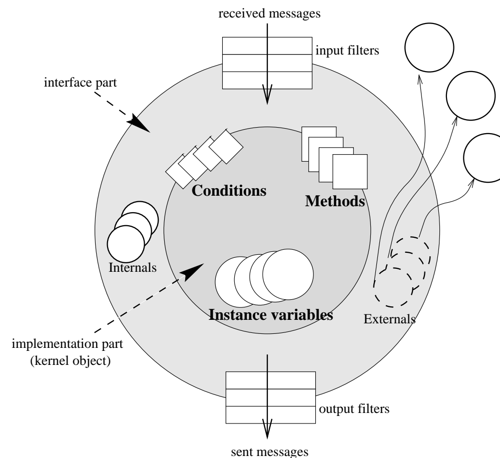


Figure 2.10: The components of the composition filter extension in the interface part.

Internals

Internals are fully encapsulated objects that are used to compose the behavior of the composition filters object. For example a superclass of a class can be defined as an internal. In this way the composition filters object model uses internals to simulate

inheritance. Internals are not the same as instance variables, instance variables represent the local data of the object.

Externals

External objects are objects that exist outside the composition filters object, such as global or shared objects. Externals are also used for state sharing. Similarly to internals, they are used to compose the behavior of the composition filters object, but do not result in an automatic object creation. In addition externals can be used to share states among objects.

Input and output filters

There are two groups of filters, input filters and output filters. The input filters handle messages that are received by the object. On the other hand, the output filters handle messages that are sent by the object. When a message is intercepted it is directed to the first filter of the input/output filter set. The message then passes all declared filters in order, until it is dispatched, sent or causes an error.

2.3 The principles of Composition Filters

The filtering mechanism of composition filters mechanism will be explained with the help of figure 2.11. A message arrives at the first filter (filter A). Each filter specifies a

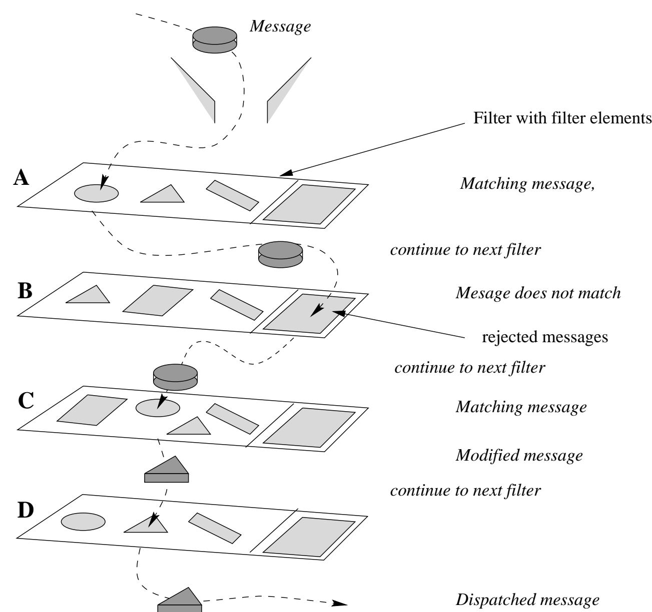


Figure 2.11: Explanation of the filter mechanism.

number of patterns to which the message should be compared. As soon as the message matches one of the patterns, the message is accepted. Depending on the filtertype, some action will be performed on the message. In the case of the example, the message will continue to the next filter. The message can be modified (see filter C). When the message is rejected (there is no matching pattern) the message is treated in a way that depends on the kind of filter (in filter B, the message continues to the next filter).

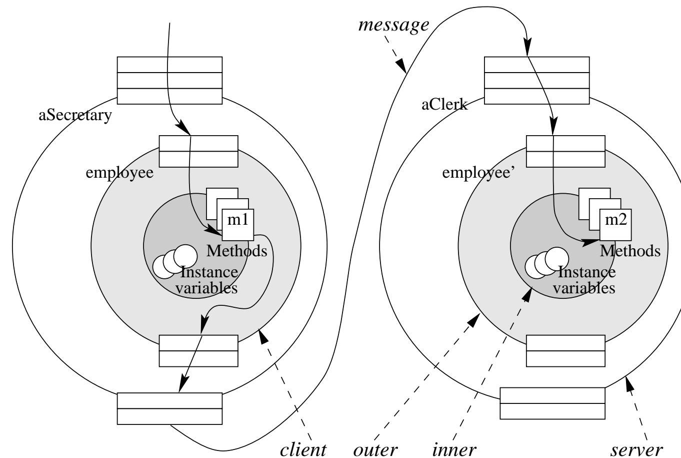


Figure 2.12: Illustration of the path a message can follow.

Figure 2.12 is an example of the path a message can follow. In this figure the following pseudo-variables are shown: *inner*, *outer*, *client*, *server* and *message*.

inner

- *inner* is the implementation part of the current object. A message that is sent to *inner* does not pass any filter. It refers to a local method;

outer

- *outer* is the current object including its interface part;

client

- *client* is the object that sent the message

server

- *server* is the original receiving object of the message.

message

- *message* is the message in reified form. This means that the message is represented as an object.

The message is sent to the object *aSecretary*, this object dispatches the message to its internal *employee*, which results in the execution of a method. This method sends a new message to the object *aClerk*, which –in its turn– dispatches the message to its internal *employee*. The *employee* part of *aClerk* executes a local method *m2*. At the moment of execution of method *m2*, the pseudo-variables *inner*, *outer*, *client* and *server* hold the following values:

- *inner*: the implementation part of *employee*;
- *outer*: the interface part of *employee*;
- *client*: *aSecretary*;
- *server*: *aClerk*;

The emphasized terms in the figure and the previous list yield at the moment of execution of the method *m2* of *employee* within *aClerk*.

Chapter 3

Filters

> The language provides a programmer with a set of conceptual tools;
> if these are inadequate for the task, they will simply be ignored.
> For example, seriously restricting the concept of a pointer
> simply forces the programmer to use a vector plus integer arithmetic
> to implement structures, pointer, etc. Good design and the absence
> of errors cannot be guaranteed by mere language features.
> *Bjarne Stroustrup, "The C++ Programming Language"*

In this chapter we will describe the Composition Filters Definition Language (CFDL). This language describes the composition filters object model interface part of a class. As with every language, it is important to define both the form (syntax) and the the meaning (semantics) of the language.
Section 3.1 describes CFDL. In section 3.2 we illustrate the syntax of the filter elements. The currently known composition filters are described in section 3.3. This chapter concludes with an example of a 'bounded stack', which shows all aspects.

3.1 The Composition Filters Definition Language

The Composition Filters Definition Language [1] is used to declare the interface part of a class. It can be defined as follows: a class name, a comment, a set internals and externals, a set of conditions and methods and a set of input and output filters. The CFDL does not however include programming language constructs, such as the source code of a method. Most of the syntax of the language is based on the Sina language.

[1] It must be stressed that the definition of the Composition Filters Definition Language has been a team effort with other members of the TRESE project.

The declaration of the interface part has the following components:

interfacepart →
  className
  commentPart ?
  internalPart ?
  externalPart ?
  conditionPart ?
  methodPart ?
  inputfilterPart ?
  outputfilterPart ?
  end ; ■

The interface part consists of a number of parts, which all are optional. The interfacepart is ended with keyword *end* and a semicolon.
Additionally, it is possible to add comment to each line using the C++ convention (so everything between '//' and 'end of line' is treated as comment).

Semantics
The interface part belongs to a class with the name *className*. It defines filters for this class specificly. The parts called after className, are described below.

3.1.1 Class declaration and comment

The class declaration component defines the name of the class. The declaration is started with the keyword *class*. The class name identifies the interfacepart uniquely. It may be followed by some comment –enclosed in double quotes– that gives a description of the class. The end of the comment part is the semicolon.

className → **class** <identifier> ■

commentPart → comment **;** ■

comment → **"** <anythingButDoubleQuote> **"** ■

Semantics
The className has to be a name of an existing class within the implementation [2] part of the composition filters object model. The comment is used to give a short description of the class.

✎ **Example 3.1** Interface part of a class

```
class Clerk
     comment "This is a clerk object.
              It is a subclass of employee.";
```

[2] In our case Smalltalk is used to define the implementation.

3.1.2 Internals declaration

The internalPart consists of a sequence of objects, separated by colons. Each object is followed by a className, which declares the type of the internal. The internals declaration starts with the keyword *internals*.

internalPart → **internals** internalDeclarations ■

internalDeclarations → (internalDeclarations internalDeclaration **;**)? ■

internalDeclaration →
 internalDeclaration **,** objectName **:** className
 | objectName **:** className ■

objectName → <identifier> ■

Semantics
This part declares a sequence of fully encapsulated objects that are used to compose the behavior of the composition filters object. A created internal object will be an instance of the declared class. The declared internals all have to be existing objects encapsulated by the class which defines them.

✎ **Example 3.2** internals

```
internals
     empl : Employee ;              // instance of Employee
     private : Housekeeper ;        // instance of Housekeeper
```

This is a declaration of two internals, one of the class employee and of of the class housekeeper. With filters (discussed later) it is possible to delegate messages to instances of employee or housekeeper.

3.1.3 Externals declaration

The externals declarations begins with the keyword *externals*. The externalPart consists of a sequence of objects, separated by colons. As with internals, each object is followed by a className, which declares the type of the external. However an external declaration does not create a new instance of a class like internals do, but it defines a reference to an object that lies outside the object, such as global objects.

externalPart → **externals** externalDeclarations ■

externalDeclarations → (externalDeclarations externalDeclaration **;**)? ■

externalDeclaration →
 externalDeclaration **,** objectName **:** className
 | objectName **:** className ■

objectName → <identifier> ■

Semantics
The external part declares a number of objects. As in the internalPart, the declared externals all have to be existing objects.

✎ **Example 3.3** externals

```
externals
     logger : TaskLogger;  // instance of TaskLogger
```

This declaration indicates that an object of the class TaskLogger is in the external scope of the class Clerk. The TaskLogger can be used in the filters of a Clerk to log his tasks.

3.1.4 Conditions declaration

The declaration of the conditions is started with the keyword *conditions*. The conditions component declares the conditions that are used within the filters of the interface part. They are separated by commas or semicolons.

conditionPart → **conditions** conditionDeclarations ■

conditionDeclarations → (conditionDeclarations conditionDeclaration **;**)? ■

conditionDeclaration →
  conditionDeclaration **,** <identifier>
  | <identifier> ■

Semantics
A condition is a method –without arguments– of a class. It always has to result in a boolean value. Furthermore, a condition has no side effects.

✎ **Example 3.4** conditions

```
conditions
     noTask,
     taskLeft;
```

In the above example two conditions are declared. The condition *noTask* returns *true* when no tasks are waiting, *taskLeft* returns *true* in case there are tasks left.

3.1.5 Method declaration

The interface part also declares the desired methods that must be implemented by the implementation part. The method declaration part begins with the keyword *methods*, methods should be available on the interface of the object. A method can be an *identifier*, a *keywordIdentifier* or an *operatorIdentifier*. With the *keywordIdentifier* we allow Smalltalk keyword methods in the method declaration, like `putTask:`.
The *operatorIdentifier*, defines a method name between single quotes, making it possible to declare methods as `+`, `*`, etcetera.

methodPart → **methods** methodDeclarations ∎

methodDeclarations → (methodDeclarations methodDeclarations2 **;**)? ∎

methodDeclarations2 →
  methodDeclarations2 **,** methodDeclaration
  | methodDeclaration ∎

methodDeclaration →
  methodName | methodName **(** objectDeclarations? **)** **returns** returntype ∎

methodName →
  <identifier>
  | <keywordIdentifier>
  | <operatorIdentifier> ∎

objectDeclarations →
  objectDeclarations objectDeclaration **;** ∎

objectDeclaration →
  objectNames **:** classDescription
  | objectNames ∎

classDescription → <identifier> ∎

objectNames →
  objectName | objectNames **,** objectName ∎

Semantics
In this part a number of methods is declared.

✎ **Example 3.5** methods

```
methods
     getTask returns String ;
     putTask(String) returns nil ;
```

In this case, two methods are declared, getTask and putTask. Execution of the method *getTask* will return a variable of the type *String*; the method needs no argument. The method *putTask* expects an argument of the type *String*. It will not return a value.

3.1.6 Inputfilters declaration

The inputfilterPart declares all input filters of the class. After the keyword *inputfilters*, a sequence of inputfilters follows; they are separated by a semicolon. An inputfilter consists of a filterName followed by a colon, a filterHandler and filter elements. The filterElements –that follow filterName and filterHandler– consist of three parts: a condition part, a matching part and possibly a parameter part (the parameter part depends on the type of the filter). The filterElements are being explained extensively in section 3.2.

inputfilterPart → **inputfilters** filterDeclarations ■

filterDeclarations → (filterDeclarations filterDeclaration **;**)? ■

filterDeclaration → filterName **:** filterHandler = { filterElements } **;** ■

filterName → <identifier> ■

filterHandler → <identifier> ■

Semantics

The filterName must be a unique name within the interface definition of a class. The filterHandler represents the type of the filter, so it has to be a valid filter type [3]. Not every filter type can be used as an input filter. The possible types are error, wait, dispatch, meta and substitution.

✎ **Example 3.6** inputfilters

```
inputfilters
     select : Error = { noTask => putTask,
                        taskLeft => getTask } ;
     delegate : Dispatch = { true => administrator.schedule } ;
```

In this case, two filters are declared, an error filter and a dispatch filter. The error filter consists of two filter elements. If the condition noTask evaluates to true, the method putTask is passed to the next filter. If the condition taskLeft becomes true, getTask is passed.

The second filter, the Dispatch filter, specifies that all *schedule* messages will be delegated to *administrator*.

3.1.7 Outputfilters declaration

The output filters declaration starts with the keyword *outputfilters*.

outputfilterPart → **outputfilters** filterDeclarations ■

filterDeclarations → (filterDeclarations filterDeclaration **;**)? ■

filterDeclaration → filterName **:** filterHandler = { filterElements } **;** ■

filterName → <identifier> ■

filterHandler → <identifier> ■

Semantics

Outputfilters are very similar to inputfilters. The difference is that inputfilters filter incoming messages and that outputfilters filter outgoing messages. As with input filters, there is a limited set of filter types allowed: The currently known output filter types are: Substitution, meta, error, wait and send. The send filter can only occur as an output filter and not as an input filter.

[3] The possible filter types depend on the types supported by the implementation environment.

3.2 The syntax of filter elements

Each filter consists of a number of filter elements. This section will describe the syntax of the filter elements and gives an example in section 3.2.4.

filterElements →
  filterElements **,** filterElement
  | filterElement ■

filterElement → conditions exclOperator matchingPart parameterPart ■

exclOperator → (=> | ~>) ■

A filter element consists of three parts:

- a condition. This condition tells when a filter element may (or may not) be evaluated further (see section 3.2.1);
- a matching part. This part is matched against a pattern consisting of a message selector and a target specification (see section 3.2.2);
- a parameter part. This part is only applicable for the following filters: Dispatch, Meta and Substitution, where the parameters consists of a target and/or a selector (see section 3.2.3).

3.2.1 Condition part

The syntax of the conditions used within a filter is as follows:

conditions → (conditionExpr)? ■

conditionExpr → conjunction ■

conjunction → disjunction
  | conjunction **or** disjunction
  | conjunction **cor** disjunction
  | conjunction **,** disjunction ■

disjunction → conditionFactor
  | disjunction **and** conditionFactor
  | disjunction **cand** conditionFactor ■

conditionFactor → condition
  | **not** condition ■

condition →
  (conditionExpr **)**
  | conditionName
  | **true**
  | **false** ■

conditionName → <identifier> ■

Semantics

All conditions that are used in a filter element have to be declared in the conditionPart (see section 3.1.4). Each condition will be evaluated to a boolean value. A condition expression can be built by single conditions, separated by logic operators (known from the boolean logic). The precedence in compound expressions is the same as in the boolean logic, so this means that the *AND* will be evaluated before the *OR*. When a different evaluation is needed, parenthesis can be used to alter the precedence.

3.2.2 Matching part

Matching is used to check the message that is being processed. The following is the syntax for the matching part.

matchingPart →
  nameMatching
  | signatureMatching
  | selectorMatching ■

nameMatching → **[** target **.** selector **]** ■

signatureMatching → target **.** selector ■

selectorMatching → selector ■

target →
  objectName
  | wildcard
  | **inner**
  | **outer** ■

selector →
  <keyword>
  | <identifier>
  | <operatorIdentifier>
  | wildcard ■

wildcard → ***** ■

Semantics

There are two kinds of matching: *name matching* and *signature matching*, where *signature matching* is used as the default matching:

- *signature matching*: first a name match takes place on the selector, after which a signature test will test if the selector is in the signature of the corresponding target object. The signature of an object is the set of all messages that the object may accept.

- *name matching*: the selector of the message will be matched with the selector specified in the filter expression. If this match is successful, then the filter element is accepted, otherwise the filter element is skipped.
  A short form of name matching, where only the selector is given, is called *selector matching*.

Both the signature match and the name match will result in a boolean value. The matching part together with the condition part, imply `'=>'` or not imply `'~>'` operator determine wheter the filter element is accepted or rejected.

Signature matching

In signature matching first the selector of the message will be matched with the selector of the filter element. If this evaluates to true, it will test if the selector of the filter element is within the signature [4] of a target object. This target object is either the target of the message or the target specified in the filter element. This results in the following formal definition of signature matching, where $\sigma(x)$ denotes the signature of the object x:

$$\begin{aligned} a.b &\equiv (selector = b \wedge b \,\epsilon\, \sigma(a)) \\ a.* &\equiv (selector \,\epsilon\, \sigma(a)) \\ *.b &\equiv (selector = b \wedge b \,\epsilon\, \sigma(target)) \\ *.* &\equiv (selector \,\epsilon\, \sigma(target)) \end{aligned}$$

Name matching

The difference between signature matching and name matching is that name matching only matches the messages' target, the messages' selector or both. Name matching is much faster than signature matching because it leaves out the time consuming signature check. We advise using name matching with care, because it does not check if the target understands the selector. This results in the following formal definition of name matching:

$$\begin{aligned} [a.b] &\equiv (selector = b \wedge target = a) \\ [a.*] &\equiv (target = a) \\ [*.b] &\equiv (selector = b) \\ [*.*] &\equiv \textbf{true} \end{aligned}$$

Selector matching

As stated above selector matching is a short notation of a name match, where only a selector needs to be matched. This results in the following definition of selector matching:

$$\begin{aligned} b &\equiv *.b \equiv (selelector = b) \\ [b] &\equiv *.b \equiv (selelector = b) \end{aligned}$$

[4] A message selector is in the signature of an object when the selector (method) is declared for the object.

3.2.3 Parameter part

The parameter part consists of zero or more parameters (this must be compelled by semantics). The number of parameters depends on the kind of filterhandler. A parameter is built by an identifier and a parameterValue.

parameterPart → (**(** parameters **)**)? ∎

parameters →
  parameters **,** parameter
  | parameter ∎

parameter → <identifier> = parameterValue ∎

parameterValue →
  <identifier>
  | <keywordIdentifier>
  | <operatorIdentifier>
  | <number> ∎

Semantics
In the current version of the syntax the identifier in the syntax rule *parameter* can only be *target* or *selector*. The parameterValue is the new target or new selector of the message. The parameter part is only applicable in the Dispatch, Meta and Substitution filter. Furthermore it is not allowed to give a new selector to the dispatch filter.

3.2.4 Example of a filter element

Example 3.7 shows the usage of the parts of the filter elements as described in the sections above. The *condition* is **NOT weekday OR holiday**. The *matching part* is **phone.***. The *parameter part* is **target = selector**. In plain language, the meaning of this filter is that in weekends and holidays the answering machine answers the phonecalls.

✎ **Example 3.7** A filter element within a substitution filter

```
subst : Substitution = { NOT weekday OR holiday =>
                         phone.* (target = answeringMachine) };
```

3.3 Filters

The most important parts of the composition filters object model are the input filters and outputfilters. In this section we give a list of filter classes known at this moment. We also explain their purpose.

Every received message passes a set of input filters and each message sent passes a set of output filters. A set of filters consists of an arbitrary number of filters, of which each filter is of a specific filter type.
At the moment the following filter classes are defined: *filter types*

- Dispatch filter
- Error filter
- Wait filter
- Substitution filter
- Meta filter
- Send filter

Generally filters manipulate messages depending on the conditions defined by the filter. The currently known filters, mentioned above, will be described in the next sections. Within TRESE there is research on other filter types, like –for example– a real time filter and an atomic filter.

filter mechanism

input / output filters

The purpose of the filter mechanism is to offer an easy way to manipulate messages that are sent to or sent by objects. There are two kinds of filters: Input and output filters. They deal with messages sent to and sent by an object respectively. Each message needs to pass a group of filters until it results in an exception, a dispatch, a send or any other action that is defined by the different filter classes.

With a dispatch filter a message can be dispatched to a local method, an internal object or an external object. The behavior of objects is therefore composed from a combination of its own methods and the methods of the internal and external objects.

The interface part defines the composition filters. The implementation part defines the behavior and state of an object.

Each filter can be seen as an object; it has an identifier associated with it, and is declared to be an instance of a specific filter class. The class of a filter defines how a filter reacts to messages that try to pass the filter. Therefore each filter has an accept and a reject handler.

A filter consists out of a group of filter elements, that is evaluated in a left-to-right order. If one of the filter elements evaluates to true, the filter will call the accept handler to accept the message. Otherwise, if none of the filter elements are true, the message will be rejected, that is the the reject handler of the filter class will decide what to do with the message. The evaluation of a single filter element consists of the following two steps:

1. condition evaluation: if the condition evaluates to true, the next step (matching) will be done, otherwise the filter element is skipped;

2. matching tries to match the message with the given pattern. This depends on the kind of match, *name matching* or *signature matching*, which is explained in section 3.2.2. *matching*

3.3.1 Dispatch filter

The dispatch filter is used to simply dispatch the message. This means that if the message is accepted, it will result in delegation of the message to its target object. The target object can be the local object, an internal object or an external object. This makes it possible to do data abstractions like (associative) inheritance and delegation. The dispatch filter can only occur as an input filter.

Dispatch filter	
Accept action	*Reject action*
➠ If the target in the parameter part is defined, target substitution will take place on the message; otherwise the target of the matching part is used. ➠ When the messages receiver is inner, the method indicated by the selector of the matching part will be invoked. Hereby it actually initiates the execution of a method; ➠ If the receiver is anything other than inner, the message will be sent to the new target.	➠ When the dispatch filter cannot accept a message it will pass the message to the next filter.

An example of a dispatch filter is the following: The first filter element dispatches messages to the inner object if they are defined for the inner object and the second filter element tries to dispatch to super. With the second filter element inheritance is realized. This means that methods of *super* are available in the current object.

✎ **Example 3.8** Dispatch filter

```
disp : Dispatch = { true => inner.*,
                    true => super.* } ;
```

3.3.2 Error filter

The functionality of the error filter is simple. When a message is accepted, it proceeds to the next filter. When the message is rejected, an error is produced and the execution is stopped. An example of the use of the error filter is screening messages: Only a restricted set of messages may pass to the next filter.

Error filter	
Accept action	*Reject action*
➠ The message will be accepted if the conditions are fulfilled and it is matched with the pattern in the filter expression. This denotes a correct message and results in passing the message to the next filter.	➠ In case a message cannot be accepted by the error filter, it will respond with raising an error. This condition should be avoided or caught by an error handler.

The following exemplifies the error filter. The filter in the example below means that the message *freeTime* will be accepted if *notNineTilFive* or *weekend* evaluates to true. If the message is accepted, it will go on to next filter. If the condition is false, an error is given and execution is halted.

✎ **Example 3.9** Error filter

```
select : Error = { notNineTilFive => *.freeTime,
                   weekend => *.freeTime } ;
```

3.3.3 Wait filter

The wait filter is used for concurrency control and synchronization. A message is delayed until a satisfying situation is reached (this means that a boolean is evaluated to true).

Wait filter	
Accept action	*Reject action*
➠ Like the error filter, the wait filter passes accepted messages to the next filter.	➠ When none of the conditions can be fulfilled, the message will be blocked, i.e. the execution of the calling thread is suspended until one of the conditions on which the message is blocked becomes true; ➠ After releasing the message when the blocking condition becomes true, the message will be reevaluated by the wait filter. This will result in passing the message to the next filter when accepted.

The next filter gives an example of the wait filter. In the example below, the message goHome will be delayed until the condition *notNineTilFive* is true.

✎ **Example 3.10** Wait filter

```
endofday : Wait = { notNineTilFive => *.goHome } ;
```

3.3.4 Meta filter

The meta filter was introduced for object interactions, abstract communication types and reflection [5]. If a message is accepted by the meta filter, the message is converted to an object representing the message. This is called *reification* [6]. The opposite of reification is dereification. This means that the object becomes a message again. When the message is reified, it can be used as an argument for another message. A useful example of the meta filter is logging, as exemplified in example 3.11.

A part of the meta filter is the parameter part, that consists of a target and a selector. The paramter part tells the meta filter to which object and selector the reified message must be sent.

[5] Reflection is a technique to structure and organize self modifying procedures and functions
[6] Reify: To regard or treat (an abstraction) as if it had concrete or material existence.

Meta filter	
Accept action	*Reject action*
➠ The execution of the calling thread will be suspended; ➠ the message will be reified[7], that is the message becomes a first-class object which can be passed as an argument. ➠ a new thread will be created which sends a message with the selector and the reified message as argument to the target. This message will not pass through the outputfilters of the owning object but will be offered to the input filters of the new target directly; ➠ the sending thread will remain suspended until the reified message is dereified –this is when it receives a reply, fire or continue message; ➠ if the, now dereified, message has a reply –when the reified form received a reply message– it will not continue in the next filter, otherwise it will continue there.	➠ The message continues in the next filter.

The next filter demonstrates the meta filter. This filter passes all messages (due to inner.*) as a parameter to the message *log* of the *logger* object. This object logs the original message and will take care that the message resumes with the next filter.

✎ **Example 3.11** Meta filter

```
log : Meta = { true => inner.*(target = logger,
                               selector = log) } ;
```

Another example of a meta filter is the following: If a message *withdraw* or *deposit* is

✎ **Example 3.12** Meta filter

```
amountChange : Meta = { inner.withdraw
                     (target = notify, selector = change),
                         inner.deposit
                     (target = notify, selector = change) } ;
```

sent to an object of the *Account* class, the meta filter sends this message as a parameter of the message *change* to the object *notify*. The *Account* class represents a bank-account, from which money can be drawn, where money can be saved, etcetera. The *notify* object notifies objects that depend on *account* and takes care that the (original) message resumes with the next filter.

[7] A reified message is an object that holds the information of the message. In this representation, the message can easily be changed or tested

3.3.5 Send filter

When sending messages from an object, the Send filter controls the sending mechanism. The send filter can only be used as an output filter.

Send filter	
Accept action	*Reject action*
➠ The target and selector will be substituted to the message, when they are defined; ➠ When this filter's owning object has no encapsulation object, or when the message's receiver lays inside its encapsulating object, the message will be offered to the input filters of the message's receiver. Otherwise the message is offered to the outputfilters of the encapsulating object.	➠ The message continues in the next filter.

The example below indicates that if the condition *weekend* is true, all messages are sent to the *housekeeper* object.

✎ **Example 3.13** Send filter

```
send : Send = { weekend => housekeeper.*  } ;
```

3.3.6 Substitution filter

We have introduced this new filter type. Its function is to substitute the target, selector or both or it. The new target and selector are specified in the parameter part of the filter element.

Substitution filter	
Accept action	*Reject action*
➠ Substitution of the target and/or the selector will take place on the active message;	➠ The message continues in the next filter.

The meaning of the example below is, that if condition *weekend* is true, all messages for the object housekeeper are redirected to the object employee. This means that *employee* must support all messages that are supported by *housekeeper*.

✎ **Example 3.14** Substitution filter

```
subst : Substitution =
        { weekend => housekeeper.*(target = employee)  } ;
```

3.4 Filter example

Example 3.15 shows a *bounded stack* to illustrate the concepts of the composition filters object model . The first item of this example is a **comment**, used for documenting the interface class. Next three conditions *empty*, *filled* and *full* are defined, preceded by the keyword **conditions**.

After the conditions the keyword **methods** introduces the definition of the following methods: *push*, *pop* and *isEmpty*. Finally the inputfilters are defined. Below the keyword **inputfilters** the three filters are defined, namely *check*, *sync* and *disp*.

The first filter, named *check*, is an error filter. The task of this filter is to raise an error and stop execution when a message cannot be accepted. The error filter consists of three filter elements, *inner.push*, *inner.pop* and *inner.isEmpty*. Each filter element will be evaluated until one can be accepted or the end of the filter elements is reached, which will result in a rejection. The pseudo-variable *inner* is the object to which the message is sent. If the target is omitted, the default object *inner* is chosen (So in the error filter *inner* is 'redundant').

The next filter is a wait filter. It has the purpose of delaying a message until the conditions can be satisfied. In the example, a *push* is delayed if the stack is full. On the other hand a *pop* is delayed if the stack is empty.

Finally the dispatch filter, dispatches the message to its target. In this filter, the wildcard character '*' is used, which denotes all the methods of the stack class.

If for some reason the message is not dispatched, so if it 'fell through' all inputfilters, an error message will be generated. The exact functionality of filters has been described in section 3.3.

✎ **Example 3.15** Bounded stack class

```
class boundedStack
comment
     "This is an interface class for the Stack class.
     The stack provides the operations 'push', 'pop' and 'isEmpty'."
conditions
     empty;  // true if the stack is empty
     filled; // true if the stack is not empty and not full
     full;   // true if the stack is full
methods
     push(item:AnyObjectOrNil)  returns nil;
     pop     returns AnyObjectOrNil;
     isEmpty returns Boolean;
inputfilters
     check: Error    = { inner.push, inner.pop, inner.isEmpty };
     sync:  Wait     = { inner.isEmpty, empty=>inner.push,
                         filled=>inner.pop,filled=>inner.push,
                         full=>inner.pop };
     disp:  Dispatch = { true=>inner.* };
end;
```

Chapter 4

Integrating composition filters in Smalltalk

Any small object that is accidentally dropped will hide
under a larger object.
Anonymous

"Hi. This is Dan Cassidy's answering machine.
Please leave your name and number...
and after I've doctored the tape,
your message will implicate you in a federal crime
and be brought to the attention of the F.B.I...
BEEEP"
Blue Devil comics

The aim of this chapter is to describe the integration of Smalltalk with the composition filters object model; this means that we can extend a Smalltalk class with one or more composition filters. Before giving an example of Smalltalk with filters in section 4.2, section 4.1 will take a look at the way things work in Smalltalk. In section 4.3 we reconsider the inheritance relation between classes and sending messages between objects, in particular between Smalltalk classes/objects with and without composition filters. Section 4.4 will explain the integration of composition filters and section 4.2 shows the changed user interface for adding composition filters to a class.

4.1 Smalltalk introduction

Studying Smalltalk is useful because it is the implementation environment for the composition filters extension.

Section 4.1.6 explains how Smalltalk reacts to a message sent and how to create a first-class message representation.

4.1.1 Objects

Object
Messages

Objects and messages are fundamental concepts of Smalltalk. Everything is based on the existence of objects and the interaction between objects by sending messages . An object is an encapsulation of information and an interface describing to which messages the object responds. A message is a request for an object to carry out one of its operations. Since objects can only communicate by sending messages, message passing is the basic means for executions in the system.

Class

Classes describe the behavior and structure for a group of common objects. Every object is an instance of a class and can be created by sending a *new* or *basicNew* message to its class. Although there can be many references to an object, each object has its own identity.

Method dictionary
Method

As can be seen from figure 4.1 behavior is defined by associating a method dictionary with a class. The method dictionary contains a collection of methods, each with a unique name, called the method selector. In addition to behavior, a class can also define the structure of its instances by defining instance variables.

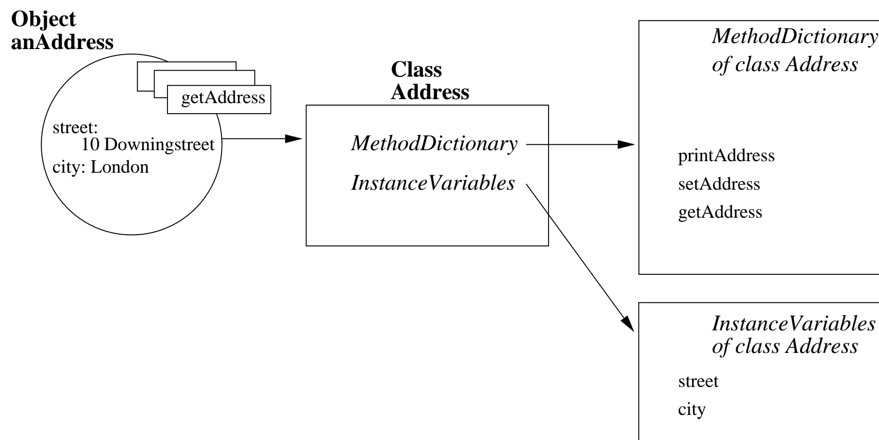


Figure 4.1: An *address* object

Figure 4.1 shows us the complete picture of an object, its class, methods and instance variables. The object *anAddress* is an instance of the class *Address*. It has two instance variables called *street* and *city* and it responds to the messages *printAddress*, *setAddress*, *getAddress*. As can be seen from the illustration the messages to which the object responds are specified in a method dictionary.

4.1.2 Inheritance

Inheritance

Superclass and subclass

The inheritance relation between classes provides a mechanism to refine a class and reuse parts that stay the same. In Smalltalk the class that is being refined is called the superclass and its refined version is the subclass.

A class inherits instance variables and methods from its super class, which in turn inherits from its super, and so on. The top class in the inheritance hierarchy is called the root class. In Smalltalk the class *Object* serves as the root class for most classes.[1]

[1] Actually *Object* is the root class for all classes, but we can define new root classes.

It is not possible to change an inherited definition, but one can add new instance variables and new methods. Furthermore it is possible to override a previously defined methods with new functionality.

4.1.3 Pseudo variables

self and super

Smalltalk provides two pseudo variables [2] *self* and *super*, which can be used as a message receiver. The variable *self* refers to the object that is currently executing the message.

The semantics of the pseudo variable *super* is as follows. Suppose there are two classes *A* and *B*, where *B* inherits from *A*. When a method from class *B* is executing, using *super* as a message receiver in this method means that the message lookup continues at its super class, that is class *A*.

4.1.4 First class message representation

Smalltalk provides two classes, *Message* and *MessageSend*, that represent messages.[3]

Message class

The *Message* class represents a message with a *selector* and *arguments*. The selector is analogous to a method name. Arguments of the message are held in an array (possibly empty). Defined by the *Message* class there is a set of methods for accessing and changing the *selector* and *arguments* of the message.

MessageSend class

Class *MessageSend* is a specialization of class *Message*, which also contains the receiver of the message. Since the *MessageSend* class has all the information to restart message execution, that is *receiver*, *selector* and *arguments* of the message, this class has a method to activate the message.

Perform methods

Message can be invoked by using the *perform* methods specified in root class *Object*. These methods provide an interface to the Smalltalk message execution mechanism. When a *perform* method is sent to the receiving object and we provide the *perform* method with a message *selector* and its *arguments*, this will result in invoking a method directly for the receiving object. This differs from sending a message, since this involves method lookup (through the class hierarchy) followed by the invocation of a method when one was found.

4.1.5 Describing classes

In Smalltalk four classes — *Behavior*, *ClassDescription*, *Metaclass*, *Class* — provide the basis for describing new classes.

class Behavior

Instances of class *Behavior* provide the minimum state necessary for compiling methods, and creating and running instances.

class ClassDescription

The class *ClassDescription* adds a number of facilities to basic Behavior, among which are named instance variables and category organization for methods. *ClassDescription* is an abstract class: its facilities are intended for inheritance by the two subclasses, *Class* and *Metaclass*.

class Class

Instances of class *Class* describe the representation and behavior of objects.

class Metaclass

Metaclasses add instance-specific behavior to various classes in the system. This typi-

[2] *self* and *super* are message recievers like objects, but are generally known as pseudo variables.

[3] We can of course create a new specialized Message class by inheriting from class *Message* or *MessageSend*.

cally includes messages for initializing class variables and instance creation messages particular to that class. There is only one instance of a metaclass, namely the class (*this-Class*) that is being described.

4.1.6 Smalltalk messages handling

Before explaining the extension of Smalltalk objects into FIST object in detail, let us first look at the normal message processing in Smalltalk.

Method lookup

When a message is sent to an Object, the Smalltalk method lookup mechanism first tries to find a corresponding method in the method dictionary of the object's class. If no such method is found, Smalltalk moves upward in the super class chain to look for the message.

doesNotUnderstand

Smalltalk provides a method `doesNotUnderstand` which is invoked whenever the method lookup fails to find a corresponding method. The `doesNotUnderstand` method is invoked with the original message in reified form as an argument which is an instance of class *Message*. With the *Message* class we can also compose messages and then activate these messages into a message invocation.

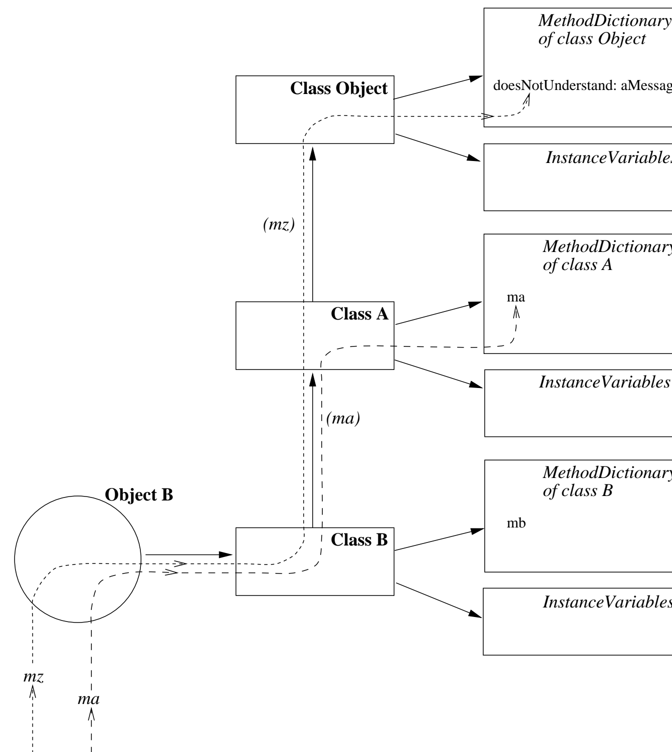


Figure 4.2: Smalltalk message lookup

Figure 4.2 illustrates sending a message *ma* and a message *mz* to an instance of class *B*. Message *ma* is not found in the method dictionary of class *B* and continues the lookup process at class *A* where *ma* is found and invoked. Message *mz* is in none of the method dictionaries of class *B* or its superclasses, so this will result in an invocation of the method `doesNotUnderstand`.

4.2 An example of using composition filters in Smalltalk

The previous section gave a introduction to Smalltalk. In this section we will illustrate the use of composition filters in Smalltalk.

4.2.1 Multiple inheritance

In Objectwork\Smalltalk — the implementation environment of our composition filter extension — only single inheritance is supported. Suppose we have two classes *Spouse* and *Secretary*. Introducing an new class *SpouseSecretary* representing a person that is both a Spouse and a Secretary creates a lot of redundancy if only single inheritance is allowed. In this section we will show how multiple inheritance is supported in Smalltalk with the composition filter extension.

For example, suppose that a bank has a loan department offering loans to customers and a credit-card department catering credit-cards to customers. In Smalltalk, given the two classes Loans and CreditCards, we obtain the following:

Loans class

```
Object subclass: #Loans
    instanceVariableNames: 'loan account'
    classVariablesNames: ''
    poolDictionaries: ''
    category: 'Bank-LoanDepartment'
```

CreditCards class

```
Object subclass: #CreditCards
    instanceVariableNames: 'creditCardAccount balance'
    classVariablesNames: ''
    poolDictionaries: ''
    category: 'Bank-CreditCardDepartment'
```

Loans has the methods *takeloan* and *repayloan*. CreditCards has the methods *charge* and *repay*.

The bank wants to offer an new service, that is the loan and credit-card service as one offering. In Smalltalk without filters we would need to duplicate the information from Loans and CreditCards in a new class LoansCreditCards.

With the Smalltalk composition filter extension we can design a dispatch filter that simulates multiple inheritance. This is done by creating the class *LoansCreditCards* and make it a subclass of *ObjectF*[4]. The next step is to specify a filter definition for this class. Both the class and filter definition are shown below.

LoansCreditCards class

```
ObjectF subclass: #LoansCreditCards
    instanceVariableNames: 'loans creditCards'
    classVariablesNames: ''
    poolDictionaries: ''
    category: 'Bank-LoanCreditCardService'
```

[4] ObjectF is the root class of all classes that can have filters

LoansCreditCards filter definition

```
class LoansCreditCards
    internals
        loans, creditCards ;
    inputfilters
        disp : Dispatch = { true => inner.* ,
                            true => loans.* ,
                            true => creditCards.* } ;
        error : Error = { true => *.* } ;
```

We will now show how the Smalltalk System Browser can be used to add a filter definition to a class. It is assumed that the *LoansCreditCards* class is already created and it is a subclass of *ObjectF*.

First select the class *LoansCreditCards* in the class subview window, then select the *filters...* option from to class menu (see figure 4.3).

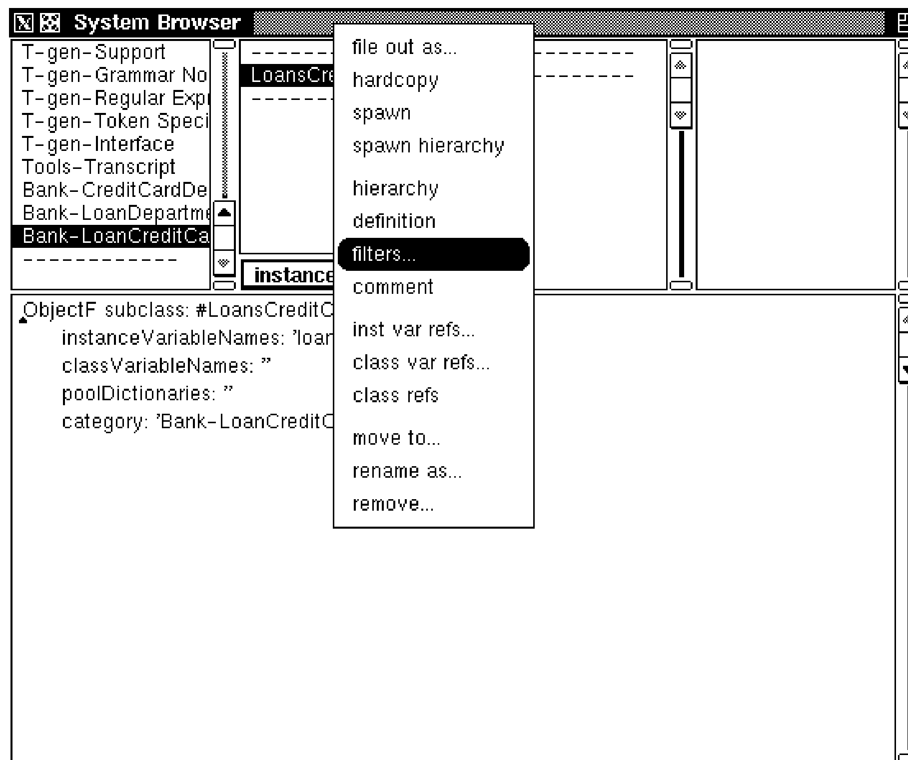


Figure 4.3: System Browser's class subview - with a class selected

Selecting the *filters...* option will switch the code subview to the filter definition windows, as is shown by figure 4.4. In this window we provide the filter definition for class *LoansCreditCards*.

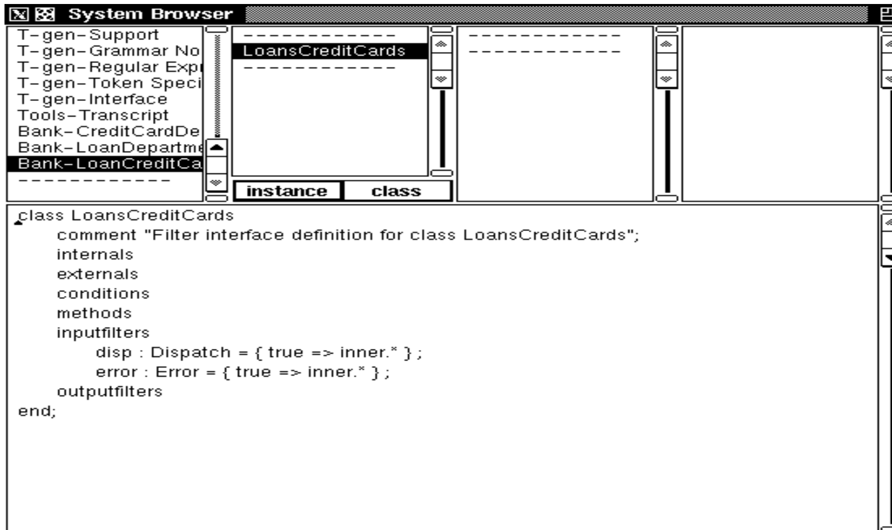


Figure 4.4: System Browser's filter definition window - with a class selected

Just as with method definitions or class definitions we can accept a filter definition by selecting the *accept* option of the code view menu, as is illustrated in figure 4.5. This will compile the filter definition and make the appropriate changes. Any syntax errors are printed in the filter definition window, while semantic errors are shown on the System Transcript window.

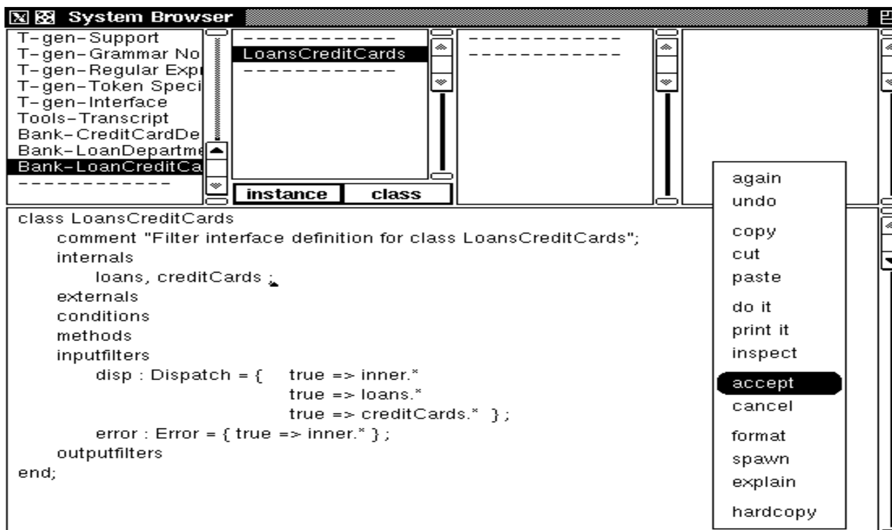


Figure 4.5: System Browser's filter definition window - with a class selected

4.3 Classes and objects with composition filters

The first step to extending Smalltalk with composition filters is adding a composition filter definition to a Smalltalk class, We realize this by adding a filter definition to the class *ClassDescription*. This creates two types of classes in the system: normal Smalltalk classes, and Smalltalk classes with a composition filter definition. To simplify the discussion, the latter will be called *FIST classes* and the first Smalltalk classes and their objects *FIST objects* and *Smalltalk objects*.

Figure 4.6 shows a *FIST* object, its class, method dictionary instance variables and filter interface specification.

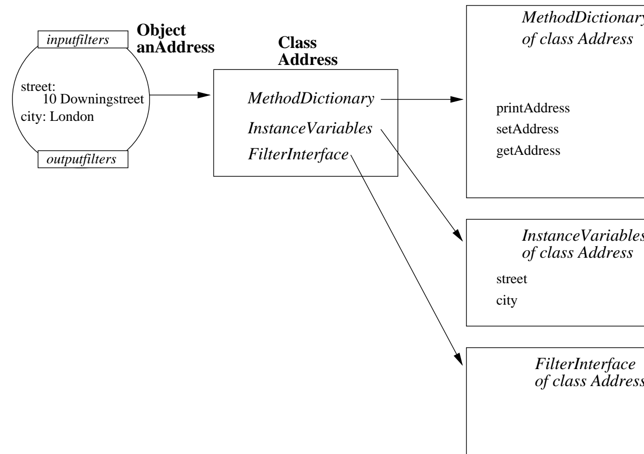
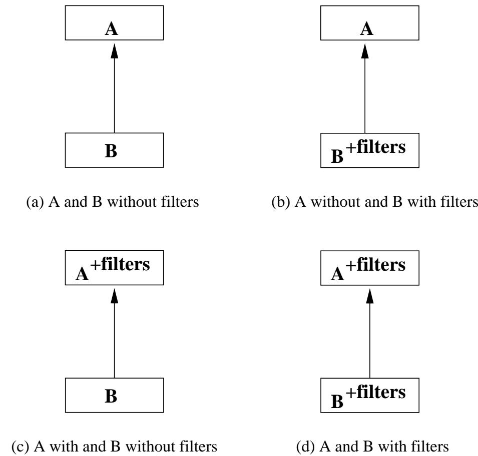


Figure 4.6: An *address* object with composition filters

In the next sections we will reconsider the inheritance relation between classes and the communication with messages between objects.

4.3.1 Inheritance between Smalltalk and FIST classes

Introducing a new kind of class, that is a class with a composition filter extension required us to take a new look at the inheritance mechanism. Figure 4.7 shows the four inheritance relations between normal Smalltalk classes and FIST classes.



(a) A and B without filters (b) A without and B with filters

(c) A with and B without filters (d) A and B with filters

Figure 4.7: Inheritance between Smalltalk classes and FIST classes

In figure 4.7a-b we can use the normal inheritance structure of Smalltalk. Class *B* either with or without composition filters inherits both the variable declarations and methods from its superclass *A*.

Consider now that filters are defined for *A*, but not for *B* as is illustrated in figure 4.7c. We will give *B* the following simple filter definition:

```
class B
  inputfilters
    disp  : Dispatch = { true => inner.* };
    error : Error    = { true => inner.* };
end;
```

Otherwise figure 4.7c has the same effect as figure 4.7a. This would be an undesirable effect since we lose information and a *B* object would not support any filters at all. Therefore the following rule is used:

- For every class *X* that has no filters defined, but has an super (ancestor) class with a filter definition, class *X* will have the above simple filter definition.

Adding this simple filter definition will certainly decrease the performance of message passing when dealing with a FIST object. However, we expect a dispatch filter can be optimized and will be able to detect a simple filter definition with some extra knowledge. With this knowledge it can use the much faster Smalltalk inheritance mechanism when this is possible.

When class *B* has filters too, as seen in figure 4.7d, there will be no need for this simple filter definition.

We should realize that an object with filters can have filters for its superclasses too. This means that an object has to deal not only with the filters of its own class, but also with the filters of its superclasses. Also once filters are introduced, all subclasses have either filters of their own or a simple filter definition supplied by the environment.

4.3.2 Message communication between objects

Communication between objects comes in four variants, depending on whether the objects are Smalltalk objects or FIST objects.

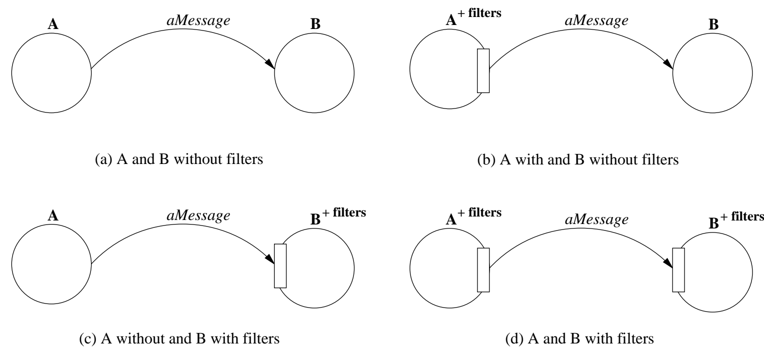


Figure 4.8: Sending Messages between Smalltalk and FIST objects

If an object sends a message, there are two possibilities:

- the object is a Smalltalk object (see 4.8a,c), in which case the message is sent immediately;
- the object is a FIST object (see 4.8b,d). In this case the message needs to be passed to the set of output filters in reified form.

If an object receives a message (from a specific sending object), again there are two possibilities:

- the object receiving the message is a Smalltalk object (see 4.8a,b), which will start the normal method lookup process of Smalltalk.
- the object receiving the message is a FIST object (see 4.8c,d), in which case the message needs to be passed to the set of input filters.

4.4 The Smalltalk extension

The key issue of Extending Smalltalk objects with composition filters is intercepting messages, because they have to be filtered. Standard in Smalltalk is method lookup and message send, this we need to intercept. This change to Smalltalk incorporates the following, which will be explained in the next sections:

- providing a Message Manager object that will manage incoming and outgoing messages for the object it is associated with;
- extending each object that has filters with a reference to *inner* and *outer*. Where *inner* is the object itself and *outer* the associated Message Manager;
- recompiling all methods for a class that has filters and for all its parent and children classes too.

Unfortunately, the above changes created some conflicts in the Smalltalk system. Because of this we chose for a limited extension by creating a new root class *ObjectF*, which is the root class for all objects extensible with composition filters.

4.4.1 Message Manager

Extending a Smalltalk object with composition filters is comparable to encapsulating an object with a Message Manager as shown in in figure 4.9. When a message is sent to the encapsulated object, a message input action is performed. A message output action is performed in case a message is sent from the encapsulated object.

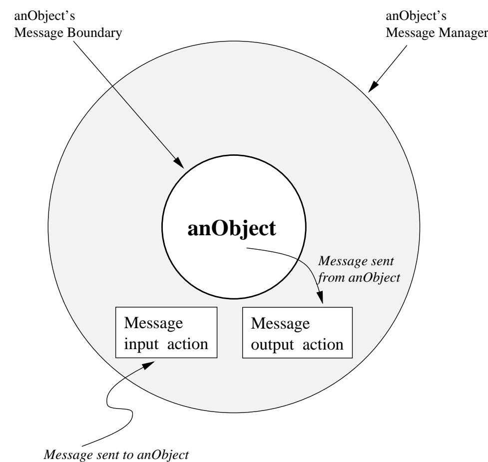


Figure 4.9: Encapsulating an object with an Message Manager

The input and output action performed by the Message Manager consist of passing the message to the set of input and output filters respectively. For this we need a way to reify [5] messages; meaning that we create a first class representation of a message that can be changed.

[5] reifying a message is an essential technique for implementing the composition filters extension

In section 4.1.4 we saw two ways to get a first class message representation in Smalltalk:

- when a message is not understood, Smalltalk invokes a special method on the receiving object. This method, with the selector *doesNotUnderstand*, receives the original message as an argument.
- we can create our own messages by creating instances of the *Message* or *MessageSend* class. i Methods can be invoked using the *perform* methods specified in the *Object* class.

4.4.2 Input filters

Each message sent to an object with filters needs to be processed by the input filters. To realize this the object's Message Manager is functioning as a message receiver for that particular object. The Message Manager can then catch the message with the *doesNotUnderstand* method. In order for the Message Manager to catch all messages with *doesNotUnderstand*, it cannot define methods for messages to which their managing object may respond. It is however necessary for the Message Manager to have some operations in order for their internal mechanisms to do the work. The methods used by the Message Manager are prefixed by *MM*, to provide selectors that will not overlap with the object's methods. Now that the message has been caught and reified, we can pass it to the input filters.

4.4.3 Output filters

When a method of an object is executing, it is possible that it sends a message which needs to be processed by the output filters. Instead of sending the message we need to reify the message and send the reified message to the output filters.

We will clarify how this works with an example. Suppose we have a method *ma* that sends a message *mb* to *anObjectB* which needs to be processed by the output filters, because the *ObjectB* class has output filters.

```
ma
  anObjectB mb.
```

We will translate the code for this Smalltalk method into the following:

```
ma
  | aMessage |
  aMessage := MessageSend new.   "create a new message"
  aMessage receiver: anObjectB. "set the message receiver"
  aMessage selector: #mb.        "set the message selector"
  aMessage arguments: #().       "message has no arguments"

  "sent the message to the outputfilters of"
  "Message Manager of this object"
  anObjectsMessageManager outputFilter: aMessage.
```

To realize this we provided a second Smalltalk Compiler. that will translate the code and make the appropriate changes. This compiler inherits most of its behavior from the *SmalltalkCompiler* class.

4.4.4 Messages sent to self and super

The pseudo variables *self* and *super* have different meaning when dealing with composition filters. In case of an object with filters sending a message to *self* means that the message has to be sent the object with filters, instead of sending it to the object without filters.

Our root class *ObjectF* provides two pseudo variables *outer* and *inner*, where *outer* refers to the object with filters, thus the object's Message Manager, while *inner* refers to the object itself. The new Smalltalk compiler we mentioned in the previous section also takes care of translating *self* into *outer*.

Sending messages to *super* needs special attention. When none of the super classes has filters, sending to *super* works as is defined by Smalltalk. That is the message lookup starts looking for the method in the super class of executing class. But when a superclass has filters, a message sent to *super* needs to pass the super class input filters. This will require another change in the Smalltalk compiler which is not implemented yet.

4.4.5 Object creation

To create an object that has filters one can use the Smalltalk *new* method, the result however, will not be a normal object creation. Upon sending a *new* message, the object itself and its Message Manager will be created. Next the association between the object and its Message Manager will be set.

In section 4.4.2 we saw that the Message Manager also catches the message for an object. For this reason an object's Message Manager will be returned on creation of a *new* object. By this we give each object with filters a private Message Manager that is catching incoming and outgoing messages.

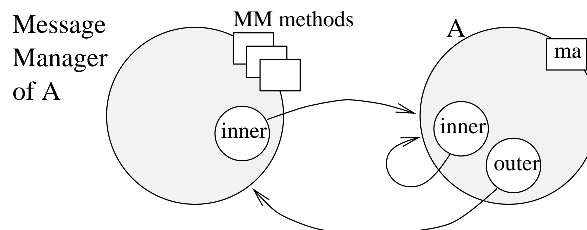


Figure 4.10: An object A with its Message Manager

Dealing with level of inheritance

In section 4.3.1 we stated that filters belong to a class. This meant that the Message Manager has to deal not only with filters of its own class, but also with the filters of the classes it inherits from.

In our extension we solved this problem by creating the Message Managers objects from shadow classes. When we create a new class in our *ObjectF* class hierarchy, this will result in the automatic creation of a new shadow class with the same name concatenated by *Shadow*. Consequently the *Shadow* classes provide the composition filters for the classes they are associated with.

Figure 4.11 provides a schematic representation of the extension.

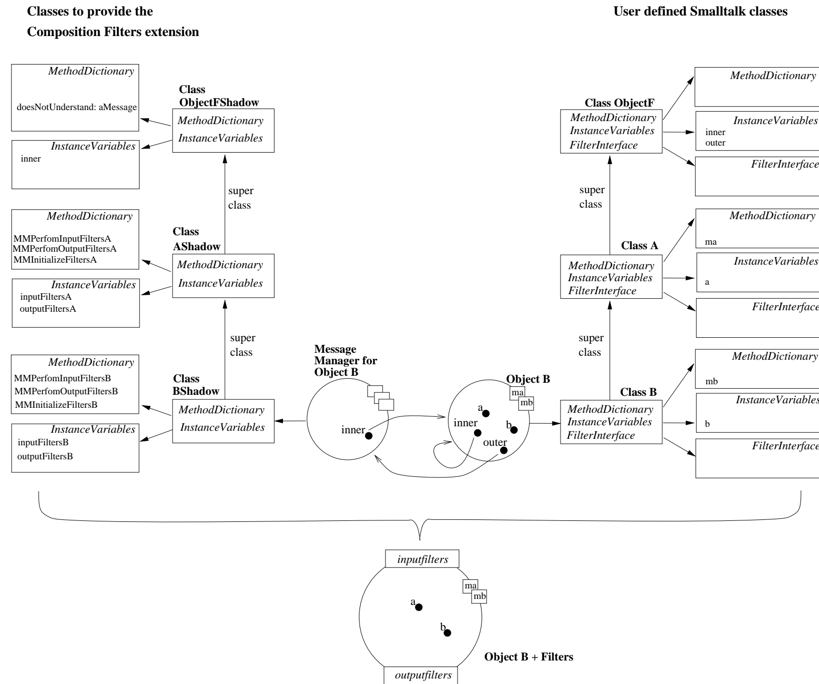


Figure 4.11: The composition filter extension in Smalltalk

Whenever a message is sent to the object *B* with filters, the message will be sent to the Message Manager instead. The Message Manager of an object *B* will not succeed in finding the method in its method dictionary and invoke the *doesNotUnderstand* method. The *doesNotUnderstand* method will than send the reified message to the input filters of class *B*, by invoking *MMPerformInputFiltersB*.

All messages sent from an object that have to go through the output filters are reified into active messages and then sent to the output filters of the Message Manager, using the *MMPerformOutputFiltersB* method. [6]

[6] *MMPerformOutputFiltersB* is invoked when executing method is of class *B*. In case the executing methods is of class *A*, the *MMPerformOutputFiltersA* is invoked.

Chapter 5

Compiler

Do not allow this language (Ada) in its present state to be used in applications where reliability is critical, i.e., nuclear power stations, cruise missiles, early warning systems, anti-ballistic missile defense systems. The next rocket to go astray as a result of a programming language error may not be an exploratory space rocket on a harmless trip to Venus: It may be a nuclear warhead exploding over one of our cities. An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations.
C. A. R. Hoare

This chapter describes the compiler for the Composition Filter Definition Language that was described in the chapter 3. This compiler is further referred to as FIST compiler.

The book *Compilers: Principles, Techniques, and Tools* [Aho *et al.*, 1986] states the importance of choosing a correct environment in which the compiler can be constructed. This will not only affect how quickly the compiler is implemented, but also its reliability.

We have chosen T-Gen[1], which is a general-purpose object-oriented tool for the automatic generation of translator. It is written in Smalltalk and it lives in the Smalltalk programming environment. Because the composition filter model should be integrated into the Smalltalk environment, this tool greatly facilitates the construction of the compiler.

[1] T-Gen (from ”translator generator”) is a program by Justin O. Graver of the University of Florida

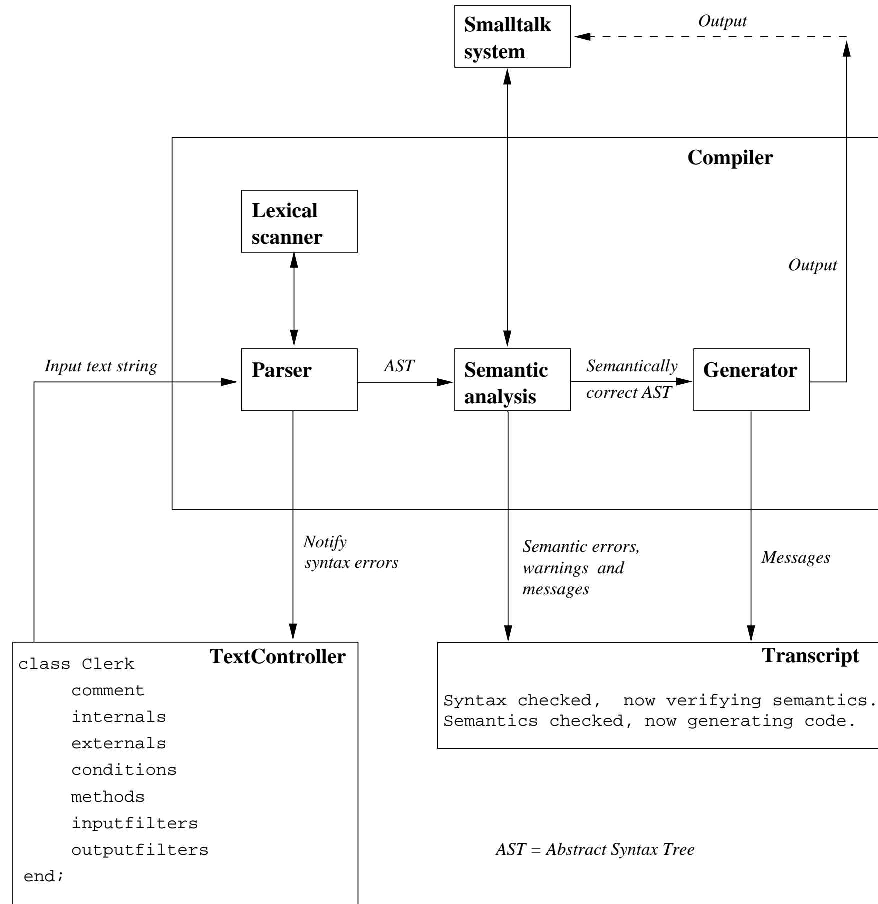


Figure 5.1: Compiler.

The task of the FIST compiler is to generate Smalltalk code for an composition filter definition. This is done in three steps: parsing, semantic analysis and code generation, as is exemplified in figure 5.1.

The parser and the lexical scanner work together to create an Abstract Syntax Tree (AST), which is a data structure that eases the semantic and code generation phases. T-Gen supports the building of an AST by providing a *ParseTreeNode* class, from which all our own AST nodes must inherit.

Once the Abstract Syntax Tree has been constructed, the system is ready for semantic analysis to obtain further meaning and see if the program is semantically correct. The process of semantic analysis is explained in section 5.1.

The generation phase of the compiler uses the semantically correct AST to create usable output. This will be explained in section 5.2.

5.1 Semantic analysis

When syntax is parsed successfully the root node of the generated Abstract Syntax Tree is given to the semantic analyzer. The semantic analyzer works in conjunction with the Smalltalk system to determine the correctness.

Semantic errors will be printed on the System Transcript window of the Smalltalk environment. The following table 5.1 gives a list of semantic errors.

Semantic errors
class not found
condition declared before
condition does not exist
condition not declared
external declared before
external does not exist
internal declared before
internal does not exist
internal declared as external
method declared before
method does not exist
method not declared
input filter declared before
output filter declared before
unknown filter type
unknown input filter type
unknown output filter type
unknown target reference

Table 5.1: Semantic errors

5.2 Code generation

As we saw in section 4.4.5, an object's Message Manager is built from shadow classes. These shadow classes control the input and output filters for an object of its original class. In this section we explain how these shadow classes are initialized, by generating code for a filter specification.

When a message passes through the filterset, be it the inputfilters or the outputfilters, it can be accepted or rejected. Accepting a message means that a filter element was found which is true for the active message. If this filter elements has any parameters, accepting the message can also change parts of the message like the *selector* or *target*.

We can only accept a message when there is a filter element that matches the active message. A filter element matches a message when: the condition of the filter element is true and the message specified by the message part[2] matches with the active message. In case of a *not imply* operator in the filter expression it matches when the condition of the filter element is true and the message specified by the message part does not match with the active message.

The matching evaluation of filter element with a message can be put into one conditional block. The condition control block, further referred as *CCB*, has the following basic structure: `[ :msg | "aCondition" ]`. The *msg* argument of the block is used to pass the active message to the control block and *aCondition* represent the condition generated from a filter expression.

When *CCB* is evaluated it results in a boolean value *true* or *false*, which tells the filter to accept or reject the active message. Furthermore the control block can change part of the active message, when this is specified the parameter part of the filter element.

The shadow classes need to be initialized with a set of input filters and a set of output filters. Each filter in this set is controlled by a condition control block, generated from the filter expression. The result of the condition control block tells the filter to accept or reject the message.

The initialize routine for a class with no output filters and a Dispatch and Error filter as input filter will look like this:

```
MMinitialize
  inputFilters := OrderedCollection new.
  outputFilters := OrderedCollection new.

  inputFilters addLast: ( DispatchFilter new
      condBlock: [ :msg | "aCondition" ] ).
  inputFilters addLast: ( Error new
      condBlock: [ :msg | "aCondition" ] ).
```

This code is straightforward except for the condition control block *CCB* which we explain in the next section.

5.3 Generating a condition control block

The *CCB* determines its boolean value in the way that is illustrated in figure 5.2.

[2] In the message part we can specify name matching or signature matching, as explained in section 3.2.2.

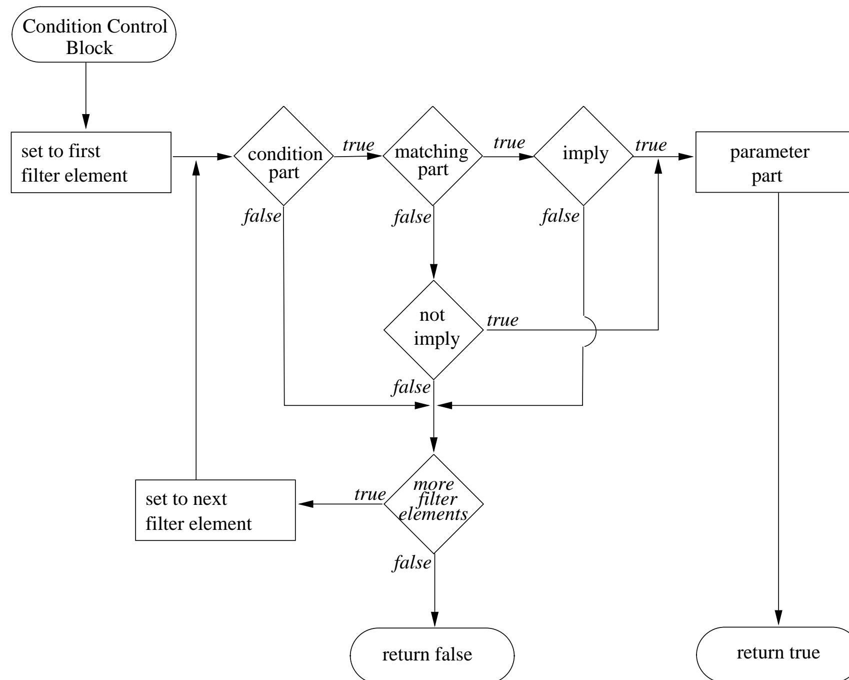


Figure 5.2: Steps in a Condition Control Block.

As we can see in the above picture, first the condition part will be evaluated. The matching part will only be evaluated when the condition part is true. Depending on the kind of enable operator[3] we will execute the parameter part and return *true* or go to the next filter element if one is available. In the end if no matching filter element is found, *false* is returned to indicate a reject.

The following parts of the filter elements are important in generating the *CCB*:

- condition part (section 5.3.1);
- matching part (section 5.3.2);
- parameter part (section 5.3.3).

[3]The enable operator is an '=>' *imply* or '∼>' *not imply* operator.

5.3.1 Condition part of a filter expression

The possible elements in a condition part, their AST nodes and the Smalltalk equivalent are given in table 5.3.1. In this table the following abbrivations are used:

- L = reference to a LeftNode;
- R = reference to a RightNode;
- N = reference to a Node;
- Id = reference to an Identifier.

Filter expr conditionPart	**AST Node**	**Smalltalk equivalent**	**Description**
cor	CorNode(L,R)	`L or:[ R ]`	conditional or
cand	CandNode(L,R)	`L and:[ R ]`	conditional and
or	OrNode(L,R)	`L | R`	unordered or
and	AndNode(L,R)	`L & R`	unordered and
not	NotNode(N)	`N not`	mondadic not
true	TrueNode	`true`	true value
false	FalseNode	`false`	false value
condition indentifier	ConditionNode(Id)	`inner Id`	condition method

Table 5.2: Condition part

The AST of the condition expression `c1 and ( c2 cor c3 ) or c4` is shown in figure 5.3.

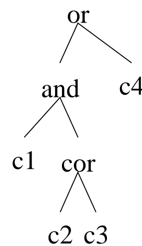


Figure 5.3: A condition expression in an AST.

For this AST the following Smalltalk condition is generated:

```
inner c1 & ( inner c2 or: [inner c3] ) | c4
```

5.3.2 Matching part of a filter expression

The following two tables 5.3 and 5.4 show what Smalltalk code is generated for signature matching and name matching respectively. Selector matching is not shown here, since it is a simple form of name matching.

Signature matching	**Definition**	**Smalltalk equivalent**
`a.b`	$sel = b \wedge b \epsilon \sigma(a)$	`msg selector == #b and: [a canUnderstand: b]`
`a.*`	$sel \epsilon \sigma(a)$	`a canUnderstand: sel`
`*.b`	$sel = b \wedge b \epsilon \sigma(tar)$	`msg selector == #b and: [msg receiver canUnderstand: b]`
`*.*`	$sel \epsilon \sigma(tar)$	`msg receiver canUnderstand: sel`

Table 5.3: Matching part for signature matching

Name matching	**Definition**	**Smalltalk equivalent**
`a.b`	$sel = b \wedge tar = c$	`msg selector == #b and: [msg receiver == a]`
`a.*`	$tar = a$	`msg receiver == a`
`*.b`	$sel = b$	`msg selector == #b`
`*.*`	$true$	`true`

Table 5.4: Matching part for name matching

5.3.3 Parameter part of a filter expression

The parameter part of a filter expression changes attributes of the active message. This is possible in the *CCB*, since the active message is presented as an argument in the *CCB*. Table 5.3.3 below shows the Smalltalk code that is generated for a parameter part.

Parameter expression	**Meaning**	**Smalltalk equivalent**
`target = a`	set new target	`msg newTarget: a`
`selector = b`	set new selector	`msg newSelector: b`

Table 5.5: Parameter part

Conclusion

During the thirteen weeks of our graduation we studied the possibilities of extending Smalltalk with the composition filters object model, developed at the University of Twente. The composition filters object model is introduced because the conventional object oriented models contain a number of deficiencies. Some of these deficiencies are: inheritance and delegation, coordinated behavior and real-time specifications.

The concept of the composition filters is a modular extension of the existing object oriented model. Integrating Smalltalk and the composition filters concept resulted in an extended Smalltalk, which is able to deal with an important class of modeling problems.

Composition filters are able to manipulate messages that are sent to or sent by an object. These messages can be manipulated with the composition filters. They can for example be delayed, redirected or modified.

The implementation consists of a compiler that translates a Composition Filters specification into in the Composition Filters Definition Language (that we redefined), which is added to the Smalltalk class.

The composition filters is *not* fully integrated in the Smalltalk environment. It is limited in the following:

- no support for output filters yet;
- not every Smalltalk class can be extended with composition filters (only subclasses of ObjectF);
- messages sent to *super* are not intercepted if the superclass has filters;
- support for only a small set of filter types, that is Dispatch Filter, Substitution Filter and Error Filter.

Efficiency is also a problem, since messages are caught by the *doesNotUnderstand* method. This requires searching the hierarchy of the Message Manager class until the *doesNotUnderstand* method can be invoked. An efficient implementation would require modifications to the Smalltalk virtual machine.

Future work

Future work and enhancements include:

- implementing output filters in Smalltalk;
- support other filter classes like em Meta Filter, *Send Filter*, *Wait Filter*;
- make it possible to extend every class in the Smalltalk system with a composition filter definition;
- intercept messages sent to *super* if the superclass has filters;
- support file in/out of composition filter definitions;
- debugger for composition filter classes in Smalltalk.

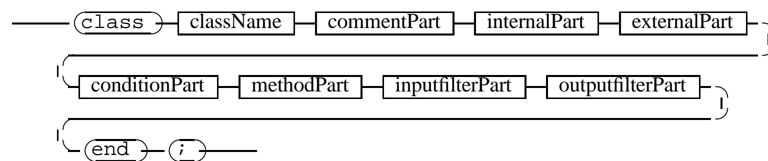
Appendix A

Filter syntax diagrams

In the next pages the syntaxdiagrams of the interfacepart of the filter language as described in chapter 3 is given (in diagram form). This syntax is used to create a parser.

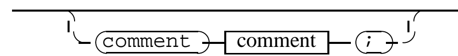
In the diagram below yields that square blocks are nonterminals and rounded blocks are terminals.

interfacePart

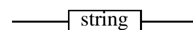


className

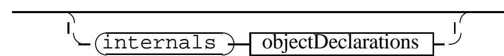
commentPart



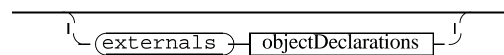
comment



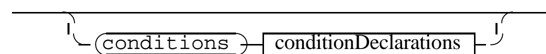
internalPart



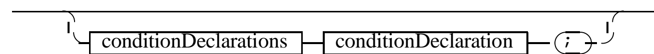
externalPart



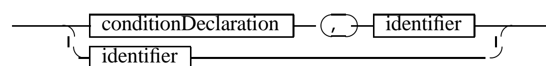
conditionpart



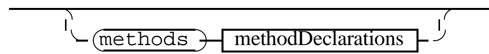
conditionDeclarations



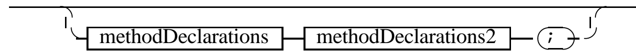
conditionDeclaration



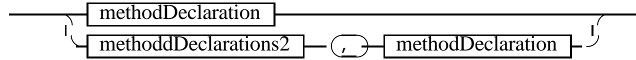
methodPart



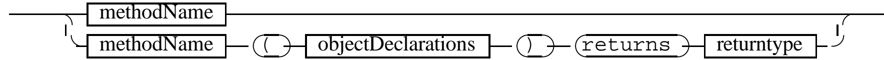
methodDeclarations



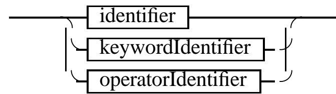
methodDeclarations2



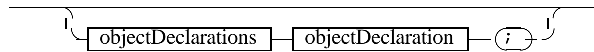
methodDeclaration



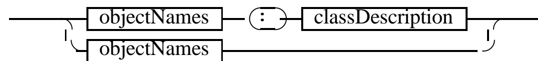
methodName



objectDeclarations

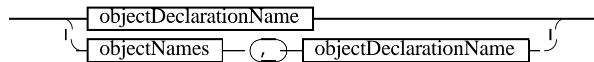


objectDeclaration

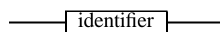


classDescription

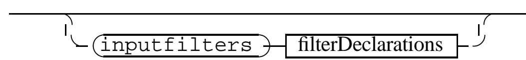
objectNames



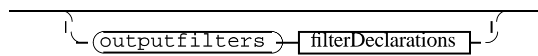
objectDeclarationName



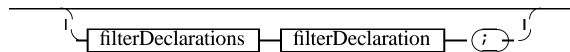
inputFilterPart



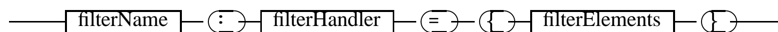
outputFilterPart



filterDeclarations



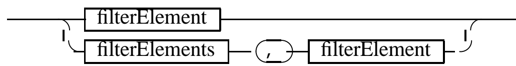
filterDeclaration



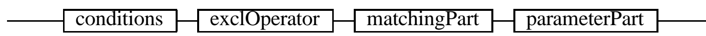
filterName

filterHandler

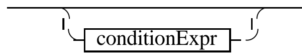
filterElements



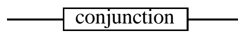
filterElement



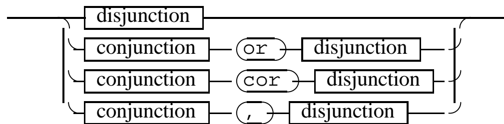
conditions



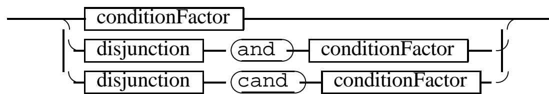
conditionExpr



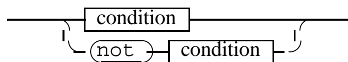
conjunction



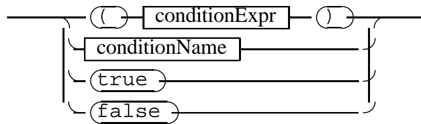
disjunction



conditionFactor

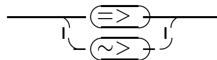


condition

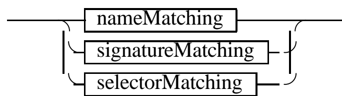


conditionName

exclOperator



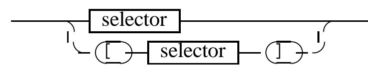
matchingPart



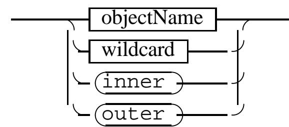
nameMatching

signatureMatching

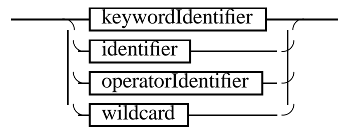
selectorMatching



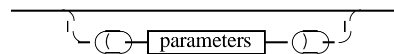
target



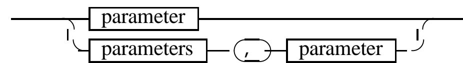
selector



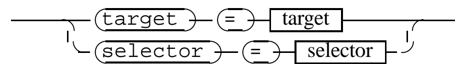
parameterPart



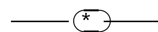
parameters



parameter



wildcard



Appendix B

Redefinition of the filter syntax

In this appendix we show the proposal we made, together with Maurice Glandrup, Coen Stuurman and Piet Koopmans for an enhanced syntax of the CFOM. We identified several problems in the Sina syntax and have found several a solution for them.

Problems with the substitution part

Within a filter there exist different meanings for the substitution part

The filters are initialized by the filter expression. The result of this filter expression is a boolean. Depending on the boolean value the filter type has a specified action. The filter expression consists of filter elements which are evaluated as a conditional or.
The syntax of a filter element is:

```
Condition => ('[' MatchingPart ']')OPT SubstitutionPart
```

For the different filter type, the meaning of the substitution part is:

- **Meta filter**: In the meta filter the substitution part is used to specify the message to be sent to the ACT. (this message will have the original message as a parameter).
- **Dispatch filter**: The dispatch filter uses the substitution part to replace the target-selector pair of the message. This will result in a dispatch to the modified target-selector.
- **Send filter**: Like the dispatch filter, the send filter also uses the substitution part to modify the target-selector pair.
- **Error filter**: As it is defined now, the error filter will use the substitution part in a same manner as the dispatch and send filter. When a message is accepted the target-selector pair is replaced and the message continues to the next filter. Only a rejection of the error filter will result that an error condition is raised.
- **Wait filter**: The wait filter ignores the substitution part. That is, it will not modify the target-selector pair.

This gives different meanings to the substitution part, depending on the filter type. It is preferable to have a uniform specification of the filter expression independent of the filter type.

Proposal for new syntax

We therefore propose to remove the dependencies out of the filter syntax. We propose to move the substitution functionality to the filters themselves.

If we use parameterized filters we can move the dependencies out of the standard filter expression into a parameter.

Next we can define a new filter which functionality is to replace the target-selector pair of the original message into a new target-selector pair and nothing else. This results in a filter called the *substitution filter*, where the new target-selector pair are provided as a parameter.

The other filter can be redefined in such a way that only the unnecessary substitution is removed from it and only the real functionality of the filter remains.
The functionality of the filters is now:

- **Substitution filter**: This filters only functionality is to replace the target-selector pair or the selector of a message. In case the message is not accepted it will be passes to the next filter.

- **Dispatch filter**: The dispatch filters functionality is to dispatch a target-selector pair. In a new parameterized form one can also specify a new target and selector. So the dispatch is extended with the substitution functionality. If no parameters are given to this filter, the matching expression will be used as a parameter as well.

- **Meta filter**: A meta filter needs the ACT as a parameter and a selector.

- **Error filter**: The only functionality of an error filter is to raise an error when a message is rejected. No parameters are needed for this filter. (Note: a future extension of this filter can be a parameter in the form of an error-level or error-handler)

- **Send filter**: A send filter will send an accepted message to the output filters of the enclosing object or to the input filters of the message's target object. Here no parameters are needed, since substitution can be done by the substitution filter.

- **Wait filter**: The functionality of the wait filter is to block a rejected message. It doesn't need any parameters.

- **Real-time filter**: For a real-time filter we could specify the timing constraints in the parameter part. The parameters of this filter need further research.

The proposed syntax is thus:

```
Substitute  = { Condition => Matching ( "Target-Selector Pair" ) }
Dispatch    = { Condition => Matching ( "Target-Selector Pair" ) }
Meta        = { Condition => Matching ( "Target-Selector Pair" ) }
Error       = { Condition => Matching () }
Send        = { Condition => Matching () }
Wait        = { Condition => Matching () }
Real-time   = { Condition => Matching ( "Timing constraint" ) }
```

Syntax for the parameter part of filter types

As stated in the previous sections we proposed a parameter part for a filter type. This part, surrounded by braces '()', consists of parameters separated by a ','. We have the following proposals for the parameter part:

- parameters are identifiers that represent a target, a selector, an ACT or a timing constraint with a number.
- parameters are of the form `ID '=' ID | Number`. In this way we can define parameters like `target = a`, `selector = b` or `minimumtime = 100`. A parameter like `target = a` for a dispatch filter can also be seen as sending a message '=' with an argument `a`.
- each filter has its own syntax-part for its parameters, which also includes a semantic check. This means we need a separate syntax for the parameters within the interface description syntax. The parameter syntax needs to have knowledge about the environment, so this needs to be passed along.

Condition syntax

In the Sina language there can only be one condition. When the condition is left out, *True* will be chosen as a default.
Our proposal is to allow conditional expressions in which the following operators are defined:

- "and", an unordered and
- "or", an unordered or
- "cand", a conditional and
- "cor", a conditional or
- "not", a monadic not
- "true"
- "false"
- balanced combination of braces '(' and ')'

Also we propose that the '=>' and '~>' are obligatory to indicate the division between condition part and matching part. This is because of some ambiguity in the syntax. Eventually this ambiguity can be solved semantically.

Matching

We have identified the following forms of matching expressions.

- *a single matching element*: this consists of a target and a selector or a selector only. The parameter part is optional.[1]
- *complex matching expression*: this consists of an arbitrary amount of single target-selector pairs in a matching expression in which the following operators are allowed: AND, OR, CAND, COR, NOT and balanced '(' ')'. This complex matching expression is surrounded by '{' and '}' and this is followed by an optional parameter part. Note that the optional parameter is the parameter for each target-selector pair and that they cannot have a parameter of their own.

Because *complex matching* probably causes a lot of problems and we are short of time, we focus on *single matching*.

Matching strategies

We have identified three matching strategies, they are:

- *signature match*: name match with signature match; this one is probably the one that is used the most, therefore we think it should be the default matching strategy. Since this is the default, it is denoted by the target selector pair.
- *name match*: the only purpose of this match is to do a name match on the target and/or selector. To choose for a name match, the target.selector pair is surrounded by '[' and ']'.
- *pure signature match*: this is a signature match, without a name match. The pure signature match is formed by placing '<' and '>' around the target selector pair.

In our assignment we will only deal with *signature matching* and *name matching*, for the same reason as we omit complex matching.

signature matching

$$
\begin{aligned}
a.b &\equiv (sel = b \wedge b \,\epsilon\, \sigma(a)) \\
a.* &\equiv (sel \,\epsilon\, \sigma(a)) \\
*.b &\equiv (sel = b \wedge b \,\epsilon\, \sigma(tar)) \\
. &\equiv (sel \,\epsilon\, \sigma(tar)) \\
b &\equiv [*.b] \equiv (sel = b)
\end{aligned}
$$

name matching

$$
\begin{aligned}
[a.b] &\equiv (sel = b \wedge tar = a) \\
[a.*] &\equiv (tar = a) \\
[*.b] &\equiv (sel = b) \\
[*.*] &\equiv \textbf{true} \\
[b] &\equiv [*.b] \equiv (sel = b)
\end{aligned}
$$

[1] Note that a single matching element has only one parameter part.

pure signature matching

$$
\begin{aligned}
\langle a.b\rangle &\equiv (b\epsilon\sigma(a))\\
\langle a.*\rangle &\equiv (*\epsilon\sigma(a)) \equiv \textbf{true}\\
\langle *.b\rangle &\equiv (b \;\epsilon\; \sigma(*)), \text{ where } * \text{ has the scope of the outer object}\\
\langle *.*\rangle &\equiv \textbf{true}\\
\langle b\rangle &\equiv \langle inner.b\rangle \equiv (b \;\epsilon\; \sigma(inner))
\end{aligned}
$$

We can use the pure signature match for the following. If we want to log messages using a Meta filter only if the method *log* is in the signature of the external *logger* we can define the following input filter:

```
externals
        logger:Logger;
inputfilters
        metalog : Meta = { => {*.* and <logger.log>}(logger.log) }
```

Change proposal for pseudo variable names

We propose that *self* is changed to *outer* and *sender* is changed to *client*. The default object when only the selector is given is changed to inner, because outer would introduce possible recursion.
The pseudo variable names are now:

- *inner*: by using inner the defined method is invoked directly; the message does not have to pass any filter.
- *outer*: a message sent to outer will be sent to the interface of the object executing the message expression. The message is then processed by the input filter of that object. [2]
- *client*: provides a reference to the sender of the message that is currently being processed.
- *server*: a message sent to server will cause the message to be sent to the receiver of the message. That is the original receiver of the message that caused the current execution.
- *message*: this pseudo variable refers to the message that caused the current execution

Language Dependent Part

We have identified the following language dependent parts of the composition filter interface syntax: *identifier* and *expression*. The identifier is used for class names, method names, selector name, etc. The expression is used for an object initialization and for the method argument part.

[2] Note that this message must be visible at the interface layer of the object

Selector identifier problem

Smalltalk defines methods uniquely as follows:

- *put*, a method without arguments
- +, a binary operator with two arguments
- *put:*, a method with one arguments
- *put:at:*, a methods with two arguments, note that the number of arguments and their names define the method name

In C++ methods are uniquely defined by the number of their arguments and the types of the arguments. And the '_' is a legal character in method names. C++ also allows operators like '+', ':=' to be overloaded.

As a selector in a filter expression is made up of characters only, the following two problem arise when we want to keep the syntax language independent.

- The ':' is part of a Smalltalk method and not of a C++ method.
- The operators of a language like '+', '*', '.' and ':=' are not a part of the current selector definition.

We propose the following changes to the selector part. To allow operators as selector we define an escape sequence to operator symbols. We can chose one of the following, where we think the first is the most pleasant in use:

- operator symbol are surrounded by quotes like '+' and ':='. A ' operator now has to be escaped with double quotes, but this is a case we think is very rare.
- we define the escape symbol as a backslash '\', which comes before each non-alpha character.
- we define a specific keyword operator for operator sequences.
- we could also define a mapping from `plus:` to +. This can be done by a preprocessor.

These three proposals are illustrated below. This shows an operator method '+' that is dispatched to inner.plus and an operator ':=' to inner.equal.

```
dispatch = { true => '+' (plus), true => ':=' (equal) }

dispatch = { true => \+ (plus), true => \:\= (equal) }

dispatch = { true => operator(+) (plus),
             true => operator(:=) (equal) }

map 'add' to '+'
map 'get' to ':='
dispatch = { true => add (plus), true => get (equal) }
```

For the ':' in Smalltalk methods and '_' in C++ methods we propose to make the syntax not language independent for selector name. Since programmers want to name their methods in a way they are used to and not with escape sequences or other difficult constructs.

Problems with the expression of an object initializer

An internal or external object can be initialized with an object initializer, consisting of expression separated by a ','. These expressions are language dependent and we propose to specify them in the form that is used for the target language. So in C++ an expression could be `2+3` and in Smalltalk it can be `2 plus: 3`.

We propose that the complete expression surrounded by '(' ')' is passed to the target language.

Method declaration and arguments

The method declaration is now defined as an *id* followed by an argument list between braces. In Smalltalk arguments have no type and are their names are part of the method-name.

Since the method names are redundant information, because they can be found in the implementation part of the target language, we propose that they are removed from the interface part.

Interface definition for a language

Since we only define an interface for a language like Smalltalk or C++ and not the implementation part we suggest that the **interface** keyword is removed from the interface definition. The declaration of a composition filter definition is then as follows:

```
class ...
     comment ...
        ;
     internals ...
        ;
     externals ...
        ;
     conditions ...
        ;
     methods ...
        ;
     inputfilters ...
        ;
     outputfilters ...
        ;
end
```

Argument signature or argument order

We note that we have identified the following problem, but haven't come with a proper solution yet. We will illustrate the problem with the following example. We have an

application that uses the *oldpoint* class to put point on the screen.

```
class oldpoint
methods
     put(x,y) returns nil ;
end
```

Assume we bought an new class library from another company that made a good improvement on the point drawing speed. The only difference between the *oldpoint* class and the *newpoint* class is the method name and the argument order.

```
class newpoint
methods
     place(y,x) returns nil ;
end
```

Suppose we want to dispatch the *oldpoint* method *put* to the new *place* method of a internal.

```
class oldpoint
internals
     N:newpoint;
methods
     put(x,y) returns nil ;
inputfilters
     disp: Dispatch = { *.put ( *.place) }
```

This gives a problem since the argument types are the same, but their context is different.

Problem with different types of arguments (i.e. the argument signatures) can be solved in C++ since there is explicit typing. But in Smalltalk an argument can be of any type.

We identified the need of some specification language for the transformation of methods. Some properties needed by these transformations are:

- filling up arguments, if the new methods has more arguments.
- changing the argument order.
- type transformations, if the types of arguments do not correspond.

A simple example of how this could be done is like this, if arguments are named and typed:

```
transformations
     put(x,y) -> place(y,x)
     begin
          place.arg[2] := put.arg[1] ;
          place.arg[1] := put.arg[2] ;
          // or we could use the argument names
          // place.x := put.x
          // place.y := put.y
     end
```

Or like this if we define the transformation on the number of arguments and not on the types of the arguments. Note that the number behind the method name represent the number of arguments:

```
transformations
     put#2 -> place#2
     begin
          place.arg[1] := put.arg[2] ;
          place.arg[2] := put.arg[1] ;
     end
```

Conclusion

The syntax for the interface part is a good basis for extending Object-Oriented Language like Smalltalk and C++ with the composition filters object model. But since most OO-languages have a very different syntax it is impossible to keep the interface syntax language independent without creating strange construct or introducing lots of new keywords.

We think that the syntax for the interface part must be clear and close to what a programmer knows of the target language. Therefore we propose to adapt the language dependent part for the target language of the compiler.

Bibliography

[Agha, 1986] G. Agha. An overview of actor languages. In *ACM SIGPLAN Notices*, volume 21:10, pages 58–67, October 1986.

[Aho *et al.*, 1986] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wessley, 1986.

[Aksit and Bergmans, 1992] Mehmet Aksit and Lodewijk Bergmans. *Obstacles in Object-Oriented Software Development*. PhD thesis, University of Twente, Enschede, the Netherlands, 1992.

[Aksit and Tripathi, 1988] M. Aksit and A. Tripathi. Data abstraction mechanisms in sina/st. In *Proceedings OOPSLA '88*, volume 23:11, pages 265–275. ACM SIGPLAN Notices, November 1988.

[Aksit *et al.*, 1992] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object oriented language-database integration model: The composition-filters approach. In *Proceedings ECOOP '92*, pages 372–395, Utrecht, the Netherlands, 1992. Springer Verlag.

[Aksit *et al.*, 1993] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In *Proceedings ECOOP '93*, pages 152–184, Kaiserslautern, Germany, 1993. Springer Verlag.

[Aksit *et al.*, 1994] Mehmet Aksit, Jan Bosch, William van der Sterren, and Lodewijk Bergmans. Real-time specification inheritance anomalies and real-time filters. In *Proceedings ECOOP '94*, pages 386–405, Bologna, Italy, 1994. Springer Verlag.

[Aksit, 1989] M. Aksit. *On the Design of the Object-Oriented Language Sina*. PhD thesis, University of Twente, Enschede, the Netherlands, 1989.

[Bergmans, 1994] Lodewijk Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, Enschede, the Netherlands, June 1994.

[DeMichiel and Gabriel, 1987] L. G. DeMichiel and R. P. Gabriel. The common lisp object system: An overview. In *Proceedings ECOOP '87*, pages 151–170, Paris, France, 1987. Springer Verlag.

[Goldberg and Robson, 1983] A. Goldberg and D. Robson. *Smalltalk 80: The Language and its implementation*. Addison Wesley, 1983.

[Graver, 1992] Justin O. Graver. *T-Gen User's Guide, version 2.1*, 1992.

[Hinkle *et al.*, 1993] Bob Hinkle, Vicki Jones, and Ralph E. Johnson. Debugging objects. *The Smalltalk Report*, 2(9), july–august 1993.

[Par, 1992] ParcPlace, Sunnyvale, California. *Objectworks\Smalltalk® 4.1 Tutorials*, 1992. Information at e-mail: info@ParcPlace.com.

[Pascoe, 1986] Geoffry A. Pascoe. Encapsulators: A new software paradigm in smalltalk-80. In *Proceedings OOPSLA '86*, pages 341–345, Productivity Products International, 1986.

[Pintado, 1993] Xavier Pintado. Gluons: a support for software component cooperation. In *Object Technologies for Advanced Software*, pages 43–60, Kanazawa, Japan, 1993. Springer Verlag.

[Stroustrup, 1986] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[Tripathi *et al.*, 1989] A. Tripathi, E. Berge, and M. Aksit. An implementation of the object-oriented concurrent programming language sina. *Software-Practice and Experience*, 19(3):235–256, March 1989.

[Ungar and Smith, 1987] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87*, volume 22:12, pages 227–242. ACM SIGPLAN Notices, December 1987.