Analysis and Redesign of the Compose* Language

A thesis submitted for the degree
of Master of Science at
the University of Twente

Ing. Dirk Doornenbal

Enschede, October 24, 2006

Graduation commission:
Prof. dr. ir. M. Akşit
Dr. ir. L.M.J. Bergmans
Ir. W.K. Havinga

Twente Research and Education
on Software Engineering
Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente

TRESE
Twente Research & Education
on Software Engineering

University of Twente
Enschede - The Netherlands

Abstract

Compose* is a research programming language, this means that the syntax changes more often then an industrial programming language. For each new feature an addition is made to the syntax and this has lead to the situation that the syntax has several redundant parts. In order to get the syntax streamlined again and to fix some major issues an analysis and redesign of the Compose* language is in place.

The analysis of the language will form the basis for the Compose* Annotated Reference Manual and will cover both syntax and semantics. The redesign will focus on increasing reusability and expressiveness while keeping the syntax as concise as possible.

Reusability of code written in Compose* will be improved with the introduction of filter module parameters. These can be compared with parameters as in an object-oriented programming language. The increase of expressiveness is achieved with replacing the old filter specification with a new canonical form.

Acknowledgments

Writing a thesis and analyzing the Compose* is not a small task. Luckily I had help and advice, for which I like to thank the following people.

First I would like to thank Lodewijk Bergmans, who entrusted me with the redesign of the Compose* language and helped me getting started with the analysis. Secondly my thanks go to Wilke Havinga, for his comments on my ideas for changing the language. I also want to thank Pascal Dürr for his technical support during my implementation efforts on Compose*.

Many thanks go to Olaf Conradi, Stephan Huttenhuis, Rolf Huisman, Johan te Winkel, and Michiel Hendriks for the great time, nice working environment, and the many discussions on Compose* and other subjects. I also want to thank the rest of the Compose* team for their comments on the language changes. And last, but certainly not least, thanks go to all the students I did not mention yet of the software engineering and formal methods lab for their advice and for making my time at the lab quite enjoyable.

Dirk Doornenbal
October 24, 2006
Enschede, The Netherlands

Reading guide

This chapter contains a short guide on the outline of this thesis. Due to the two separate tasks of the work, the thesis is split up into two parts: the analysis and the redesign.

The first two chapters contains the general information on aspect-oriented programming and Compose*. The third chapter contains the motivation for this thesis and the requirements for the redesign.

The first part of the thesis is on the changes of the Compose* language. It covers both major changes and some minor changes which are combined in chapter 6.

The second part covers the Annotated Reference Manual and its chapters contains the analysis of the language. This is the basic version of the manual and the port specific comments are left out.

The conclusion contains the summary of the results, related work and the work that lays ahead.

Contents

List of Figures

Chapter 1

Introduction to AOSD

The first two chapters have originally been written by seven M. Sc. students [Hol04, Dür04, Vin04, Bos04, Sta05, Hav05, Bos06] at the University of Twente. The chapters have been rewritten for use in the following theses [Oud06, Con06, Doo06, Spe06, Hut06, Hui06, Win06]. They serve as a general introduction into Aspect-Oriented Software Development and Compose* in particular.

1.1 Introduction

The goal of software engineering is to solve a problem by implementing a software system. The things of interest are called concerns. They exist at every level of the engineering process. A recurrent theme in engineering is that of modularization: separation and localization of concerns. The goal of modularization is to create maintainable and reusable software. A programming language is used to implement concerns.

Fifteen years ago the dominant programming language paradigm was procedural programming. This paradigm is characterized by the use of statements that update state variables. Examples are Algol-like languages such as Pascal, C, and Fortran.

Other programming paradigms are the functional, logic, object-oriented, and aspect-oriented paradigms. Figure 1.1 summarizes the dates and ancestry of several important languages [Wat90]. Every paradigm uses a different modularization mechanism for separating concerns into modules.

Functional languages try to solve problems without resorting to variables. These languages are entirely based on functions over lists and trees. Lisp and Miranda are examples of functional languages.

A logic language is based on a subset of mathematical logic. The computer is programmed to infer relationships between values, rather than to compute output values from input values. Prolog is currently the most used logic language [Wat90].

A shortcoming of procedural programming is that global variables can potentially be accessed

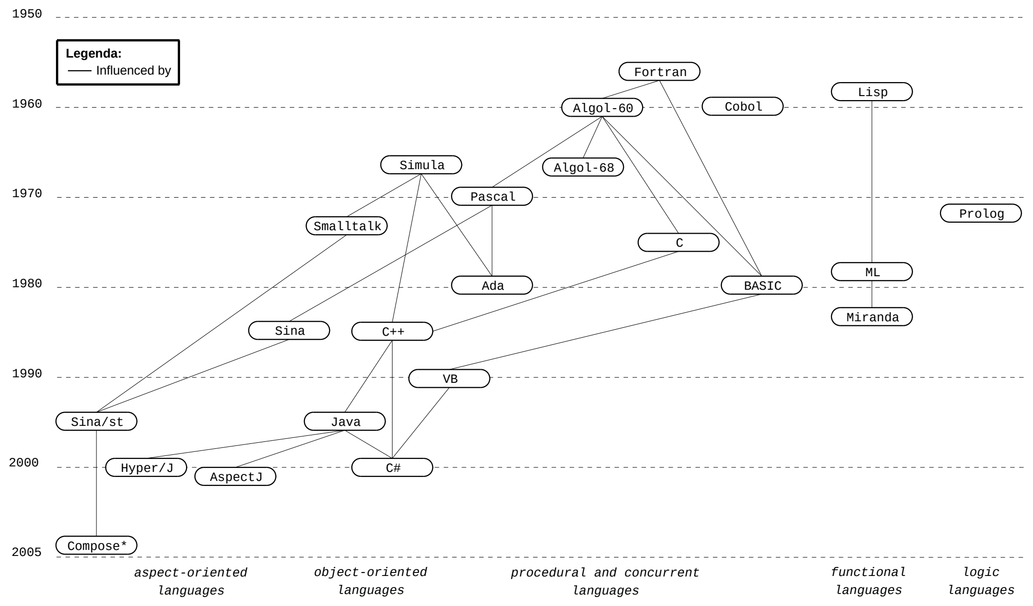

Figure 1.1: Dates and ancestry of several important languages

and updated by any part of the program. This can result in unmanageable programs because no module that accesses a global variable can be understood independently from other modules that also access that global variable.

The Object-Oriented Programming (OOP) paradigm improves modularity by encapsulating data with methods inside objects. The data may only be accessed indirectly, by calling the associated methods. Although the concept appeared in the seventies, it took twenty years to become popular [Wat90]. The most well known object-oriented languages are C++, Java, C#, and Smalltalk.

The hard part about object-oriented design is decomposing a system into objects. The task is difficult because many factors come into play: encapsulation, granularity, dependency, adaptability, reusability, and others. They all influence the decomposition, often in conflicting ways [GHJV95].

Existing modularization mechanisms typically support only a small set of decompositions and usually only a single dominant modularization at a time. This is known as the tyranny of the dominant decomposition [TOSH05]. A specific decomposition limits the ability to implement other concerns in a modular way. For example, OOP modularizes concerns in classes and only fixed relations are possible. Implementing a concern in a class might prevent another concern from being implemented as a class.

Aspect-Oriented Programming (AOP) is a paradigm that solves this problem. AOP is commonly used in combination with OOP but can be applied to other paradigms as well. The following sections introduce an example to demonstrate the problems that may arise with OOP and show how AOP can solve this. Finally, we look at three particular AOP methodologies in more detail.

1.2 Traditional Approach

Consider an application containing an object `Add` and an object `CalcDisplay`. `Add` inherits from the abstract class `Calculation` and implements its method `execute(a, b)`. It performs the addition of two integers. `CalcDisplay` receives an update from `Add` if a calculation is finished and prints the result to screen. Suppose all method calls need to be traced. The objects use a `Tracer` object to write messages about the program execution to screen. This is implemented by a method called `write`. Three concerns can be recognized: addition, display, and tracing. The implementation might look something like Listing 1.1.

From our example, we recognize two forms of crosscutting: *code tangling* and *code scattering.*

The addition and display concerns are implemented in classes `Add` and `CalcDisplay` respectively. Tracing is implemented in the class `Tracer`, but also contains code in the other two classes (lines 5, 10, 15, and 22 in (a) and 2, 5, and 10 in (b)). If a concern is implemented across several classes it is said to be scattered. In the example of Listing 1.1 the tracing concern is scattered.

Usually a scattered concern involves code *replication.* That is, the same code is implemented a number of times. In our example the classes `Add` and `CalcDisplay` contain similar tracing code.

In class `Add` the code for the addition and tracing concerns are intermixed. In class `CalcDisplay` the code for the display and tracing concerns are intermixed. If more then one concern is implemented in a single class they are said to be tangled. In our example the addition and tracing concerns are tangled. Also display and tracing concerns are tangled. Crosscutting code has the following consequences:

Code is difficult to change
Changing a scattered concern requires us to modify the code in several places. Making modifications to a tangled concern class requires checking for side-effects with all existing crosscutting concerns;

Code is harder to reuse
To reuse an object in another system, it is necessary to either remove the tracing code or reuse the (same) tracer object in the new system;

Code is harder to understand
Tangled code makes it difficult to see which code belongs to which concern.

1.3 AOP Approach

To solve the problems with crosscutting, several techniques are being researched that attempt to increase the expressiveness of the OO paradigm. Aspect-Oriented Programming (AOP) introduces a modular structure, the aspect, to capture the location and behavior of crosscutting concerns. Examples of Aspect-Oriented languages are Sina, AspectJ, Hyper/J, and Compose*. A special syntax is used to specify aspects and the way in which they are combined with regular objects. The fundamental goals of AOP are twofold [GL03]: first to provide a mechanism to express concerns that crosscut other components. Second to use this description to allow for the separation of concerns.

```java
public class Add extends Calculation{

  private int result;
  private CalcDisplay calcDisplay;
  private Tracer trace;

  Add(){
    result = 0;
    calcDisplay = new CalcDisplay();
    trace = new Tracer();
  }

  public void execute(int a, int b){
    trace.write
      ("void Add.execute(int, int)");
    result = a + b;
    calcDisplay.update(result);
  }

  public int getLastResult(){
    trace.write("int
      Add.getLastResult()");
    return result;
  }
}
```

(a) Addition

```java
public class CalcDisplay{
  private Tracer trace;

  public CalcDisplay(){
    trace = new Tracer();
  }

  public void update(int value){
    trace.write("void
      CalcDisplay.update(int)");
    System.out.println("Printing
      new value of calculation: "+value);
  }
}
```

(b) CalcDisplay

Listing 1.1: Modeling addition, display, and logging without using aspects

```
public class Add extends Calculation{
  private int result;
  private CalcDisplay calcDisplay;

  Add(){
    result = 0;
    calcDisplay = new CalcDisplay();
  }

  public void execute(int a, int b){
    result = a + b;
    calcDisplay.update(result);
  }

  public int getLastResult(){
    return result;
  }
}
```

(a) Addition concern

```
aspect Tracing{
  Tracer trace = new Tracer();

  pointcut tracedCalls():
    call(* (Calculation+).*(..)) ||
    call(* CalcDisplay.*(..));

  before(): tracedCalls(){
    trace.write(
      thisJoinPoint.getSignature().toString()
    );
  }
}
```

(b) Tracing concern

Listing 1.2: Modeling addition, display, and logging with aspects

Join points are well-defined places in the structure or execution flow of a program where additional behavior can be attached. The most common join points are method calls. *Pointcuts* describe a set of join points. This allows us to execute behavior at many places in a program by one expression. *Advice* is the behavior executed at a join point.

In the example of Listing 1.2 the class `Add` does not contain any tracing code and only implements the addition concern. Class `CalcDisplay` also does not contain tracing code. In our example the tracing aspect contains all the tracing code. The pointcut `tracedCalls` specifies at which locations tracing code is executed.

The crosscutting concern is explicitly captured in aspects instead of being embedded within the code of other objects. This has several advantages over the previous code.

Aspect code can be changed
Changing aspect code does not influence other concerns;

Aspect code can be reused
The coupling of aspects is done by defining pointcuts. In theory, this low coupling allows for reuse. In practice reuse is still difficult;

Aspect code is easier to understand
A concern can be understood independent of other concerns;

Aspect pluggability
Enabling or disabling concerns becomes possible.

1.3.1 AOP Composition

AOP composition can be either symmetric or asymmetric. In the symmetric approach every component can be composed with any other component. This approach is followed by e.g. Hyper/J.

In the asymmetric approach, the base program and aspects are distinguished. The base program is composed with the aspects. This approach is followed by e.g. AspectJ (covered in more detail in the next section).

1.3.2 Aspect Weaving

The integration of components and aspects is called *aspect weaving*. There are three approaches to aspect weaving. The first and second approach rely on adding behavior in the program, either by weaving the aspect in the source code, or by weaving directly in the target language. The target language can be intermediate language (IL) or machine code. Examples of IL are Java byte code and Common Intermediate Language (CIL). The remainder of this chapter considers only intermediate language targets. The third approach relies on adapting the virtual machine. Each method is explained briefly in the following sections.

1.3.2.1 Source Code Weaving

The source code weaver combines the original source with aspect code. It interprets the defined aspects and combines them with the original source, generating input for the native compiler. For the native compiler there is no difference between source code with and without aspects. Hereafter the compiler generates an intermediate or machine language output (depending on the compiler-type).

The advantages of using source code weaving are:

High-level source modification
Since all modifications are done at source code level, there is no need to know the target (output) language of the native compiler;

Aspect and original source optimization
First the aspects are woven into the source code and hereafter compiled by the native compiler. The produced target language has all the benefits of the native compiler optimization passes. However, optimizations specific to exploiting aspect knowledge are not possible;

Native compiler portability
The native compiler can be replaced by any other compiler as long as it has the same input language. Replacing the compiler with a newer version or another target language can be done with little or no modification to the aspect weaver.

However, the drawbacks of source code weaving are:

Language dependency
Source code weaving is written explicitly for the syntax of the input language;

Limited expressiveness
Aspects are limited to the expressive power of the source language. For example, when using source code weaving, it is not possible to add multiple inheritance to a single inheritance language.

1.3.2.2 Intermediate Language Weaving

Weaving aspects through an intermediate language gives more control over the executable program and solves some issues as identified in section 1.3.2.1 on source code weaving. Weaving at this level allows for creating combinations of intermediate language constructs that can not be expressed at the source code level. Although IL can be hard to understand, IL weaving has several advantages over source code weaving:

Programming language independence
All compilers generating the target IL output can be used;

More expressiveness
It is possible to create IL constructs that are not possible in the original programming language;

Source code independence
Can add aspects to programs and libraries without using the source code (which may not be available);

Adding aspects at load- or runtime
A special class loader or runtime environment can decide and do dynamic weaving. The aspect weaver adds a runtime environment into the program. How and when aspects can be added to the program depend on the implementation of the runtime environment.

However, IL weaving also has drawbacks that do not exist for source code weaving:

Hard to understand
Specific knowledge about the IL is needed;

More error-prone
Compiler optimization may cause unexpected results. Compiler can remove code that breaks the attached aspect (e.g. inlining of methods).

1.3.2.3 Adapting the Virtual Machine

Adapting the virtual machine (VM) removes the need to weave aspects. This technique has the same advantages as intermediate language weaving and can also overcome some of its disadvantages as mentioned in section 1.3.2.2. Aspects can be added without recompilation, redeployment, and restart of the application [PGA02, PAG03].

Modifying the virtual machine also has its disadvantages:

Dependency on adapted virtual machines
Using an adapted virtual machine requires that every system should be upgraded to that version;

Virtual machine optimization
People have spend a lot of time optimizing virtual machines. By modifying the virtual machine these optimizations should be revisited. Reintegrating changes introduced by newer versions of the original virtual machine, might have substantial impact.

1.4 AOP Solutions

As the concept of AOP has been embraced as a useful extension to classic programming, different AOP solutions have been developed. Each solution has one or more implementations to demonstrate how the solution is to be used. As described by [EFB01] these differ primarily in:

How aspects are specified
Each technique uses its own aspect language to describe the concerns;

Composition mechanism
Each technique provides its own composition mechanisms;

Implementation mechanism
Whether components are determined statically at compile time or dynamically at run time, the support for verification of compositions, and the type of weaving;

Use of decoupling
Should the writer of the main code be aware that aspects are applied to his code;

Supported software processes
The overall process, techniques for reusability, analyzing aspect performance of aspects, is it possible to monitor performance, and is it possible to debug the aspects.

This section will give a short introduction to AspectJ [KHH+01] and Hyperspaces [OT01], which together with Composition Filters [BA01] are three main AOP approaches.

1.4.1 AspectJ Approach

AspectJ [KHH+01] is an aspect-oriented extension to the Java programming language. It is probably the most popular approach to AOP at the moment, and it is finding its way into the industrial software development. AspectJ has been developed by Gregor Kiczales at Xerox's PARC (Palo Alto Research Center). To encourage the growth of the AspectJ technology and community, PARC transferred AspectJ to an open Eclipse project. The popularity of AspectJ comes partly from the various extensions based on it, build by several research groups. There are various projects that are porting AspectJ to other languages, resulting in tools such as AspectR and AspectC.

One of the main goals in the design of AspectJ is to make it a compatible extension to Java. AspectJ tries to be compatible in four ways:

Upward compatibility
All legal Java programs must be legal AspectJ programs;

Platform compatibility
All legal AspectJ programs must run on standard Java virtual machines;

Tool compatibility
It must be possible to extend existing tools to support AspectJ in a natural way; this includes IDEs, documentation tools and design tools;

Programmer compatibility
Programming with AspectJ must feel like a natural extension of programming with Java.

AspectJ extends Java with support for two kinds of crosscutting functionality. The first allows defining additional behavior to run at certain well-defined points in the execution of the program

```
aspect DynamicCrosscuttingExample {
  Log log = new Log();

  pointcut traceMethods():
    execution(edu.utwente.trese.*.*(..));

  before() : traceMethods {
    log.write("Entering " + thisJointPoint.getSignature());
  }

  after() : traceMethods {
    log.write("Exiting " + thisJointPoint.getSignature());
  }
}
```

Listing 1.3: Example of dynamic crosscutting in AspectJ

and is called the *dynamic crosscutting mechanism*. The other is called the *static crosscutting mechanism* and allows modifying the static structure of classes (methods and relationships between classes). The units of crosscutting implementation are called aspects. An example of an aspect specified in AspectJ is shown in Listing 1.3.

The points in the execution of a program where the crosscutting behavior is inserted are called *join points*. A *pointcut* has a set of join points. In Listing 1.3 is `traceMethods` an example of a pointcut definition. The pointcut includes all executions of any method that is in a class contained by package `edu.utwente.trese`.

The code that should execute at a given join point is declared in an advice. Advice is a method-like code body associated with a certain pointcut. AspectJ supports *before*, *after* and *around* advice that specifies where the additional code is to be inserted. In the example both before and after advice are declared to run at the join points specified by the `traceMethods` pointcut.

Aspects can contain anything permitted in class declarations including definitions of pointcuts, advice and static crosscutting. For example, static crosscutting allows a programmer to add fields and methods to certain classes as shown in Listing 1.4.

The shown construct is called inter-type member declaration and adds a method `trace` to class `Log`. Other forms of inter-type declarations allow developers to declare the parents of classes (superclasses and realized interfaces), declare where exceptions need to be thrown, and allow a developer to define the precedence among aspects.

With its variety of possibilities AspectJ can be considered a useful approach for realizing software requirements.

```
aspect StaticCrosscuttingExample {
  private int Log.trace(String traceMsg) {
    Log.write(" --- MARK --- " + traceMsg);
  }
}
```

Listing 1.4: Example of static crosscutting in AspectJ

```
Hyperspace Pacman
  class edu.utwente.trese.pacman.*;
```

Listing 1.5: Creation of a hyperspace

1.4.2 Hyperspaces Approach

The *Hyperspaces* approach is developed by H. Ossher and P. Tarr at the IBM T.J. Watson Research Center. The Hyperspaces approach adopts the principle of multi-dimensional separation of concerns [OT01], which involves:

- Multiple, arbitrary dimensions of concerns;
- Simultaneous separation along these dimensions;
- Ability to dynamically handle new concerns and new dimensions of concern as they arise throughout the software life cycle;
- Overlapping and interacting concerns. It is appealing to think of many concerns as independent or orthogonal, but they rarely are in practice.

We explain the Hyperspaces approach by an example written in the *Hyper/J* language. Hyper/J is an implementation of the Hyperspaces approach for Java. It provides the ability to identify concerns, specify modules in terms of those concerns, and synthesize systems and components by integrating those modules. Hyper/J uses bytecode weaving on binary Java class files and generates new class files to be used for execution. Although the Hyper/J project seems abandoned and there has not been any update in the code or documentation for a while, we still mention it because the Hyperspaces approach offers a unique AOP solution.

As a first step, developers create hyperspaces by specifying a set of Java class files that contain the code units that populate the hyperspace. To do this is, you create a hyperspace specification, as demonstrated in Listing 1.5.

Hyper/J will automatically create a hyperspace with one dimension—the class file dimension. A dimension of concern is a set of concerns that are disjoint. The initial hyperspace will contain all units within the specified package. To create a new dimension you can specify concern mappings, which describe how existing units in the hyperspace relate to concerns in that dimension, as demonstrated in Listing 1.6.

The first line indicates that, by default, all of the units contained within the package `edu.utwente.trese.pacman` address the kernel concern of the feature dimension. The other mappings specify that any method named `trace` or `debug` address the logging and debugging concern respectively. These later mappings override the first one.

Hypermodules are based on concerns and consist of two parts. The first part specifies a set of hyperslices in terms of the concerns identified in the concern matrix. The second part spec-

```
package edu.utwente.trese.pacman: Feature.Kernel
operation trace: Feature.Logging
operation debug: Feature.Debugging
```

Listing 1.6: Specification of concern mappings

```
hypermodule Pacman_Without_Debugging
  hyperslices: Feature.Kernel, Feature.Logging;
  relationships: mergeByName;
end hypermodule;
```

Listing 1.7: Defining a hypermodule

ifies the integration relationships between the hyperslices. A hyperspace can contain several hypermodules realizing different modularizations of the same units. Systems can be composed in many ways from these hypermodules.

Listing 1.7 shows a hypermodule with two concerns, kernel and logging. They are related by a `mergeByName` integration relationship. This means that units in the different concerns correspond if they have the same name (`ByName`) and that these corresponding units are to be combined (`merge`). For example, all members of the corresponding classes are brought together into the composed class. The hypermodule results in a hyperslice that contains all the classes without the debugging feature; thus no `debug` methods will be present.

The most important feature of the hyperspaces approach is the support for on-demand remodularisation: the ability to extract hyperslices to encapsulate concerns that were not separated in the original code. Which makes hyperspaces especially useful for evolution of existing software.

1.4.3 Composition Filters

Composition Filters is developed by M. Akşit and L. Bergmans at the TRESE group, which is a part of the Department of Computer Science of the University of Twente, The Netherlands. The composition filters (CF) model predates aspect-oriented programming. It started out as an extension to the object-oriented model and evolved into an aspect-oriented model. The current implementation of CF is Compose*, which covers .NET, Java, and C.

One of the key elements of CF is the *message*, a message is the interaction between objects, for instance a method call. In object-oriented programming the message is considered an abstract concept. In the implementations of CF it is therefore necessary to reify the message. This *reified message* contains properties, like where it is send to and where it came from.

The concept of CF is that messages that enter and exit an object can be intercepted and manipulated, modifying the original flow of the message. To do so, a layer called the *interface part* is introduced in the CF model, this layer can have several properties. The interface part can be placed on an object, which behavior needs to be altered, and this object is referred to as *inner*.

There are three key elements in CF: messages, filters, and superimposition. Messages are sent from one object to another, if there is an interface part placed on the receiver, then the message that is sent goes through the input filters. In the filters the message can be manipulated before it reaches the inner part, the message can even be sent to another object. How the message will be handled depends on the filter type. An output filter is similar to an input filter, the only difference is that it manipulates messages that originate from the inner part. The latest addition to CF is superimposition, which is used to specify which interfaces needs to be superimposed on which inner objects.

Chapter 2

Compose*

Compose* is an implementation of the composition filters approach. There are three target environments: .NET, Java, and C. This chapter is organized as follows, first the evolution of Composition Filters and its implementations are described, followed by an explanation of the Compose* language and a demonstrating example. In the third section the Compose* architecture is explained, followed by a description of the features specific to Compose*.

2.1 Evolution of Composition Filters

Compose* is the result of many years of research and experimentation. The following time line gives an overview of what has been done in the years before and during the Compose* project.

1985 The first version of Sina is developed by Mehmet Akşit. This version of Sina contains a preliminary version of the composition filters concept called semantic networks. The semantic network construction serves as an extension to objects, such as classes, messages, or instances. These objects can be configured to form other objects such as classes from which instances can be created. The object manager takes care of synchronization and message processing of an object. The semantic network construction can express key concepts like delegation, reflection, and synchronization [Koo95].

1987 Together with Anand Tripathi of the University of Minnesota the Sina language is further developed. The semantic network approach is replaced by declarative specifications and the interface predicate construct is added.

1991 The interface predicates are replaced by the dispatch filter, and the wait filter manages the synchronization functions of the object manager. Message reflection and real-time specifications are handled by the meta filter and the real-time filter [Ber94].

1995 The Sina language with Composition Filters is implemented using Smalltalk [Koo95]. The implementation supports most of the filter types. In the same year, a preprocessor providing C++ with support for Composition Filters is implemented [Gla95].

1999 The composition filters language ComposeJ [Wic99] is developed and implemented. The implementation consists of a preprocessor capable of translating composition filter specifications into the Java language.

2001 ConcernJ is implemented as part of a M. Sc. thesis [Sal01]. ConcernJ adds the notion of superimposition to Composition Filters. This allows for reuse of the filter modules and to facilitate crosscutting concerns.

2003 The start of the Compose* project, the project is described in further detail in this chapter.

2004 The first release of Compose*, based on .NET.

2005 The start of the Java port of Compose*.

2006 Porting Compose* to C is started.

2.2 Composition Filters in Compose*

A Compose* application consists of concerns that can be divided in three parts: filter module specifications, superimposition, and implementation. A filter module contains the filter logic to filter on incoming or outgoing messages on objects it is superimposed on. Messages have a target, which is an object reference, and a selector, which is a method name. A superimposition part specifies which filter modules, annotations, conditions, and methods are superimposed on which objects. An implementation part contains the class implementation of a concern. How these parts are placed in a concern is shown in Listing 2.1.

```
concern {
  filtermodule {
    internals
    externals
    conditions
    inputfilters
    outputfilters
  }

  superimposition {
    selectors
    filtermodules
    annotations
    constraints
  }

  implementation
}
```

Listing 2.1: Abstract concern template

The working of a filter module is depicted in Figure 2.1. A filter module can contain input and output filters. The difference between these two sets of filters is that the first is used to filter on incoming messages, while the second is used to filter on outgoing messages. The return of a method is not considered an outgoing message. A filter has three parts: a filter identifier, a filter type, and one or more filter elements. A filter element exists out of an optional condition part, a matching part, and a substitution part. These parts are shown below:

$$\overbrace{stalker\_filter}^{identifier} : \overbrace{Dispatch}^{filter\ type} \ = \ \{\overbrace{!pacmanIsEvil}^{condition\ part} =>$$

$$\overbrace{[*.getNextMove]}^{matching\ part} \ \overbrace{stalk\_strategy.getNextMove}^{substitution\ part} \ \}$$

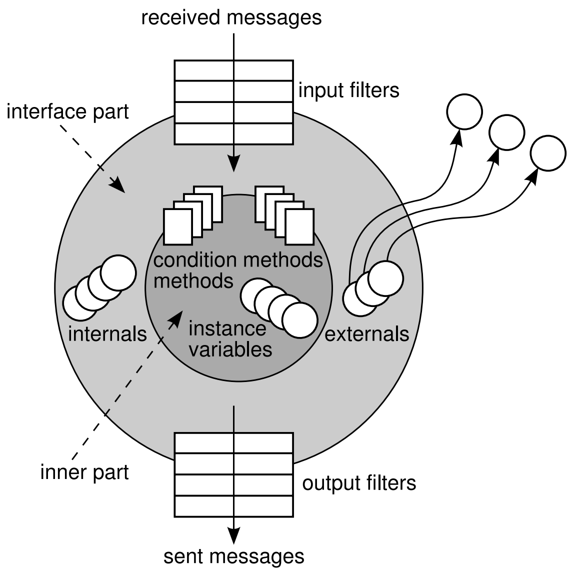

Figure 2.1: Components of the composition filters model

A filter identifier is a unique name for a filter in a filter module. Filters match when both the condition part and the matching part evaluate to true. In the demonstrated filter, every message where the selector is `getNextMove` matches. If an asterisk (`*`) is used in the target, every target will match. When the condition part and the matching part are true, the message is substituted with the values provided in the substitution part. How these values are substituted and how the message continues, depends on the used type of filter. At the moment there are four basic filter types defined in Compose*. It is also possible to write custom filter types.

Dispatch If the message is accepted, it is dispatched to the specified target of the message, otherwise the message continues to the subsequent filter. This filter type can only be used for input filters;

Send If the message is accepted, it is sent to the specified target of the message, otherwise the message continues to the subsequent filter. This filter type can only be used for output filters;

Error If the filter rejects the message, it raises an exception, otherwise the message continues to the next filter in the set;

Meta If the message is accepted, the message is sent as a parameter of another meta message to an internal or external object, otherwise the message just continues to the next filter. The object that receives the meta message can observe and manipulate the message and can re-activate the execution of the message.

The identifier `pacmanIsEvil`, used in the condition part, must be declared in the conditions section of a filter module. Targets that are used in a filter can be declared as internal or external. An internal is an object that is unique for each instance of a filter module, while an external is an object that is shared between filter modules.

Filter modules are superimposed on classes using filter module binding, which specifies a selection of objects on the one side and a filter module on the other side. The selection is specified in a selector definition. This selector definition uses predicates to select objects, such as `isClassWithNameInList`, `isNamespaceWithName`, and `namespaceHasClass`. In addition to filter modules, it is possible to bind conditions, methods, and annotations to classes using superimposition.

The last part of the concern is the implementation part, which can be used to define the behavior of a concern. For a logging concern, for example, we can define specific log functions.

2.3 Demonstrating Example

To illustrate the Compose* toolset, this section introduces a *Pacman* example. The Pacman game is a classic arcade game in which the user, represented by pacman, moves in a maze to eat vitamins. Meanwhile, a number of ghosts try to catch and eat pacman. There are, however, four mega vitamins in the maze that make pacman evil. In its evil state, pacman can eat ghosts. A simple list of requirements for the Pacman game is briefly discussed here:

- The number of lives taken from pacman when eaten by a ghost;
- A game should end when pacman has no more lives;
- The score of a game should increase when pacman eats a vitamin or a ghost;
- A user should be able to use a keyboard to move pacman around the maze;
- Ghosts should know whether pacman is evil or not;
- Ghosts should know where pacman is located;
- Ghosts should, depending on the state of pacman, hunt or flee from pacman.

2.3.1 Initial Object-Oriented Design

Figure 2.2 shows an initial object-oriented design for the Pacman game. Note that this UML class diagram does not show the trivial accessors. The classes in this diagram are:

Game
: This class encapsulates the control flow and controls the state of a game;

Ghost
: This class is a representation of a ghost chasing pacman. Its main attribute is a property that indicates whether it is scared or not (depending on the evil state of pacman);

GhostView
: This class is responsible for painting ghosts;

Glyph
: This is the superclass of all mobile objects (pacman and ghosts). It contains common information like direction and speed;

Keyboard
: This class accepts all keyboard input and makes it available to pacman;

Main
: This is the entry point of a game;

Pacman
: This is a representation of the user controlled element in the game. Its main attribute is a property that indicates whether pacman is evil or not;

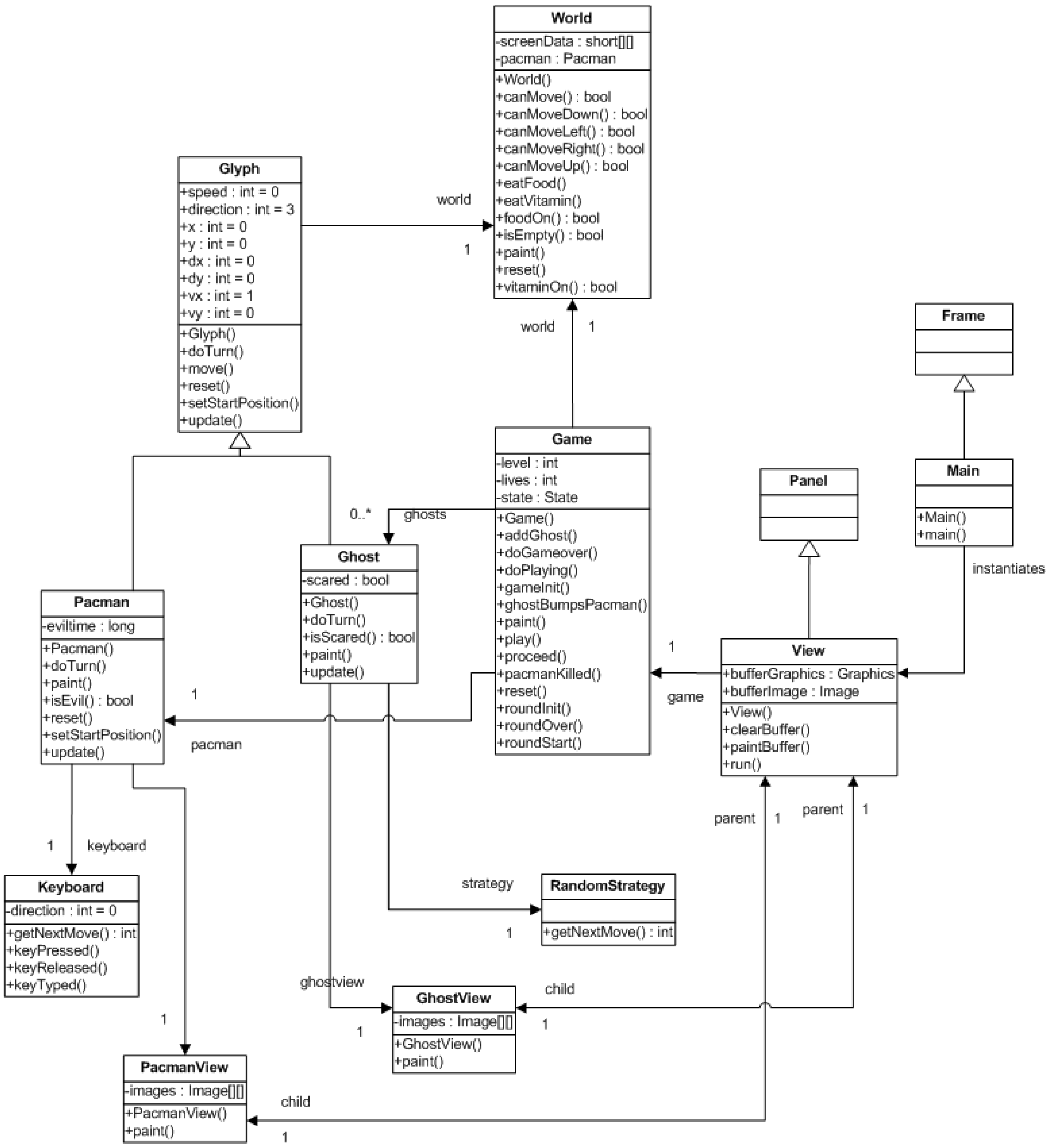

Figure 2.2: UML class diagram of the object-oriented Pacman game

PacmanView
: This class is responsible for painting pacman;

RandomStrategy
: By using this strategy, ghosts move in random directions;

View
: This class is responsible for painting a maze;

World
: This class has all the information about a maze. It knows where the vitamins, mega vitamins and most importantly the walls are. Every class derived from class `Glyph` checks whether movement in the desired direction is possible.

2.3.2 Completing the Pacman Example

The initial object-oriented design, described in the previous section, does not implement all the stated system requirements. The missing requirements are:

- The application does not maintain a score for the user;
- Ghosts move in random directions instead of chasing or fleeing from pacman.

In the next sections, we describe why and how to implement these requirements in the Compose* language.

2.3.2.1 Implementation of Scoring

The first system requirement that we need to add to the existing Pacman game is scoring. This concern involves a number of events. First, the score should be set to zero when a game starts. Second, the score should be updated whenever pacman eats a vitamin, mega vitamin or ghost. And finally, the score itself has to be painted on the maze canvas to relay it back to the user. These events scatter over multiple classes: `Game` (initializing score), `World` (updating score), `Main` (painting score). Thus scoring is an example of a crosscutting concern.

To implement scoring in the Compose* language, we divide the implementation into two parts. The first part is a Compose* concern definition stating which filter modules to superimpose. Listing 2.2 shows an example Compose* concern definition of scoring.

This concern definition is called `DynamicScoring` (line 1) and contains two parts. The first part is the declaration of a filter module called `dynamicscoring` (lines 2–11). This filter module contains one *meta filter* called `score_filter` (line 6). This filter intercepts five relevant calls and sends the message in a reified form to an instance of class `Score`. The final part of the concern definition is the superimposition part (lines 12–18). This part defines that the filter module `dynamicscoring` is to be superimposed on the classes `World`, `Game` and `Main`.

The final part of the scoring concern is the so-called *implementation part*. This part is defined by a class `Score`. Listing 2.3 shows an example implementation of class `Score`. Instances of this class receive the messages sent by `score_filter` and subsequently perform the events related to the scoring concern. In this way, all scoring events are encapsulated in one class and one Compose* concern definition.

```
concern DynamicScoring in pacman {
  filtermodule dynamicscoring {
    externals
      score : pacman.Score = pacman.Score.instance();
    inputfilters
      score_filter : Meta = {[*.eatFood] score.eatFood,
                             [*.eatGhost] score.eatGhost,
                             [*.eatVitamin] score.eatVitamin,
                             [*.gameInit] score.initScore,
                             [*.setForeground] score.setupLabel}
  }
  superimposition {
    selectors
      scoring = { C | isClassWithNameInList(C, ['pacman.World',
                                 'pacman.Game', 'pacman.Main']) };
    filtermodules
      scoring <- dynamicscoring;
  }
}
```

Listing 2.2: `DynamicScoring` concern in Compose*

2.3.2.2 Implementation of Dynamic Strategy

The last system requirement that we need to implement is the dynamic strategy of ghosts. This means that a ghost should, depending on the state of pacman, hunt or flee from pacman. We can implement this concern by using the strategy design pattern. However, in this way, we need to modify the existing code. This is not the case when we use Compose* *dispatch filters*. Listing 2.4 demonstrates this.

This concern uses *dispatch* filters to intercept calls to method `RandomStrategy.getNextMove` and redirect them to either `StalkerStrategy.getNextMove` or `FleeStrategy.getNextMove`. If pacman is not evil, the intercepted call matches the first filter, which dispatches the intercepted call to method `StalkerStrategy.getNextMove` (line 9). Otherwise, the intercepted call matches the second filter, which dispatches the intercepted call to method `FleeStrategy.getNextMove` (line 11).

2.4 Compose* Architecture

An overview of the Compose* architecture is illustrated in Figure 2.3. The Compose* architecture can be divided in four layers [Nag06]: IDE, compile-time, adaptation, and run-time.

2.4.1 Integrated Development Environment

One of the purposes of an integrated development environment (IDE) layer is to provide an interface to the native IDE and create a build configuration. A build configuration specifies which source files and settings are required to build an application with Compose*.

A build configuration can be created manually or by using a plug-in. Examples of these plug-ins are the Visual Studio add-in for Compose*/.NET and the Eclipse plug-in for Compose*/J and

```java
import Composestar.Runtime.FLIRT.message.*;
import java.awt.*;

public class Score
{
  private int score = -100;
  private static Score theScore = null;
  private Label label = new java.awt.Label("Score: 0");

  private Score() {}

  public static Score instance() {
    if(theScore == null) {
      theScore = new Score();
    }
    return theScore;
  }

  public void initScore(ReifiedMessage rm) {
    this.score = 0;
    label.setText("Score: "+score);
  }

  public void eatGhost(ReifiedMessage rm) {
    score += 25;
    label.setText("Score: "+score);
  }

  public void eatVitamin(ReifiedMessage rm) {
    score += 15;
    label.setText("Score: "+score);
  }

  public void eatFood(ReifiedMessage rm) {
    score += 5;
    label.setText("Score: "+score);
  }

  public void setupLabel(ReifiedMessage rm) {
    rm.proceed();
    label = new Label("Score: 0");
    label.setSize(15*View.BLOCKSIZE+20,15*View.BLOCKSIZE);
    Main main = (Main)Composestar.Runtime.FLIRT.message.MessageInfo
                                .getMessageInfo().getTarget();
    main.add(label,BorderLayout.SOUTH);
  }
}
```

Listing 2.3: Implementation of class `Score`

```
concern DynamicStrategy in pacman {
  filtermodule dynamicstrategy {
    internals
      stalk_strategy : pacman.Strategies.StalkerStrategy;
      flee_strategy : pacman.Strategies.FleeStrategy;
    conditions
      pacmanIsEvil : pacman.Pacman.isEvil();
    inputfilters
      stalker_filter : Dispatch = {!pacmanIsEvil =>
                          [*.getNextMove] stalk_strategy.getNextMove};
      flee_filter : Dispatch = {
                          [*.getNextMove] flee_strategy.getNextMove}
  }
  superimposition {
    selectors
      random = { C | isClassWithName(C,
                          'pacman.Strategies.RandomStrategy') };
    filtermodules
      random <- dynamicstrategy;
  }
}
```

Listing 2.4: `DynamicStrategy` concern in Compose*

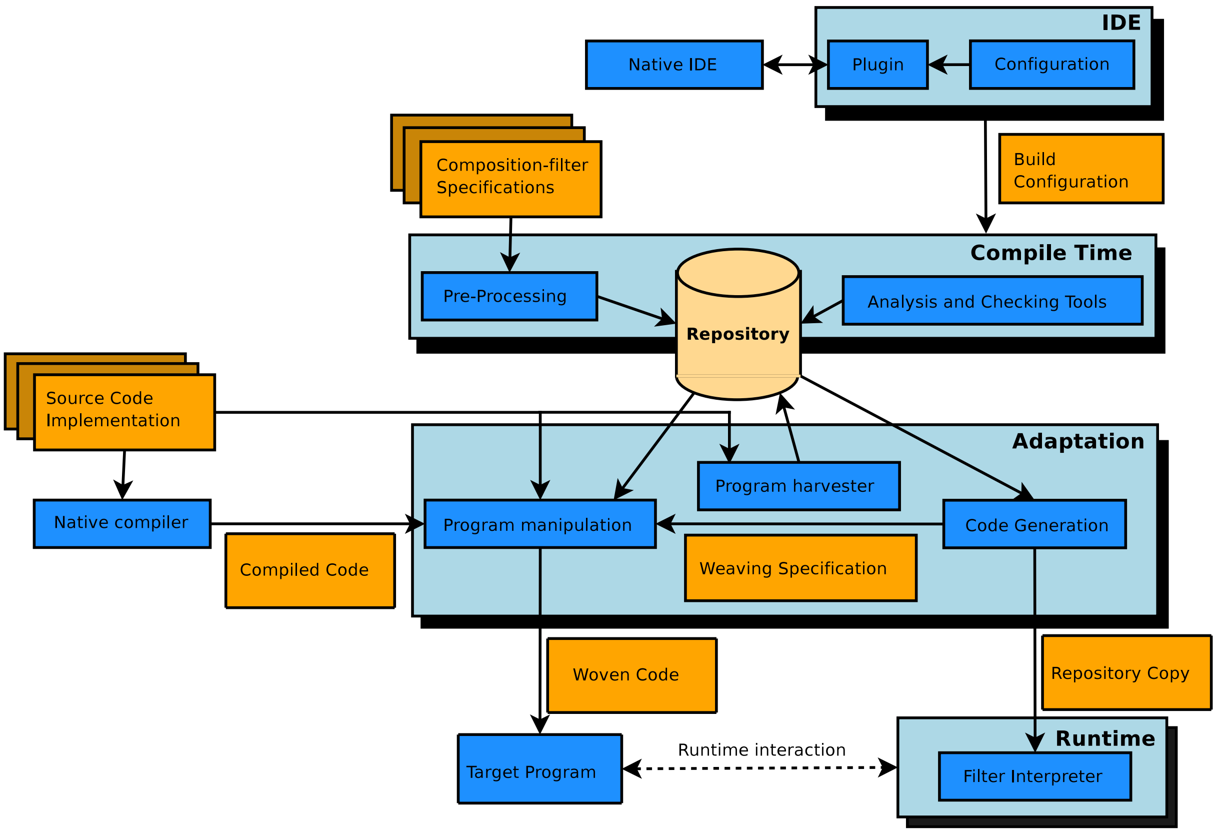

Figure 2.3: Overview of the Compose* architecture

Compose*/C. After creating a build configuration, the compile-time is started.

2.4.2 Compile-time

The compile-time layer is platform independent and reasons about the correctness of a composition filter specification with respect to an application. This allows the target application to be build by the adaptation layer.

The compile-time 'pre-processes' the composition filter specifications by parsing the specification, resolving the references, and checking its consistency. To provide an extensible architecture to facilitate this process, a blackboard architecture is chosen. This means that the compile-time uses a general knowledgebase, called the 'repository'. This knowledgebase contains the structure and metadata of the application. It is used by different modules to base their activities on. Examples of analysis and validation tools are the three modules SANE, LOLA, and FILTH. These three modules are responsible for (some) of the analysis and validation of the superimposition specification. The compile-time layer creates a weave specification that is used as input by the adaptation layer.

2.4.3 Adaptation

The adaptation layer consists of modules for program manipulation, type harvesting, and code generation. These modules connect the platform independent compile-time to the target platform. The harvester is responsible for gathering the structure and the annotations within the source program. And the harvester adds this information to the knowledgebase. During code generation a reduced copy of the knowledgebase is generated along with a weave specification. This weave specification is used by the weaver during program manipulation to instrument the target program with hooks to the run-time environment of Compose*.

2.4.4 Run-time

The Compose* run-time environment is responsible for execution of concerns at join points. It is activated by the hooks present in the application, as woven by the adaptation layer. A reduced copy of the knowledgebase, containing the necessary information for filter evaluation and execution, is used for evaluation of messages. When an instrumented function is called, that message is evaluated by the superimposed filter modules. Depending on the condition part and matching part of a filter, accept or reject behavior is executed.

2.5 Supported Platforms

The composition filters concept of Compose* can be applied to any programming language, given that certain assumptions are met. Currently, Compose* supports three platforms: .NET, Java, and C. For each platform different tools are used for compilation and weaving. They all share the same platform independent compile-time.

Compose*/.NET targets the .NET platform and is the oldest implementation of Compose*. Its weaver operates on CIL byte code. Compose*/.NET is programming language independent as long as the programming language can be compiled to CIL code. An add-in for Visual Studio is provided for ease of development. Compose*/J targets the Java platform and provides a plug-in for integration with Eclipse. Compose*/C contains support for the C programming language. The implementation is different from the Java and .NET counterparts, because it does not have a run-time environment. The filter logic is woven directly in the source code. Because the language C is not based on objects, filters are woven on funtions based on membership of sets of functions. Like the Java platform, Compose*/C provides a plug-in for Eclipse.

2.6 Features Specific to Compose*

The Composition Filters approach uses a restricted (pattern matching) language to define filters. This language makes it possible to reason about the semantics of the concern. Compose* offers three features that use this possibility, which originate in more control and correctness over an application under construction. These features are:

Ordering of filter modules
It is possible to specify how the superimposition of filter modules should be ordered. Ordering constraints can be specified in a fixed, conditional, or partial manner. A fixed ordering can be calculated exactly, whereas a conditional ordering is dependent on the result of filter execution and therefore evaluated at runtime. When there are multiple valid orderings of filter modules on a join point, partial ordering constraints can be applied to reduce this number. These constraints can be declared in the concern definition;

Filter consistency checking
When superimposition is applied, Compose* is able to detect if the ordering and conjunction of filters creates a conflict. For example, imagine a set of filters where the first filter only evaluates method m and another filter only evaluates methods a and b. In this case the latter filter is only reached with method m; this is consequently rejected and as a result the superimposition may never be executed. There are different scenarios that lead to these kinds of problems, e.g. conditions that exclude each other;

Reason about semantic problems
When multiple pieces of advice are added to the same join point, Compose* can reason about problems that may occur. An example of such a conflict is the situation where a real-time filter is followed by a wait filter. Because the wait filter can wait indefinitely, the real-time property imposed by the real-time filter may be violated.

The above mentioned conflict analyzers all work on the assumption that the behavior of every filter is well-defined. This is not the case for the meta filter, its user-undefined, and therefore unpredictable, behavior poses a problem to the analysis tools.

Furthermore, Compose* is extended with features that enhance the usability. These features are briefly described below:

Integrated Development Environment support
The Compose* implementations all have a IDE plug-in; Compose*/.NET for Visual Studio, Compose*/J and Compose*/C for Eclipse;

Debugging support
The debugger shows the flow of messages through the filters. It is possible to place breakpoints to view the state of the filters;

Incremental building process
When a project is build and not all the modules are changed, incremental building saves time.

Some language properties of Compose* can also be seen as features, being:

Language independent concerns
A Compose* concern can be used for all the Compose* platforms, because the composition filters approach is language independent;

Reusable concerns
The concerns are easy to reuse, through the dynamic filter modules and the selector language;

Expressive selector language
Program elements of an implementation language can be used to select a set of objects to superimpose on;

Support for annotations
Using the selector, annotations can be woven at program elements. At the moment annotations can be used for superimposition.

Chapter 3

Motivation

The Compose* language is an ever evolving language, because it is a research project. It has some known shortcomings and legacy constructs. To get the language in line with the current requirements and wishes, a redesign of the Compose* language is in place. The process of redesign starts with making an inventory of the language and the known problems. The inventory of the language will also serve as the Compose* Annotated Reference Manual (ARM). From the inventory of the known problems we will identify the possible improvements. The main problems are that the application code written in the Compose* language is not reusable enough, the filter specification lacks expressiveness, and the language has some old redundant features. After identifying the possible improvements we will design the improvements according to the language requirements. The language requirements are stated in section 3.2 and the improvements, with their own problem statement and examples in chapter 4, chapter 5, and chapter 6.

3.1 Background

When the Compose* project started, the grammar of its predecessor, ConcernJ [Sal01], was copied and slightly altered. A ConcernJ concern is similar to a Compose* concern, it can be divided into filter modules, superimposition, and implementation. ConcernJ is the combination of the earlier implementations of Composition Filters and the then new introduced superimposition. As we can see, several parts of the Compose* language date back from the times when CF was considered an object-oriented approach.

After the release of version 0.1 of Compose*, certain short comings were identified, such as the lack of reusable filter modules and limitations in the expressiveness of filter specifications. To solve the short comings and to remove parts which have become obsolete, a redesign of the language is needed.

3.2 Requirements for the Compose* Language

The redesign of the Compose* language has to follow these requirements:

Language independent

Composition filters is a language and platform independent approach, and one of the goals of Compose* is to be language and platform independent as well. We are using the term *language and platform independent* instead of language independent, because some languages have different specifications for each platform, for instance C++. We want this independence so that the Compose* language can be used along side any other programming language without losing any syntax or semantics of Compose*. This means that the Compose* language should not contain any language dependent parts. Whereas this sounds trivial, it does have severe consequences. It means that keywords such as *new* cannot be used or we must map it to a similar construct for each language, for the same reason we also need to be careful for using primitive types such as integer and reals.

Expressive

A user must be able to write down what he wants and does not need any workarounds to do it. If we take for instance the filtering of the message, then we see that it is only possible to filter on the target and the selector of a message. This is not very expressive, because a message has more properties then just those two. The new language constructs should open the full possibilities, such as the properties of the message, to the user.

Robust

The code written in the Compose* language must be robust. Robustness can be a conflicting requirement with conciseness, so it is important to find the balance between robustness and conciseness.

Concise

If the language is concise it is easier to apply an algorithm on the code written in the Compose* language. Therefore it is preferable to only keep the items in the language that really matter.

Reusable

The current Compose* syntax is based on the principle that concerns are written for one particular set of classes. To get better reusable concerns the syntax and semantics of Compose* should change.

Backwards compatible

Compose* is not a production language and while redesigning the Compose* language we do not have to worry about that the user should be able to compile his old code on a new version of Compose*. However, some parts of the language have been around for twenty years and are described in numerous articles and papers. It would be preferable if the code examples from back then can still work on the new version of the language. Backwards compatibility is a requirement that we may break if needed, however, we should for every new construct look for the possibility to be backwards compatible.

Part I

Proposed Compose* Language Changes

Chapter 4

Introduction of Filter Module Parameters

One of the current shortcomings of Compose* is the lack of possibilities to create generic filter modules. Currently it is needed to write filter modules which are custom made for one application. Reusing filter modules usually does not work because you have to refer to specific program elements (in a specific application).

In this chapter, two examples demonstrate this problem, one example models a company structure [Ber94] and the other example implements logging. After the examples we look to two solutions for introducing generic filter modules. Section 4.3 covers the detailed design and decisions on the usage of the generic construct, this is followed by the considerations on the implementation.

4.1 Examples

4.1.1 Company Example

Figure 4.1 shows the diagram of the model, two of the inheritance relations are worked out in Listing 4.1.

```
filtermodule inheritFromPerson{
  internals
    person : Person;
  inputfilters
    inheritFilter : Dispatch =
      {<inner.*> inner.*, <person.*> person.*}
}
filtermodule inheritFromEmployee{
  internals
    employee : Employee;
  inputfilters
    inheritFilter : Dispatch =
      {<inner.*> inner.*, <employee.*> employee.*}
```

Listing 4.1: Inheritance concern

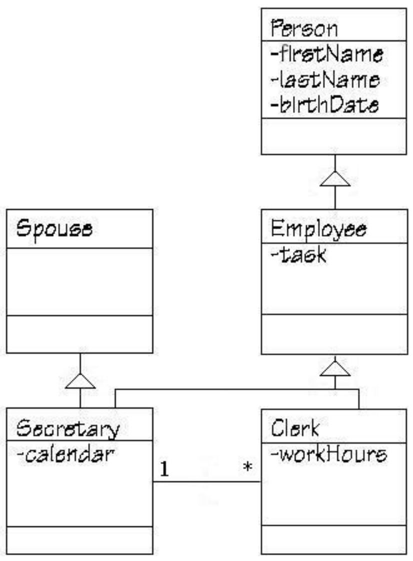

Figure 4.1: UML diagram of the company model

For every inheritance relation we have to write a filter module with an internal of the parent class type and a dispatch filter which dispatches messages to the parent class when the selector is in the signature of the internal. Thus when we would have a number N of inheritance relations we would also need N filter modules[1]. Looking to Listing 4.1, we see that the two filter modules do have the same pattern and if we write another inheritance filter module it will have the same pattern as well. To solve this problem we want to write only one filter module, that we can reuse for every inheritance relation we want to specify.

4.1.2 Logging Example

Logging is a common concern, so it is most likely that a concern like logging will show up in a generic library. To make a logging filter module fully generic we need to make two elements of it generic, first is what we want to log and second how we log it.

We start with one of the common ways to implement a logging filter module, which is shown in Listing 4.2. In the example the filtering on the methods a and b is static declared. So for every object where we will bind this filter module on, the methods it logs are `a` and `b`. The logging takes place with the method `logger.log()`. This construct is not flexible, because when we reuse the filter module Logging we are stuck with these identifiers.

4.1.3 Problem Statement

In the two earlier mentioned examples we see that the language needs a better way of reusing a filter module. We want to achieve this by making the filter modules more generic. Besides improving the reusability the construct also must be flexible. To illustrate the problem, Listing 4.2 is demonstrated schematically in Figure 4.2. In the schema the application specific concern is

[1]Multiple inheritance is not taken into account.

```
filtermodule Logging1{
  externals
    logger : Logger;
  inputfilters
    log : Meta ={[*.a] logger.log, [*.b] logger.log}
```

Listing 4.2: A common logging filter module

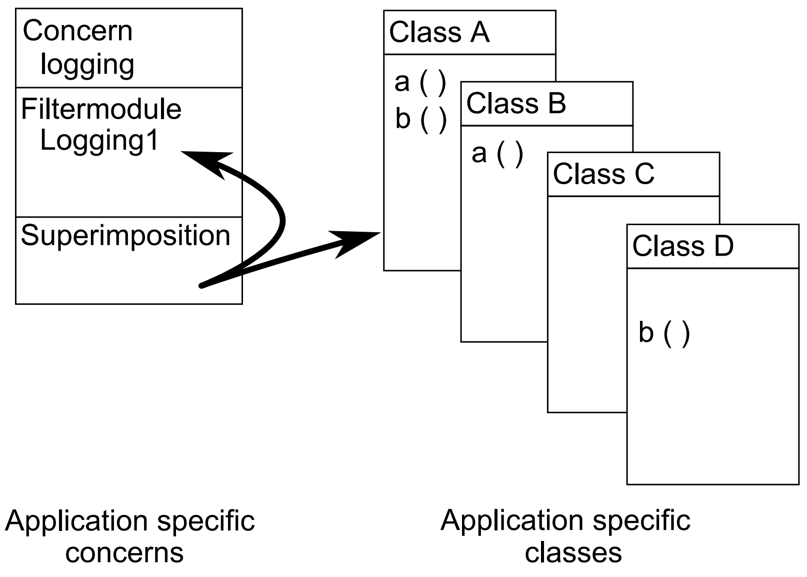

Figure 4.2: Schematic representation of the Logging example

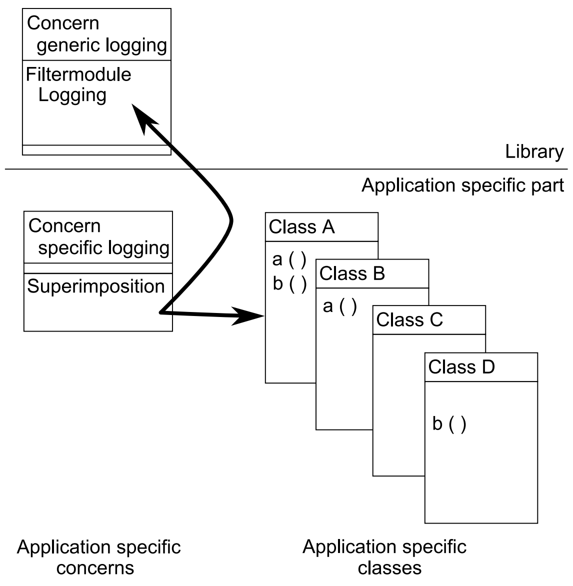

Figure 4.3: Schematic representation of the preferred situation

```
filtermodule Logging2{
  externals
    logger : Logging1:logger;
  inputfilters
    log : Meta ={[*.c] logger.log}
```

Listing 4.3: A common logging filtermodule with reused references

superimposed on the set of classes A, B, C, and D. The filter module Logging1 from the application specific concern is the filter module Logging1 in Listing 4.2 and this filter logs when the methods `a` and `b` are used. We want to get the situation shown in Figure 4.3. In this schema a generic filter module logging is placed in a library. We want to achieve that the only thing that remains in the application specific concern is the superimposition, which superimposes the filter module Logging to the selection of classes.

4.2 Possible Solutions

There several ways of making a programming language more generic and adaptable. We look to two of these options, the first is reuse by reference, which is an idea that survived the transition of ConcernJ to Compose* and is still mentioned in the Compose* documentation [Vin04]. The second option is to use parameters in the filter modules. This is the option that comes in mind first when we think of adding more generics and adaptability to the filter module syntax.

4.2.1 Reuse by Reference

The idea of this language construct is to reuse parts of another filter module by referring to the orginal declaration. For example, if we rewrite the filter module `inheritFromEmployee` from Listing 4.1 with reuse then we get Listing 4.4. Listing 4.2 can be rewritten as Listing 4.3, which demonstrates the reuse of an external.

Referring to Internals, Externals, and Conditions
The problem with reusing internals, externals, or conditions is that you are not certain whether the instance to which you refer to really exists. If we take for instance Figure 4.4, which shows a possible reuse construction, we can see that if we implement two filtermodules `A` and `B` and B's external points to the external of A, and A is never instantiated by the superimposition then the external does not exist and the external of B has a null value[1]. This will be catched by Compose* on compile time, so we could argue that the problem lays by the user. However, it can be the case that a reused object can be different in each instance of a filter module. An example of this scenario is given in Figure 4.5. In this scenario we reuse an internal as an external, the problem then is that because every instance of the filter module gets its own instance of the internal we can never determine which instance we need or get. This applies for the reuse of all three parts: internals, externals, and conditions.

The idea of reusing these three parts with this construct is Compose* legacy, it probably became obsolete with the introduction of superimposition. The only correct use of the reuse by reference is to get a construction like in Figure 4.6, where we place a filtermodule B on filtermodule A.

[1] Another scenario is that the filter module gets superimposed to a class, and this class never gets instantiated.

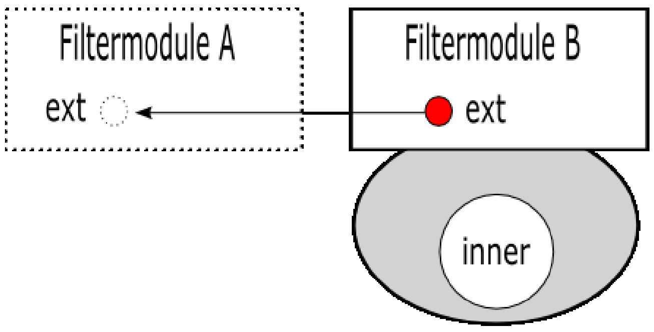

Figure 4.4: Reusing an external

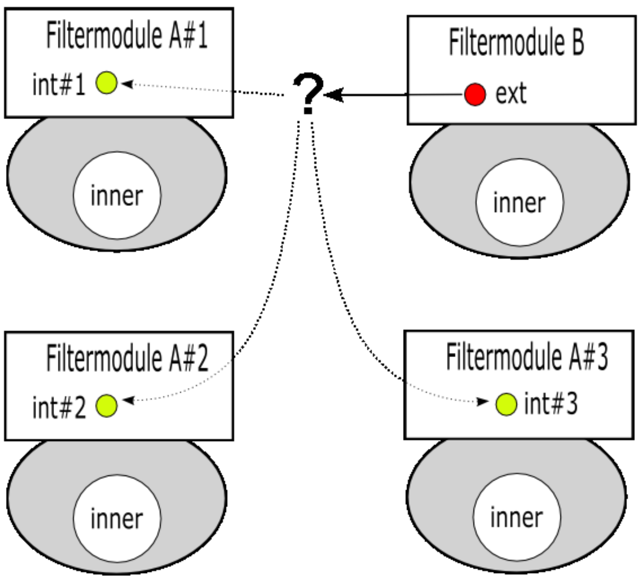

Figure 4.5: Reusing an internal

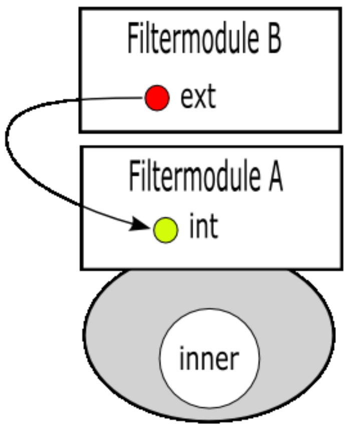

Figure 4.6: Another form of reusing

However, this is achievable by introducing the externals and conditions from `self.internal`[1]. The only (current) technical problem is that the signature of self needs to be determined when B gets bounded with A and the superimposed object, however it is possible to determine the dependencies and from that we can work out how to get the correct signature.

Referencing to filters
The concept of reusing filters has a different base than the reuse of the internals, externals and conditions. When you reuse a filter, like in Listing 4.4, you only copy the filter structure and you have to take care that the filter module has internals, externals, and conditions with the same identifiers as the "donor" filter module. Thus saving you copying the code manually from one filter module to another. However, although the reuse of the internals, externals, and conditions is already in the language, the referencing to filters has not been implemented yet.

4.2.2 Generics by Filter Module Parameters

A common way to make a programming language generic and adaptable is to use parameters. With parameters it is possible to implement code with free variables, which you fill in later. If we apply this idea on the example of the company then we rewrite the filter modules of Listing 4.1 into a generic (single) inheritance filter module demonstrated in Listing 4.5. This generic filter module is instantiated in the filter module binding part.

Using generic filter modules for multiple inheritance can be achieved in two ways, creating a filter module with multiple parameters or by binding the single inheritance filter module multiple times on the same object. For example, if we take Listing 4.6, we can can create a

[1]The signature of self is the signature of inner extended with the filtermodule signature.

```
filtermodule inheritFromEmployee{
  internals
    parent : Employee;
  inputfilters
    inheritFilter = self::inherit:inheritFilter
```

Listing 4.4: The company example with reused filters

```
filtermodule inherit(?parent){
  internals
    parent : ?parent;
  inputfilters
    inheritFilter : Dispatch =
      {<inner.*> inner.*, <parent.*> parent.*}

superimposition{
  selectors
    ...
  filtermodules
    selectEmployee <- inherit(Person);
    selectClerk <- inherit(Employee);
```

Listing 4.5: The generic single inheritance filtermodule

generic double inheritance filter module with two parameters, `filtermodule doubleInherit (first, second)`. The drawback of this solution is that we need a filter module for every *N-inheritance*. The other option, binding the filter module N times to the object, that is to be superimposed on does not have this problem. However, if we implement multiple inheritance in this way we got another problem: we need to set constraints, but the constraints currently only supports different filter modules and we cannot specify the constraints for filter module with the same identifier but with different parameters, like in Listing 4.7. This short coming is not exclusive for this example, it is likely that we encounter more situations in which we want to make a distinction between filter module instances with the same name but with different parameters. Therefore the language will be adapted to solve this issue.

For the logging example we can make the two elements for logging generic with parameters, the external declaration can be changed into a parameter. The methods to log on cannot be declared as flexible with a single parameter. In Listing 4.2 we have two methods `A` and `B`, we can replace them with two parameters `?method1` and `?method2`, but that limits us to only log on two methods. We actually want to use a list, so that we do not have to define on how many methods we want to log. There are no conceptual problems with the parameter list, we only need to define in which places we can use the list and what it then exactly means. In Listing 4.2 we also see the problem with the constraints, if we want to place two log filter modules on one class, then we do want to define ordering[1], this demonstrates again the need for more distinction between filter modules with the same name.

```
filtermodule inheritForSecretary{
  internals
    employee : Employee;
    spouse : Spouse;
  inputfilters
    inheritFilter : Dispatch = {<inner.*> inner.*,
      <employee.*> employee.*, <spouse.*> spouse.*}
```

Listing 4.6: Multiple inheritance

```
filtermodules
  sel1 <- inherit("Employee");
  sel1 <- inherit("Spouse");
constraints
  pre (inherit("Employee"), inherit("Spouse"));
```

Listing 4.7: Multiple inheritance with multiple bindings

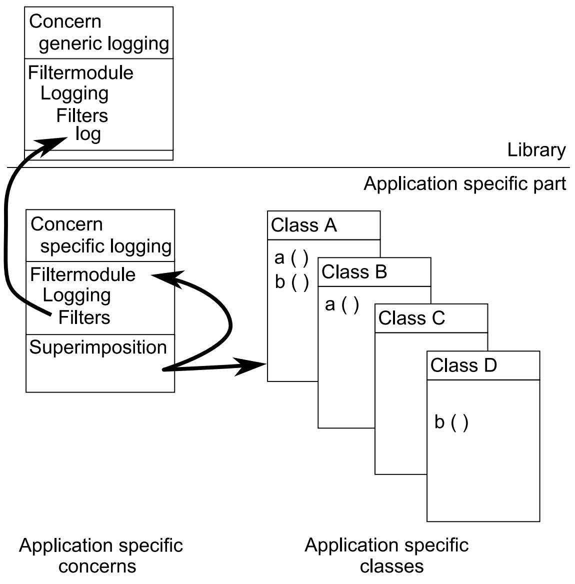

Figure 4.7: Schematic representation of reuse by reference

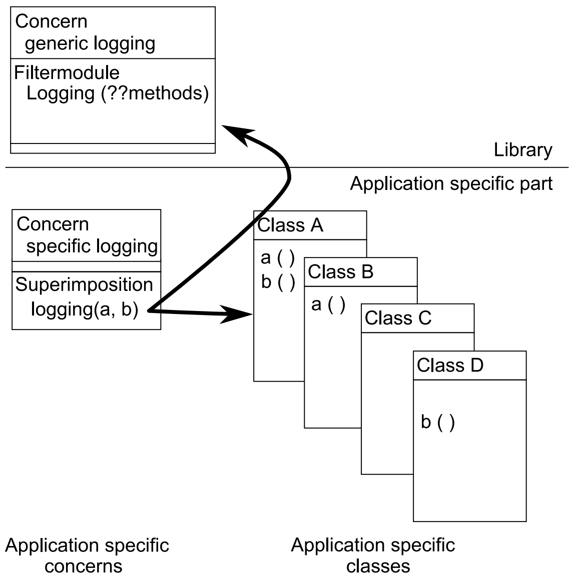

Figure 4.8: Schematic representation of generics by filter module parameters

4.2.3 Evaluation

In the problem statement in the beginning of this chapter, section 4.1.3, the current situation for filter module usage is shown in Figure 4.2. When we make a schema for both solutions for the logging example (section 4.1.2), we get Figure 4.7 for reuse by reference and Figure 4.8 for filter module parameters.

For the reuse of filter elements we see that we can only reuse filter specifications. The filter module of the application specific code must have the same identifiers as the filter specification in the generic filter module that is placed in the library. So instead of moving the whole filter module to the library as in Figure 4.3, the application specific concern still needs to have its own filter module specification. On the other hand, Figure 4.7, which show the schema of the filter module parameters, is similar to Figure 4.3. Therefore we can say that filter module parameters are a possible solution to the problem mentioned in section 4.1.3.

As mentioned earlier, referencing has several drawbacks and we do not have these drawbacks with filter module parameters. With the introduction of the filter module parameters to the language we can rewrite the code that uses the reuse by referencing construct, into generic code with parameters. So by introducing filter module parameters to the language, we do not longer need the construct for reuse by reference. Therefore we can conclude that reuse by reference is not the option for introducing generics and adaptability to Compose* and we will adopt filter module parameters to achieve more generic code.

4.3 Design of the Filter Module Parameters

To keep a difference between a literal and a parameter, the parameters have a prefix. This prefix can be a single question mark or a double question mark. The single question mark means that you only have one value, the double means that you are dealing with a list. To give a small example of the different semantics of both prefixes, we execute the operation *size* on a list of persons. *Size ?persons* gives the size of the list of person, thus the number of persons in the list, the other option: *Size ??persons*, gives a list of the sizes of all the persons in the list. The question mark prefixes are inspired from the AOP languages Sally [Sal03] and LogicAJ [Log05]. They use this notation to add parameters to AspectJ, with as goal to make AspectJ code more generic. Their approach has also a distinction between a single parameter and a list of parameters.

A point of consideration is whether we take the ordering of the list into account, thus is {a,b} different from {b,a}? From the examples given earlier on, it is clear that some lists will come from the selectors. The selectors do return non ordered sets, so in most cases when we use a parameter list then there is not an ordering, therefore the ordering of a list is not taken into account.

Parameters are declared, like in most programming languages, at the start of the language element that has the parameters. In this case this is in the filter module heading. Another point of consideration is where we will allow to use the parameters? In order to determine this we will take a shorthand syntax (Appendix E) of the filter module grammar and look whether it means anything if parameters are allowed there. For a complete overview of the grammar the

[1] This of course depends on the application, in certain cases you do not mind that something is logged double.

nr.	EBNF name	Line	?-	??-	Example
1	FilterModuleName	1			
2	Identifier-LIST	5			
3	Type (aka Constructor)	5	X		person : ?type;
4	Identifier-LIST	6			
5	Type	6	X		person : ?type;
6	ConstructorReference	8	X		person : Person = ?NewPerson
8	ConditionName	10			
9	ConstructorReference	10	X		condition : ?ConditionReference

Table 4.1: Filter module parameters in the first part of the filter module

references are added, however they differ from the real world grammar (Version 0.5). To keep things simple it is better to split the rest of the grammar into two parts, the filters and the rest. We start with the latter.

The tables use the names of elements of the left hand side of the syntax, because some names occur multiple times, the line numbers from Appendix E are also put in the table. In the columns "?-" and "??-" we will mark whether we will allow the use of a parameter or list of parameters on that point. In the following column an example will be given of how the possible use of a parameter will look like. Further on we discuss the choices and their impact on the use of the language.

4.3.1 Using Parameters in Internals, Externals, and Conditions

We start with the first three fields of the filter module. The first thing that strikes from Table 4.1 is that the parameter list is never used in this part of the language. Second, identifiers of filter modules, internals, externals, and conditions cannot be parameters. The Type can be a parameter and since every object can only have one type, the usage of a list of parameters is not meaningful here. For the right hand side it is possible to use a single parameter. A list of parameters in the right hand side of the conditions is a meaningful construction, however there too many drawbacks on the construction, as shown in section 4.3.6. So we decided not to use lists of parameters in condition declarations.

When you use a parameter in the type of an external declaration, it does imply that you also should use a parameter in the Constructor reference. However the construction used in the example column in row six is correct. When you change the function which will give the external, the type can be the same or it can be a subtype of it. Using a parameter in the constructor does not imply that you should use one in the Type.

For the values, which can be filled in the parameters, is required that they are fully qualified names. This means that constructions like `?package.?class` and `PackageA.?package.classC` are not allowed. We enforce this to keep the typing simple and to keep the language clear.

nr.	EBNF name	Line	?-	??-	Example
10	GeneralFilterSet	12			
11	GeneralFilter	15			
12	FilterCompositionOperator	16			
13	FilterName	18			
14	FilterType	18	X		d : ?Filter =
15	ActualFilterParameters	19	X		m : Meta (?args) =
16	FilterElements	20			
17	Value-LIST	21	X		m : Meta (a, ?b, c, ?d) =
18	FilterElement	22			
19	ElemCompositionOperator	23			
20	ConditionExpression	25			
21	ConditionOperator	25			
22	MessagePatternSet	26			
23	MessagePattern	32			
24	Matching	33			
25	Substitute	33			
26	Target	40	X		[?Foo.bar]
27	Selector	40	X	X	[Foo.??bar]
28	MethodName	42	X		?Method(String, Person)
29	Type-SEQ	42	X		Method(?Types)

Table 4.2: Filter module parameters in the filters

4.3.2 Using Parameters in Filters

The other part of the filter module to be analysed is the filterpart, this is where the usage of the list of parameters will be demonstrated. In Table 4.2 more of the half of the grammar elements can not be filled in with parameters, for instance a complete filter or any operator. Allowing such parameters would weaken the language on points like robustness and usability. The filtertype can be a parameter because Compose* has the option of using custom filtertypes. The filter has a place to put arguments, in that place it also possible to use parameters from the filter module. There are no parameters in the condition part of the filter, if a condition is to be passed through the filter module parameters, then you should declare it in the conditions field. This closes the way to pass the boolean values "True" and "False" along as arguments. This can be fixed with two static methods. These methods give back one of these two values and we can introduce these methods for use in the conditions field.

In the matching and the substitution part there are some rules which cannot be parameterized either, the idea of `?TargetAndSelector` which can have the value "Target.Selector" is wrong because we do not know whether the dot in the value belongs to the name of the selector. When we look to Appendix E, we see that a matching or substitution part can consist of only a selector. The usage of the list of parameters in the selector means that a message must match with one value of the list.

4.3.3 Parameter Typing

To improve the robustness typing is added to the filter module parameters. For the grammar we need to choose between explicit typing, thus an user must give the type, and implicit typing, the compiler concludes which types are used. A parameter in Compose* can be used on more then one spot at the time, this means that if explicit typing is applied we also need morphism constructs in the language. Implicit typing does not have this problems because the compiler can determine the type by itself. Therefore the typing of the parameters will be implicit. The effect is that the syntax does not have to cover any types and we can just use `(Parameter | ParameterList)-LIST` to list the parameters in the filter module heading.

4.3.4 Converting Types

In the parameter heading we get different types for values then we use in the filters. For instance, in order to get a String in the parameters to be filled as a selector we need to transform it. We need to analyse which combinations of a given type and demanded type we can encounter and how we will transform these. We look first where we can insert parameters and after that we see which type of parameter gets which preferred type.

Table 4.3 has the grammar parts listed where parameters can be placed, the table number is the number from Table 4.1 and Table 4.2. The last column is what type we expect for that field. A remark about this analysis is that we take the analysis from the viewpoint of the filter module, we only look to what we expect and not to the items that can be placed there.

nr.	Name	Table nr.	Type
1	Internal type	3	Class reference
2	External type	5	Class reference
3	External initialization	6	Method call
4	Condition initialization	9	Method call
5	FilterType	14	Filter type
6	FilterParameters	15	Variable
7	Target	26	Literal
8	Selector (1)	27	Literal
9	Selector (N)	27	Literal list
14	Method name	28	Literal
16	Method parameter type	29	Type

Table 4.3: Typing of the filter module parameters

Table 4.3 gives us an idea of what we actually expect in the places for the parameters. With this we can build our coercion table (Table 4.4). The action stated in the table depends on where the change between parameter and actual value takes place. The easiest is to first transform it to a string and make it look like a non parameterized value. Then let the rest of Compose* do their transformations.

nr.	Parameter type	Convert to	Action
1	String	Internal type	N.A.
2	Class reference	Internal type	Convert to String
3	String	External type	N.A.
4	Class reference	External type	Convert to String
5	String	External initialization	N.A.
6	String	Condition initialization	N.A.
7	Method reference	Condition initialization	Convert to String
8	String	Filtertype	N.A.
9	Class reference	Filtertype	Convert to String
10	Object reference	Filterparameters	Ignore
11	String	Target	N.A.
12	String	Selector (1)	N.A.
13	Method reference	Selector (1)	Convert to String
14	String list	Selector (N)	N.A.
15	Method reference list	Selector (N)	Convert to String list
16	String	Methodname	N.A.
17	String	Method parameter type	N.A.

Table 4.4: Coercion table

In general we want that each language part has only one type. This sometimes means that we get a reference. We will convert the parameter to a string and later on it will be converted back to a reference. This does looks like a redundant job, however if we are certain that only one type can be found in each place, then the code will be less complex then when we allow multiple types. Because on most places we can have two types of values, like the type and reuse of an external, or there are typical strings, such as the "*" in the selector, it is the easiest to convert to strings. The "ignores" in the action column are for parameters which will be passed through as argument. So if we place parameters from the filter module through to the filterparameters we do not execute any operations on them and we will just pass them through unchanged. The subparts are all strings, it would be strange to fill in other items on these language parts.

For the Target field only the Literal is allowed, without this Literal we cannot use the inner keyword as parameter. The usage of an object reference is not possible due to several problems with the direct use of objects out of the parameters. These problems will be covered in the next section.

4.3.5 Objects in the Parameters

There are several ideas about objects in the parameters, however some of them are technical not possible or will limit the Compose* language or the base language. The main idea is to place an object in the parameters and then use it in the internals and externals blocks, or directly in a filter. We start with the latter.

When we combine Listing 4.8 with Listing 4.9, we see that the typing of the object does matter for applications written in languages where methods are (or can be) hidden (For example C#, C++, and Delphi). If we would place the external person in the parameters we loose the option to use the hidden functions of the class Person. So in any case it is more correct and robust in the first place to type an object in the externals. To keep all filter module usage the same for all base languages we are not making an exception for languages which always override methods (For example Java).

```
filtermodule externaltype{
  externals
    person : Person = getNewStudent();
  inputfilters
    d : Dispatch = {[*.getName] person.getName, [*.getID] person.getID}
```

Listing 4.8: An example of external typing

```
class Person{
  public virtual string getName(){}
  public string getID(){} // returns social security number
}

class Student : Person{
  public override string Name(){}
  new public string getID(){}  // returns student number
}
```

Listing 4.9: An example C# application

All objects that are introduced by means of parameters must be declared as an external. If you want to use the object further on you need to use the identifier of the external, this because you can introduce the object with different types.

It is not possible to introduce an object as an internal, whereas we could clone the object, but then we have to choose between a shallow or a deep clone. This choice lays actually by the user and therefore we will not clone objects in order to use them as an internals. Because if we do so we need to create a syntax to let the user assign the type of cloning.

4.3.6 Condition Lists

In section 4.3.1 we see that we cannot use parameter lists in conditions, however it is interesting to see what a list on that place could mean and what it would add to the language. There are two possible meanings for such construct, these will be illustrated by extending the Pacman example [Vin04].

The first meaning is to use a parameter list to give a serie of conditions to use in a filter. Looking to the earlier mentioned filter module DynamicStrategy of the Pacman example (Listing 4.10), and we make the filter module more generic by placing the condition `isEvil()` as a parameter, we are stuck with just one free variable. If we want to add a feature to the game that when Pacman eats a fruit the ghosts will also flee but Pacman cannot eat the ghosts, we want to use two conditions instead of one. This is not possible because there is only one free variable. One way to solve this is by a using a parameter list. However, if we work this out we see that there is a problem with this construct. We want to say `isEvil OR hasEatenFruit`, this means we have to define the operator in the condition list on "OR". If we would set the operator in the condition list on a default "AND" or "OR", then in half of the cases this would not be the operator you want. Adding a language construction to say which operator you want, means a increase in grammar complexity. You also might not want to use just one operator for the whole statement, but probably a combination of both. Therefore we conclude that writing a wrapper method, which gets introduced as a condition and will define the relation between both methods `isEvil` and `hasEatenFruit`, is the best way to solve this particular problem.

The second meaning is that a parameter list in the conditions is a part of a case statement. If we take again the basic Pacman example and we want to use four different hunt strategies for each of the four ghosts. If we could determine which request comes from which ghost we can provide each ghost with its own behavior. To make this reusable for an easy, normal, or hard game, we must give a filter module five strategies, one for fleeing and one for each of the four ghosts. If we want the number of ghosts to be flexible, then it is possible as demonstrated in Listing 4.11[1]. We have the problem that the parameter lists are not ordered, thus we can not guarantee that the lists are staying in the correct ordering. Even if the list was ordered we still can better apply a generic filtermodule DynamicStrategy five times to achieve the same. So we conclude that this second meaning can not be worked out due to our decision to let go on the ordering of the parameter list.

After these two analysis of the use of the conditionlist we can conclude that we can omit the use of parameter lists in conditions from the grammar. The possibilities that it gives are achievable in other ways or are just impractical to use.

[1]The used condition form is currently available, it is to demonstrate the idea. The functionality is obtainable by creating a static function which could say whether the sender was a certain ghost.

```
concern DynamicStrategy in pacman{
  filtermodule dynamicstrategy
      (?stalk_strategy : type; ?flee_strategy : type;
        ?stalk : method){
    internals
      stalk_strategy : ?stalk_strategy;
      flee_strategy : ?flee_strategy;
    conditions
      pacmanIsEvil : ?stalk;
    inputfilters
      stalker_filter : Dispatch = {?stalk =>
        [*.getNextMove] flee_strategy.getNextMove };
      flee_filter : Dispatch =
        {[*.getNextMove]stalk_strategy.getNextMove }
  }

  superimposition{
    selectors
      strategy = { Random | isClassWithName
                   (Random, 'pacman.Strategies.RandomStrategy') };
    filtermodules
      strategy <- dynamicstrategy
                    (pacman.Strategies.HardStalkerStrategy,
                    pacman.Strategies.HardFleeStrategy,
                    pacman.Pacman.isEvil());
  }
}
```

Listing 4.10: Generic Dynamic Strategy pattern from Pacman

```
concern DynamicStrategy in pacman{
  filtermodule dynamicstrategy
      (??strategies : object; ??stalk : method){
    inputfilters
      behaviour_filter : Dispatch =
       {#??stalk => [*.getNextMove] #??strategies.getNextMove }
  }

  superimposition{
    selectors
      strategy = { Random | isClassWithName
                   (Random, 'pacman.Strategies.RandomStrategy') };
    filtermodules
      strategy <- dynamicstrategy
       ({pacman.Strategies.FleeStrategy, pacman.Strategies.PinkyStrategy,
       pacman.Strategies.BlinkyStrategy, pacman.Strategies.InkyStrategy,
       pacman.Strategies.ClydeStrategy}, {pacman.Pacman.isEvil(),
       sender.isPinky(), sender.isBlinky(), sender.isInky(),
       sender.isClyde() });
  }
}
```

Listing 4.11: Generic Dynamic Strategy pattern from Pacman

4.3.7 Initialization String Parameters

By allowing parameters in the initialization of the external we get more options to use an external function that distributes objects. With passing parameters from the filter modules to the filter module elements, we get even more options for making generic code.

In the original (Version 0.5) grammar the parameters for the initialization String do not exist. We will not argue whether this a design issue of just a bug, because it is not in the original syntax. It is however taken into account in the analysis.

The parameters that we will pass do have one restriction, they must be a primitive type or an object. Since these can be passed through the filter module parameters it does not give any problems. The only issue is that you can only add objects which come from the parameters and it is not possible to use a function in the initialization string. This is the main counter argument to this construc: it breaks the orthogonality of the language, by allowing to place objects from the parameters, but you cannot create them there.

The best alternative is to set the parameters of the object outside the filter module code and then pass it through the filter module parameters.

4.3.8 The References

In the start of the analysis we looked to the option of placing parameters in the references. With allowing them, we need an analysis of how a parameter may be broken up in pieces for every place where a parameter can be placed. This results in a huge analysis and an unreadable grammar for all the rules and exceptions. If we break up the parameters in to sub parameters we also get problems with typing. For instance if we take an internal declaration as `packageA .?packageB.?Class`, then we do not know when ?Class is a the fully qualified name or just a part of it.

The construct of using parameters as sub parts opens the way of writing reusable libraries, so if we take the Pacman example we write for every difficulty level a set of hunting and fleeing classes, this is demonstrated in Listing 4.12. However, it is also possible to place this whole construction of difficulty and strategy in the superimposition.

Too keep things as simple as possible in the filter module, it is better not to use sub parameters and only use fully qualified names. It is possible to work around the limitations this choice brings.

4.4 Implementation of the Filter Module Parameters

The implementation of the filter module parameters can be divided into two parts. The first part is applying the changes in the syntax and the second part are the changes in the repository and the modules that use the repository. To start with the first part, the proposed changes for the syntax are shown in Appendix F.

```
concern DynamicStrategy in pacman{
  filtermodule dynamicstrategy (?difficulty){
    internals
      stalk_strategy : ?difficulty.stalk_strategy;
      flee_strategy : ?difficulty.flee_strategy;
    conditions
      pacmanIsEvil : Pacman.pacman.isEvil();
    inputfilters
      stalker_filter : Dispatch = {pacmanIsEvil =>
        [*.getNextMove] flee_strategy.getNextMove };
      flee_filter : Dispatch =
        {[*.getNextMove]stalk_strategy.getNextMove }
  }

  superimposition{
    selectors
      strategy = { Random | isClassWithName
                   (Random, 'pacman.Strategies.RandomStrategy') };
    filtermodules
      strategy <- dynamicstrategy ("hard");
      strategy2 <- dynamicstrategy ("easy");
  }
}
```

Listing 4.12: Strategy pattern from Pacman with difficulty levels

```
concern aConcern{
  filtermodule fm(?type){
    internals
      int : ?type;
  }

  superimposition{
    selectors
      ...
    filtermodules
      selA <- fm(A);
      selB <- fm(B);
  }
}
```

Listing 4.13: Implementation example

```
getType(){
  if(!parameter){
    return super.getType();
  }else{
    getParent().getArgument(MyName).getType();
  }
}
```

Listing 4.14: GetType method of the Internal class

4.4.1 The Problem with the Current Repository

The second part is harder to implement, this is because we have to redesign the repository on a conceptual level. If we take Listing 4.13, then we get into problems when we want to map it onto the current repository layout. The current repository layout is shown in Figure 4.9. In Figure 4.9 we make the difference between abstract and concrete classes. The concrete classes are the real entities like parsed filter modules and filters. The abstract layer are the classes where the concrete ones inherit from.

All the objects in the repository on this point come directly from the concern code itself. So in the situation of Listing 4.13 this means that we have one object of the class *filtermodule* and two objects of the class *filtermodulebinding*. The problem that we have here is that we need to store the arguments A and B somewhere and we need to excess these values from the filter module class. It is not possible to just fill the argument on the parameter spot, in this case we have to choose between one the two arguments.

4.4.2 Proposed Solution

To solve this problem we need to make a difference between the parsed filter modules and filter modules which parameters have been filled in, so the instances of the parsed filter modules. This situation is demonstrated in Figure 4.10. In that layout the concrete layer is divided into two groups: the AST classes and the instance classes. The AST (Abstract Syntax Tree) classes are parsed and the instance classes are created later on and delegate to the AST objects.

With this change it is possible to keep the arguments of each *filtermodulebinding* in the attributes of the *filtermodules*. When the parameter for the internal type is introduced the parent link needs to be fixed. Because as demonstrated in Figure 4.10 an internal instance will have a parent link to the AST filter module. Therefore it is not possible to access the value of the argument. To solve this we have to define an abstract and an instance class for internal as well. This changes the repository to the schema shown in Figure 4.11. The new structure uses the same delegation construction as the filter module. To get the type of an internal, the method *getType* need to get overloaded to the method shown in Listing 4.14. When this is applied on all parts of the syntax were parameters are allowed, it actually means that the syntax tree of the filter module gets duplicated, because the AST tree is similar to instances tree.

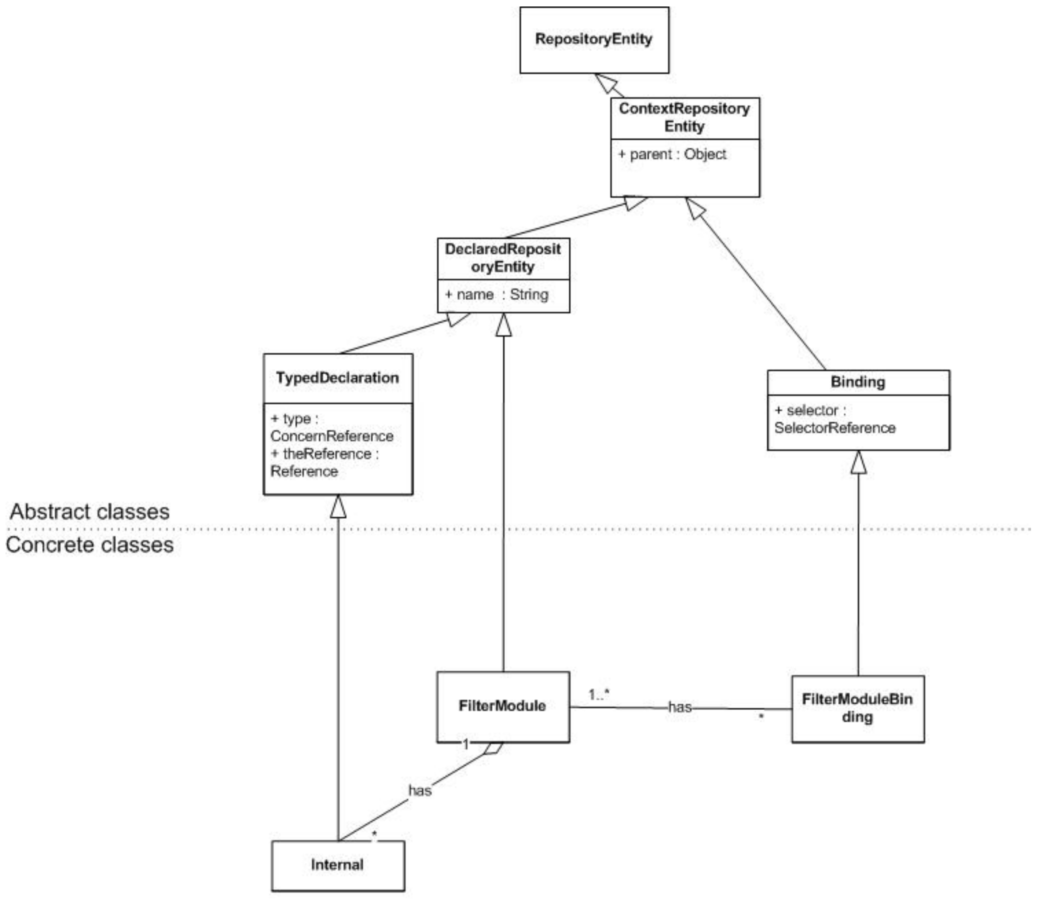

Figure 4.9: Schema of the old repository

```
filtermodule log(?logger, ??walkfunction){
  internals
    logger : ?logger;
  inputfilters
    m : Meta ={[*.??walkfunction] logger.log}
}

superimposition{
  selectors
    ...
  filtermodules
    selA <- log( FilterModuleParameter.Logger, {walk, makeNoise});
}
```

Listing 4.15: Filter module with parameters

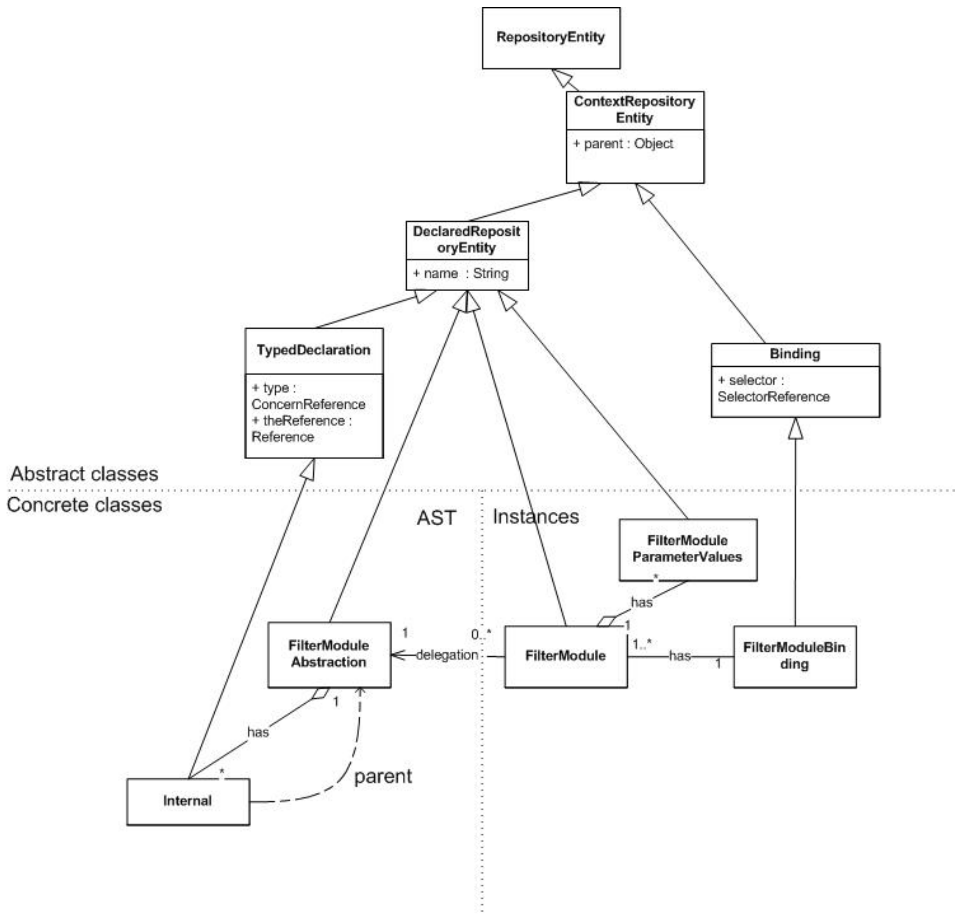

Figure 4.10: First step of changing the repository

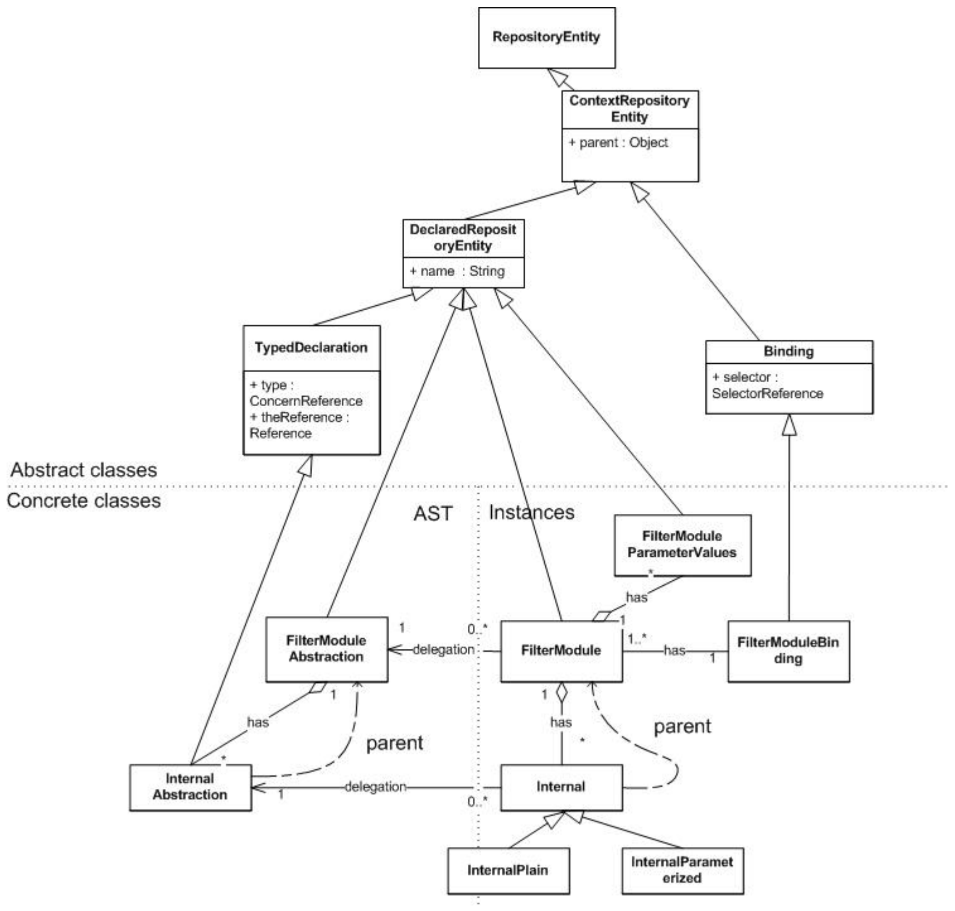

Figure 4.11: Second step of changing the repository

4.4.3 The Status of the Implementation

To demonstrate that the proposed solution is feasible, there are three uses of the parameters implemented for Compose*. This are the parameter for the type of the internal, the single parameter for the selector, and the list of parameters for the selector. The additions make it possible to use the filter module log from the FilterModuleParameters example [Com06], which is shown in Listing 4.15. To get this example working several adjustments to Compose* are in place. In this section we look to how it is implemented and why certain features are implemented differently then it was initially designed.

The Compose* modules which are in involved in the design are COPPER, LOLA, and REXREF. COPPER is the parser of Compose*. LOLA resolves the selectors of the superimposition and REXREF resolves references.

In general we try to resolve the parameters before the rest of the modules are executed. So the rest of Compose* does not need to be adjusted to handle the parameters. This means that the best spot to resolve the parameters is after the module LOLA, because we need that module in the near future to get arguments for the filter module bindings. The resolving is added to the module REXREF, which also handles the other revolving operations. Besides changing REXREF we also need to change the module COPPER, which must create the newly introduced AST classes. The creation of the instances of filter modules and the other elements happens in the module REXREF.

The parameter of the internal type works as follows, COPPER creates an AST class for the parameterized internal. When the instance of the filter module is created from the AST object, the instances of the other objects are created recursively. And when the parameterized internal is being created the type is filled with the parameter, just like an ordinary internal. This is the first operation in REXREF. So when the operation is reached where the internals are resolved, this operation does not see the difference between an non-parameterized internal and a parameterized internal.

The single parameter for the selector works with the same idea as the parameterized internal, when the selector instance is created the value of the parameter is filled in the parameterized selector. The list of parameters for the selector works different because it also needs to be transformed besides being resolved. To demonstrate this transformation we look to the `??walkfunction` from Listing 4.15. We want to transform that filter to:

```
    m : Meta = { {*.walk, *.makeNoise} logger.log}
```

However this multiple matching part does not work in the current implementation of Compose*. This means that we temporary need another transformation in order to get the list of parameters working. The best option to get it working without changing the other modules of Compose* is by transforming the filter element with the list of parameters to multiple filter elements. Each of these filter elements get one selector and copies of the condition and substitution part. For the given example it means that after the transformation we get the filter:

```
    m : Meta ={[*.walk] logger.log, [*.makeNoise] logger.log}
```

The problem with this transformation is that with the ~> operators you get a different result as designed. Because with the list it would mean that a message must not match with any of the list and the transformed filter has a different meaning.

Chapter 5

Towards a Canonical Filter Specification

The current filter element syntax, `target.selector`, only allows matchings on two properties of a message. This is considered as a limiting factor. In this chapter we look at the alternatives to use more attributes of a message and we determine which alternative is the best suitable syntax for Compose*. This syntax is the basis for a design and implementation plan.

5.1 Definitions and Concepts

Canonical Form
We use the term canonical form in this context for a standard and concise form. An example of such form is the current filter element syntax of `target.selector`. Each element has the same form and this form is concise. So the old syntax is a canonical form.

Reified
In the context of Compose* reifying means that something abstract from the base language model is made concrete.

Messages
To explain the concept of a message we take two objects A and B, these are demonstrated in Figure 5.1. In object A the method `bar` of object B is called. In an object-oriented language this call is abstract. It is possible to reify the call to a concrete object: the message. The message object has attributes, such as the target object, the targeted method, and the sender object.

5.2 The Limitations of the Current Syntax

The filter specification is one the key elements of Compose*, it originates from the programming language Sina and is a vital part of the Composition Filters (CF) approach. In the current form we can only filter on two attributes of a message, the target and the selector. The other attributes of the message can only be used for filtering with a condition. To demonstrate why this is limiting for writing filters, we look to an example filter module that is superimposed to a mass mailer class. The idea is that in order to let the mass mailer send mail, the person that

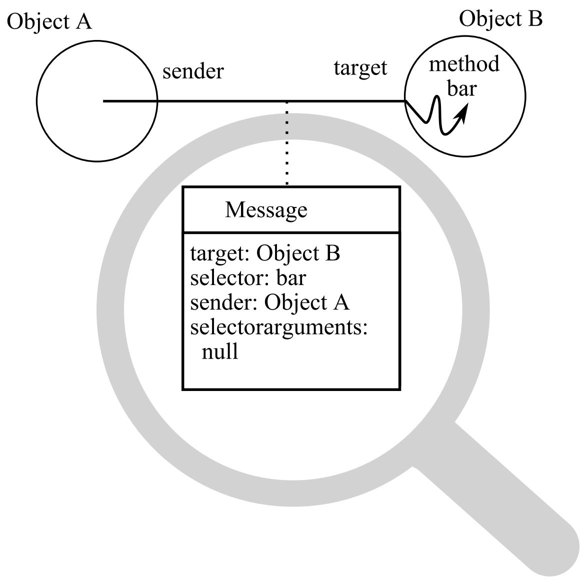

Figure 5.1: An abstract representation of a message

requests the action must have proper authorization to mail at a certain time of the day.

In the example we use a basic company example with the job types as classes. The problem of the company is that too many mass mail is sent and to solve this restrictions are placed on sending mass mail. The restrictions are given in the following set of rules:

1. All secretaries with an access level of five can mail on all times;
2. All secretaries can mail in the morning or in the evening and all other personnel with access level three or higher can mail in the morning or in the evening;
3. Everybody is allowed to mail in the evening.

From this set of rules the filter module from Listing 5.1 is programmed. Because we are limited to only use the target and the selector properties of a message, we need to write conditions to let the filter follow the given rules. This also brings up a second limitation, we cannot use arguments in the used conditions, otherwise we would only have need three conditions: `isTimeStampIn(begintime, endtime)`, `isSender(jobtype)`, and `hasAccesLevel (levelNumber)`. Whereas it looks that using conditions with arguments already solves our problems, we are still stuck writing methods which can be embedded within the filters. The filter module parameter `?rules` makes it possible to provide a different set of conditions for each binding of the filter module authorization.

In the given example we have only used three rules to have a small example, if we want to extend the rule set the filter module becomes more complex and less readable. By applying other syntax options we can determine which syntax provides the functionality we want, and is still short and readable. The message properties we want to use in this scenario are the target, selector, accesslevel, timestamp, and the sender.

```
filtermodule authorization(?rules){
internals
  ruleBase = ?rules;
externals
  mailer = MassMailer.instance();
conditions
  isTimeStampInMorning : ruleBase.isTimeStampInMorning;
  isTimeStampInMidday : ruleBase.isTimeStampInMidday;
  isTimeStampInEvening : ruleBase.isTimeStampInEvening;
  isSenderSecretary : ruleBase.isSecretary;
  hasAccesLevelThree : ruleBase.hasAccesLevelThree;
  hasAccesLevelFive : ruleBase.hasAccesLevelFive;
inputfilters
  rule1 : Dispatch = {isSenderSecretary & hasAccesLevelFive => <mailer.*> mailer.*};
  rule2 : Dispatch = {(isSenderSecretary | hasAccesLevelThree) &
                      (isTimeStampInMorning |  isTimeStampInEvening) =>
                      <mailer.*> mailer.*};
  rule3 : Dispatch = {isTimeStampInEvening => <mailer.*> mailer.*};
  error : Error = {[*.*]}
}
```

Listing 5.1: Example of a filter module for mass mailing

5.3 Possible Syntax Options

5.3.1 Keeping the Current Syntax

The first option we have is to leave things as they are, this means that the given example, Listing 5.1, does not change. The current form is actually a short hand notation, for instance `[foo.bar]` is short for `message.target = foo, message.selector = bar`. In the early years of CF the decision was made to only filter on the target and the selector, this opens the possibility to use the canonical form `target.selector`. With `< >` and `[ ]` the language is extended to indicate the difference between name and signature matching. The advantages of this option are: it is a short notation, it is intuitive, the matching and the substitution have the same structure, and it covers two of the important properties of a message. The last one is also one of the disadvantages, it *only* covers two properties of the message. It limits you because you cannot write filters without the use of conditions, that will filter on something else then target or selector. If we want to construct filters which will only use the time stamp, then we cannot use the matching part of the filter. Another consideration is, that besides the basic set of filters, you can also use a custom filter. With just name and signature matching in combination with the target and the selector, we are limited in our options to customize a filter.

Another disadvantage that shows up in some example applications is the lack of adding arguments to a method call, so if we want to dispatch a method `methodA(int a, int b)` to `methodB(int a, int b, Object c)`, we have to write a custom filter that handles this situation. Listing 5.2 shows the code for this example, it uses a custom dispatch filter which appends the given arguments to the selector argument list. It is also not possible to match on the values of the arguments, however it is possible to create work arounds for it.

```
filtermodule fm(?addedArgument){
  inputfilters
    d : CustomDispatch (?addedArgument) = {[*.methodA] *.methodB}
}
```

Listing 5.2: Adding an argument in a Custom Dispatch filter

5.3.2 Static Forms

The second option is to extend the original canonical form with the missing properties of the message. An example is the addition of the annotations of both target and selector. This gives that we move from `[a.b] c.d` to `[@(classAnnotation)a. @(methodAnnotation)b] c.d`. This already demonstrates the disadvantages of expanding the static forms, first the simple notation becomes bloated and second the added property is just chosen at random; we could have used the sender property instead of annotations. We could spend time to find out which properties are used the most and add them to the standard static form, however using more than two properties gives us a form which is not longer clear and easy. If only the annotation are to be added to the filter syntax then it is acceptable to just add them to the syntax as described in [Nag06].

To relate back on our example, adding the annotations to the static form only makes the example bigger and does not solve any of the given problems. One way to solve the problems is by placing the time and sender properties in the static matching form, for instance we use the form `[timestamp:sender->target.selector] target.selector`, were timestamp can be one time or a time zone, then our example becomes the filter module as demonstrated in Listing 5.3. In the example we have made the sender and the timestamp optional. The disadvantage of this approach is that the static form is good this particular example, but if we want to apply it, for example, on the Pacman example this expended form is useless.

This gives us the insight that expanding the canonical form will limit the language a bit less than the original form, however we must give up the clear and easy layout for it. It is possible to choose the fields for the form with some empirical research, however we still have to make a choice because it will not be possible to include all fields. Moreover, we lose the symmetry between the matching and the substitution part. In the earlier mentioned annotation case, it is

```
filtermodule authorization(?rules){
internals
  ruleBase = ?rules;
externals
  mailer = MassMailer.instance();
conditions
  hasAccesLevelThree : ruleBase.hasAccesLevelThree;
  hasAccesLevelFive : ruleBase.hasAccesLevelFive;
inputfilters
  rule1 : Dispatch = {hasAccesLevelFive => <secretary->mailer.*> mailer.*};
  rule2a: Dispatch = {<!(12.00-15.00):secretary->mailer.*> mailer.*);
  rule2b: Dispatch = {hasAccesLevelThree => <!(12.00-15.00):*->mailer.*> mailer.*);
  rule3 : Dispatch = {<(16.00-8.00):*-> mailer.*> mailer.*};
  error : Error = {[*.*]}
}
```

Listing 5.3: Example of a filter module for mass mailing with a different filter element syntax

```
filtermodule authorization(?rules){
internals
  ruleBase = ?rules;
externals
  mailer = MassMailer.instance();
conditions
  hasAccesLevelThree : ruleBase.hasAccesLevelThree;
  hasAccesLevelFive : ruleBase.hasAccesLevelFive;
inputfilters
  rule1 : Dispatch !sender.target.selector! =
    {hasAccesLevelFive => <secretary.mailer.*> mailer.*};
  rule2a: Dispatch !(timestamp).sender.target.selector! =
    {<(15.00-12.00).secretary.mailer.*> mailer.*);
  rule2b: Dispatch !(timestamp).sender.target.selector! =
    {hasAccesLevelThree => <(15.00-12.00).*.mailer.*> mailer.*);
  rule3 : Dispatch !(timestamp).sender.target.selector! =
    {<(16.00-8.00).*.mailer.*> mailer.*};
  error : Error = {[*.*]}
}
```

Listing 5.4: Mass mailing example with dynamic filter element specification

possible to place annotations in the substitution, however it would not mean anything[1].

Another issue is how we must place a form with more than two fields in a syntax, if we separate them with dots then we would get a lengthy shorthand like `[a.*.b.*.*]` which is error prone. Adding prefixes, for example “@” for annotations, “♯” for targets, will produce an unusable language with an enormous amount of operators, it also gives some problems with the filter module parameters, causing constructs such as `@?parameter`, which are not easy to use.

5.3.3 Dynamic Forms

To solve the problems of the static form, we can also look to options that will make the form dynamic, solving the syntax problem mentioned earlier with the wildcards in section 5.3.2. An example is given in Listing 5.4, were we have worked out the mass mailer example into a dynamic form. If you take a good look to the example then you notice that this does not differ much from the idea with prefixes for every message property. They only differ in syntax, but in both you can define the properties of the message you want to use. This also gives that we have the same advantages and disadvantages in the conceptual field, but the grammar technical considerations are different.

With this option we gain the possibility to use all the message properties in the filter specification and it is even possible to use specific properties of a child class of message. So if we would use our extended message TimedCustomMessage, we can just use `target.(TimedCustomMessage .Timestamp)`. This gives that we can use customized filters and messages easier then in the current situation. A disadvantage is that the readability drops, since `foo.bar` can have different meanings. In the examples the substitution part is kept in the `target.selector` form, it is of course possible to write the substitution part in the same form as the matching part.

[1]It does not mean anything in *most* cases, it is however possible to get a meaningful construction. If you change the annotation, you can trigger the next filter in line. We will not discuss whether that is a correct use of filters.

```
filtermodule authorization(?rules){
internals
  ruleBase = ?rules;
externals
  mailer = MassMailer.instance();
conditions
  hasAccesLevelThree : ruleBase.hasAccesLevelThree;
  hasAccesLevelFive : ruleBase.hasAccesLevelFive;
inputfilters
  rule1 : Dispatch = {hasAccesLevelFive, sender = secretary, selector = mailer,
                      target := mailer};
  rule2 : Dispatch = {!(timestamp > 12.00 & timestamp < 15.00),
    (sender = secretary | hasAccesLevelThree,
     selector sig mailer, target := mailer};
  rule3 : Dispatch = {!(timestamp > 8.00 & timestamp < 16.00) selector sig mailer,
                      target := mailer};
  error : Error = {}
}
```

Listing 5.5: The mass mailer with filtering on message properties directly

In the demonstrated example the declaration of the form is making the code bloated, one way to solve this is by declaring the form once in a filter module. This can also be done for the prefixes, thus the set of prefixes can be filter module specific. While this solves one problem, the problem that the filter elements semantics becomes filter bounded does move to the filter module level. And a filter element can still have different semantics in different filter modules.

5.3.4 Message Properties

In the other options one feature is always coming forward and that is the possibility to use all the properties of a message. Besides using the standard properties of the message, we also would like to have a possibility to use properties of custom messages, which are extended from the original message object. The most obvious language construct is to write it like Listing 5.5. With the use of the properties we also need to introduce operators. In the example we see the use of the operators '=', ':=', and 'sig'. Their respective meanings are: match, assign, and signature match. Adding operators introduces more complexity to the language, because we will have more options then before. However the increase in complexity can be very limited if we do not introduce too much operators.

The advantage is that we can use every property we want. And like the second option it is possible that the user introduces new properties. It also opens the possibility to use constructs like `target = foo | selector sig bar`, where the message target is foo or the message selector is in the signature of bar. This gives an increase in expressiveness. The disadvantage is that we cannot longer use the symmetric matching and substitution part. Without this symmetry we can stop using the wildcard. The wildcard becomes obsolete in this form, because it is no longer needed to fill up the unused properties in a canonical form. A conceptual disadvantage with the custom message fields is that a combination of field and operator can be meaningless or just plain wrong, like doing a signature match on a method instead of a class. This is solvable by implementing it with correct error reports for the user.

To limit the amount of operators we also allow the use of methods op the properties. So `target = foo` can also be written as `target.equals(foo)`. More important is that we do not need to assign an operator to constructs like `TimedMessage.day.whenPigsFly()`. The last example also shows the handling of custom messages. Whereas `target` is a shorthand for `Massage.target`, we can also write filters for extended messages, which are ignored by messages of another type.

5.3.5 Evaluation of the Filter Syntax Options

The main part of the evaluation is the choice between an easy and simple syntax or a full expressive syntax. Extending the syntax so that we can keep a canonical form is not practical, we can add one or two properties, but with more than two properties the syntax becomes bloated and unreadable. Creating a dynamic form, thus that we define the meaning for `firstProperty .secondProperty` for each filter module, is not user friendly because the user cannot directly see how the filter matches. So the last option is the preferred one, thus the combination of a message property name, an operator, and a value.

With the introduction of the new filter syntax we also need to determine whether it will replace the old syntax or whether we allow both in the language. In order to keep them both it must be possible to determine whether a filter syntax is old or new style and it must be possible to transform the old style to the new form to achieve backwards compatibility. The first issue is solved fast, the old syntax uses the tokens ".", "[ ]", and "<>", in such a way that we can create a rule to determine whether a part is old style or new style. This brings us to the second hurdle, whether it is possible to transform the old style to the new style. In Appendix D the transformation from the old to the new syntaxis defined and we see that this transformation does not give any problems. So we can have both the old and new syntax in the Compose* language.

Whereas normally the suggestion is not to keep two styles in the language and just use one. In this case the argument that all the documentation written in the last twenty years is written in the old style gives the advantage for keeping both styles in the language. However, it is necessary to write good documentation in the new syntax and to point out the differences between the old and the new style. When everything is implemented and it all works correctly, we can research whether we still need to support the old syntax. The choice whether we will still support them both in the future will depend on usability, readability, and how hard it is to support two layouts. But to make a well founded decision we need empirical values and we can only gather such information after we have created the new syntax with both styles and use it for while.

5.4 Design of the New Filter Syntax

With the introduction of the new filter syntax we can also introduce other features. The new syntax is shown in Listing 5.6. We have chosen to combine the condition and matching parts, making it possible for instance to place a condition part between two matchings. To keep the filter element structured, we keep the substitutions parts at the end of a filter specification. The ElemCompositionOperator changes from "," to ";" in order to see the difference between filter elements. Now the condition parts and matching parts are mixed, they also get the same logical operators. We stick to the functional complete set [BKLM91] of AND, OR, and NOT. Because this set is functional complete we do not need to add other logical operators. We also keep the standard precedence of the logical operators, which can derived from the syntax.

```
GeneralFilterSet ::= GeneralFilter (FilterCompositionOperator GeneralFilter)*
FilterCompositionOperator ::= ‘;’
GeneralFilter ::= FilterName ‘:’ FilterType [‘(’ ActualFilterParameters ‘)’];
                         ‘=’ ‘{’ [FilterElements] ‘}’
FilterElements ::= FilterElement (ElemCompositionOperator FilterElement)*
ElemCompositionOperator ::= ‘;’

FilterElement ::=  [ConditionAndMatchingPart ‘,’] AssignmentPart
                   | ConditionAndMatchingPart

ConditionAndMatchingPart ::= OrExpr
OrExpr ::= AndExpr (‘OR’ AndExpr)*
AndExpr ::= PrimaryExpr (‘AND’ PrimaryExpr)*
PrimaryExpr ::= [NOT] PlainExpr;
PlainExpr ::= ‘(’ OrExpr ‘)’ | ConditionPart | MatchingPart

ConditionPart ::= ConditionLiteral
ConditionLiteral ::= ‘True’ | ‘False’ | ConditionName
MatchingPart ::= FieldName MatchingOperator (Value | ‘{’ Value-LIST ‘}’)
                 | FieldOperation
MatchingOperator ::= equals | equalsOne | equalsAll | sig | OtherOperators

AssignmentPart ::= Assignment (AND Assignment)*
Assignment ::= FieldName ‘:=’ Value | ‘{’ Value-LIST ‘}’ | FieldOperation
AssignmentOperator ::= ‘:=’ | OtherOperators

FieldName ::= Identifier
FieldOperation ::= fqnwithArguments
AND ::= ‘&’ | ‘,’
OR ::= ‘|’
NOT ::= ‘!’
Value ::= Identifier | Number | Parameter | ParameterList
Methodcall ::= Identifier ‘(’ Value-LIST ‘)’
OtherOperator ::= (* To Be Specified *)
```

Listing 5.6: New filter element syntax

The matchings can be written in two ways: with full method property functions, like `message.target.equals(foo)`, or with shorthands, like `target = foo`. For now we have only identified five shorthands: the equals, assign, signature match, equalsOne, and equalsAll. The exact operator designation is implementation dependent and trivial. Other short hands will be included when we identify more frequently used operations.

5.4.1 Operator Designation

In the syntax there are two kinds of operators, the ones in a matching or substitution part and the ones between the matching and the condition parts. To start with the latter, we will use the logical operators which are now solely being used for the condition part of a filter. This are the AND ("&"), OR ("|"), and NOT ("!"). These operators are completed with a set of parenthesis to indicate ordering. For the assignment parts only an AND is useful, so we will use a comma for that particular AND.

The other operators are short hands for message field operations, so we do not add any functionality with them. The actual set of operators that we will use is not relevant for the analysis. The only thing that is relevant is that the operators can only have one meaning, thus they must mean the same in the matching or the substitution, to be more precise we will determine whether a filter part is a matching or a substitution part by its operator. Thus a filter part with a "=" is a matching part and one with ":=" is a substitution part.

5.4.2 Limiting the Operations

With the possibility to use the operations of a message we must also define the limitations of this construct. The goal of allowing the operations besides the operators is that we cannot define a symbol for each operation, so we will allow simple operations in the matching parts. The filter specification `selector.getParameter(1).getSignature().equals(foo)` is one the examples of what we do not want: long operation calls. Even without the number 1 in the `getParameter` call, this is something you might want to place in a custom method. Limiting the possibilities to the user with help of the syntax will probably result in unwanted situations.

Chapter 6

Adjustments to the Language

Besides the larger language changes there are some smaller changes as well. These smaller changes have a low impact on the syntax and the semantics. However, the changes still need to be thoroughly examined before they are applied.In this chapter we look to these minor changes in the language.

6.1 Language Style

Most programming languages contains syntactic sugar so that the code can be easily read and still be easily parsed. This syntactic sugar for a syntax often contains the common tokens like a semicolon to mark the end of an operation and curly brackets to mark syntax blocks. Because it is just the syntactic sugar, it does not affect the semantics and actually it is just a part of the user interface of a programming language.

In the begining of the Compose* project[1] there was a debate on the use of *curly bracket* style from languages like C, Java, and C#, versus the *begin-end* style from Pascal and related languages. The main part of the discussion is not the use of the tokens but whether the syntax must contain the possibility to optional use the starting keyword with the identifier:

```
filtermodule = ‘filtermodule’ NAME ‘{’ filtermoduleParts ‘}’
             |‘filtermodule’ NAME ‘begin’ filtermoduleParts
              ‘end’ [‘filtermodule’ [NAME]]‘;’;
```

The possibility of writing down the keyword and the identifier is only possible with a look-ahead of two because of the used semicolon in the end. Without this semicolon the syntax would need a look-ahead of three, which is a sign of a bad language design from a parser building point of view. In the end of the discussion the choice was made to keep both styles. After two years we can see that this was not the best choice, mainly because the begin-end style was never programmed correctly, thus with fixed semicolon instead of an optional one[2].

If we take the requirements from section 3.2, we can directly conclude that to make the language

[1]The discussion might already be there with the predecessors of Compose*.

[2]If you check the CVS you can find that the syntax had a fixed semicolon, however that token was placed wrongly in the syntax as well [Com06].

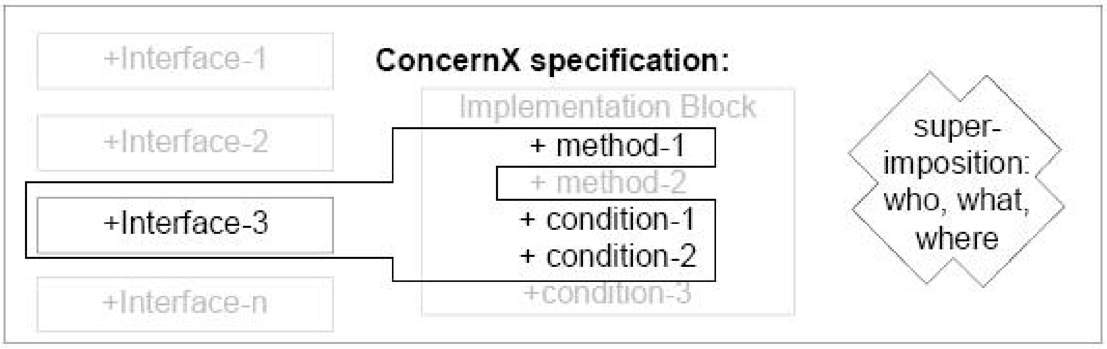

Figure 6.1: The idea behind condition and method binding

more concise, we must switch to one style. It does not affect the expressiveness, because the two styles offer a way to express the same thing. So if we have to choose for a style, the best choice is to go for the C style, because this is widely used and the old Pascal style is nearly obsolete. The last (main stream) language which actually supports begin-end style is Delphi. So to end this old debate, the single style base has been introduced into Compose*.

6.2 Filter Module Method Block

The purpose of the method block in the filter module is to provide an extra check to look whether the used methods in the filters exists. However, this is not the original purpose of the method fields, original it was used to introduce methods which are stated in the implementation part. The implementation part is different implemented then it was originally designed and there are no loose methods but classes in the implementation part. Thus making the methods field obsolete for its old task. Adding robustness in a programming language is a good practise, however, in this case it is better to remove the method block. The methods that are used in the filters can be derived by the compiler and therefore the methods block is redundant. Another problem is that it does look like the conditions block, but it means something completely different.

6.3 Condition and Method Binding

The condition and method binding were introduced in the language with the initial superimposition part [Sal01]. In the design of the superimposition the assumption was made that the conditions and methods would be declared directly in the concern and therefore a construct, like Figure 6.1, was needed.

In the later implementation of Compose* the choice is made to use class declarations in the implementation part of a concern instead of loose methods. This results in the situation that all the methods are in one class and can be used in a filtermodule by creating an internal or an external. This conclusion was also drawn in [Vin04].

So in theory, as long as we keep the implementation part so that it is uses classes instead of loose methods, the condition and method binding are redundant. Therefore we remove them from the Compose* language. As long we keep the implementation part as it is, we must reevaluate

```
filtermodule queue(?internalType){
  internals
    req, ans : ?internalType;
  conditions
    ifBusy = Queue.isBusy(req, ans);
    isIdle = Queue.isIdle(req, ans);
  inputfilters
    d1 : Dispatch = {isIdle => [*.*] *.*};
    d2 : Meta = {ifBusy => [*.*] req.add}
}

implementation in C# by Queue as Queue.cs{
  public class Queue{
    public static boolean isBusy(Object req, Object ans){
      return ((Counter) req).getAmount() >= ((Counter) ans).getAmount());
    }

    public static boolean isIdle(Object req, Object ans){
      return ((Counter) req).getAmount() == ((Counter) ans).getAmount());
    }
  }
}
```

Listing 6.1: Using an argument in a condition declaration

this decision when the implementation part changes. There are two reasons for temporary removing redundant syntax, first it saves the maintenance of both code and documentation and second it closes any possible problems those two block might cause. The idea of condition and method binding is correct and useful, however it is not needed in the current implementation of Compose*.

6.4 Arguments in Internals, Externals and Conditions

In the current implementation of Compose* it is not possible to add arguments to the internal, external and condition declaration. With the introduction of filter module parameters it is necessary to have the ability to use arguments in both declarations. One of the uses is demonstrated in Listing 6.1[1]. In that listing we try to make a queue filter module where we use two internals as arguments. The interesting part of this example is that using the internals or externals for a condition makes it possible to write the filter modules in a more generic way. Of course there are more arguments you might want to add, like primitive values such as integers or strings. However, primitive values break the language independence requirement. Therefore only filter module parameters, internals, and externals can be used as arguments for an external or condition declaration.

[1]We have chosen for a static method to write a short example. An alternative is to include Queue as an internal.

Part II

Annotated Reference Manual

Chapter 7

Concern

Concerns are the distinctive building blocks of a Compose* application, in addition to the building blocks of the base language. Conceptually concerns are an extension to classes. Concerns are declared in files with the .cps extension.

A concern has three different parts: zero or more filter modules, an optional superimposition part, and an optional implementation part. The filter modules are superimposed on classes by the filter module binding field of the superimposition. The implementation part contains language dependent code of the concern.

Syntax

The syntax of a concern is shown in Listing 7.1. A concern name must be unique for the package where it is declared. It often has the same name as the .cps file in which it is declared. It is not possible to place two concerns in one file. The ordering of the filter modules, superimposition, and implementation is fixed.

Semantics

The concern is the main language entity of Compose*, it consists of one or more filter modules, an optional superimposition part, and an optional implementation part. How these are combined depends on how you use a Compose* concern. There are two different types of usage of a concern. We will look here to the usage that comes from the conceptual idea, which assumes

```
Concern ::= ‘concern’ Identifier [‘in’ Namespace]
            ‘{’ (FilterModule)* [SuperImposition] [Implementation] ‘}’
Namespace ::= Identifier (‘.’ Identifier)*
ConcernBlock ::= ‘{’ (FilterModule)* [SuperImposition] [Implementation] ‘}’
```

Listing 7.1: Concern syntax

```
concern aConcern in aNamespace{
  filtermodule A{
    ...
  }

  filtermodule B{
    ...
  }

  superimposition{
    ...
  }

  implementation{
    ...
  }
}
```

Listing 7.2: Abstract example of a concern

that a class is a concern. This means that every class can be written as a concern, with an implementation part and one or more filter modules. To superimpose these concerns onto other classes, a concern is used with a superimposition part; this concern can be seen as a sort of aspect specification.

Examples

A concern heading consists of a concern nameand the package in which the concern is defined. The package is also optional, if none is specified the concern is located in the root of a project, the usage of packages works the same as in other languages that supports packages or namespaces.

In the concern it is possible to create one or more filter modules, one superimposition block and an implementation part. A possible concern is demonstrated in Listing 7.2.

Legality Rules

- The identifier of a concern must be unique for the namespace where it is declared in;
- The ordering of filter modules, superimposition, and implementation is fixed;
- There can be maximal one superimposition and implementation part.

Commentary

Different Usages of a Concern

The Compose* concern has a multi-role usage due to the way it is built up. The different combinations of concern elements and their explanations are shown in Table 7.1 [BA05]. The usages in this table are all the possible usages of concerns.

Filter Module(s)	Super-imposition	Implemen-tation	Explanation
No	No	Yes	CF conventional class
Yes	only to self	Yes	CF conventional CF class
Yes	Yes	Yes	Crosscutting concern with implementation
Yes	Yes	No	"Pure" crosscutting concern, no implementation
Yes	No	No	CF abstract advice without crosscutting definition
No	Yes	No	Superimposition only (of reused filter specs.)
No	Yes	Yes	CF class or aspect with only reused filter specs.

Table 7.1: Different concern usages

Concern Parameters
There has been the idea to use global variables in a concern. These variables were declared in a concern with the concern parameters block. However it became clear that it is not possible to assign these parameters anywhere in the application, because with the introduction of superimposition it is no longer useful to instantiated concerns manually with arguments. Therefore the concern parameters are removed from the Compose* syntax.

Chapter 8

Filter Module

The filter module is a reusable entity that holds two sets of filters: the input filters and the output filters. These filters can use objects and conditions, which are also defined in the filter module. A filter module is superimposed on a class and the filter module can extend the signature and the behavior of the class on which it is superimposed.

Syntax

A filter module has a unique identifier in a concern; the uniqueness is defined over the name and not the name in combination with the parameters.The ordering of the blocks of a filter module is fixed, thus it is only possible to use them in the ordering as shown in the syntax of Listing 8.1. The filter module template is demonstrated in the concern template in Listing 8.2. All the blocks are optional, making it possible to write a completely empty filter module.

Semantics

A filter module is an entity that holds filter sets which are composed together. The objects and the conditions that are used in the filters are also declared in the filter module. Figure 8.1 demonstrates how a filter module can be visualized in a schema. The schema shows a filter module superimposed to an object, this object is called the "inner" object. Any message that is sent to the object goes through the input filters and any message that is sent goes through the output filters. Both filter sets can use internals, externals, and conditions. These elements are declared in the internals, externals, and conditions blocks.

```
FilterModule ::= ‘filtermodule’ FilterModuleName [FilterModuleParameters] ‘{’
                 [Internals] [Externals] [Conditions]
                 [InputFilters] [OutputFilters] ‘}’;
```

Listing 8.1: Filter module syntax

```
filtermodule ( ... ) {
  internals
    ...
  externals
    ...
  conditions
    ...
  inputfilters
    ...
  outputfilters
    ...
}
```

Listing 8.2: Filter module template

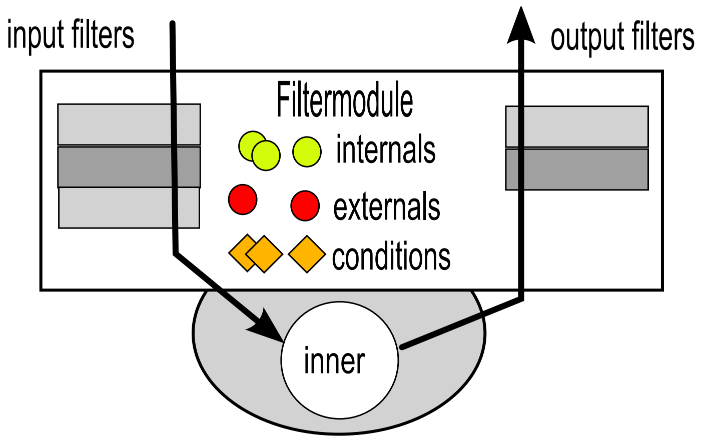

Figure 8.1: Schematic representation of a filter module

```
filtermodule dynamicstrategy{
  internals
    stalk_strategy : pacman.Strategies.StalkerStrategy;
    flee_strategy : pacman.Strategies.FleeStrategy;
  conditions
    pacmanIsEvil : pacman.Pacman.isEvil();
  inputfilters
    stalker_filter : Dispatch = {!pacmanIsEvil =>
      [*.getNextMove] stalk_strategy.getNextMove};
    flee_filter : Dispatch = {[*.getNextMove] flee_strategy.getNextMove}
}
```

Listing 8.3: Dynamic strategy filter module from Pacman

Examples

A filter module can consist of parameters, internals, externals, conditions, input filters, and output filters. The blocks in which these filter elements are declared are all optional, so the shortest filter module you can write is `filtermodule fm {}`. Listing 8.3 and Listing 8.4 show two example filter modules taken from the Compose* example directory [Com06]. What the separate filter module elements do can be found in their respective chapters, the examples are given to get an idea of how to build up a filter module. The ordering of the blocks is fixed, so for instance, it is not possible to place the conditions block before the internals block.

Legality Rules

- The identifier of a filter module must be unique for the concern where it is declared in;
- The blocks in a filter module must be in the following ordering: internals, externals, conditions, input filters, and output filters;
- You can only have one declaration of the above mentioned blocks, for example there can be only one set of internals.

Commentary

Language Independence

It is not possible to use primitive values, like int, char, and bool, as internals and externals, this is a result from the language independence we want to have in Compose*. This means that only objects can be used as internals and externals. Object references are language independent and thus usable in a language independent environment. The usage of primitive values in the

```
filtermodule Enqueue{
  externals
    playlist: Jukebox.Playlist = Jukebox.Playlist.instance();
  inputfilters
    meta : Meta = { True => [*.play] playlist.enqueue}
}
```

Listing 8.4: Enqueue filter module from Jukebox

```
1 filtermodule rollbackcounter{
2   internals
3     counter : int;
4   inputfilters
5     e : Error = {(if counter > 10) => [*.*]};
6     m : Meta = {[*.rollback] counter++;}
7 }
```

Listing 8.5: A filter module with a primitive value

language would break language independence because the primitives have different ranges in each language[1]. There are ideas to introduce a set of primitive values for usage in the filter module, but these ideas always get stuck on the fact that you need to set a range for the primitives and they often conflict with the requirement to keep the language as concise as possible. An example of a rejected solution is the subtyping of integer values like in the programming language Ada [Ada95]. With Ada it is possible to create a subtype of integer by declaring it as a subtype of integer and by setting a range, for instance from one to ten. A possible Ada type declaration is `type OwnInteger is Integer range 0 .. 256;`

If we look on how we would put a primitive into action we can see that we can do the same with an object. For example, if we take a filter module that allows ten times the call `rollback` and gives an error the eleventh time, then the filter module does need a counter to keep track on how many times the call `rollback` has been send to that filter module. In Listing 8.5 the code with a primitive value is worked out. In line 3 of Listing 8.5 an integer is declared, we assume that by default the value will be zero. An alternative is to create an internals declaration like `counter : int = 0;`. This value counter is used in line 5 to check whether the amount of rollback is still below ten. To get the filter module count every call to the method `rollback` we need a statement as `counter++` (Line 6). Because the internal is only accessible in the filter module we need to use a statement in the filter that raises the value of counter. So with the use of primitive values in the filter module we also need to add operators to use the values.

Listing 8.6 shows the same behavior only with an object that inherits from the Integer object[2]. The internal declaration now uses a Counter class type, the if-statement is replaced with a condition declaration as seen in line 5 and the raising of the counter is now been handled with a

[1]For bool (or boolean) this is not a real problem, because generally it has only two values: True and False. However, if we would find (or create) a language that uses fuzzy values for its booleans then we have the same range problem with booleans as with, for example, integers. In such scenario the question is whether the fuzzy range is from one to zero or from hundred to zero?

[2]Extending from Integer makes that you can add your own custom methods like *raise()* and *isGreaterThenTen()*.

```
1 filtermodule rollbackcounter{
2   internals
3     counter : Counter; // extends of Integer
4   conditions
5     greaterThenTen : counter.isGreaterThenTen();
6   inputfilters
7     e : Error = {greaterThenTen => [*.*]};
8     m : Append = {[*.rollback] counter.raiseCounter}
9 }
```

Listing 8.6: A filter module with an object instead of primitive value

```
1  filtermodule agenda(?externalObject){
2    externals
3      secretary : Example.Secretary = ?externalObject;
4  }
5
6  superimposition{
7    filtermodules
8      selA <- example(Example.Secretary);
9      selB <- example(Example.Secretary);
10 }
```

Listing 8.7: A parameter as external object

function of the class Counter. This gives that using an object is preferable to a primitive value, because we can use the methods of the internal and we do not need to introduce mathematical operators, like ">" and "+", in the filter syntax. So we can achieve the same with an object or a primitive, only with the primitive we have problems with breaking language independence and we have a syntax that is less concise.

Therefore we conclude that we do not need primitive values in the language and we only need method calls and object references. Not adding primitive values to the syntax saves us from the issues mentioned above. First it offers us the possibility to keep the filter module as concise as possible and second if we would consider adding primitive values and some operators, we would never become as expressive as the base language, and it is not our goal to copy all the possibilities of the base language. With this we can close the discussion whether to introduce primitive values in the filter module.

The Methods Block

Originally, the filter module also had a method block. As mentioned earlier in section 6.2 it has become obsolete and it has been removed from the language.

Combining Internals and Externals

The internals and externals are the local variables of a filter module. If we look at the example programs in Compose*, then we see that the externals are only used in combination with the singleton design pattern [GHJV95]. Because the usage of the externals is limited, we can consider combining both blocks into one block called *variables*. Whether the combination of both blocks is desirable depends whether we want to keep a visible difference between an internal and external declaration. A solution is to mark externals with a "*", like C++ uses for pointers, so that all the variables are in one field and there is still a (visible) difference between internals and externals.

However, with the introduction of filter module parameters, the usages of the internals and externals fields are extended. For the externals it means that we can get programs like Listing 8.7. In that example one instance of the given object is used for all the classes that are bound in the filter module binding, thus in this particular example there are two instances of the class `Example.Secretary`, one for the selection of classes of selA and of the classes of selB[1]. The external declaration on line 3 has an identifier, a fixed type, and a flexible object. We can apply the "*" construction also in this situation, but then you can only derive the meaning of the code with the "*". If we would introduce constructor calls for internal declaration we get a readability problem. Consider the line of code: `variable : type = ?parameter;`. If this

[1]The selections can have an overlap.

would be an internal in the variable block it means that the parameter is cloned, an external in the same variable field is then `*variable : type = ?parameter;`, which means the same as the construct in Listing 8.7. Due to the new options introduced by the parameters, keeping the distinction between internals and externals becomes more preferable over the combination of the two blocks.

So with the introduction of parameters, we need to reevaluate the idea of combining the internals and externals fields. And can conclude that with the choice of adding parameters to the filter module, it is better to make a clear distinction between externals and internals, and the best way to do so is to keep them separated.

Block Ordering
The order of the blocks in the filter module is fixed, making it impossible, for instance, to place externals before internals and to have two conditions blocks. It is a matter of readability whether mixed ordering and multiple occurrences of blocks are better then fixed ordering and single occurrence of blocks. Because of the filter operators and how filters work together we only allow one input and output filter set per filter module and that we do not break it up in sub parts. If we would break up the filter sets, then it becomes harder to read how a filter set behaves. Another point is that if we would allow the declaration of internals and externals between filters, then users can get confused whether the scope is the whole filter module or just until the next internals or externals block. Therefore we chose to use the fixed one-of-a-type syntax and that no mixes of blocks are allowed.

Chapter 9

Filter Module Parameters

Filter module parameters can be used to bind variables to a filter module when the filter module is bound to an object. It is possible to use a single parameter value or a list of parameter values; these will be referred to as *single parameter* and *list of parameters*. The scope of the filter module parameters is the filter module where they are declared.

Syntax

Parameters declarations are placed right after the name of the filter module. A parameter must have a “?” or a “??” prefix, followed by an identifier. The single question mark is for a single value and the double question mark is to mark a parameter list. After you have declared the parameters you can use them in the filter module. The formal syntax is stated in Listing 9.1.

You can use a single parameter to parameterize the following elements of a filter module:

- Internal type
- External type
- External initialization
- Condition declaration
- Filter type
- Filter arguments
- Target
- Selector

```
FilterModule ::= ‘filtermodule’ FilterModuleName [FilterModuleParameters] ‘{’
                   [Internals] [Externals] [Conditions]
                   [InputFilters] [OutputFilters] ‘}’
FilterModuleParameters ::= ‘(’ [ParameterDefinition-LIST] ‘)’
ParameterDefinition ::= Parameter | ParameterList
Parameter ::= ‘‘?’’Identifier
ParameterList ::= ‘‘??’’Identifier
```

Listing 9.1: Parameter declaration syntax

```
 1 filtermodule logging(?externalDeclaration, ?methodreference, ??setOfSelectors){
 2   externals
 3     logger : Logger = ?externalDeclaration;
 4   conditions
 5     c : ?methodreference;
 6   inputfilters
 7     m : Meta = {c => [*.??setOfSelectors] logger.log}
 8 }
 9 
10 superimposition
11   filtermodules
12     self <- logging(Logger.instance(), inner.isActive(), {print, printAll});
```

Listing 9.2: Example of a filter module for logging

Lists of parameters can only be used in the selector of a matching.

Semantics

The parameters are variables that can be used in the scope of a filter module. The values of the parameters are assigned with the filter module binding. Parameter types are inferred automatically; the declaration of a parameter is only the identifier of the parameter with a prefix.

Examples

The first example, Listing 9.2, shows a filter module that handles logging. Logging is a common concern and it is likely that a filter module like this will end up in a concern library. The given arguments, in line 1 of the example, are an external declaration, which can be an object or a method that will return an object, a method reference, which must refer to a method that returns a boolean value, and a set of selectors. The external and condition declaration are like an ordinary declaration only the right hand side of the declaration has now a parameter, thus this part is known when the filter module is bound to an object. In the filter `??setOfSelectors` is used as argument for the selector part, it means that every item of the list will be evaluated, in this example this means that there will be a name matching for the functions `print` and `printAll`, which are given in the filter module binding in line 12. In that binding the other two arguments are given as well.

The second example, Listing 9.3, is a generic single inheritance filter module. The internal declaration is like an internal declaration with a parameter and with the filter module binding a proper class type must be provided. The filter in line 5 first uses a signature matching on inner to have the possibility of overriding methods. Listing 9.4 demonstrates a generic encryption filter module with a parameterized filter type and a parameter list for the selector of the matching part.

```
filtermodule inherit(?internaltype){
  internals
    parent : ?internalType;
  inputfilters
    d : Dispatch = {<inner.*> inner.*, <parent.*> *.*}
}
```

Listing 9.3: Example of a generic inheritance filter module

Legality rules

- The identifiers of the parameters must be unique for one filter module, so declaring both `?para` and `??para` is not allowed. So an identifier cannot occur in one filter module with different prefixes;
- Although you do not need to specify types, there is a typing system for the parameters. If there is a typing error the compiler will let you know, for more information on the technical details consult the implementation details in chapter 4;
- When you bind a filter module on an object, but you do not provide the correct amount of arguments, Compose* will give an error.

FAQ and Hints

FAQ

Q: *Because ?parameter is a single parameter and ??parameter a list of parameters, does that mean that ???parameter is a list of parameter lists?*
A: No, because all “???(?)*” prefixes are too complex to use and during the design of the filter module parameters no general usage could be found for ???parameter.

Q: *How can you determine the types, for example an internal type or a selector, that you need to fill in the filter module binding?*
A: You must derive the type information from the filter module specification.

Hints

The given arguments for the parameters can be a string or a reference to an object. Compose* handles the type conversions for string to object reference and vice versa, but when a wrongly typed argument is provided, Compose* will give an error.

In Listing 9.2, line 3, the type of the external is fixed and the initialization string is made generic with a parameter. This construction is more robust than the construction where the type of the external is a parameter as well. Because with a static type we know that initialization String

```
filtermodule encryption(??setOfSelectors, ?filtertype){
  outputfilters
    e : ?filtertype = {[*.??setOfSelectors]}
}
```

Listing 9.4: Example of an encryption filter

must give back the static type or a type that inherits from the static type, so if the external declaration is type sound then we also know a part of the signature. If we take Listing 9.2, we see that we can write a reusable logging filter module, the user that will reuse this filter module must provide an object with a certain signature, in this case we want to have at least the method `log` in the signature. With an external with a generic type it is not possible to enforce this.

Commentary

The prefixes are inspired by Sally [Sal03] and LogicAJ [Log05], which both extend AspectJ with parameters. They make the distinction between a single parameter and a parameter list with the different prefixes "?" and "??". When we were looking to parameters we wanted a way to make them distinct from literals, so adding a prefix is an obvious solution for this.

Typing
Typing is left out of the grammar because we infer types automatically in Compose*, this means that the user does not have to write down the type, which lowers the complexity of the syntax. If there is a problem with incompatible types or parameters used on two places and those places do require different types, then Compose* will give an error.

Chapter 10

Internals

Internals are the object instances that are instantiated for each instance of a filter module. Because of this, internals can be used in situations where each instance of a filter module must have its own instance of an object, for example to hold a state or for inheritance by delegation.

Syntax

The internal declaration has two parts, which are separated with a colon. The left hand side contains the identifiers of the internals and the right hand side the type of those identifiers. The formal syntax is defined in Listing 10.1.

The types are defined by fully qualified names. You can refer to an internal in the conditions field, the filter parameters, and the Target. Internals can only be objects, primitive types are not allowed.

Semantics

When the filter module gets instantiated, then all its internals get instantiated using the constructor with no parameters, this is often the default constructor. Listing 10.2 shows the binding of a filter module isActive to classes A and B. What this code does at run time is shown in Figure 10.1, every instance of A and B gets its own instance of the filter module isActive and each instance of the filter module gets its own instance of the internal `state`.

```
FilterModule ::= ‘filtermodule’ FilterModuleName
                  [FilterModuleParameters] ‘{’
                  [Internals] [Externals] [Conditions]
                  [InputFilters] [OutputFilters] ‘}’
Internals ::= ‘internals’ (Identifier-LIST ‘:’ Type ‘;’)*
```

Listing 10.1: Internals syntax

```
filtermodule isActive{
  internals
    state : State;
  conditions
    notActive : state.notActive();
  inputfilters
    d : Dispatch ={<state.*> state.*};
    e : Error (state) = {notActive => [*.*]}
}
superimposition{
  selectors
    sel = {C | isClassWithNameInList(C,['A', 'B'])};
  filtermodules
    sel <- isActive;
}
```

Listing 10.2: Example 2, a filter module to hold a state

The effect is that every instance of an object gets its own instance of the filter module and thus indirect the internal, making it suitable for storing states independently and for inheritance by delegation. With the proper filter construction, like `d : Dispatch = { <internal.*> internal.*}`, it is possible to extend the signature of a superimposed object with the signature of the internal. This means that you do not need to access the internal from outside the filter module, because you can directly use the object on which is superimposed.

Examples

In the Pacman example (Listing 10.3), the strategies are internals; this gives that they are instantiated for each filter module. An internal is declared with a fully qualified name of a class.

Legality Rules

- The identifier of an internal must be unique for the filter module where it is declared in;
- The type must be a fully qualified name of a class and this class must exist;
- In order to be instantiated, the class for the internal must have a constructor without parameters;
- When an internal is declared but not used you get a warning from the compiler.

```
filtermodule dynamicstrategy{
  internals
    stalk_strategy : pacman.Strategies.StalkerStrategy;
    flee_strategy : pacman.Strategies.FleeStrategy;
  conditions
    ...
```

Listing 10.3: Example 1, a piece of Pacman

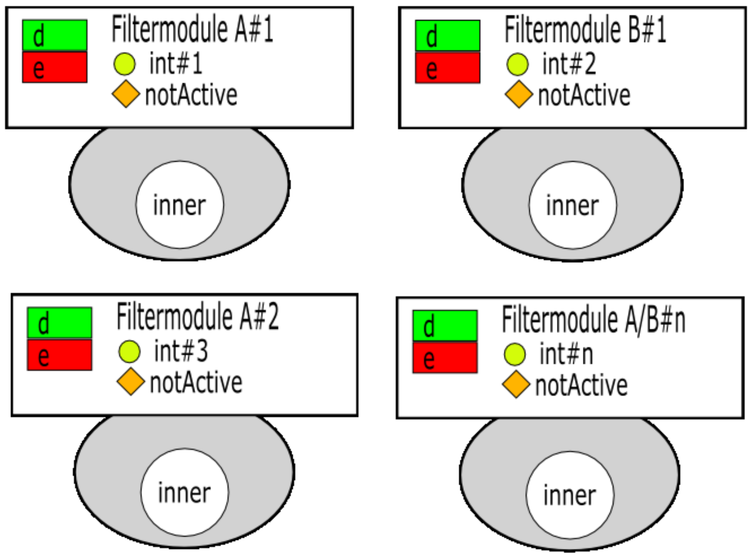

Figure 10.1: Instances of Listing 10.2

FAQ and Hints

FAQ

Q: *What is the use of declaring multiple instances of one type? Is it not possible to rewrite the class of the type, so that you only need to declare one internal?*
A: This depends on the situation, whereas for holding a state you can also write another class that takes care of the complete state, however for constructions where you delegate to a tree-like structure, like `left, right : Node;`, you do not want to combine the internals into one class.

Q: *Is it possible to access an internal of another filter module?*
A: It is not possible, on conceptual and implementation level, to access an internal with a filter module element reference.

Q: *I want to declare an internal with a type that is in a library, but the class does not have a default constructor. How can I solve this?*
A: The best way to solve this is by extending the class you want to use with a default constructor that sets the needed default values. If that is not possible because the class has been made final, then you can write a static method which calls a constructor with arguments and returns the result, this static method can be used in the declaration of an external[1].

Hints

The instantiation of an internal happens with a constructor with no parameters. In most object-oriented languages this is the default constructor, which is used when the user does not create a constructor. Sometimes people forget to create a constructor without parameters, when there

[1]N.B.: Only use this construction if you are stuck in such particular situation, see also the commentary.

is already a constructor with parameters, which results in an error.

Commentary

The use of Arguments in Constructors Calls
The internal initialization is done with a constructor without arguments; this is a result of the requirement to be language independent. The internal declaration `internal: Person = Person(‘‘Albert’’, 24, true);` is considered as language dependent and therefore only constructors without arguments are used. We will not re-discuss the point of allowing primitive values in the internal declaration, because is has been discussed in chapter 8. However, with the introduction of the filter module parameters it is possible to use objects, which are passed through the filter module parameters, in the declaration of the internals. This feature is not added to the syntax yet, because it relies on the full implementation of the filter module parameters. When this is realized then the syntax can change to the one given in Listing 10.4.

```
Internals ::= ‘internals’ (Identifier-LIST ‘:’ Type
              [‘=’ InitialisationExpression ‘(’ [ Argument-LIST ] ‘)’]‘;’)*
```
Listing 10.4: Proposed internal syntax

Because we already use default constructors, we already have the mapping from the language independent declaration to language specific constructor calls, thus we do not have to work them out for adding arguments in a constructor.

Visibility of Internals
As mentioned in the semantics, it is possible to extend the signature of the class on which is superimposed with the signature of the internal. What is not mentioned in the text, is whether the internal is public or private value, and even more important can we directly access the internal? To start with the public or private matter, addressing the filter module directly from outside the filter module is not correct programming for Compose*, because it breaks with the filter interface. So whether it is public or private, you should not access it anyway.

Internals as Externals
As mentioned earlier in chapter 8 it is good to keep a difference between internals and externals. It is however, possible to write all internals as externals, if you write a static method `static public getnewPerson(){ return new Person()}` and you use that in an external declaration, `externals person : Person = person.getNewPerson()`, then you get the same result as when you would create an internal with `internals person : Person`. The construction showed here is not what we want, because we break the decoupling of the main code and the concerns, the class Person, which is in the main code, does have a method which is solely written for a concern. The only correct use for this construction is when you want to use an internal of a class out of a library, and this class does not have a default constructor and the class is declared as final[1]. It is for the best to avoid this kind of constructions; however it is good to know that there are workarounds for this kind of situations.

[1]It is actually your only option when you are stuck with such a library.

Chapter 11

Externals

Externals are objects that are instantiated outside a filter module. They can be instances which are used in multiple places in the application, for instance as a Logger object, or they are shared between multiple filter module instances.

Syntax

As we can see in Listing 11.1, we can split up the external in three parts: the identifiers, type, and the initialization. The parts are separated with syntactic sugar, between the identifiers and type there is a colon (':') and between the type and initialization an equation mark ('='). The initialization expression is the method that returns the reference to the external object. It is possible to use arguments in the initialization expression.

Semantics

If we take Listing 11.2 and look how the instances are created, we get Figure 11.1. Every instance of the filter module `dynamicscoring` gets a reference to the common object Score. If we draw a schema of Listing 11.3 then we get a different picture, because every instance of delegatePlanning gets a pointer to one of the two shared externals.

Getting a shared object can be done on several ways, however you must write a construct that holds the objects so that you can select them, thus you should keep a collection of externals. If

```
FilterModule ::= ‘filtermodule’ FilterModuleName [FilterModuleParameters] ‘{’
                  [Internals] [Externals] [Conditions]
                  [InputFilters] [OutputFilters] ‘}’
Externals ::= ‘externals’ (Identifier ‘:’ Type
              ‘=’ InitialisationExpression ‘(’ [ ArgumentList ] ‘)’ ‘;’)*
InitialisationExpression ::= MethodReference
```

Listing 11.1: Externals syntax

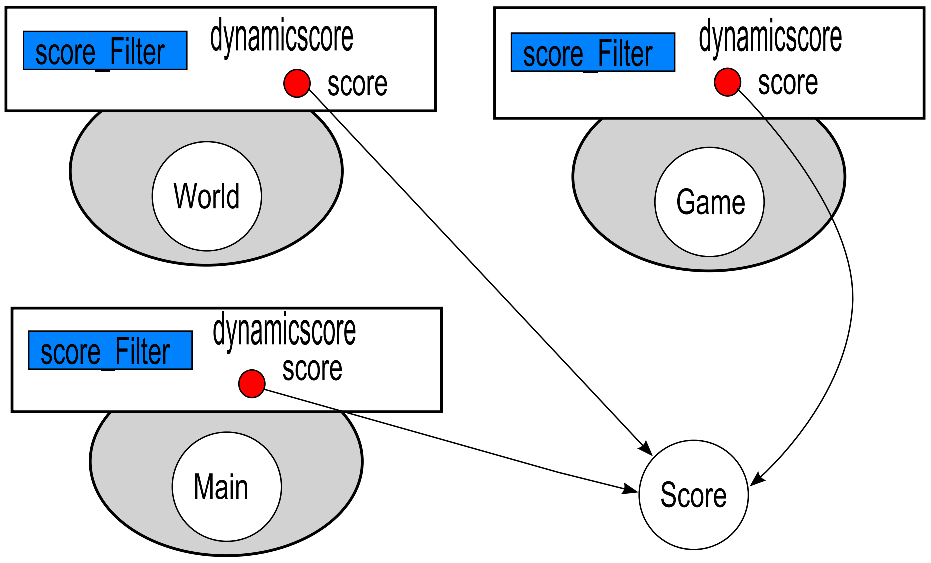

Figure 11.1: Schema of Listing 11.2

there is only one instance of an external, then the Singleton pattern is a good pattern to apply.

Examples

In the Pacman example, the filter module `dynamicscoring` uses an external to keep track of the score (Listing 11.2). For this concern it means that every instance of the classes `pacman.World`, `pacman.Game`, and `pacman.Main` gets its own instance of the filter module dynamicscoring and each of the instances of dynamicscoring has a reference to the same instance of `pacman.Score`. The construction with scoring in Pacman is known as the Singleton pattern [GHJV95]. It is also possible to get constructions without the Singleton pattern as demonstrated in Listing 11.3, where an object is used as parameter and is assigned through the filter module binding.

Legality rules

- The identifier of an external must be unique for the filter module where it is declared in;
- The type must be a fully qualified name of a class and this class must exist;
- The initialization expression must point to a valid method, this often means that you use a static method.

FAQ and Hints

FAQ

Q: *How do I create an external that is unique for a certain number of filter module instances?*
A: If you want to use an external that is not shared with every instance of the filter module,

```
filtermodule dynamicscoring{
  externals
    score : pacman.Score = pacman.Score.instance();
  inputfilters
    score_filter  : Meta = { [*.eatFood] score.eatFood, [*.eatGhost] score.eatGhost,
            [*.eatVitamin] score.eatVitamin, [*.gameInit] score.initScore,
            [*.setForeground] score.setupLabel }
}

superimposition{
  selectors
        scoring = { C | isClassWithNameInList
                    (C, ['pacman.World', 'pacman.Game', 'pacman.Main']) };
  filtermodules
    scoring <- dynamicscoring;
}
```

Listing 11.2: Dynamic scoring filter module from the Pacman example

```
filtermodule delegatePlanning(?external){
  externals
    e : Secretary = ?external;
  inputfilters
    d : Dispatch = {<inner.*> *.*, <e.*> e.*};
}

superimposition{
  ...
  filtermodules
    selA <- delegatePlanning(Secretary());
    selB <- delegatePlanning(Secretary());
}
```

Listing 11.3: An external without the singleton construction

you can use the filter module parameters to define the different instances for the external.

Commentary

The advantage of using the Singleton pattern becomes visible when we would use a parameter as initialization String. In the given example Listing 11.2, we see that we can address the same instance of Score from three filter modules and that there will only be one instance of Score in the application. We could do the same without a filter module, by just adding the Score object to every class. The advantage of the filter is that when we would write `score : pacman.Score = ?instance;`, then we only need to change the initialization String in one spot instead of all the classes where we want to access Score.

The Alternative to Singletons
With the addition of filter module parameters it is also possible to use a given object from the parameters as an external. This fixes the old limitation that you had to choose between a singleton construction, which results in that every filter module points the same object, or to use an internal, which results in that very instance has it own instance of the internal. In Listing 11.3 we have an example were two selections, selA and selB, get a different instance of the class secretary. Using an object as a filter module parameter makes it possible to use an instance for a certain selection of filter module instances instead of all filter module instances.

Arguments in the Method Calls
Currently it is only possible to use the Singleton design pattern for an external declaration. This construct is mentioned earlier in section 6.4 for arguments in conditions calls.

Introducing Static Methods
In the old syntax it was possible to declare an external without an initialization expression. So the syntax was:

```
Externals ::= ‘externals’ (Identifier ‘:’ Type
              [‘=’ InitialisationExpression ‘(’ [ ArgumentList ] ‘)’] ‘;’)*
```

The purpose of such construct is that with an external declaration without an initialization expression it is possible to introduce static methods. A static method from a class can be used without having instantiated an object from this class. Because the target needs to be an internal, external, or the keyword “inner”, the only way you are able to substitute the target, so that it is send to a static method, is by declaring the class as an external. However, this construct has been removed because the construct probably was not placed in the syntax to support the usage of static methods and that it is just an error in the grammar. The theory mentioned above is a possible explanation what the old syntax could mean.

Chapter 12

Conditions

The conditions are used to introduce (boolean) methods in a filter module so that these methods can be used in a filter specification to influence the behavior of a filter at runtime. To keep the filter specification simple and to reuse declarations, all the conditions of a filter module are declared in one place: the conditions declaration block. The methods for implementing the conditions can be defined in the inner object (the class where the filter module is superimposed to), internals, externals, or are static methods.

Syntax

From the syntax, Listing 12.1, we see that the condition name and declaration are a one-to-one relation. Each condition name must be unique for each filter module and the name may also not be used for an internal or external. There are two forms of declaration: a condition can be declared from an internal, external, or the inner object and the other option is to use a fully qualified name of a static method.

Semantics

A condition declaration labels a method reference with a identifier so that you can use this identifier in the filter specification, instead of a method reference. The method reference of the condition declaration needs to point to a method that returns a boolean.

```
FilterModule ::= ‘filtermodule’ FilterModuleName [FilterModuleParameters] ‘{’
                 [Internals] [Externals] [Conditions]
                 [InputFilters] [OutputFilters] ‘}’
Conditions ::= ‘conditions’ (Identifier ‘:’ MethodReference ‘;’)*
MethodReference ::= (FilterModuleElement‘‘.’’MethodName | FullyQualifiedName)
                   ‘(’ ‘)’
```

Listing 12.1: Conditions syntax

```
filtermodule someConditions{
  internals
    a : VenusFlyTrapExample.Animal;
  externals
    jbframe : JukeboxFrame.JBFrame = JukeboxFrame.JBFrame.instance();
  conditions
    isfly : a.hasPrey(); // Venusfly trap
    isStateChanged : jbframe.isStateChanged(); //Jukebox
    pacmanIsEvil : pacman.Pacman.isEvil(); //Pacman
    isFrame : Composestar.Patterns.ChainOfResponsibility.Click.isFrame();
    // Command of responsibility pattern
    isBlue : inner.isBlue();
```

Listing 12.2: Some condition declarations combined

Examples

In Listing 12.2 five examples of condition declarations are given. Four of these declarations come from the Compose* examples [Com06], for each of them is noted in the listing from which example they come. The first one shows the usage of an internal, the second one shows the usage of an external. The third and fourth examples show the use of static methods. The last example does not come from the example directory, there is a design technical issue why this option is not much used. There is no example that uses all the different condition declarations in one condition declaration block.

Legality rules

- The identifier of a condition must be unique for the filter module it is declared in;
- The condition implementation method that you reference to must exist and it must return a boolean value;
- A condition implementation method must not have any side effects in the application.

Commentary

Parenthesis and Arguments

The current implementation does not allow the usage of arguments in the condition declaration. As mentioned in section 6.4 it is a possible addition to the language to add arguments in the condition declaration, because of the introduction of the filter module parameters.

The Use of Conditions from the Inner Object

One of the possibilities to declare a condition is to use a boolean method of the inner object. As mentioned earlier, this option is not used in the current set of examples. We can say that it is not widely used due to two limitations: the methods must exist and you should know this when you write the selector. Filter modules are imposed on multiple classes and when a method of the inner class is used, then all the multiple classes must contain this method.

There are three ways to use a condition from an inner object without getting any problems with non-existing methods. The first is to write filter modules for just one class or its child classes. If a filter module is written solely for one class then it might not be a crosscutting concern and the behavior of the filter module can be included in the original class. The second use for conditions from the inner object is if you select all the classes which implement a certain interface. In that scenario you know whether the boolean method is available if it is in the signature of the interface. The third one is code conventions, if you use a code convention that classes with a certain annotation all have a method called foo, then it is also possible to use the conditions from the inner object safely.

Removing the Direct Use of Static Methods in the Filter Definition
In the old grammar it was possible to directly call static methods in the filter specification. This is sometimes not practical because with the direct calling you might end up using the same method twice and in that case it would be better to use an identifier, in order to have a readable filter specification. Another argument for not allowing them in the filter specification, is that they can become quite long as seen in Listing 12.2 line 10.

Language Independent Conditions
The selector in the superimposition uses predicate queries to select certain attributes of an application. It uses a language independent model of language constructs and each language gets translated to this independent model [Hav05]. For instance it uses the term namespace, which gets translated to package for the Java language. In theory it is possible to write such language independent model on objects and its attributes, so that we can write language independent conditions with predicate queries. This would make it possible to write conditions that are reusable even for another platform.

Chapter 13

Filters

Filters are the main part of a filter module, a filter module can have input filters and output filters. Both the input filters and output filters have the same syntax and therefore we handle them both in this chapter.

A filter has five parts: the filter identifier, the filter type, the condition part, the matching part, and the substitution part. These parts are shown below:

$$\overbrace{stalker\_filter}^{identifier} : \overbrace{Dispatch}^{filter\ type} \ = \{\overbrace{!pacmanIsEvil}^{condition\ part} =>$$
$$\overbrace{[*.getNextMove]}^{matching\ part} \ \overbrace{stalk\_strategy.getNextMove}^{substitution\ part} \ \}$$

Except for the filter identifier they all are separately described in the reference manual. In this chapter we look to how these parts work together and how sequential filters behave.

Syntax

The syntax in Listing 13.1 covers the full syntax of the filter, so it also covers the condition, matching, and substitution part. The filter type contains the type of the filter, this can be a predefined one (currently Dispatch, Meta, Send, or Error) or a custom type. A filter element contains an optional condition part and one of more message patterns. The conditional part contains a logical statement, possibly containing logical operators in combination with conditions and boolean values. The message pattern contains a name or signature matching and an optional substitution part. The filters are separated with a filter operator. Currently the only operator is the “;”.

```
FilterModule ::= ‘filtermodule’ FilterModuleName [FilterModuleParameters] ‘{’
                  [Internals] [Externals] [Conditions]
                  [InputFilters] [OutputFilters] ‘}’
Inputfilters ::= ‘inputfilters’ FilterSet
Outputfilters ::= ‘outputfilters’ FilterSet

FilterSet ::= Filter (FilterOperator Filter)*
FilterOperator ::= ‘;’
Filter ::= FilterName ‘:’ FilterType [‘(’ ArgumentList ’)’] ‘=’
           ‘{’ FilterElements ‘}’
FilterElements ::= FilterElement (ElementCompositionOperator FilterElement)*
ElementCompositionOperator ::= ’,’

FilterElement ::= [ORExpression ConditionOperator] MessagePattern
ORExpression ::= ANDExpression [‘|’ ANDExpression]
ANDExpression ::= NOTExpression [‘&’ NOTExpression]
NOTExpression ::= [!] (ConditionLiteral | ‘(’ORExpression‘)’)
ConditionLiteral ::= ConditionName | ‘True’| ‘False’
ConditionOperator ::= ‘=>’| ‘~>’
MessagePattern ::= Matching [SubstitutionPart]
                   | MatchPattern
                   | ‘{’ Matching (‘,’ Matching)* ‘}’ [SubstitutionPart]
Matching ::= SignatureMatching | NameMatching
SignatureMatching ::= ‘<’ MatchPattern ‘>’
NameMatching ::= ‘[’ MatchPattern ‘]’ | Quote MatchPattern Quote
SubstitutionPart ::= [Target ‘.’] Selector
MatchPattern ::= [Target ‘.’] Selector
```

Listing 13.1: Filter syntax

```
filtermodule dynamicstrategy{
  internals
    stalk_strategy : pacman.Strategies.StalkerStrategy;
    flee_strategy : pacman.Strategies.FleeStrategy;
  conditions
    pacmanIsEvil : pacman.Pacman.isEvil();
  inputfilters
    stalker_filter : Dispatch = {!pacmanIsEvil =>
      [*.getNextMove] stalk_strategy.getNextMove };
    flee_filter : Dispatch = {[*.getNextMove]flee_strategy.getNextMove }
}
```

Listing 13.2: Dynamic strategy filter module in Pacman

Semantics

As mentioned earlier, a filter can consist of five parts. The filter identifier can be used to get the corresponding filter, like other filter module elements. The filter element is the combination of condition, matching, and substitution part. How the filter element behaves, depends on the filter type. If the condition and matching part matches then the substitution part gets executed.

Filters are separated by a “;”, this operator means that if the filter does not match, the next filter in the filter module will be evaluated. After the last superimposed filter module a *default dispatch filter* to the *inner object* is created. This means that the last filter that is declared in a filter module is always followed by another filter, be it from the next filter module or it is the default dispatch filter. To keep the user from placing another filter operator between these couplings, it is not possible to use the “;” after the last filter in a filter module.

Input filters reason about messages going to the object and output filters on the messages that are sent from an object. The return call of a method is not considered a message.

Examples

When we look at the dynamic strategy concern of Pacman, Listing 13.2, we have two filters: stalker filter and flee filter. Both filters are of the filter type dispatch, this means that if the condition part and matching part matches, the target and selector gets substituted with the values in the substitution part and the message is then dispatched to the target of the changed message.

The combination of the two filters means that the stalker filter is evaluated first and if it does not match then the next filter is being evaluated, this is the flee filter in this situation. The matching is based on the condition part, in the example this is the condition whether Pacman is evil, and the matching part, which filters on messages with “getNextMove” as selector value. When a filter matches, the target and selector of the message gets changed with the values of the substitution part. What happens with the message then depends on the filter type. To specify how sequential filters behave there is an filter operator placed between the filters. Currently only the “;” is used and it means “if not then”. It is not possible to place an operator after the last filter in a filter set.

```
filtermodule logger(??inputMethods, ??outputMethods){
  externals
    logger : Logger = Logger.instance();
  conditions
    isLoggingOn : logger.isOn();
  inputfilters
    inlog : Meta = {isLoggingOn => [*.??inputMethods] logger.log}
  outputfilters
   outlog : Meta = {isLoggingOn => [*.??outputMethods] logger.log}
}
```

Listing 13.3: Custom logging filter module

An example on how you can place input and output filters in one filter module is demonstrated in Listing 13.3. In the example the working of both filters is the same, if the condition matches and the selector is in the set of the given methods, then the message getss wrapped up and is send to the external object and to be more precise to the method `log`.

Legality Rules

- The identifier of a filter must be unique for the filter module where it is declared in, so it is not possible to have the same name for an input and output filter;
- Filters are separated by a “;”, which is called the composition operator;
- The last filter is not followed by a “;”.

FAQ and Hints

FAQ

Q: *Why is it not possible to place a “;” after the last filter?*
A: The “;” between filters is often mistaken for a terminator symbol, but it actually says something about the sequential ordering of filters. The last filter in a filter module is always followed by the filter of the next filter module or the default dispatch filter. It would be incorrect to let the user have a say about that coupling and therefore it is not possible to place a “;” after the last filter.

Hints

To keep a message from being evaluated by the default dispatch filter it is possible to place a last filter that catches all messages. This could be an error or a dispatch filter.

Commentary

Filter Identifier

The filter has a unique identifier which gives the opportunity to get a filter from its filter module by using a filter module element reference. Currently this construct is not used in Compose* and it comes from the original idea to reuse filter specifications. So at the moment the identifier

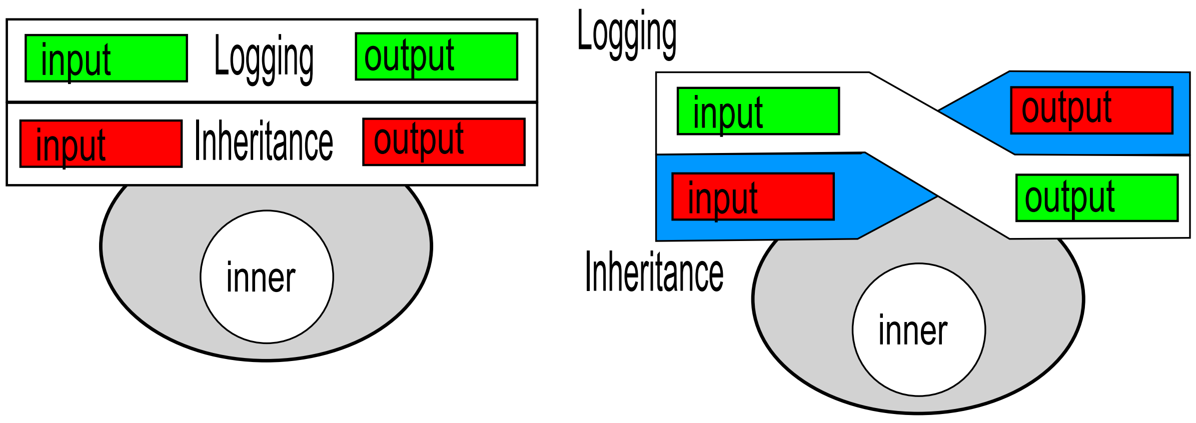

Figure 13.1: Two filter modules with input and output filters

is redundant, but we keep it in the syntax for usability and user-friendliness. Because talking about the fleeStrategy filter is easier than talking about the first filter of filter module dynamic strategy.

Filter Modules with both Input and Output Filters
As mentioned earlier it is possible to have input and output filters in one filter module. However, the option of using both input and output filters in a filter module is barely used. One of the reasons for this is that for most usages of the filter module you only need one of the two filter sets. Some of the used constructs for filters only need an input filter, like for instance the dynamic strategy filter module in Pacman (Listing 13.2) or a generic inheritance filter module (Listing 9.3).

A limitation for having both input and output filters in one filter module is how the constraints are defined. For instance, if we take two filter modules logging and inheritance and they both have input and output filters and we want to log before the input filters of inheritance and we want to log after the output filters of inheritance, this is demonstrated on the left side of Figure 13.1. However in reality you end up with the situation on the right, and the only way to get it like the left image is when you break up one of the two filter modules. To avoid situations like this, it is advisable to only combine input and output filters if it is really necessary, for instance if they share an internal.

Filters as State Machines
An idea for the usage of the filter operators is to use them for building a state machine. An example is given in Listing 13.4, in this example we want to let the first filter evaluate and when it matches it moves to the second one until that one matches. Whereas this looks like a nice idea, but there are drawbacks. First you get problems with the syntax and semantics, because you must be able to scope your REPEAT statement and you also need to define what the commando's mean. Second, designing such filter sets is hard and it is much better to write

```
inputfilters
  first : Dispatch = {[*.log]} REPEAT
  second : Dispatch = {[*.print]}
}
```

Listing 13.4: Idea for a filter state machine

```
internals
  state = State;
conditions
  ifRaisedState : state.ifRaisedState();
inputfilters
  first : Before = {[*.log] *.raiseSate};
  second : Dispatch = {ifRaisedState => [*.print]}
}
```

Listing 13.5: State machine with filter operators

such a machine with an internal that keeps the state in combination with conditions that handle the state, like Listing 13.5. The filter of the message for the state change can be done with a meta filter or even the prepend and append filter [Min06]. Writing such construction is easier then using state machine operators.

Filter Operators and Different Filter Combinations

The filter operator is placed between filters. At the moment only the "if not then" operator is used which is in the ";" token in the syntax. There are two points to comment on the filter operator. The first question is why the usage of ";"? And the second question is can the filter operator mean anything at all?

To start with the second question, as mentioned above with designing the state machine it hard to define new filter operators because you probably end up adding parenthesis and other syntax constructs. Therefore there is no final answer on this question yet, there are some ideas on more filter operators around. But these ideas have problems on the what the semantics are of these operators. And to answer the first question, it is indeed a bit strange to use the semicolon as an operator. In most programming languages it marks the end of a statement. We do not allow a semicolon after the last filter declaration and this often regarded as annoying by the users. We could add the "+" sign as a stand in for ";" to make things less confusing. We have not done it yet, due to the fact that we cannot answer the second question yet and it would be strange to start adding operators while we have not answered some of the problems with combining filters.

Chapter 14

Filter Type

The type of a filter specifies how a filter behaves for an accept and a reject of the condition and the matching part. This includes the specification of how to execute the substitution part.There are currently four predefined filter types and it is possible to create custom filter types. It is possible to add arguments in a filter specification, it depends on the filter type what the semantics are of the arguments.

Syntax

As demonstrated in Listing 14.1, a filter type it must be a predefined filter type or custom defined one. It is not possible to overwrite the predefined filters, so a dispatch filter always behaves the same.

Semantics

The four filter types that are currently in use are described shortly in section 2.2. The description below is more technical then the earlier description.

Dispatch
If the message is accepted, then the target and the selector of the message get substituted with the values of the target and selector of the substitution part. If the value is a wildcard ("*"), then the original value remains unchanged. After that, the message is dispatched to the target of the message; this can be either the old target or a new one. In the situation that the target is the old value, the message starts again at the begin of the input filter

```
Filter ::= FilterName ‘:’ FilterType [‘(’ ArgumentList ’)’] ‘=’ ’{’ FilterElements ’}’
FilterType ::= Identifier
```

Listing 14.1: Filter type syntax

```
...
inputfilters
  d : Dispatch = {<inner.*> inner.*};
  rotate : Rot13 = { [*.ReadLine] };
  l : Log (?argument) = {[*.log]};
  e : Error = {[*.*]}
}
```

Listing 14.2: Some different filter types

set. A dispatch to inner goes directly to the inner object. Dispatch can only be used for input filters;

Send
: The send filter is the dispatch filter for the output filters. When the filter matches the target and the selector gets substituted with the values that are in the substitution part;

Error
: An error filter raises an exception when it rejects. It ignores the substitution part;

Meta
: When a meta filters matches it reifies the current message and add it as an argument of a new message. This new message is sent to the object and method declared in the substitution part. This method must have a single parameter of the type *ReifiedMessage*. In this method it is possible to alter the message and to let it resume or to send it back.

A custom filter gets extended from a meta filter and it is possible to define how it handles the different parts and values.

Examples

In Listing 14.2 four different filters are used, the first and last filter declaration use predefined filter types, the other two use custom filter types. An example of how to write a custom filter type is demonstrated in the Rot13 example [Com06]. It is possible to initialize a custom filter with the filter parameters.

Legality Rules

- A custom filter type cannot have the same name as a predefined one;
- A custom filter type must a have an implementation in the application.

Commentary

It is not possible to redefine predefined filter types, this is done to keep the language understandable for a user because in this way a dispatch filter will always behave like a dispatch filter.

The advantage of predefined filters is that we know what they do and therefore we can do a static analysis on the flow of a filter set. With a meta filter we do no longer know what happens, because the resume and return commandos of a message are called in the implementation. On the other hand, predefined filter types are only a selection of possibilities and the meta filter is a necessary type for creating other non-predefined behavior. So it is a constant trade-off between being all flexible by only using custom filters or just trying to add predefined or custom defined filters for every situation, just to have filter sets where you can reason about. Therefore we keep the current set of predefined filter types, however there is room for some more predefined filter types.

Chapter 15

Condition Part

The condition part of a filter contains the conditions and operators which say something of the matching part, to reason whether a filter is accepted or not. The used methods must be first declared in the conditions block of a filter module. The logical operators are the basic set of *and*, *or*, and *not* added with two operators to denote “if true” (=>) and “if false” (~>)., which are placed in front of the matching part.

Syntax

The syntax of the condition part, as demonstrated in Listing 15.1, follows the standard way to program a set of AND (&), OR (|), and NOT(!) operators. The precedence is that the NOT is executed first, then the AND and the OR. This means that *A AND B OR C* is actually *(A AND B) OR C*. The chosen precedence is the commonly used one.

The *ConditionLiteral* can be either a condition name, which must be declared in the condition block of a filter module, or the predefined conditions “True” or “False”. It does not matter whether the keywords are written like this or without capitals. If there is an expression then there also must be a *ConditionOperator*, there are two options for this operator: => and ~>.

```
FilterElement = [ORExpression ConditionOperator] MessagePattern
ORExpression = ANDExpression ['|' ANDExpression]
ANDExpression = NOTExpression ['&' NOTExpression]
NOTExpression = [!] (ConditionLiteral | '('ORExpression')')
ConditionLiteral = ConditionName | 'True'| 'False'
ConditionOperator = '=>'| '~>'
```

Listing 15.1: Filter condition part syntax

```
conditions
  pacmanIsEvil : pacman.Pacman.isEvil();
inputfilters
  stalker_filter : Dispatch = {!pacmanIsEvil =>
                              [*.getNextMove] stalk_strategy.getNextMove};
  flee_filter : Dispatch = {[*.getNextMove] flee_strategy.getNextMove}
```

Listing 15.2: The dymanic strategy filtermodule

Semantics

In the condition part it is only possible to use the boolean values *true* and *false* or conditions that are declared in the conditions block of a filter module. This means that the number of possible syntax constructs is small. The meaning of the logical operators is the same as the usual meaning in most programming languages. The condition operator does say something on the message part and not as it might suggest on the condition part. The `=>` operator provides a true if the message part matches. The other operator, `~>`, gives a true when the message part does not match.

The condition part is optional; however when it is left out then the syntactic sugar places a `True =>` in the filter.

Examples

In the Pacman example of section 2.3 we have already seen the filter demonstrated in Listing 15.2. In that example the method *pacmanIsEvil* is checked in order to determine how the ghosts must behave. So when Pacman is not evil it only depends on the matching part whether the filter matches.

Some examples of condition parts are given in Listing 15.3. We see that d1 (on line five) depends on the condition isBlue whether the message will pass. The ~> means exclusion so it will match if the message does not match with the matching part of the filter. Thus filter d2 (on line six) will never match, because the matching part will always match. Filter d3 (on line seven) shows a combination of operators and values.

```
1 conditions
2   isBlue : inner,isBlue();
3   isMorning : inner.isMorning();
4 inputfilters
5   d1 : Dispatch = {!isBlue => *.*};
6   d2 : Dispatch = {isBlue ~> *.*};
7   d3 : Dispatch = {(isBlue | isMorning) & True => *.*}
```

Listing 15.3: Possible condition parts

Legality rules

- A condition name must be declared in the condition block of the same filter module;
- The keywords true and false are written as “True”, “true”, “False”, and “false”, anything else, also with different capital letters, is considered a condition name.

FAQ and Hints

Hints
Whereas the condition part is optional, the matching part is not. It is however easy to let only a condition part determine the outcome of a filter, with the matching `[*.*]` every message matches and it only depends on the conditions whether a message is accepted.

Commentary

Basic Set of Logical Operators
The supported set operators are the *and*, *or*, and *not*. These three are enough to create any other possible logic operator because they form a *functional complete* set [BKLM91]. Because we want to reason about the filters, we keep the syntax as simple as possible on this point. It is easier to write an algorithm that only has to take care of three operators instead of a whole set.

We could go further by removing all the operators and just write one method that calls all the separate conditions, however this would also mean that this method should also know the references to the internals, externals, and inner object, and even more important this method would be quite complex and not be reusable at all. Therefore we adopt a limited set of logical operators.

Chapter 16

Filter Matching Part

The matching part of a filter is where the message is being matched. There are two different types of matching: name and signature matching.

Syntax

The syntax of the matching part is shown in Listing 16.1. The basic form is `target.selector` surrounded by the tokens that mark the type of matching. It is possible to leave out these tokens when only a single MatchPattern is used, in that situation signature matching is applied by default. The target needs to be an internal, external, or the keyword *inner*. The selector must be a method name. To indicate whether you want to use name matching or signature matching you can use square brackets (`[]`) or double quotes (`‘‘ ’’`) for name matching and angle brackets (`< >`) for signature matching.

Semantics

The name matching checks the target part of the match and the selector part. The use of the keyword “inner” is, as you might expect, quite trivial for a name matching in an input filter.

Signature matching checks whether the selector of the message is in the signature of the given

```
FilterElement ::= [ConditionExpression ConditionOperator] MessagePattern
MessagePattern ::= Matching [SubstitutionPart]
                 | MatchPattern
                 | ‘{’ Matching (‘,’ Matching)* ‘}’ [SubstitutionPart]
Matching ::= SignatureMatching | NameMatching
SignatureMatching ::= ‘<’ MatchPattern ‘>’
NameMatching ::= ‘[’ MatchPattern ‘]’ | Quote MatchPattern Quote
MatchPattern ::= [Target ‘.’] Selector
```

Listing 16.1: Filter matching part syntax

```
1 filtermodule example{
2   inputfilters
3     d : Dispatch = {[tar.sel] *.*, [*.sel] *.*, <tar.*> *.*};
4     d2 : Dispatch = {True ~> {[*.foo] , [*.bar]} inner.*}
```

Listing 16.2: Some possible matching parts

target. This means that it is only useful to fill in the target when a signature matching is used. Filling in the selector in the signature matching has no use, because `<foo.bar>` will always hold or always fail.

Both the target and the selector can be a parameter, for the selector it is also possible to use a list of parameters. With a list of parameters it means that one entry in the list must match.

When you have a dispatch filter ands the selector is not in the signature of the object on which is superimposed, then the selector is added to the signature of this object. This mechanism is not used when a wild card is used on the selector spot, because that would imply that the signature becomes every possible method. The mechanism also works on singnature matching, which makes it possible to add the signature of the internal or external to the object on which is superimposed.

Examples

Three matching parts are demonstrated in Listing 16.2 line 3. The first two are of the *name matching* type and the third is an example of *signature matching*. The matching `[tar.sel]` first whether the target points to the same object as where "tar" refers to and it will try to match whether the selector is "sel". The second one, `[*.sel]` ignores the target, because a wild card ("*") is used and the selector is matched with "sel". The last one, `<tar.*>` checks whether the selector is in the signature of "tar".

The second filter d2 on line 4 demonstrates how you can do a name matching where the selector must match when the selector is not `foo` and not `bar`. The matching works as follows, when a selector is one of the values of the list you get a match. Because of the ~> this match becomes a reject, thus making this filter to reject when the selector is `foo` or `bar`. Especially for this type on of constructions a matching list is introduced.

It is also possible to use parameters as the target and the selector, for the target only a single parameter can be used, while for the selector both a single parameter and a list of parameters can be used.

Legality rules

- The used target must be a declared internal, external, the keyword "inner", or a parameter;
- A selector can be a method, a single parameter, or a list of parameters.

FAQ and Hints

FAQ
Q: *Why is it not possible to use class names as targets?*
A: The messages are sent from object to object, therefore the choice is made to use objects as targets instead of class names. The used identifier in the filters refers to internals, externals, or is the inner object (the object where is superimposed on). It is however possible to achieve filtering on class names with a meta filter.

Hints
Keep an eye on the `[*.*]` because any declaration afterwards does not get reached at all. Another important thing is that in order to make exceptions for a `[*.*]` matching, you can place a filter before that matching.

Commentary

Name matching had two syntax variants, `[foo.bar]` is the same as `"foo.bar"`. To keep the language concise the quotes have been removed.

With the new proposed filter layout (chapter 5) it is possible to filter on more then just the target and the selector. It is known that the current system is too limited to the users.

Wildcards
In AspectJ it is possible to define pointcuts with wildcards in the method name, like `set*`. We are aware that Compose* is indeed not AspectJ, but the idea of using wildcards is tempting, because it is a quick language construct. The disadvantage is that `set` is open-ended and that a method like `setup` is taken into the selection as well. A better solution is to use design documentation, like annotations, instead of a wildcard. In [Nag06] a proposal is made to use annotations in the matching part, it is has not been implemented yet and therefore the language changes are not mentioned in this chapter. Filtering on annotations is covered by the new filter element syntax, however the syntax is different from the original proposal.

Signatures
Despite of its name, signature matching only matches on the names of the methods in a class, instead of the whole signature. To check on the whole signature you need a meta filter and you have to write the check function yourself. Also filling in the selector in the signature matching has no use, because `<foo.bar>` will always hold or always fail.

Inner, Self, Server, and Sender
In [Ber94] there are more predefined keywords that say something about the object, on which is superimposed, then just the keyword *inner*. In that plan the keywords *self*, *server*, and *sender* can be used in the matching and substitution part. Self is the object on which is superimposed plus the extended signatures, thus the original object plus the filter modules.

The server keyword points to the original target and selector of a message, thus the original message before it goes into the set of filter modules. Sender is the object that sends the message. The server and sender can be retrieved in a meta filter.

With self we need to determine whether self is the extended object after all filter module bindings or just after some of the filter module bindings. With this definition you are never sure how the self object looks like, because the self object is flexible due to the possibility that more filter modules are added to the object. So it is better to leave self out of the language.

Chapter 17

Filter Substitution Part

The substitution part of a filter is where you can specify the changes in a message. Just like the matching part it consists of a target and a selector.

Syntax

The syntax is almost similar to the matching part and is shown is Listing 17.1. The difference is that the substitution part does not have any brackets, creating a visible difference between matching and substitution part. Another difference is that it is not possible to have a list of substitution parts.

Semantics

A substitution part gets executed when a filter matches, how it gets executed depends on the filter type. The meaning of the substitution part is simple, if there is a wild card on the target or selector spot the attribute of the message stays the same. Otherwise the attribute gets substituted with the given value.

```
FilterElement = [ConditionExpression ConditionOperator] MessagePattern
MessagePattern ::= Matching [SubstitutionPart]
                   | MatchPattern
                   | ‘{’ Matching (‘,’ Matching)* ‘}’ [SubstitutionPart]
Matching ::= SignatureMatching | NameMatchingSubstitutionPart = [Target ‘.’] Selector
```

Listing 17.1: Filter substitution part syntax

```
filtermodule example{
  inputfilters
    d : Dispatch = {[tar.sel] tar1.*, [*.sel] tar2.sel2, <tar.*> *.*}
```

Listing 17.2: Some possible substitution parts

Examples

In Listing 17.2 three out of the four possible substitution part combinations are demonstrated. The matching parts are different as well to get a filter specification that means something. The fourth is that the target stays the same and that the selector changes.

Legality rules

- The used target must be a declared internal, external, the keyword "inner", or a single parameter;
- A selector can be a method or a single parameter;
- A selector cannot be a list of parameters.

FAQ and Hints

FAQ

Q: *Why is it not possible to use a parameter list in the selector of a substitution part, like in the matching part?*
A: A message can only be send to one method, so a list of methods would be meaningless.

Hints

Keep an eye on where you send your messages to. If the combination of target and selector does not exist you get an error, but it is easier to find the error before it occurs.

Commentary

Just as with the matching part we can only work with the target and the selector. With the proposed new filter layout (chapter 5) we can alter more attributes of a message.

Chapter 18

Superimposition

The superimposition part of a concern is where you can make a selection of program elements, such as classes, and then superimpose certain entities, such as filter modules and annotations, on this selection.

The superimpositions parts are executed during the initialization of the application, this is necessary because the bindings need to be known before the execution starts.

With the changes in the filter module syntax, the superimposition syntax needs some minor adjustments as well. The superimposition and its fields are all quite young, and moreover it is possible for all fields to point to the original proposal, were the field is proposed.

The superimposition part with the original selector and filter module binding are introduced by [Sal01]. The new improved selector and annotation binding are introduced by [Hav05].

Syntax

The syntax, as demonstrated in Listing 18.1, shows the syntax of the superimposition part. The result is the skeleton as demonstrated in Listing 18.2.

Semantics

The superimposition itself is a holder of blocks for selecting elements and defining bindings. It is executed during initialization of an application.

```
Superimposition ::= ‘superimposition’ ‘{’ [Selectors]
                    [FilterModuleBindings] [AnnotationBindings]‘}’
```

Listing 18.1: Superimposition syntax

```
superimposition{
  selectors
    ...
  filtermodules
    ...
  annotations
    ...
}
```

Listing 18.2: The superimposition skeleton

Examples

A superimposition can have three blocks and the ordering of these blocks are fixed, the skeleton is demonstrated in Listing 18.2. There is also only zero or one instance of a block. The selectors makes a selection of language elements, which can be used in the filter module and annotation binding.

Legality Rules

- The ordering of the sub blocks is fixed and have the following ordering: selector, filter modules, and annotations;
- Each sub block can only have one instance.

Commentary

The Ordering of the Selector Parts
The ordering and number of occurrences of the sub blocks is fixed, this is characteristic for the whole Compose* syntax. The main reason to do so, is that this ensures that all the declarations of a certain type are all together placed on the same spot. Thus you can find all the selectors within one concern on the selectors block. It is possible to allow random and multiple placement of the superimposition and its sub blocks. However, the readability would drop because users do not longer know where to expect certain blocks. To keep everything organized is also the reason why there is only one superimposition block per concern.

Removal of Condition and Method Binding
In the original design of the superimposition [Sal01], there were two blocks called condition and method binding. As described in section 6.3 they are redundant for the current implementation and therefore removed.

Selection of Program Elements
Currently the selector is used to select only classes. However, with the adding the filter module parameters and some other upcoming research it will be possible to select much more then classes and probably you can also superimpose items to the extended selections. Therefore the term *program elements* is used instead of classes.

Constraints
When multiple filter modules are superimposed on one object, it can be necessary to define

an ordering for the filter modules. In [Nag06] a model is defined to specify the ordering. The module for ordering the filter modules, called SECRET, is in place and be accessed with a XML file. More on this module can be found in [Dür04].

Chapter 19

Selectors

The selector field is where you can make a selection of certain program elements. The selections can be used to bind filter modules or annotations, and it is possible to use the selection as an argument for a filter module parameter.

Syntax

The identifier of a selector must be unique for the superimposition where you declare it in. The list of possible predicate expressions can be found in [Hav05].

Semantics

With the declaration of a selector you make a selection of program elements and you label these with an identifier. This identifier can be used in the different bindings and as filter module argument. There is a default selector “self” which selects a class with the same name as the concern.

Examples

In Listing 19.2 there are three example selections demonstrated. The selectors are written with queries, that are modeled to a common language model.

```
Selectors ::= ‘selectors’ (SingleSelector)*
SingleSelector ::= Identifier ‘=’ ‘{’ PredicateExpression ‘}’ ‘;’
PredicateExpression ::= Identifier PROLOG_EXPRESSION
```

Listing 19.1: Selector syntax

```
selectors
  strategy =
    { Random | isClassWithName(Random, 'pacman.Strategies.RandomStrategy') };
  levels =
    { C | isClassWithNameInList (C, ['pacman.World', 'pacman.Pacman']) };
  player =
    { C | methodHasAnnotationWithName (M, 'Jukebox.StateChange'),
        classHasMethod(C, M)};
```

Listing 19.2: Some selector examples

Legality rules

- A selector identifier must be unique for the superimposition block it is declared in;
- A selector identifier may not be named self.

Commentary

The design issues for the new selectors are described in [Hav05], however the usage changes with the addition of new features and some things can be added to the selector language.

Extended Selector Syntax
With the introduction of filter module parameters, we also want to select program elements to use as arguments in the filter module binding. And it would be even better if we can get multiple selectors from one query. So the new proposed layout is like Listing 19.3, were it is possible to use the variables of the query further on in the superimposition. Another proposal is to reuse queries in other queries, which offer good reusable code.

```
selectors
  selection(C, M, A) = {MethodHasAnnotation(M, A),
           classHasMethods(C, M), classWithName(C, "certainName")};
  selectionB(C) = selection(C, M, A);
filtermodules
  selection.C <- fm(selection.A);
```

Listing 19.3: A selector with extended syntax

Chapter 20

Filter Module and Annotation Binding

The binding parts are the places to bind certain properties to elements of a selector. Currently we can bind filter modules and annotations to classes.

Syntax

The syntax in Listing 20.1 shows the common layout for bindings. The weave operator is legacy, from the times when unweaving was still considered. The left hand side identifier must be a selector reference and the right hand side depends on what kind of binding you are doing.

Semantics

The meanings of the bindings is that you superimpose whatever is on the right hand side on the left hand side. Currently Compose* only binds filter modules and annotations. The bindings gets executed during the initialization of the application.

```
FilterModuleBindings ::= ‘filtermodules’ (FilterModuleBinding ‘;’)*
FilterModuleBinding ::= Identifier WeaveOperator
                        (FilterModuleReference-LIST
                        | ‘{’ FilterModuleReference-LIST ‘}’)
FilterModuleReference ::= Identifier [‘(’ Argument-LIST ’)’]
AnnotationBindings ::= ‘annotations’ (AnnotationBinding ‘;’)*
AnnotationBinding ::= Identifier WeaveOperator
                        (Identifier-LIST | ‘{’ Identifier-LIST ‘}’)
WeaveOperator ::= ‘<-’
```

Listing 20.1: Bindings syntax

```
filtermodules
  selA <- filtermoduleA;
  selB <- filtermoduleB(‘‘Argument’’, selC), filtermoduleA;
annotations
  selA <- anAnnotation;
```

Listing 20.2: Examples of bindings

Examples

A binding has a left hand side and a right hand side, the left hand side contains a selector identifier, which contains a selection of classes. The right hand side has one or more filter module identifiers with the appropriate arguments or one of more annotations. Some examples are shown in Listing 20.2.

Legality Rules

- The used selectors must be defined in the same superimposition block;
- The used filter module references must point to an existing filter module.

Commentary

More Binding Operations

In the begin days of Compose* there were ideas on binding and unbinding on events, however these never got implemented. This leaves us with the only weave operator: “`<-`”.

Introduction of Parameters

The binding of filter modules differs from the annotation binding because it has the possibility to attach arguments. In this argument list we allow constructors to get an unique external that is only shared for that binding.

Chapter 21

Implementation Part

The implementation part is were the base language dependent code of a concern is specified. The code is copied to a format known to the base language and is compiled by the compiler of the base language. This is done to save the effort of writing our own compiler.

Syntax

The heading of the implementation contains the name of the source language, the class name and the file name. The file name needs to be marked with quotes. The source code must be written according the specification of the base language. The Compose* syntax for the implementation part is shown in Listing 21.1.

Semantics

The implementation part is the base language dependent part of a concern. As we have already seen in chapter 12 is a concern with only an implementation part just a class.

Examples

Listing 21.2 shows a C# class that is fully placed in a concern. Besides adding a complete class by stating it in the *cps* file it also possible to point to a dll file. Every concern can only have one implementation part.

```
Implementation ::= ‘implementation’(‘by’ ClassName ‘;’
                 | ‘in’ SourceLanguage ‘by’ ClassName ‘as’ FileName ‘{’ Source ‘}’
Source ::= (?)*
```

Listing 21.1: Implementation syntax

```
concern Bar in GrammarTest{
  implementation in Csharp by Bar as "Bar.cs"{
    using System;

    namespace GrammarTest{
      public class Bar{
        public void write(){
          System.Console.WriteLine("Bar");
        }

      public bool isBar(){
        return true;
      }
    }
  }
}
```

Listing 21.2: The class Bar as concern

Legality rules

- When the code is added in a concern, you need the full heading;
- The extension of the file name must be appropriate for the base language.

Commentary

There have been several ideas on the usage of the implementation part. In [Sal01] the proposal is made to add implementations for each base language. So for the source for a logger you write for instance a Java and a C♯ source.

Part III

Conclusions

Chapter 22

Conclusions

In this chapter we take a look to work related to this thesis, followed by the conclusions of the analysis of the Compose* language and the evaluation of the overall work. At the end there is an overview of the work that lays ahead.

22.1 Related Work

22.1.1 Sally

"Sally is a general-purpose aspect language that is highly inspired by AspectJ." [HU03] The purpose of Sally is to demonstrate that AspectJ and Hyper/J are not reusable enough and that the reusability increases with the introduction of parameters. Sally has a similar pointcut language as AspectJ.

The syntax for the parameters has been a source of inspiration for designing the parameter syntax of Compose*. There are some differences in the semantics, these come from the difference between Compose* and AspectJ.

22.1.2 LogicAJ

LogicAJ is the successor of the work on Sally [KRH04]. It is based on idea that the introduction of parameters is only the first step to a generic aspect language, and to get a general solution for reusable and evolvable pattern implementations, a different language is needed. LogicAJ is an extension of AspectJ and it goes further with parameterizing than Sally, because almost every language part can be parameterized, with the exception of keywords. A new idea is the introduction of a parameter list, which applies the statement, where it is a part of, to every member of the list.

Sally and LogicAJ have been a source of inspiration for the parameters in Compose*, however LogicAJ also demonstrates some ideas that we do not want in Compose*, like the possibility to use parameters in every language construct. This possibility does provide generic code, but this code becomes difficult to read and maintain. For Compose* we have decided to limit the places where the user can place a parameter. Another difference is the usage of the parameter list, whereas LogicAJ uses to apply the statement to each member of the list, Compose* uses it as a set, where sometimes it must match one of the list and sometimes all members must match.

22.1.3 Eos

Eos is a AspectJ like extension to C# [Eos04]. Eos is the basis language for some spin offs. For the analysis on the Compose* language the Eos-U is the most interesting spino off. Eos-U combines class and join point specifications into one element: the *classpect*. This classpect element is treated like a class in object-oriented programming. It is possible to create instances from the classpect and these instances can be used for example as arguments in functions.

A characteristic of the base language Eos is that it uses the before, after, and around advice of AspectJ. However Eos has limited the advice options, so that only method calls can be used. This is done to improve the means for reasoning on the behavior of the application.

The idea of combining aspects and classes into one element is a good comparison for the Compose* approach of placing filter modules, superimposition and implementation in one element. The idea of limiting the language model for the possibility to reason about an application is a trade off which also occurs in the Compose* language.

22.1.4 Composable Message Manipulators

In [Stu95] a model is demonstrated as an alternative filter representation. The model is called Composable Message Manipulators and the goal of this model is to describe Composition Filters in an object-oriented way. The model uses objects, called Message Manipulators, to reason on a message. A Message Manipulator object has a decision, an accept action, and a reject action. The decision can be a condition or a matching. When a condition returns a true or a matching matches with the message, then the accept action is executed and otherwise the reject action.

The decision of a Message Manipulator object can be composed of two other Message Manipulator objects, creating a recursive system of decisions. Each sub-decision can have its own accept and reject actions.

The Composable Message Manipulators model demonstrates an idea for building a filter representation and the model is not a proposal for the syntax. It differs from the current filter specification of Compose*, because in Compose* a filter element has one decision, composed of conditions and matchings, in combination with one accept action, the substitution part. With the Composable Message Manipulators model the composition of conditions and methods is broken up in atomic elements and each element gets its own accept and reject action. The composition of these atomic elements can also have accept and reject actions, this is demonstrated in Figure 22.1. In this figure there are two decisions, `Evaluate(userview)` and `Signature(*\ deliver)`, and they have both their own accept and reject actions. They are composed by the *CAND* operator which is a *Conditional AND* operator, which means that there is an order in which the components are evaluated. The combination of the two decisions has also their own accept and reject actions.

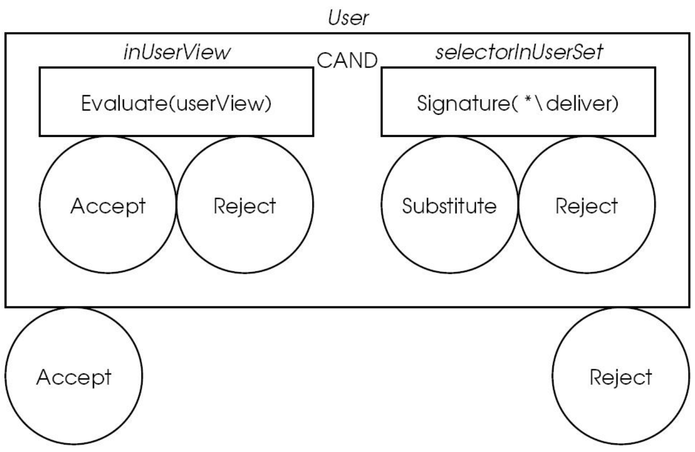

Figure 22.1: Demonstration of the Composable Message Manipulators

22.2 Conclusion

After the analysis we have determined the shortcomings of the Compose* language and identified redundant parts. The shortcomings are that the filter modules are not reusable enough and the filter specification lacks expressiveness. The redundant parts are the method block in the filter module and the condition and method binding in the superimposition. The reasons for their removal can be found in section 6.2 and section 6.3.

In chapter 4 the introduction of filter module parameters is described. With this addition to language it is now possible to write filter modules that can be reused without altering the original code. So it is now possible to write libraries of generic concerns.

The proposal for the new filter syntax in chapter 5, demonstrated the lack of expressiveness for filtering on messages. Filtering on only the target and the selector of a message is too limiting. The proposed filter syntax allows reasoning on all the attributes of a message. Using all the attributes of a message improves the expressiveness. It is possible to be backwards compatible with the old filter syntax.

Another existing idea are primitive values, such as integers and booleans, in the filter module. However we have demonstrated that the constructs, were the primitive values could be useful for, can already be solved with the current syntax . This gives that we can close the discussion and the search on how to introduce primitive values in the syntax, because we do not need them for the expressiveness of the language.

With the specification of the Annotation Reference Manual, the syntax and semantics of the Compose* language is recorded. This ensures that the semantics are now defined and will not change gradually over time without any notice. It ensures that all the different platforms have the same basis semantics for the basis designs and that the different platforms will not drift apart.

22.3 Evaluation

In the previous section we looked to the changes in the design of the Compose* language, in this section we look to how these changes are implemented in Compose*. All the parts of the language which can be removed as stated in chapter 6 are removed from the syntax. The new filter syntax has not been implemented yet, this is because we decided to first ratify the proposal and then implement it.

The implementation plan for the filter module parameters, in section 4.4, is tested. The filter module parameters for the type of the internal and the matching selector can now be parameterized. The selector can have a single parameter or a list of parameters.

22.4 Future Work

In this section we look to future work and the road that lays ahead. We look to specific items and we leave out the maintenance work, like redesigns and over thinking concepts, which should be done on a regular basis.

Fine Tuned Design for Filter Specification

In this thesis a proposal is made for the new filter specification. Whereas this is just a draft proposal, it must be worked out in more detail when the general layout is being approved to replace the old filter element syntax of Compose*. The fine tuning will exist of working out the syntax and semantics in further detail and looking to other opportunities to improve the design where possible.

Extending the Constraint Model

In the current constraint model [Nag06] it is possible to define constraints on which filter module must be placed in front of another filter module. The introduction of the filter module parameters makes it possible that one filter module is bound twice to an object with different arguments. The current constraint model cannot handle this, so it is necessary to work out a model that can handle the filter module parameters. This model also needs a syntax, so besides the model there most also be a design for the syntax of the new constraint model.

Applying New Selector Syntax

In chapter 19 we have already seen that the selector syntax can be extended to offer better support to be used as arguments in filter module bindings. Currently only one set is retrieved from each selector definition. So changing the syntax so that more then one set can be retrieved and by reusing definitions the selectors become more useful for selecting arguments. A proposal for the design for the new selector is made (Commentary of chapter 19), the only thing that now has to be done is the implementation.

Part IV

Appendices

Appendix A

Old Grammar from the Rembrandt Release

```
Concern ::= ‘concern’ Identifier [‘(’ FormalParameters ‘)’]
            [‘in’ Namespace] ConcernBlock
FormalParameters ::= FormalParameterDef (‘;’ FormalParameterDef)*
FormalParameterDef ::= Identifier (‘,’ Identifier)* ‘:’ Type
Namespace ::= Identifier (‘.’ Identifier)*
ConcernBlock ::= Begin (FilterModule)* [SuperImposition] [Implementation]
                 End [‘concern’ [Indentifier]]

FilterModule ::= ‘filtermodule’ FilterModuleName Begin [Internals]
                  [Externals] [Conditions] [Methods] [InputFilters]
                  [OutputFilters] End [‘filtermodule’ [FilterModuleName]];

Internals ::= ‘internals’ (Identifier-LIST ‘:’ Type ‘;’)*
Externals ::= ‘externals’ (Identifier ‘:’ Type
              [‘=’ InitialisationExpression ‘(’ ‘)’]‘;’)*
InitialisationExpression ::= MethodReference
Conditions ::= ‘conditions’ (Identifier ‘:’ MethodReference ‘;’)*
MethodReference ::= (FilterModuleElement‘‘.’’MethodName | FullyQualifiedName)
Methods ::= ‘methods’ MethodDeclarations*
MethodDeclarations ::= MethodName ‘(’ [FormalParameterTypeDef-SEQ] ‘)’
                       [‘;’ ReturnType] ‘;’
FormalParameterTypeDef ::= [Identifier-LST ‘:’] Type

Inputfilters ::= ‘inputfilters’ FilterSet
Outputfilters ::= ‘outputfilters’ FilterSet

FilterSet ::= Filter (‘;’ Filter)*
Filter ::= FilterName ‘:’ FilterType [Arguments] ‘=’ ‘{’ FilterElements ‘}’
FilterType ::= Identifier
Arguments ::= ‘(’ Value-LIST ‘)’
FilterElements ::= FilterElement (ElementCompositionOperator FilterElement)*
ElementCompositionOperator ::= ’,’

FilterElement ::= [ORExpression ConditionOperator] MessagePatternSet
ORExpression ::= ANDExpression [‘|’ ANDExpression]
ANDExpression ::= NOTExpression [‘&’ NOTExpression]
NOTExpression ::= [!] (ConditionLiteral | ‘(’ORExpression‘)’)
ConditionLiteral ::= ConditionName | ‘True’| ‘False’
ConditionOperator ::= ‘=>’ | ‘~>’
MessagePatternSet ::= ‘{’ MessagePattern
                      (ElementCompositionOperator MessagePattern)* ‘}’
                     | MessagePattern
```

```
MessagePattern ::= SignatureMatching [SubstitutionPart]
                   | NameMatching [SubstitutionPart]
                   | MatchPattern
SignatureMatching ::= ‘<’ MatchPattern ‘>’
NameMatching ::= ‘[’ MatchPattern ‘]’ | Quote MatchPattern Quote
SubstitutionPart ::= [Target ‘.’] Selector
MatchPattern ::= [Target ‘.’] Selector

Superimposition ::= ‘superimposition’ Begin [Selectors] [ConditionBindings]
                    [MethodBindings] [FilterModuleBindings]
                    [AnnotationBindings] [Constraints] End [‘superimposition’]

Selectors ::= ‘selectors’ (SingleSelector)*
SingleSelector ::= Identifier ‘=’ ‘{’
                    ((OldSelectionExpression (’,’ OldSelectionExpression)*)
                    | PredicateExpression) ‘}’ ‘;’
OldSelectionExpression ::= ‘*’ (‘=’ | ‘:’) ConcernReference
PredicateExpression ::= Identifier PROLOG_EXPRESSION

ConditionBindings ::= ‘conditions’ (ConditionBinding ‘;’)*
ConditionBinding ::= ‘{’ ConditionName-LIST ‘}’ | ConditionName-LIST
MethodBindings ::= ‘methods’ (MethodBinding ‘;’)*
MethodBinding ::= ‘{’ MethodName-LIST ‘}’ | MethodName-LIST

FilterModuleBindings ::= ‘filtermodules’ (FilterModuleBinding ‘;’)*
FilterModuleBinding ::= Identifier WeaveOperator
                        (FilterModuleReference-LIST
                        | ‘{’ FilterModuleReference-LIST ‘}’)
FilterModuleReference ::= Identifier [‘(’ FilterArgument-LIST ’)’]
AnnotationBindings ::= ‘annotations’ (AnnotationBinding ‘;’)*
AnnotationBinding ::= Identifier WeaveOperator
                        (Identifier-LIST | ‘{’ Identifier-LIST ‘}’)
WeaveOperator ::= ‘<-’

Constraints ::= ‘constraints’ (ConstraintElement ‘;’)+
ConstraintElement ::= ConstraintCondition ‘(’ FilterModuleReference
                      ‘,’ FilterModuleReference ‘)’
ConstraintCondition ::= ‘pre’ | ‘presoft’ | ‘prehard’

Implementation ::= ‘implementation’( ‘by’ ClassName ‘;’
                   | ‘in’ SourceLanguage ‘by’ ClassName
                     ‘as’ FileName ‘{’ Source ‘}’ )

ELEMENT-LIST ::= ELEMENT (‘,’ ELEMENT)*
ELEMENT-SEQ ::= ELEMENT (‘;’ ELEMENT)*

Begin = ‘{’ | ‘begin’
End = ‘}’ | ‘end’
ClassName ::= Identifier (‘.’ Identifier)*
FileName ::= Quote (Letter | Digit | Special | Dot)* Quote
FilterArgument ::= SelectorIdentifier | MethodReference | Value
FilterModuleElement ::= [PackageName (‘.’ PackageName)* ‘::’]
                        FilterModuleName ‘:’ FilterElement
FilterModuleName ::= Identifier
FilterName ::= Identifier
Identifier ::= (Letter | Special) (Letter | Digit | Special)*
MethodName ::= Identifier
Quote ::= ‘"’
Selector ::= MethodName [‘(’ Type-SEQ’‘)’] | ‘*’
SelectorIdentifier ::= Identifier
SourceLanguage ::= Identifier
Special ::= ‘_’
```

```
Target ::= Identifier | ‘inner’| ‘*’
Type ::= ClassName
Value ::= Identifier | Number

PROLOG_EXPRESSION ::= VarName ‘|’ PrologBody
VarName ::= UpperCase (LowerCase)*
PrologBody ::= PrologFun-LIST
PrologFun ::= ConstString [‘(’ [Arg-LIST] ‘)’ ]
Arg::= PrologFun | PrologVar | PrologList | ConstNum
PrologVar ::= ‘_’ | VarName
PrologList ::= ‘[’ ‘]’ | ‘[’ ListElems ‘]’
ListElems ::= [Arg-LIST] [ ‘|’ (PrologList | PrologVar) ]
```

Appendix B

Proposed Grammar

```
Concern ::= ‘concern’ Identifier [‘in’ Namespace]
            ‘{’ (FilterModule)* [SuperImposition] [Implementation] ‘}’
Namespace ::= Identifier (‘.’ Identifier)*

FilterModule ::= ‘filtermodule’ FilterModuleName [FilterModuleParameters] ‘{’
                  [Internals] [Externals] [Conditions]
                  [InputFilters] [OutputFilters] ‘}’;
FilterModuleParameters ::= ‘(’ [ParameterDefinition-LIST] ‘)’
ParameterDefinition ::= Parameter | ParameterList
Parameter ::= ‘‘?’’Identifier
ParameterList ::= ‘‘??’’Identifier

Internals ::= ‘internals’ (Identifier-LIST ‘:’ Type
              [‘=’ InitialisationExpression ‘(’ [ Argument-LIST ] ‘)’]‘;’)*
Externals ::= ‘externals’ (Identifier ‘:’ Type
              ‘=’ InitialisationExpression ‘(’ [ Argument-LIST ] ‘)’ ‘;’)*
InitialisationExpression ::= MethodReference
Conditions ::= ‘conditions’ (Identifier ‘:’ MethodReference ‘;’)*
MethodReference ::= (FilterModuleElement‘‘.’’MethodName | FullyQualifiedName)
                    [Arguments]
Arguments ::= ‘(’ Value-LIST ‘)’

Inputfilters ::= ‘inputfilters’ FilterSet
Outputfilters ::= ‘outputfilters’ FilterSet

FilterSet ::= Filter (FilterOperator Filter)*
FilterOperator ::= ‘;’
Filter ::= FilterName ‘:’ FilterType [Arguments] ‘=’ ‘{’ FilterElements ‘}’
FilterType ::= Identifier
FilterElements ::= FilterElement (ElementCompositionOperator FilterElement)*
ElementCompositionOperator ::= ’,’

FilterElement ::= [ORExpression ConditionOperator] MessagePattern
ORExpression ::= ANDExpression [‘|’ ANDExpression]
ANDExpression ::= NOTExpression [‘&’ NOTExpression]
NOTExpression ::= [!] (ConditionLiteral | ‘(’ORExpression‘)’)
ConditionLiteral ::= ConditionName | ‘True’| ‘False’
ConditionOperator ::= ‘=>’ | ‘~>’
MessagePattern ::= Matching [SubstitutionPart]
                   | MatchPattern
                   | ‘{’ Matching (‘,’ Matching)* ‘}’ [SubstitutionPart]
Matching ::= SignatureMatching | NameMatching
```

```
SignatureMatching ::= ‘<’ MatchPattern ‘>’
NameMatching ::= ‘[’ MatchPattern ‘]’
SubstitutionPart ::= [Target ‘.’] Selector
MatchPattern ::= [Target ‘.’] Selector

Superimposition ::= ‘superimposition’ ‘{’ [Selectors] [FilterModuleBindings]
                    [AnnotationBindings] [Constraints] ‘}’

Selectors ::= ‘selectors’ (SingleSelector)*
SingleSelector ::= Identifier ‘=’ ‘{’ PredicateExpression ‘}’ ‘;’
PredicateExpression ::= Identifier PROLOG_EXPRESSION

FilterModuleBindings ::= ‘filtermodules’ (FilterModuleBinding ‘;’)*
FilterModuleBinding ::= Identifier WeaveOperator
                        (FilterModuleReference-LIST
                        | ‘{’ FilterModuleReference-LIST ‘}’)
FilterModuleReference ::= Identifier [‘(’ FilterArgument-LIST ’)’]
AnnotationBindings ::= ‘annotations’ (AnnotationBinding ‘;’)*
AnnotationBinding ::= Identifier WeaveOperator
                        (Identifier-LIST | ‘{’ Identifier-LIST ‘}’)
WeaveOperator ::= ‘<-’

Constraints ::= //Not yet implementend

Implementation ::= ‘implementation’(‘by’ ClassName ‘;’
                   | ‘in’ SourceLanguage ‘by’ ClassName ‘as’ FileName ‘{’ Source ‘}’

ELEMENT-LIST ::= ELEMENT (‘,’ ELEMENT)*
ELEMENT-SEQ ::= ELEMENT (‘;’ ELEMENT)*

ClassName ::= Identifier (‘.’ Identifier)*
FileName ::= Quote (Letter | Digit | Special | Dot)* Quote
FilterArgument ::= SelectorIdentifier | MethodReference | Value
FilterModuleElement ::= [PackageName (‘.’ PackageName)* ‘::’]
                        FilterModuleName ‘:’ FilterElement
FilterModuleName ::= Identifier
FilterName ::= Identifier
Identifier ::= (Letter | Special) (Letter | Digit | Special)*
MethodName ::= Identifier
Quote ::= ‘"’
Selector ::= MethodName [‘(’ Type-SEQ’‘)’] | ‘*’ | Parameter | ParameterList
SelectorIdentifier ::= Identifier
SourceLanguage ::= Identifier
Special ::= ‘_’
Target ::= Identifier | ‘inner’| ‘*’ | Parameter
Type ::= ClassName | Parameter
Value ::= Identifier | Number | Parameter | ParameterList

PROLOG_EXPRESSION ::= VarName ‘|’ PrologBody
VarName ::= UpperCase (LowerCase)*
PrologBody ::= PrologFun-LIST
PrologFun ::= ConstString [‘(’ [Arg-LIST] ‘)’ ]
Arg::= PrologFun | PrologVar | PrologList | ConstNum
PrologVar ::= ‘_’ | VarName
PrologList ::= ‘[’ ‘]’ | ‘[’ ListElems ‘]’
ListElems ::= [Arg-LIST] [ ‘|’ (PrologList | PrologVar) ]
```

Appendix C

Filter Elements

The filter elements are the parts of the right hand side of a filter. They are the proposed new filter syntax, as described in chapter 4. The layout for the condition, matching, and substitution part, are alike in this syntax and therefore they have been combined into one chapter.

In the filter element part it is possible to specify the matching and substitution behavior of a filter. The old syntax is still usable, because it can be transformed to the new syntax. How this transformation is done is shown in Appendix D.

Syntax

Listing C.1 shows the syntax for the filter elements. The conditions and matching elements are first, followed by the substitution elements. The conditions elements are the labels defined in the condition field of the filter module. The matching elements are the name of a message property, an operator, and one or more values. It is also possible to use a function of the message. The set of usable operators in the matching has not been fully worked out yet. It is possible to combine both with the standard set of logical operators, AND, OR, and NOT. The substitution element has only the ":=" operator to assign a value to a message property.

Semantics

The condition and matching elements provide boolean values which can be used in a logical expression to determine whether a filter matches or not. If a filter matches the substitution elements are being executed.

The matching elements can have predefined operators, the proposed ones are "=" and "sig". The operator "=" is short hand for the equals operator and for a set it means of the out of the set. And the operator "sig" means signature matching, this operator can only be used to check on a selector whether it is in a certain signature. For lists we also have the "equalsOne" and "equalsAll". Other operators have to be assigned yet. The matching elements has only the assignment operator, which is defined as ":=".

```
GeneralFilterSet ::= GeneralFilter (FilterCompositionOperator GeneralFilter)*
FilterCompositionOperator ::= ‘;’
GeneralFilter ::= FilterName ‘:’ FilterType [‘(’ ActualFilterParameters ‘)’];
                         ‘=’ ‘{’ [FilterElements] ‘}’
FilterElements ::= FilterElement (ElemCompositionOperator FilterElement)*
ElemCompositionOperator ::= ‘;’

FilterElement ::=  [ConditionAndMatchingPart ‘,’] AssignmentPart
                   | ConditionAndMatchingPart

ConditionAndMatchingPart ::= OrExpr
OrExpr ::= AndExpr (‘OR’ AndExpr)*
AndExpr ::= PrimaryExpr (‘AND’ PrimaryExpr)*
PrimaryExpr ::= [NOT] PlainExpr;
PlainExpr ::= ‘(’ OrExpr ‘)’ | ConditionPart | MatchingPart

ConditionPart ::= ConditionLiteral
ConditionLiteral ::= ‘True’ | ‘False’ | ConditionName
MatchingPart ::= FieldName MatchingOperator (Value | ‘{’ Value-LIST ‘}’)
                 | FieldOperation
MatchingOperator ::= equals | equalsOne | equalsAll | sig | OtherOperators

AssignmentPart ::= Assignment (AND Assignment)*
Assignment ::= FieldName ‘:=’ Value | ‘{’ Value-LIST ‘}’ | FieldOperation
AssignmentOperator ::= ‘:=’ | OtherOperators

FieldName ::= Identifier
FieldOperation ::= fqnwithArguments
AND ::= ‘&’ | ‘,’
OR ::= ‘|’
NOT ::= ‘!’
Value ::= Identifier | Number | Parameter | ParameterList
Methodcall ::= Identifier ‘(’ Value-LIST ‘)’
OtherOperator ::= (* To Be Specified *)
```

Listing C.1: Filter element syntax

```
filtermodule dynamicstrategy{
  internals
    stalk_strategy : pacman.Strategies.StalkerStrategy;
    flee_strategy : pacman.Strategies.FleeStrategy;
  conditions
    pacmanIsEvil : pacman.Pacman.isEvil();
  inputfilters
    stalker_filter : Dispatch = {!pacmanIsEvil, selector = getNextMove,
                                 target := stalk_strategy };
    flee_filter : Dispatch = {selector = getNextMove, target := flee_strategy }
}
```

Listing C.2: Dynamic strategy filter module in Pacman in new filter syntax

The comma's between the elements are AND operators. It also has become possible to write filters without any condition or matching elements, so that the substitution elements are always executed.

Examples

In the old syntax entry in the reference manual we have seen two examples, Listing 13.2 and Listing 13.3, these can be rewritten to the new syntax. The result for Pacman is demonstrated in Listing C.2 and the other is demonstrated in Listing C.3. Listing C.4 is an example which is already stated in chapter 4.

Legality Rules

- It is not possible to place an assignment part before a condition or matching element;
- When a attribute of a message is used, it must be available for that type of message.

Commentary

Removed from the Old Syntax

Several options of the old syntax have not been ported to the new syntax. Two operators which are now removed are the `=>` and ~`>`, this is done because they have become we can use logical operators on the matching elements.

Separated Substitution Elements

In the new design we can see when the first substitution element is being parsed, however it is not certain this still works when we also allow functions on message attributes. Because these substitution elements can also been seen as matching elements. In that scenario we must introduce a separator symbol like "`/`", to mark the transition between condition and matching elements and the substitution elements. This separator comes from the original ideas, but is dropped because we could parse the filter correctly without. When we would come to the conclusion that this is indeed impossible, we need to add it again to the syntax.

Filter Operators

```
filtermodule logger(??inputMethods, ??outputMethods){
  externals
    logger : Logger = Logger.instance();
  conditions
    isLoggingOn : logger.isOn();
  inputfilters
    inlog : Meta = {isLoggingOn, selector = ??inputMethods, target := logger,
                    selector := log}
  outputfilters
   outlog : Meta = {isLoggingOn, selector = ??outputMethods, target := logger,
                    selector := log}
}
```

Listing C.3: Custom logging filter module, in new filter syntax

```
filtermodule authorization(?rules){
internals
  ruleBase = ?rules;
externals
  mailer = MassMailer.instance();
conditions
  hasAccesLevelThree : ruleBase.hasAccesLevelThree;
  hasAccesLevelFive : ruleBase.hasAccesLevelFive;
inputfilters
  rule1 : Dispatch = {hasAccesLevelThree, sender = admin, selector sig mailer,
                      target := mailer};
  rule2 : Dispatch = {hasAccesLevelFive, sender = secretary, selector = mailer,
                      target := mailer};
  rule3 : Dispatch = {!(timestamp > 12.00 & timestamp < 15.00),
    (sender = (secretary | admin))| hasAccesLevelThree,
     selector sig mailer, target := mailer};
  rule4 : Dispatch = {!(timestamp > 8.00 & timestamp < 16.00) selector sig mailer,
                      target := mailer};
  error : Error = {}
}
```

Listing C.4: The mass mailer from the filter syntax analysis

```
inputfilters
prefilter : Check ={selector = getNextMove} AND
(
  stalker_filter : Dispatch = {!pacmanIsEvil, target := stalk_strategy } OR
  flee_filter : Dispatch = {target := flee_strategy }
}
```

Listing C.5: Dynamic strategy filter module in Pacman, with filter operators

With the introduction of the new filter syntax, we can look again to the filter operators. With filter operators we can rewrite the input filter set of Listing C.2 to Listing C.5. However we get again stuck on the parenthesis and this example could be better solved by allowing filters like Listing C.6., the only drawback is that such filter constructions are not allowed with the proposed syntax. We should not be eager to add this construct to the syntax, because the behavior filter is harder to read then the two filter version.

```
inputfilters
behavior : Dispatch ={selector = getNextMove AND ((!pacmanIsEvil,
                      target := stalk_strategy ) OR target := flee_strategy )
```

Listing C.6: Dynamic strategy filter module in Pacman, done in one filter

Appendix D

Converting the Old Filter Syntax

```
FilterSet ::= Filter (‘;’ Filter)*
Filter ::= FilterName ‘:’ FilterType [Arguments] ‘=’ ‘{’ FilterElement ‘}’

FilterElement ::= [ORExpression ConditionOperator] MessagePatternSet
ORExpression ::= ANDExpression [‘|’ ANDExpression]
ANDExpression ::= NOTExpression [‘&’ NOTExpression]
NOTExpression ::= [!] (ConditionLiteral | ‘(’ORExpression‘)’)
ConditionLiteral ::= ConditionName | ‘True’| ‘False’
ConditionOperator ::= ‘=>’ | ‘~>’
MessagePatternSet ::= ‘{’ MessagePattern-LIST ‘}’ | MessagePattern
MessagePattern ::= SignatureMatching [SubstitutionPart]
                 | NameMatching [SubstitutionPart]
                 | MatchPattern
SignatureMatching ::= ‘<’ MatchPattern ‘>’
NameMatching ::= ‘[’ MatchPattern ‘]’
SubstitutionPart ::= [Target ‘.’] Selector
MatchPattern ::= [Target ‘.’] Selector
```

Listing D.1: ”The old filter syntax”

With the introduction of the new filter element syntax we also want to be backwards compatible with the old style. This means that we must write transformation rules for the syntax demonstrated in Listing D.1, in order to transform it to Listing 5.6. The left hand side of the filter specification stays the same, so we only have to look to condition, matching, and substitution part.

The condition part can be transformed without much effort, the logical operators and the usage of conditions are the same so we can leave them as they are. This brings us to the *Condition-Operator*, the `=>` translates to an AND operator and the ~> to the code `!(...)`, where on the place of the dots the matching part is placed.

The next part to transform is the *MessagePattern*, the *MessagePattern-LIST* will be done later. There are three possibilities for the MessagePattern, it is name matching or signature matching with a substitution part or a *MatchPattern* without any substitution part, which is by default a signature matching. A name matching of the form `[matchTarget.matchSelector] substitutionTarget.subsitutionSelector` transforms to `target = matchTarget AND selector = matchSelector AND target := substitutionTarget AND selector := subsitutionSelector`.

The signature matching `<matchTarget.matchSelector> substitutionTarget.subsitutionSelector` transforms to `selector SIG matchTarget AND target := substitutionTarget AND selector := subsitutionSelector`.

The default MatchPattern transforms to `selector SIG matchTarget`.

To clean up the transformed code it is possible to remove the elements containing a wildcard, but this is not necessary.

The MessagePattern-LIST transforms the separate MessagePatterns as described above and places them in a `(MessagePattern1 | MessagePattern2 | ... | MessagePatternX)` construct.

The described transformation can be done in the normal two step approach, thus first compiling an old filter to its AST and then transform the old style AST to the new style AST. However, the transformation is not that complex and it is even possible to create a new style AST from the old style EBNF, which saves us a step in the compiling. This can be done without much effort, because there are only six transformation rules.

Appendix E

The Old Short Hand Syntax of the Filter Module

```
FilterModule ::= ‘filtermodule’ FilterModuleName ‘{’
                  [Internals] [Externals] [Conditions]
                  [InputFilters] [OutputFilters] ‘}’

Internals ::= ‘internals’ (Identifier-LIST ‘:’ Type ‘;’)*
Externals ::= ‘externals’ (Identifier-LIST ‘:’ Type
               [‘=’ InitializationExpression] ‘;’)*
InitializationExpression ::= ConstructorReference
Conditions ::= ‘conditions’ (ConditionDecl)*
ConditionDecl ::= ConditionName ‘:’ ConditionReference ‘;’

InputFilters ::= ‘inputfilters’ GeneralFilterSet
OutputFilters ::= ‘outputfilters’ GeneralFilterSet

GeneralFilterSet ::= GeneralFilter
                     (FilterCompositionOperator GeneralFilter)*
FilterCompositionOperator ::= ‘;’
GeneralFilter ::= FilterName ‘:’ FilterType
                  [‘(’ ActualFilterParameters ‘)’]
                  ‘=’ ‘{’ [FilterElements] ‘}’
ActualFilterParameters ::= Value-LIST
FilterElements ::= FilterElement
                   (ElemCompositionOperator FilterElement)*
ElemCompositionOperator ::= ‘,’
FilterElement ::= [ConditionExpression ConditionOperator]
                  MessagePatternSet
ConditionOperator ::= ‘=>’  | ‘~>’
ConditionExpression ::= ConditionLiteral | ‘!’ ConditionLiteral |
                        ‘(’ ConditionExpression (‘|’ | ‘&’)
                        ConditionExpression ‘)’
ConditionLiteral ::= ConditionName| ‘True’ | ‘False’
MessagePatternSet ::= ‘{’ MessagePattern-LIST ‘}’ | MessagePattern
MessagePattern ::= SignatureMatching SubstitutionPart
                   | NameMatching SubstitutionPart
                   | DefaultMatchAndSubstitute
SignatureMatching ::= ‘<’ MatchPattern ‘>’
NameMatching ::= ‘[’ MatchPattern ‘]’ | Quote MatchPattern Quote
DefaultMatchAndSubstitute ::= MatchPattern
SubstitutionPart:: MatchPattern
```

```
MatchPattern ::= [Target ‘.’] Selector
Target ::= Identifier | ‘inner’ | ‘*’
Selector ::= MethodName [‘(’ [Type-SEQ] ‘)’] | ‘*’

ConcernReference ::= [PackageReference ‘.’] ConcernName
ConditionReference ::= ConstructorReference
ConstructorReference ::= ConcernReference ‘()’
PackageReference ::= (PackageName ‘.’)* PackageName
Type ::= ConcernReference
```

Appendix F

The Proposed Short Hand Syntax of the Filter Module

```
FilterModule ::= ‘filtermodule’ FilterModuleName
                  [FilterModuleParameters] ‘{’
                  [Internals] [Externals] [Conditions]
                  [InputFilters] [OutputFilters] ‘}’

FilterModuleParameters ::= ’(’ [ParameterDefinition-LIST] ’)’
ParameterDefinition ::= Parameter | ParameterList

Internals ::= ‘internals’ (Identifier-LIST ‘:’ Type ‘;’)*
Externals ::= ‘externals’ (Identifier-LIST ‘:’ Type
               [‘=’ InitializationExpression] ‘;’)*
InitializationExpression ::= ConstructorReference | Parameter
Conditions ::= ‘conditions’ (ConditionDecl)*
ConditionDecl ::= ConditionName ‘:’ ConditionReference ‘;’

InputFilters ::= ‘inputfilters’ GeneralFilterSet
OutputFilters ::= ‘outputfilters’ GeneralFilterSet

GeneralFilterSet ::= GeneralFilter
                     (FilterCompositionOperator GeneralFilter)*
FilterCompositionOperator ::= ‘;’
GeneralFilter ::= FilterName ‘:’ FilterType
                  [‘(’ ActualFilterParameters ‘)’]
                  ‘= {’ [FilterElements] ‘}’
ActualFilterParameters ::= Value-LIST

FilterElements ::= FilterElement
                   (ElemCompositionOperator FilterElement)*
ElemCompositionOperator ::= ‘,’
FilterElement ::= [ConditionExpression ConditionOperator]
                  MessagePattern
ConditionOperator ::= ‘=>’  | ‘~>’
ConditionExpression ::= ConditionLiteral | ‘!’ ConditionLiteral |
                        ‘(’ ConditionExpression (‘|’ | ‘&’)
                        ConditionExpression ‘)’
ConditionLiteral ::= ConditionName | ‘True’ | ‘False’
MessagePattern ::= MessagePattern ::= Matching [SubstitutionPart]
                   | MatchPattern
                   | ‘{’ Matching (‘,’ Matching)* ‘}’ [SubstitutionPart]
```

```
Matching ::= SignatureMatching | NameMatching
SignatureMatching ::= ‘<’ MatchPattern ‘>’
NameMatching ::= ‘[’ MatchPattern ‘]’ | Quote MatchPattern Quote
DefaultMatchAndSubstitute ::= MatchPattern
SubstitutionPart:: MatchPattern
MatchPattern ::= [Target ‘.’] Selector
Target ::= Identifier | ‘inner’ | ‘*’ | Parameter
Selector ::= MethodName [‘(’ [Type-SEQ]‘)’] | ‘*’
             | Parameter | ParameterList

ConcernName ::= Identifier| Parameter
ConcernReference ::= [PackageReference ‘.’] ConcernName
ConditionName ::= Identifier
ConditionReference ::= ConcernReference
                       | ConstructorReference
                       | Parameter
ConstructorReference ::= ConcernReference ‘(’ Value-LIST ‘)’
                         | Parameter
FilterModuleName ::= Identifier
FilterType ::= Identifier | Parameter
PackageReference ::= (PackageName ‘.’)* PackageName
Type ::= ConcernReference | Parameter
Value ::= Indentifier | Number | Parameter
Parameter ::= ‘?’Identifier
ParameterList ::= ‘??’Identifier
```

Bibliography

[Ada95] *Ada Reference Manual*, 1995.

[BA01] L. Bergmans and M. Akşit. *Composing Crosscutting Concerns Using Composition Filters. Comm. ACM*, volume 44(10):pp. 51–57, October 2001.

[BA05] L. Bergmans and M. Akşit. *Principles and Design Rationale of Composition Filters.* In Filman et al. [FECA05], (pp. 63–95).

[Ber94] L. Bergmans. *Composing Concurrent Objects.* Ph.D. thesis, University of Twente, 1994.
URL `http://trese.cs.utwente.nl/publications/paperinfo/bergmans.phd.pi.top.htm`

[BKLM91] J. van Benthem, J. Ketting, J. Lodder, and W. Meyer-Viol. *Logica voor informatica.* Pearson, 1991. 3rd edition November 2003, ISBN : 90-430-0722-6, (Dutch).

[Bos04] R. Bosman. *Automated Reasoning about Composition Filters.* Master's thesis, University of Twente, The Netherlands, November 2004.

[Bos06] S. R. Boschman. *Performing transformations on .NET Intermediate Language Code.* Master's thesis, University of Twente, The Netherlands, August 2006.

[Com06] *Compose* homepage*, 2006.
URL `http://composestar.sf.net`

[Con06] O. Conradi. *Fine-grained Join Point Model in Compose*.* Master's thesis, University of Twente, The Netherlands, August 2006.

[Doo06] D. Doornenbal. *Analysis and Redesign of the Compose* Language.* Master's thesis, University of Twente, The Netherlands, October 2006.

[Dür04] P. E. A. Dürr. *Detecting semantic conflicts between aspects (in Compose*).* Master's thesis, University of Twente, The Netherlands, April 2004.

[EFB01] T. Elrad, R. E. Filman, and A. Bader. *Aspect-Oriented Programming. Comm. ACM*, volume 44(10):pp. 29–32, October 2001.

[Eos04] *Eos homepage*, 2004. Retrieved on 13 March 2006.
URL `http://www.cs.iastate.edu/~eos/`

[FECA05] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit (Editors). *Aspect-Oriented Software Development.* Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software.* Addison Wesley, 1995.

[GL03] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java.* John Wiley and Sons, 2003. ISBN 0471431044.

[Gla95] M. Glandrup. *Extending C++ using the concepts of Composition Filters.* Master's thesis, University of Twente, 1995.
URL `http://trese.cs.utwente.nl/publications/paperinfo/glandrup.thesis.pi.top.htm`

[Hav05] W. Havinga. *Designating join points in Compose* - a predicate-based superimposition language for Compose*.* Master's thesis, University of Twente, The Netherlands, May 2005.

[Hol04] F. J. B. Holljen. *Compilation and Type-Safety in the Compose* .NET Environment.* Master's thesis, University of Twente, The Netherlands, May 2004.

[HU03] S. Hanenberg and R. Unland. *Parametric Introductions.* In *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (edited by M. Akşit), (pp. 80–89). ACM Press, March 2003.

[Hui06] R. L. R. Huisman. *Debugging Composition Filters.* Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[Hut06] S. H. G. Huttenhuis. *Patterns within Aspect Orientation.* Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[KHH^{+}01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. *An overview of AspectJ.* In *Proc. ECOOP 2001, LNCS 2072* (edited by J. L. Knudsen), (pp. 327–353). Springer-Verlag, Berlin, June 2001.

[Koo95] P. Koopmans. *Sina user's guide and reference manual.* Technical report, Dept. of Computer Science, University of Twente, 1995.
URL `http://trese.cs.utwente.nl/publications/paperinfo/sinaUserguide.pi.top.htm`

[KRH04] G. Kniesel, T. Rho, and S. Hanenberg. *Evolvable Pattern Implementations Need Generic Aspects.* In *ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution (RAM-SE)* (edited by W. Cazzola, S. Chiba, and G. Saake), (pp. 111–126). June 2004.
URL `http://www.disi.unige.it/person/CazzolaW/RAM-SE04Proceedings/RAM-SE04-proceedings.pdf`

[Log05] *LogicAJ homepage*, 2005. Retrieved on 13 March 2006.
URL `http://roots.iai.uni-bonn.de/research/logicaj/`

[Min06] W. Minnen. *Design of new Filter Types for Data Abstraction.* Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[Nag06] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution.* Ph.D. thesis, University of Twente, The Netherlands, June 2006.

[OT01] H. Ossher and P. Tarr. *Multi-Dimensional Separation of Concerns and The Hyperspace Approach.* In *Software Architectures and Component Technology* (edited by M. Akşit). Kluwer Academic Publishers, 2001. ISBN 0-7923-7576-9.

[Oud06] M. D. W. van Oudheusden. *Automatic Derivation of Semantic Properties in .NET.* Master's thesis, University of Twente, The Netherlands, August 2006.

[PAG03] A. Popovici, G. Alonso, and T. Gross. *Just in Time Aspects.* In *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (edited by M. Akşit), (pp. 100–109). ACM Press, March 2003.

[PGA02] A. Popovici, T. Gross, and G. Alonso. *Dynamic Weaving for Aspect-Oriented Programming.* In *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)* (edited by G. Kiczales), (pp. 141–147). ACM Press, April 2002.

[Sal01] P. Salinas. *Adding Systemic Crosscutting and Super-Imposition to Composition Filters.* Master's thesis, Vrije Universiteit Brussel, August 2001.

[Sal03] *Sally homepage*, 2003. Retrieved on 13 March 2006.
URL `http://dawis.icb.uni-due.de/research/aosd/sally/`

[Spe06] D. R. Spenkelink. *Compose* Incremental.* Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[Sta05] T. Staijen. *Towards Safe Advice: Semantic Analysis of Advice Types in Compose*.* Master's thesis, University of Twente, April 2005.

[Stu95] C. Stuurman. *Techniques for defining Composition Filters Using Message Manipulators.* Master's thesis, University of Twente, August 1995.

[TOSH05] P. Tarr, H. Ossher, S. M. Sutton, Jr., and W. Harrison. *N Degrees of Separation: Multi-Dimensional Separation of Concerns.* In Filman et al. [FECA05], (pp. 37–61).

[Vin04] C. Vinkes. *Superimposition in the Composition Filters Model.* Master's thesis, University of Twente, The Netherlands, October 2004.

[Wat90] D. A. Watt. *Programming language concepts and paradigms.* Prentice Hall, 1990.

[Wic99] J. C. Wichman. *The Development of a Preprocessor to Facilitate Composition Filters in the Java Language.* Master's thesis, University of Twente, 1999.
URL `http://trese.cs.utwente.nl/publications/paperinfo/wichman.thesis.pi.top.htm`

[Win06] J. W. te Winkel. *Bringing Composition Filters to C.* Master's thesis, University of Twente, The Netherlands, 2006. To be released.