Compose* Annotated Reference Manual

October 31, 2006

Reading guide

Each chapter of the ARM handles a part of the Compose* language. In each chapter there is first an abstract of what that particular part of the language should do. After that the syntax and the semantics are demonstrated. How it can be used is shown in the examples. After that the legality rules and the FAQ are stated.

The how, what, and the design decisions are mentioned in the comments. If added there can also be specific platform comments for each Compose* platform.

When there is more to read about a chapter then this is mentioned in the "further reading" section of a chapter.

Contents

List of Figures

Chapter 1

Glossary

Compose*/Java Compose*.Java is the Java port of Compose*.

Compose*/DotNET Compose*.Net is the .Net port of Compose*.

Compose* developer A member of the Compose* team, thus who programs on Compose*. This term is used to avoid the ambiguous term Compose* programmer.

Compose* user Somebody who uses Compose*, this does not exclude the person from being a Compose* developer. This term is used to avoid the ambiguous term Compose* programmer.

Default constructor A constructor without any argument.

Functional complete A set of logical operators is functional complete when it can express any operator from the set [vBKLM91]. So a XOR (exclusive OR) is not in a functional complete set, because $A\ XOR\ B\ =\ (A\ AND\ !B)\ OR\ (!A\ AND\ B)$. The smallest functional complete set only contains OR and NOT.

Language independence Compose* is designed to be used for any language, that gives that Compose* needs to be language independent. This language in dependency influences Compose* in two ways, first on the syntax and semantics, and second on the under laying mechanisms. True language independence means that a concern can be added to an application to a random parent language and it behaves he same as if you would place it in the same application written in a different parent language. For the syntax and semantics this means that we cannot add primitive types to Compose*, because for instance *int*, *char*, and *double*, can have different ranges. These limitations can be solved by automating some conversions, for instance we could introduce our own primitive type i which is an integer of 32 bits, then we can transform this primitive to the primitive value we want by creating a conversion method in Compose*. However for every work around in the language applies that it is just more code and even more important that it solves the problem for most of the

cases, but is would cause errors in a small number of situations. For instance we go further with our primitive i, we find ourselves in the situation that this idea only works if the application stays on one platform and one parent language, if we would import a library then we cannot say for certain to what primitive type we must convert. So if we apply the our own defined primitives we limit the use of the parent language, which is something we do not want.

This gives that there are a few basic comments on language independence which are common for all the parts of the Compose* language:

- Keywords from parent languages have language specific meanings, that means that we cannot use them directly in Compose*.
- The things that are language independent are Objects, Methods, Packages, and Annotations. However these might not be available in every language, for instance C does not know Objects and Annotations are not common for all languages.
- Workarounds with conversion mechanisms might work, but probably it conceptual not sound or you limit yourself in another way.
- If something, for example an integer is 32 bit, in most languages, say Java, C#, C++, Delphi, then it dos not mean that it can be made language independence. There is probably one language around where the integer is not 32 bit.

Signature of a class There are various definitions of the signature of a class, the one used in Compose* is that the signature of a class is the collection of signatures of the methods of the class. The signature of a class is the name, return type, and the list of parameter types. The signature of the class in Listing 1.1 is {*Sting getName, void setBirthDate(Date), void setBirthDate(String)* }.

```
1 public class Person{
2   public String getName(){}
3   public void setBirthDate(Date date){}
4   public void setBirthDate(String date){}
5 }
```

Listing 1.1: Example class in Java

Base language Compose* is an extension to other programming languages, the base language is the language that gets extended. So for Compose*/Java the programming language Java is the parent language.

Wild card The used term for the '*' token, whereas it normally means everything, it means in Compose* ignore.

Chapter 2

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 2.1. 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 that a class is a concern. This means that every class can be written as a concern, with an implementation part and

```
1 Concern ::= 'concern' Identifier ['in' Namespace]
2             '{' (FilterModule)* [SuperImposition] [Implementation] '}'
3 Namespace ::= Identifier ('.' Identifier)*
4 ConcernBlock ::= '{' (FilterModule)* [SuperImposition] [Implementation] '}'
```

Listing 2.1: Concern syntax

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

  filtermodule B{
    ...
  }

  superimposition{
    ...
  }

  implementation{
    ...
  }
}
```

Listing 2.2: Abstract example of a concern

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 2.2.

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 2.1 [BA05]. The usages in this table are all the possible usages of concerns.

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

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 2.1: Different concern usages

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.

Comments on Compose*.Net

Concern Parameters

As mentioned in the Commentary of the concerns the concern parameters are removed from the Compose* syntax. This change only exist in the design and in Compose*/DOTNet they still exist in the syntax. It is however not clear whether they work or not. In Listing 2.3 the concern is taken from the JoinPointInformation example, which can be found in the Compose* example directory. In this concern specification the concern parameters are declared, but never used in a filter module.

Further Reading

In the comments section of this chapter [BA05] is already mentioned, this is a good source for the theory behind concerns.

```
concern JoinPointInfoConcern
    (ext_foo : JPInfo.Foo ; ext_bar : JPInfo.Bar) in JPInfo{

  filtermodule JPInfoModule{
    internals
      in_foo : JPInfo.Foo;
      in_bar : JPInfo.Bar;
      jpie   : JPInfo.JoinPointInfoExtractor;
    conditions
      cond : jpie.doCondition();
    inputfilters
      meta : Meta = { cond => [*.*] jpie.extract }
  }

  superimposition{
    selectors
      selectees = { C | isClassWithName(C, 'JPInfo.JoinPointInfoConcern') };
    filtermodules
      selectees <- JPInfoModule;
  }

  implementation by JPInfo.JoinPointInfoConcern;
}
```

Listing 2.3: JoinPointInfoConcern from the JoinPointInformation example

Chapter 3

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 3.1. The filter module template is demonstrated in the concern template in Listing 3.2. 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 3.1 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.

```
1 FilterModule ::= 'filtermodule' FilterModuleName [FilterModuleParameters] '{'
2                  [Internals] [Externals] [Conditions]
3                  [InputFilters] [OutputFilters] '}';
```

Listing 3.1: Filter module syntax

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

Listing 3.2: Filter module template

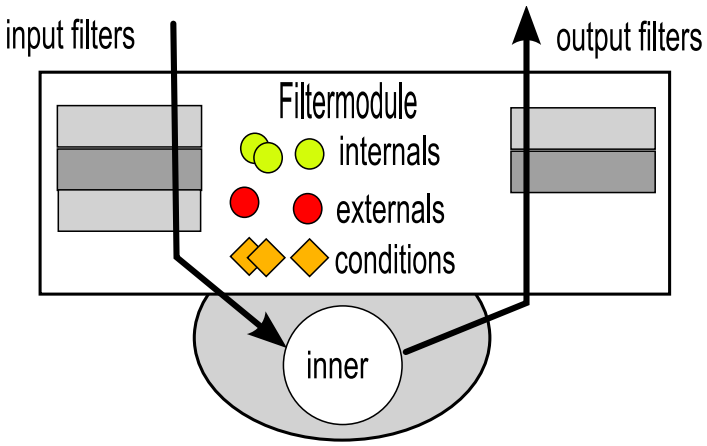

Figure 3.1: Schematic representation of a filter module

```
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 3.3: 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 3.3 and Listing 3.4 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 language would break

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

Listing 3.4: 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 3.5: A filter module with a primitive value

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 3.5 the code with a primitive value is worked out. In line 3 of Listing 3.5 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 3.6 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 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 “>”

[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 3.6: 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 3.7: A parameter as external object

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 [Doo06] 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 3.7. 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 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 3.7. 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.

[1]The selections can have an overlap.

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 4

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 4.1.

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

```
1 FilterModule ::= 'filtermodule' FilterModuleName [FilterModuleParameters] '{'
2                  [Internals] [Externals] [Conditions]
3                  [InputFilters] [OutputFilters] '}'
4 FilterModuleParameters ::= '(' [ParameterDefinition-LIST] ')'
5 ParameterDefinition ::= Parameter | ParameterList
6 Parameter ::= ''?''Identifier
7 ParameterList ::= ''??''Identifier
```

Listing 4.1: 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 4.2: 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 4.2, 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 4.3, 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 4.4 demonstrates a generic encryption filter module with a parameterized filter type and a parameter list for the selector of the matching part.

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;

```
1 filtermodule inherit(?internaltype){
2   internals
3     parent : ?internalType;
4   inputfilters
5     d : Dispatch = {<inner.*> inner.*, <parent.*> *.*}
6 }
```

Listing 4.3: Example of a generic inheritance filter module

- 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 [Doo06];
- 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 4.2, 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 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 4.2, 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.

```
1 filtermodule encryption(??setOfSelectors, ?filtertype){
2   outputfilters
3     e : ?filtertype = {[*.??setOfSelectors]}
4 }
```

Listing 4.4: Example of an encryption filter

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.

Further Reading

The filter module parameters are introduced in [Doo06], here you can find more information on how the syntax was formed and what the other alternatives were.

Chapter 5

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 5.1.

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 5.2 shows the binding of a filter module isActive to classes A and B. What this code does at run time is shown in Figure 5.1, 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`.

```
1 FilterModule ::= 'filtermodule' FilterModuleName
2                  [FilterModuleParameters] '{'
3                  [Internals] [Externals] [Conditions]
4                  [InputFilters] [OutputFilters] '}'
5 Internals ::= 'internals' (Identifier-LIST ':' Type ';')*
```

Listing 5.1: 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 5.2: 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 5.3), 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 5.3: Example 1, a piece of Pacman

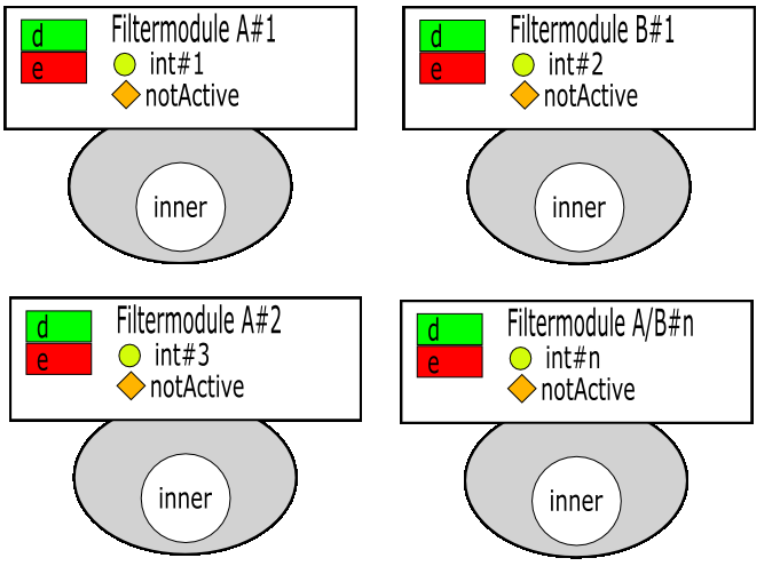

Figure 5.1: Instances of Listing 5.2

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

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

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 3. 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 5.4.

```
1 Internals ::= 'internals' (Identifier-LIST ':' Type
2               ['=' InitialisationExpression '(' [ Argument-LIST ] ')']';')*
```

Listing 5.4: 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 3 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.

Comments on Compose*.Net

If you really want to access an internal then there are two options, first you can access the filter information from the repository, which has at runtime the reference to the internal you want. Second,

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

```
filtermodule returnInternal{
  internals
    state : State;
  inputfilters
    d : Dispatch = {[*.getState] s.getSelf}
}
```

Listing 5.5: How to get hold of an internal

it is possible to write a filter that returns the internal and with the extension of the signature of the superimposed object you can get a *get-method* for the internal, this construction is demonstrated in Listing 5.5. The second option is preferable above the first one, because it is more clear of what you do, only in certain cases where you cannot extend the signature you are stuck with the first option.

Chapter 6

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 6.1, 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 6.2 and look how the instances are created, we get Figure 6.1. Every instance of the filter module `dynamicscoring` gets a reference to the common object Score. If we draw a schema of Listing 6.3 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 there is

```
1 FilterModule ::= 'filtermodule' FilterModuleName [FilterModuleParameters] '{'
2                  [Internals] [Externals] [Conditions]
3                  [InputFilters] [OutputFilters] '}'
4 Externals ::= 'externals' (Identifier ':' Type
5               '=' InitialisationExpression '(' [ ArgumentList ] ')' ';')*
6 InitialisationExpression ::= MethodReference
```

Listing 6.1: Externals syntax

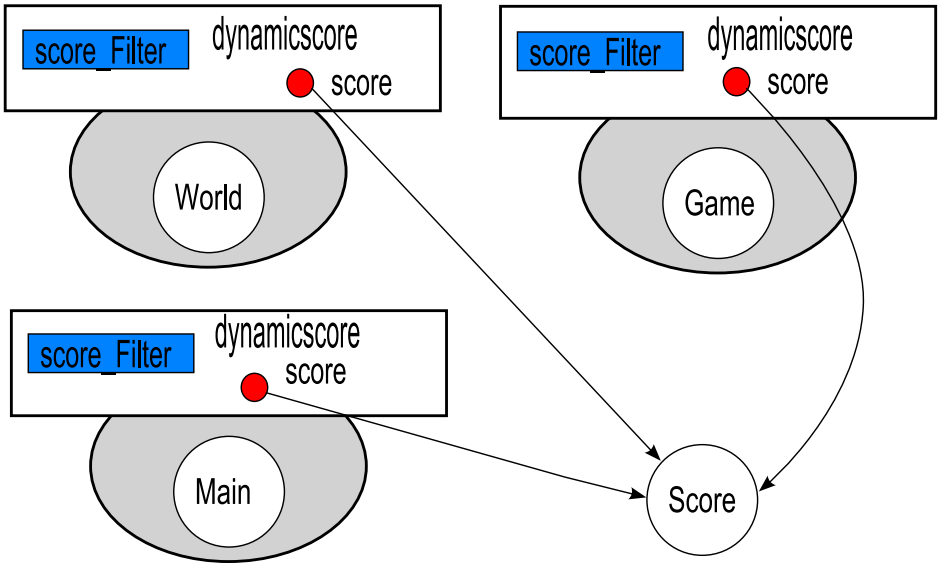

Figure 6.1: Schema of Listing 6.2

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 6.2). 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 6.3, 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, you can use the filter module parameters to define the different instances for the external.

```
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 6.2: 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 6.3: An external without the singleton construction

Commentary

The advantage of using the Singleton pattern becomes visible when we would use a parameter as initialization String. In the given example Listing 6.2, 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 6.3 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 [Doo06] 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.

Comments on Compose*.Net

Arguments in the initialization expression
The possibility to use arguments in the initialization expression is currently not supported by Compose*.Net. This is because the grammar does not support it and it is not certain whether the other parts can handle them. So for now you should try to work around it, it will be checked whether it can be fixed as soon as the filter module parameters are being implemented.

```
filtermodule pending(){
  internals
    i : Distribution;
  externals
    e : Object = i.getSingleton();
}
```

Listing 6.4: A filtermodule with an external from an internal

Pending Questions

Which are declared first, the internals or the externals? To be more concrete, will Listing 6.4 work or not? And does it always work? If the internals are earlier declared then the external, then the example works and there are no further problems. (The given example is probably not the best way to combine internals and externals.)

Chapter 7

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 7.1, 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.

```
41 FilterModule ::= 'filtermodule' FilterModuleName [FilterModuleParameters] '{'
42                  [Internals] [Externals] [Conditions]
43                  [InputFilters] [OutputFilters] '}'
44 Conditions ::= 'conditions' (Identifier ':' MethodReference ';')*
45 MethodReference ::= (FilterModuleElement''.''MethodName | FullyQualifiedName)
46                    '(' ')'
```

Listing 7.1: Conditions syntax

```
47 filtermodule someConditions{
48   internals
49     a : VenusFlyTrapExample.Animal;
50   externals
51     jbframe : JukeboxFrame.JBFrame = JukeboxFrame.JBFrame.instance();
52   conditions
53     isfly : a.hasPrey(); // Venusfly trap
54     isStateChanged : jbframe.isStateChanged(); //Jukebox
55     pacmanIsEvil : pacman.Pacman.isEvil(); //Pacman
56     isFrame : Composestar.Patterns.ChainOfResponsibility.Click.isFrame();
57     // Command of responsibility pattern
58     isBlue : inner.isBlue();
```

Listing 7.2: Some condition declarations combined

Examples

In Listing 7.2 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 [Doo06] 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 7.2 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 8

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 8.1 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 ";".

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

```
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 8.1: 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 8.2: Dynamic strategy filter module in Pacman

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 8.2, 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.

An example on how you can place input and output filters in one filter module is demonstrated in Listing 8.3. 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`.

```
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 8.3: Custom logging filter module

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 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

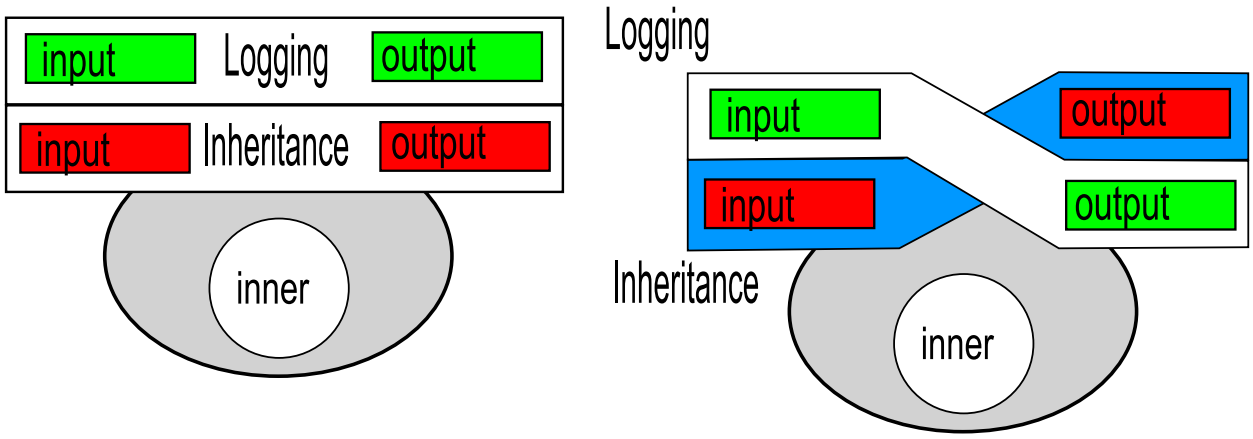

Figure 8.1: Two filter modules with input and output filters

module in Pacman (Listing 8.2) or a generic inheritance filter module (Listing 4.3).

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 8.1. 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 8.4, 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 such a machine with an internal that keeps the state in combination with conditions that handle the state, like Listing 8.5. 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

```
80 inputfilters
81   first : Dispatch = {[*.log]} REPEAT
82   second : Dispatch = {[*.print]}
83 }
```

Listing 8.4: Idea for a filter state machine

```
84 internals
85   state = State;
86 conditions
87   ifRaisedState : state.ifRaisedState();
88 inputfilters
89   first : Before = {[*.log] *.raiseSate};
90   second : Dispatch = {ifRaisedState => [*.print]}
91 }
```

Listing 8.5: State machine with filter operators

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 9

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 9.1, 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 below:

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 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

```
1 Filter ::= FilterName ':' FilterType ['(' ArgumentList ')'] '=' '{' FilterElements '}'
2 FilterType ::= Identifier
```

Listing 9.1: Filter type syntax

```
92 ...
93 inputfilters
94   d : Dispatch = {<inner.*> inner.*};
95   rotate : Rot13 = { [*.ReadLine] };
96   l : Log (?argument) = {[*.log]};
97   e : Error = {[*.*]}
98 }
```

Listing 9.2: Some different filter types

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 9.2 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.

Comments on Compose*.Net

Other predefined filter types have already been described in various publications on Compose*, however these are currently not implemented. This are the *wait* [Ber94] and *real-time* filters which date back before Compose* and the *prepend* and *append* filters [Min06].

Chapter 10

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 10.1, 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 ~>.

```
1 FilterElement = [ORExpression ConditionOperator] MessagePattern
2 ORExpression = ANDExpression ['|' ANDExpression]
3 ANDExpression = NOTExpression ['&' NOTExpression]
4 NOTExpression = [!] (ConditionLiteral | '('ORExpression')')
5 ConditionLiteral = ConditionName | 'True'| 'False'
6 ConditionOperator = '=>'| '~>'
```

Listing 10.1: Filter condition part syntax

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

Listing 10.2: 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 [Com06] we have already seen the filter demonstrated in Listing 10.2. 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 10.3. 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.

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.

```
105 conditions
106   isBlue : inner,isBlue();
107   isMorning : inner.isMorning();
108 inputfilters
109   d1 : Dispatch = {!isBlue => *.*};
110   d2 : Dispatch = {isBlue ~> *.*};
111   d3 : Dispatch = {(isBlue | isMorning) & True => *.*}
```

Listing 10.3: Possible condition parts

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 [vBKLM91]. 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.

Comments on Compose*.Net

Current Implementation
Currently the condition part is not programmed as described in this chapter; the nesting is not in the logical operators yet. Thus `(True & False)& True` will give an error. Because the Compose* grammar is the same for all ports it therefore affects every port.

Chapter 11

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 11.1. 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 target. This means that it is only useful to fill in the target when a signature matching is used. Filling in the

```
1 FilterElement ::= [ConditionExpression ConditionOperator] MessagePattern
2 MessagePattern ::= Matching [SubstitutionPart]
3                  | MatchPattern
4                  | '{' Matching (',' Matching)* '}' [SubstitutionPart]
5 Matching ::= SignatureMatching | NameMatching
6 SignatureMatching ::= '<' MatchPattern '>'
7 NameMatching ::= '[' MatchPattern ']' | Quote MatchPattern Quote
8 MatchPattern ::= [Target '.'] Selector
```

Listing 11.1: Filter matching part syntax

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

Listing 11.2: Some possible matching parts

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 11.2 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 [Doo06] 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.

Comments on Compose*.Net

Whether the mechanism to add the selectors of a name matching to the superimposed object depends on the filter type is still an open question. Practically we can apply the mechanism to all filters, the only conceptual point is whether we should also include it for the error filter. Because this last filter does not add any behavior as in adding methods to a superimposed object, the mechanism must not work for the error filter. In chapter 9 there is an overview of which filter type uses the mechanism and which does not.

Chapter 12

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 12.1. 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.

Examples

In Listing 12.2 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

```
1 FilterElement = [ConditionExpression ConditionOperator] MessagePattern
2 MessagePattern ::= Matching [SubstitutionPart]
3                    | MatchPattern
4                    | '{' Matching (',' Matching)* '}' [SubstitutionPart]
5 Matching ::= SignatureMatching | NameMatchingSubstitutionPart = [Target '.'] Selector
```

Listing 12.1: Filter substitution part syntax

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

Listing 12.2: Some possible substitution parts

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 ([Doo06]) we can alter more attributes of a message.

Chapter 13

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 13.1, shows the syntax of the superimposition part. The result is the skeleton as demonstrated in Listing 13.2.

Semantics

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

```
1 Superimposition ::= 'superimposition' '{' [Selectors]
2                     [FilterModuleBindings] [AnnotationBindings]'}'
```

Listing 13.1: Superimposition syntax

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

Listing 13.2: The superimposition skeleton

Examples

A superimposition can have three blocks and the ordering of these blocks are fixed, the skeleton is demonstrated in Listing 13.2. 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 [Doo06] 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].

Comments on Compose*.Net

Constraints

The earlier mentioned constraint model [Nag06] is not yet implemented in Compose*/.NET. However an earlier model is placed in the syntax, but the semantics are not yet implemented. So the ordering can now be done by providing the ordering in the *filterdesc* XML file of your application, so that (insert module) knows the ordering.

Further Reading

The initial design of the superimposition can be found in [Sal01]. More on the design of the new selector language and the binding of annotations is stated in [Hav05]. The ideas on the constraints model can be found in [Nag06] and the inner working of the filter module ordering in [Dür04]. For the theory behind the removal of the method and condition binding see [Doo06].

Chapter 14

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 Appendix A.

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

```
1 Selectors ::= 'selectors' (SingleSelector)*
2 SingleSelector ::= Identifier '=' '{' PredicateExpression '}' ';'
3 PredicateExpression ::= Identifier PROLOG_EXPRESSION
```

Listing 14.1: 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 14.2: 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 14.3, 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.

Comments on Compose*.Net

Currently the original declaration style of [Sal01] is supported besides the new style with predicate queries [Hav05]. The original selector syntax is still available in Compose*, an example is shown in Listing 14.4. even more important to know is that the "self" construct still works with the old selector and as long this is not rewritten in the syntactic sugar expander, the old style cannot be removed.

```
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 14.3: A selector with extended syntax

```
selectors
  stream = { *=Rot13Filter.MyReader };
```

Listing 14.4: An example of the old style selector

The current style has problems with namespace and concerns with the same name, the old selector does not have this problem. See also the Platypus example.

Further Reading

In [Hav05] the new selector language is introduced. The considerations on the chosen design and the implementation can be found there.

Chapter 15

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 15.1 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.

```
1 FilterModuleBindings ::= ‘filtermodules’ (FilterModuleBinding ‘;’)*
2 FilterModuleBinding ::= Identifier WeaveOperator
3                         (FilterModuleReference-LIST
4                         | ‘{’ FilterModuleReference-LIST ‘}’)
5 FilterModuleReference ::= Identifier [‘(’ Argument-LIST ’)’]
6 AnnotationBindings ::= ‘annotations’ (AnnotationBinding ‘;’)*
7 AnnotationBinding ::= Identifier WeaveOperator
8                         (Identifier-LIST | ‘{’ Identifier-LIST ‘}’)
9 WeaveOperator ::= ‘<-’
```

Listing 15.1: Bindings syntax

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

Listing 15.2: 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 15.2.

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.

Comments on Compose*.Net

Introduction of Parameters

Both constructs mentioned in the comments do not work currently, only literals can be used as arguments for now.

Further Reading

More on filter module binding can be found in [Sal01]. Annotation binding is introduced in [Hav05].

Chapter 16

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 16.1.

Semantics

The implementation part is the base language dependent part of a concern. As we have already seen in chapter 7 is a concern with only an implementation part just a class.

Examples

Listing 16.2 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.

```
1 Implementation ::= 'implementation'('by' ClassName ';'
2                    | 'in' SourceLanguage 'by' ClassName 'as' FileName '{' Source '}'
3 Source ::= (?)*
```

Listing 16.1: 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 16.2: 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.

Comments on Compose*.Net

The code that is placed in this block, is copied to the directory embedded, in the file with the name you have specified.

The syntax cannot handle "♯" in the source language, therefore you need to write sharp full out.

Appendix A

Selector functions

There are 90[1] different selector functions currently defined. They can be divided in two families: the ones derived from program elements and those derived from relations. The different program elements are:

- isNamespace(NS)
- isType(Type)
- isInterface(Interface)
- isClass(Class)
- isMethod(Method)
- isField(Field)
- isParameter(Parameter)
- isAnnotation(Annotation)

A short description of the elements is listed below:

Namespace element
The name of a namespace is its fully qualified name, i.e., the complete path up to this point in the namespace hierarchy, such as *"java.util"* or *"mycompany.myapp.gui"*. Therefore the name of a namespace is unique. Currently a namespace can not have any attributes.

Type element
The name of a type is its fully qualified name, e.g., *"java.util.Hashtable"* or *"mycompany.myapp.gui.SomeWidget"*. Therefore the name of a Type is unique. Types can have a "public" attribute. "Type" is a supertype of Interfaces and Classes.

[1] Each function has the possibility to apply only on one or on a whole list. 45 times 2 is 90.

Interface element
See Type, but matches only Interfaces.

Class element
See Type, but matches only Classes.

Namespace element
See Type.

Method element
Method names are e.g. 'getThis' or 'setThat' - they are not generally unique (not even within a single class). Methods can have the following attributes: public, private, static, final.

Field element
Field names are e.g. 'mCounter', 'dataStore'. Field names are not generally unique, because a field with the same name can occur in different classes. Fields can have the following attributes: public, private, static.

Parameter element Parameters are named like fields. Names are not unique. Parameters do not have attributes.

From these elements the following functions are derived:

- isNamespaceWithName(NS, Name)
- isNamespaceWithAttribute(NS, Attr)
- isTypeWithName(Type, Name)
- isTypeWithAttribute(Type, Attr)
- isInterfaceWithName(Interface, Name)
- isInterfaceWithAttribute(Interface, Attr)
- isClassWithName(Class, Name)
- isClassWithAttribute(Class, Attr)
- isAnnotationWithName(NS, Name)
- isAnnotationWithAttribute(NS, Attr)
- isAnnotationWithName(NS, Name)
- isAnnotationWithAttribute(NS, Attr)
- isMethodWithName(Method, Name)
- isMethodWithAttribute(Method, Attr)

- isFieldWithName(Field, Name)
- isFieldWithAttribute(Field, Attr)
- isParameterWithName(Parameter, Name)
- isParameterWithAttribute(Parameter, Attr)

The relations can be divided into sub groups. This gives the following groups with functions:

Namespace relations

- namespaceHasInterface(Namespace, Interface) - the Namespace contains Interface.
- namespaceHasClass(Namespace, Class) - the Namespace contains Class.

Type relations

- typeHasAnnotation(Type, Annotation) - the Type has Annotation attached.
- isSuperType(SuperType, SubType) - SubType directly inherits from SuperType.

Class relations

- classHasAnnotation(Class, Annotation) - the Class has Annotation attached.
- classHasAnnotationWithName(Class, AnnotName) - the Class has an annotation with the specified name attached.
- classHasMethod(Class, Method) - Class has Method.
- classHasField(Class, Field) - the Class has Field.
- isSuperClass(SuperClass, SubClass) - SubClass directly inherits from SuperClass.
- inherits(Parent, Child) - Child is in the inheritance tree of Parent (e.g. any subclass of Parent).
- inheritsOrSelf(Parent, Child) - Child is in the inheritance tree of Parent or Parent == Child (e.g. Parent and all subclasses).
- implements(Class, Interface) - the Class implements Interface.

Interface relations

- isSuperInterface(SuperInterface, SubInterface) - SubInterface directly inherits from SuperInterface.
- interfaceHasAnnotation(Interface, Annotation) - Interface has Annotation attached.
- interfaceHasAnnotationWithName(Interface, AnnotName) - the Interface has an annotation with the specified name attached.
- interfaceHasMethod(Interface, Method) - Interface declaration contains Method.

Method relations

- methodReturnClass(Method, Class) - Method returns a result of type Class. Don't confuse with classHasMethod!
- methodHasParameter(Method, Parameter) - Method has the specified Parameter. Parameters are not ordered, but can be selected by name of the argument. It is (at least currently) not possible to select e.g. the first argument of a method.
- methodHasAnnotation(Method, Annotation) - Method has Annotation attached.
- methodHasAnnotationWithName(Method, AnnotName) - the Method has an annotation with the specified name attached.

Field relations

- fieldClass(Field, Class) - the class (type) of a field - not to be confused with the Type that the field is a part of (classHasField).
- fieldInterface(Field, Interface) - the interface (type) of a field.
- fieldHasAnnotation(Field, Annotation) - Field has Annotation attached.
- fieldHasAnnotationWithName(Field, AnnotName) - the Field has an annotation with the specified name attached.

Parameter relations

- parameterClass(Parameter, Class) - the class (type) of a parameter.
- parameterHasAnnotation(Parameter, Annotation) - Parameter has Annotation attached.
- parameterHasAnnotationWithName(Parameter, AnnotName) - the Parameter has an annotation with the specified name attached.

This are all the basic functions. It is possible to use all functions with the postfix *InList*. So *isClassWithName* has a similar function *isClassWithNameInList*. This brings the total number of functions on 90.

Appendix B

Message API

When a meta filter is used a method with the argument `Reifiedmessage rd` is being called. This Reifiedmessage object contains the information of where it is sent to and who sent it. it also has some methods to let the message go to its target or to let the message go the the object that sent it.

The explanation of the of the API can be found own the Compose* Wiki at `http://janus.cs.utwente.nl:8000/twiki/bin/view/Composer/ReifiedMessage`. The full API can be found in `http://wwwhome.cs.utwente.nl/~composestar/messageapi/Composestar/Runtime/FLIRT/message/ReifiedMessage.html`, which is part of the message API `http://wwwhome.cs.utwente.nl/~composestar/messageapi/`.

Appendix C

Grammar

This is the full design of the grammar. The current used grammar can differ from this one.

```
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 D

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 [Doo06]. 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 [Doo06].

Syntax

Listing D.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 D.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 D.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 8.2 and Listing 8.3, these can be rewritten to the new syntax. The result for Pacman is demonstrated in Listing D.2 and the other is demonstrated in Listing D.3. Listing D.4 is an example from [Doo06].

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
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 D.2 to Listing D.5. However we get again

```
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 D.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 D.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 D.5: Dynamic strategy filter module in Pacman, with filter operators

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

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

stuck on the parenthesis and this example could be better solved by allowing filters like Listing D.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.

Further Reading

For the full proposal on the new filter element syntax see [Doo06].

Bibliography

[Ada95] Ada reference manual, 1995.

[BA05] Lodewijk Bergmans and Mehmet Akşit. Principles and design rationale of composition filters. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 63–95. Addison-Wesley, Boston, 2005.

[Ber94] L. Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994.

[Com06] Compose* homepage, 2006.

[Doo06] Dirk Doornenbal. Analysis and redesign of the Compose* language. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

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

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison Wesley, 1995.

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

[Log05] LogicAJ homepage, 2005. retrieved on 13 March 2006.

[Min06] Wim Minnen. Design of new filter types for data abstraction. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

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

[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.

[vBKLM91] J.F.A.K. van Benthem, J. Ketting, J.S. Lodder, and W.P.M. Meyer-Viol. *Logica voor informatica*. Pearson, 1991. 3rd edition November 2003, ISBN : 90-430-0722-6, (Dutch).