

Bugspray for composition filters

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

Ing. Rolf Huisman

Enschede, February 26, 2007

Graduation commission:
Prof. dr. ir. M. Akşit
Dr. ir. L.M.J. Bergmans
Ir. ing. P. Dürr

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



Abstract

The behavior of computer programs can be hard to understand because of the execution complexity of a program. Because of the composition of programs with composition filters and/or other aspect oriented languages the execution in computer programs can be even more complex and therefore even more difficult to understand. Besides the increased difficulty we also found that the conventional debugging tools and techniques are less suitable for use within aspect oriented programming. This report will concentrate on improving the representation, editing, compiling, and debugging of composition filter programs while remaining extensible and programming language independent. The primary focus will be on the debugging of composition filter programs within the Compose★ framework.

We approached the breakpoint issue by allowing the programmer to set breakpoints on the behavior of a program, to allow the programmer to see how the execution of the implementation behaves. This behavior of the program can be specified with use of the proposed LTL breakpoints.

We approached the comprehensibility problem by a proposed Composition Filter representation. Because of the sound conceptual model of Composition Filters, if the side effects of a filter are limited to a message, the behavior becomes stutter equivalent. This stutter equivalent behavior can be used to reduce validations and allows a representation of a complete filters evaluation at a joinpoint.

Acknowledgements

I like to thank Dr. Ir. L. Bergmans, Ir. Pascal Dürr, and Prof. Dr. Ir. Mehmet Akşit for supervising and allowing me to complete this master project.

Ir. Wilke Havinga, Dr. István Nagy, Ir. Tom Staijen, and M.Sc. Gurcan Gulesir for assisting and their input in writing this thesis and other related work.

Sverre Boschman, Dennis Spenkel, Roy Spenkel, Michiel van Oudheusden, Olaf Conradi, Johan te Winkel, Ing. Stephan Huttenhuis, Ing. Dirk Doornenbal, Arjan de Roo, Wim Minnen, Johan Wiskerke, Frans Overbeek, and the rest of the graduation crew for their assisting, helping spirits, opinions, and reading of the concepts.

My thanks also go out to Scot Guthrie and Mike Stall for information about visual studio and debugging within the CLR.

And everyone who I forgot to mention here, but helped in any way.

Rolf Huisman

`r.l.r.huisman@student.utwente.nl`

February 26th, 2007

Enschede, The Netherlands

Contents

1	Introduction to AOSD	1
1.1	Introduction	1
1.2	Traditional Approach	3
1.3	AOP Approach	4
1.3.1	AOP Composition	5
1.3.2	Aspect Weaving	6
1.4	AOP Solutions	8
1.4.1	AspectJ Approach	8
1.4.2	Hyperspaces Approach	10
1.4.3	Composition Filters	12
2	Compose[★]	13
2.1	Evolution of Composition Filters	13
2.2	Composition Filters in Compose [★]	14
2.3	Demonstrating Example	17
2.3.1	Initial Object-Oriented Design	17
2.3.2	Completing the Pacman Example	19
2.4	Compose [★] Architecture	20

2.4.1	Integrated Development Environment	20
2.4.2	Compile Time	23
2.4.3	Adaptation	23
2.4.4	Runtime	23
2.5	Platforms	24
2.6	Features Specific to Compose*	24
3	Introduction to the .NET Framework	27
3.1	Introduction	27
3.2	Architecture of the .NET Framework	28
3.2.1	Version 2.0 of .NET	30
3.3	Common Language Runtime	30
3.3.1	Java VM vs .NET CLR	31
3.4	Common Language Infrastructure	32
3.5	Framework Class Library	33
3.6	Common Intermediate Language	34
4	Problem Statement	39
4.1	Bug Anatomy	39
4.2	Difficulties in debugging software programs	40
4.2.1	Breakpoints	43
4.3	Added difficulties in debugging AOP Programs	45
5	Conceptual Solution	47
5.1	Execution behavior	48
5.2	Breakpoints	50
5.3	Behavior prediction	51

6	Conceptual Design	53
6.1	LTL Propositions	53
6.1.1	Concerns	54
6.1.2	Superimposition	55
6.1.3	FilterModules	55
6.1.4	Filters	55
6.1.5	Message	56
6.1.6	Operations	57
6.1.7	Language dependent code	57
6.2	Filter Representation	58
6.2.1	8Ball representation	58
6.2.2	3DBox representation	59
6.2.3	Source iterator representation	60
6.2.4	Visual Composition Filters representation	61
6.2.5	Tree representation	62
6.2.6	Vortex representation	63
6.2.7	Message box representation	64
6.2.8	Action list representation	64
6.2.9	Face the music representation	65
6.2.10	Conclusion	66
6.3	Break Navigation	67
6.3.1	Continue	67
6.3.2	Step Over	68
6.3.3	Step Into	68
6.3.4	Step Outside	69
6.4	Debugging Example	69

6.4.1	Non-Compositional Filter Behavior	69
6.4.2	Compositional Filter Behavior	70
6.4.3	Origin Of The Divorce Function Call	70
6.4.4	Arranging a marriage license	72
7	Implementation Design	75
7.1	Business Context	75
7.2	Functional Requirements	76
7.3	Project Classification	77
7.3.1	Interaction with the environment	77
7.3.2	Termination of the process	77
7.3.3	Interrupt driven	77
7.3.4	State-dependent response	78
7.3.5	Environment-oriented response	78
7.3.6	Parallel processes	78
7.3.7	Real-Time constraints	78
7.3.8	Conclusion	79
7.4	Technical Requirements	79
7.5	Information Analysis	79
7.5.1	Step 1: Assumptions	80
7.5.2	Step 2: Describing	80
7.5.3	Step 3: Generalization	80
7.5.4	Step 4: Cardinality	81
7.5.5	Step 5: Transformations	83
7.6	Use Cases	84
7.6.1	Programmers perspective	84

7.6.2	Program perspective	85
7.7	Architecture	86
7.8	Sequences	87
7.8.1	Define LTL Breakpoint	87
7.8.2	Program actions	88
7.8.3	Continue, Step Over, Step Into, Step Out	88
7.9	Component Design	89
7.9.1	Subject	89
7.9.2	Runtime	89
7.9.3	Profiler	90
7.9.4	Symbol manager	90
7.9.5	Design Time	91
7.9.6	The Publisher	92
8	Conclusion and Future Work	93
8.1	Related work	93
8.1.1	Aspect Slicing	93
8.1.2	NAspect	94
8.1.3	SPIN	94
8.1.4	AspectJ Debugger	95
8.1.5	Control-flow Breakpoints	95
8.2	Conclusion	95
8.3	Future Research	97
A	Source Code Wedding Example	98
B	Non serious representations	99

Chapter 1

Introduction to AOSD

*“Many people see AOSD as a solution.
Others see it as a research subject.”*

Rolf Huisman

The first two chapters have originally been written by seven M. Sc. students [Hol04; Dür04; Vin04; Bos04; Sta05; Hav05; Bos06] at the University of Twente. The chapters have been rewritten for use in the following theses [vO06; Con06; tW06; Hut06; Doo06; Hui07; Spe06]. They serve as a general introduction into Aspect-Oriented Software Development and Compose* in particular.

1.1 Introduction

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

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

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

Functional languages try to solve problems without resorting to variables. These lan-

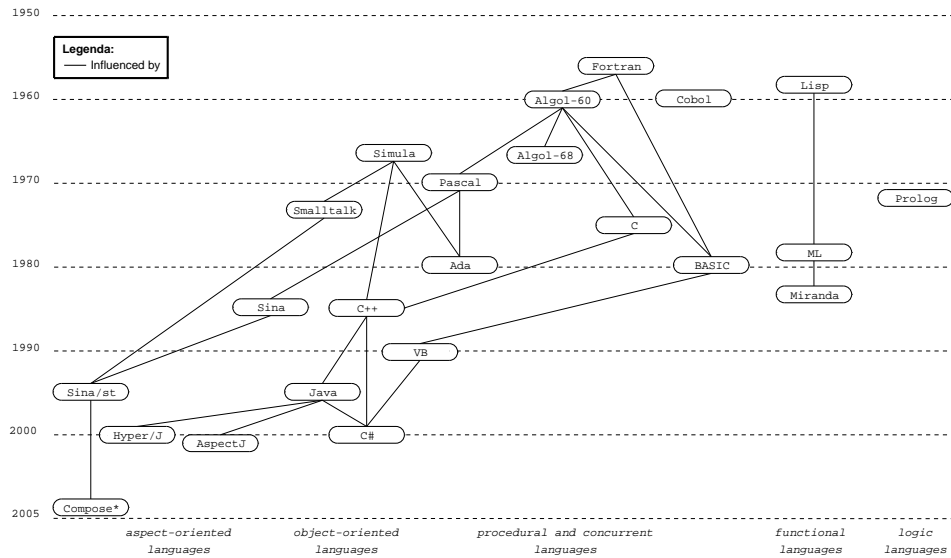


Figure 1.1: Dates and ancestry of several important languages

languages are entirely based on functions over lists and trees. Lisp and Miranda are examples of functional languages.

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

A shortcoming of procedural programming is that global variables can potentially be accessed and updated by any part of the program. This can result in unmanageable programs because no module that accesses a global variable can be understood independently from other modules that also access that global variable.

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

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

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

```

1 public class Add extends Calculation{
2
3     private int result;
4     private CalcDisplay calcDisplay;
5     private Tracer trace;
6
7     Add() {
8         result = 0;
9         calcDisplay = new CalcDisplay();
10        trace = new Tracer();
11    }
12
13    public void execute(int a, int b) {
14        trace.write("void Add.execute(int, int
15        );");
16        result = a + b;
17        calcDisplay.update(result);
18    }
19
20    public int getLastResult() {
21        trace.write("int Add.getLastResult()");
22        ;
23        return result;
24    }
25 }

```

```

1 public class CalcDisplay {
2     private Tracer trace;
3
4     public CalcDisplay() {
5         trace = new Tracer();
6     }
7
8     public void update(int value) {
9         trace.write("void CalcDisplay.update(
10        int)");
11        System.out.println("Printing new value
12        of calculation: "+value);
13    }
14 }

```

(a) Addition

(b) CalcDisplay

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

Aspect-Oriented Programming (AOP) is a paradigm that solves this problem.

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

1.2 Traditional Approach

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

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

The addition and display concerns are implemented in classes `Add` and `CalcDisplay` respectively. Tracing is implemented in the class `Tracer`, but also contains code in the other two classes (lines 5, 10, 14, and 20 in (a) and 2, 5, and 9 in (b)). If a concern is implemented across several classes it is said to be scattered. In the example of [Listing 1.1](#) the tracing concern is scattered.

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

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

Code is difficult to change

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

Code is harder to reuse

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

Code is harder to understand

Tangled code makes it difficult to see which code belongs to which concern.

1.3 AOP Approach

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

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


```

1 public class Add extends Calculation{
2     private int result;
3     private CalcDisplay calcDisplay;
4
5     Add() {
6         result = 0;
7         calcDisplay = new CalcDisplay();
8     }
9
10    public void execute(int a, int b) {
11        result = a + b;
12        calcDisplay.update(result);
13    }
14
15    public int getLastResult() {
16        return result;
17    }
18 }

```

```

1 aspect Tracing {
2     Tracer trace = new Tracer();
3
4     pointcut tracedCalls():
5         call(* (Calculation+).*(..)) ||
6         call(* CalcDisplay.*(..));
7
8     before(): tracedCalls() {
9         trace.write(thisJoinPoint.getSignature()
10                    .toString());
11 }

```

(a) Addition concern

(b) Tracing concern

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

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

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

Aspect code can be changed

Changing aspect code does not influence other concerns;

Aspect code can be reused

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

Aspect code is easier to understand

A concern can be understood independent of other concerns;

Aspect pluggability

Enabling or disabling concerns becomes possible.

1.3.1 AOP Composition

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

In the asymmetric approach, the base program and aspects are distinguished. The base

program is composed with the aspects. This approach is followed by e.g. AspectJ (covered in more detail in the next section).

1.3.2 Aspect Weaving

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

1.3.2.1 Source Code Weaving

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

The advantages of using source code weaving are:

High-level source modification

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

Aspect and original source optimization

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

Native compiler portability

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

However, the drawbacks of source code weaving are:

Language dependency

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

Limited expressiveness

Aspects are limited to the expressive power of the source language. For example,

when using source code weaving, it is not possible to add multiple inheritance to a single inheritance language.

1.3.2.2 Intermediate Language Weaving

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

Programming language independence

All compilers generating the target IL output can be used;

More expressiveness

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

Source code independence

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

Adding aspects at load- or runtime

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

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

Hard to understand

Specific knowledge about the IL is needed;

More error-prone

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

1.3.2.3 Adapting the Virtual Machine

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

Modifying the virtual machine also has its disadvantages:

Dependency on adapted virtual machines

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

Virtual machine optimization

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

1.4 AOP Solutions

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

How aspects are specified

Each technique uses its own aspect language to describe the concerns;

Composition mechanism

Each technique provides its own composition mechanisms;

Implementation mechanism

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

Use of decoupling

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

Supported software processes

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

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

1.4.1 AspectJ Approach

AspectJ [KHH⁺01] is an aspect-oriented extension to the Java programming language. It is probably the most popular approach to AOP at the moment, and it is finding its way into the industrial software development. AspectJ has been developed by Gregor Kiczales at Xerox's PARC (Palo Alto Research Center). To encourage the growth of the AspectJ technology and community, PARC transferred AspectJ to an open Eclipse project. The popularity of AspectJ comes partly from the various extensions based on

```
1 aspect DynamicCrosscuttingExample {
2     Log log = new Log();
3
4     pointcut traceMethods():
5         execution(edu.utwente.trese.**(..));
6
7     before() : traceMethods {
8         log.write("Entering " + thisJointPoint.getSignature());
9     }
10
11    after() : traceMethods {
12        log.write("Exiting " + thisJointPoint.getSignature());
13    }
14 }
```

Listing 1.3: Example of dynamic crosscutting in AspectJ

it, build by several research groups. There are various projects that are porting AspectJ to other languages, resulting in tools such as AspectR and AspectC.

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

Upward compatibility

All legal Java programs must be legal AspectJ programs;

Platform compatibility

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

Tool compatibility

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

Programmer compatibility

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

AspectJ extends Java with support for two kinds of crosscutting functionality. The first allows defining additional behavior to run at certain well-defined points in the execution of the program and is called the *dynamic crosscutting mechanism*. The other is called the *static crosscutting mechanism* and allows modifying the static structure of classes (methods and relationships between classes). The units of crosscutting implementation are called aspects. An example of an aspect specified in AspectJ is shown in Listing 1.3.

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

The code that should execute at a given joinpoint is declared in an advice. Advice is a method-like code body associated with a certain pointcut. AspectJ supports *before*,

```
1 aspect StaticCrosscuttingExample {
2     private int Log.trace(String traceMsg) {
3         Log.write(" --- MARK --- " + traceMsg);
4     }
5 }
```

Listing 1.4: Example of static crosscutting in AspectJ

after and *around* advice that specifies where the additional code is to be inserted. In the example both before and after advice are declared to run at the joinpoints specified by the `traceMethods` pointcut.

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

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

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

1.4.2 Hyperspaces Approach

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

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

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

```

1 Hyperspace Pacman
2   class edu.utwente.trese.pacman.*;

```

Listing 1.5: Creation of a hyperspace

```

1 package edu.utwente.trese.pacman: Feature.Kernel
2 operation trace: Feature.Logging
3 operation debug: Feature.Debugging

```

Listing 1.6: Specification of concern mappings

```

1 hypermodule Pacman_Without_Debugging
2   hyperslices: Feature.Kernel, Feature.Logging;
3   relationships: mergeByName;
4 end hypermodule;

```

Listing 1.7: Defining a hypermodule

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

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

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

Hypermodules are based on concerns and consist of two parts. The first part specifies a set of hyperslices in terms of the concerns identified in the concern matrix. The second part specifies the integration relationships between the hyperslices. A hyperspace can contain several hypermodules realizing different modularizations of the same units. Systems can be composed in many ways from these hypermodules.

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

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

1.4.3 Composition Filters

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

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

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

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

Chapter 2

Compose★

*“The difficult part of composition filters
is understanding its simplicity.”*

Lodewijk Bergmans

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

2.1 Evolution of Composition Filters

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

- 1985** The first version of Sina is developed by Mehmet Akşit. This version of Sina contains a preliminary version of the composition filters concept called semantic networks. The semantic network construction serves as an extension to objects, such as classes, messages, or instances. These objects can be configured to form other objects such as classes from which instances can be created. The object manager takes care of synchronization and message processing of an object. The semantic network construction can express key concepts like delegation, reflection, and synchronization [Koo95].
- 1987** Together with Anand Tripathi of the University of Minnesota the Sina language is further developed. The semantic network approach is replaced by

- declarative specifications and the interface predicate construct is added.
- 1991** The interface predicates are replaced by the dispatch filter, and the wait filter manages the synchronization functions of the object manager. Message reflection and real-time specifications are handled by the meta filter and the real-time filter [Ber94].
 - 1995** The Sina language with Composition Filters is implemented using Smalltalk [Koo95]. The implementation supports most of the filter types. In the same year, a preprocessor providing C++ with support for Composition Filters is implemented [Gla95].
 - 1999** The composition filters language ComposeJ [Wic99] is developed and implemented. The implementation consists of a preprocessor capable of translating composition filter specifications into the Java language.
 - 2001** ConcernJ is implemented as part of a M. Sc. thesis [Sal01]. ConcernJ adds the notion of superimposition to Composition Filters. This allows for reuse of the filter modules and to facilitate crosscutting concerns.
 - 2003** The start of the Compose★ project, the project is described in further detail in this chapter.
 - 2004** The first release of Compose★, based on .NET.
 - 2005** The start of the Java port of Compose★.
 - 2006** Porting Compose★ to C is started.

2.2 Composition Filters in Compose★

```

1 concern {
2   filtermodule {
3     internals
4     externals
5     conditions
6     inputfilters
7     outputfilters
8   }
9
10  superimposition {
11    selectors
12    filtermodules
13    annotations
14    constraints
15  }
16
17  implementation
18 }

```

Listing 2.1: Abstract concern template

A Compose★ application consists of concerns that can be divided in three parts: filter module specifications, superimposition, and implementation. A filter module contains the filter logic to filter on incoming or outgoing messages on superimposed objects. Messages have a target, which is an object reference, and a selector, which is a method

name. A superimposition part specifies which filter modules, annotations, conditions, and methods are superimposed on which objects. An implementation part contains the class implementation of a concern. How these parts are placed in a concern is shown in Listing 2.1.

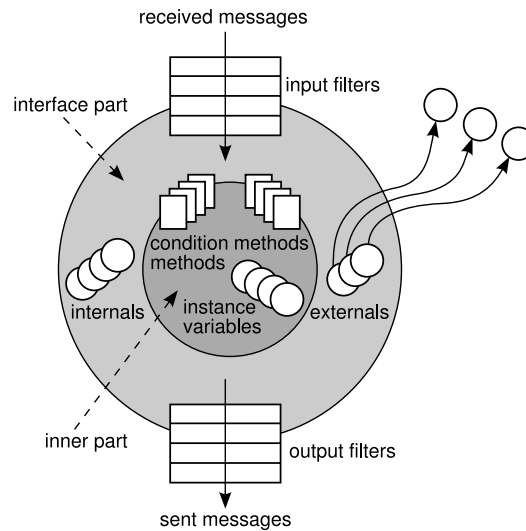


Figure 2.1: Components of the composition filters model

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

$$\begin{array}{c}
 \overbrace{\text{stalker_filter}}^{\text{identifier}} : \overbrace{\text{Dispatch}}^{\text{filter type}} = \overbrace{\{\text{!pacmanIsEvil}\}}^{\text{condition part}} => \\
 \underbrace{[*.\text{getNextMove}]}_{\text{matching part}} \quad \underbrace{\text{stalk_strategy.getNextMove}}_{\text{substitution part}} \}
 \end{array}$$

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

At the moment there are four basic filter types defined in Compose★. It is, however, possible to write custom filter types.

- Dispatch** If the message is accepted, it is dispatched to the specified target of the message, otherwise the message continues to the subsequent filter. This filter type can only be used for input filters;
- Send** If the message is accepted, it is sent to the specified target of the message, otherwise the message continues to the subsequent filter. This filter type can only be used for output filters;
- Error** If the filter rejects the message, it raises an exception, otherwise the message continues to the next filter in the set;
- Meta** If the message is accepted, the message is sent as a parameter of another meta message to an internal or external object, otherwise the message just continues to the next filter. The object that receives the meta message can observe and manipulate the message and can re-activate the execution of the message.

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

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

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

2.3 Demonstrating Example

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

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

2.3.1 Initial Object-Oriented Design

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

Game	This class encapsulates the control flow and controls the state of a game;
Ghost	This class is a representation of a ghost chasing pacman. Its main attribute is a property that indicates whether it is scared or not (depending on the evil state of pacman);
GhostView	This class is responsible for painting ghosts;
Glyph	This is the superclass of all mobile objects (pacman and ghosts). It contains common information like direction and speed;
Keyboard	This class accepts all keyboard input and makes it available to pacman;
Main	This is the entry point of a game;
Pacman	This is a representation of the user controlled element in the game. Its main attribute is a property that indicates whether pacman is evil or not;
PacmanView	This class is responsible for painting pacman;
RandomStrategy	By using this strategy, ghosts move in random directions;
View	This class is responsible for painting a maze;
World	This class has all the information about a maze. It knows where the vitamins, mega vitamins and most importantly the walls are. Every class derived from class <code>Glyph</code> checks whether movement in the desired direction is possible.

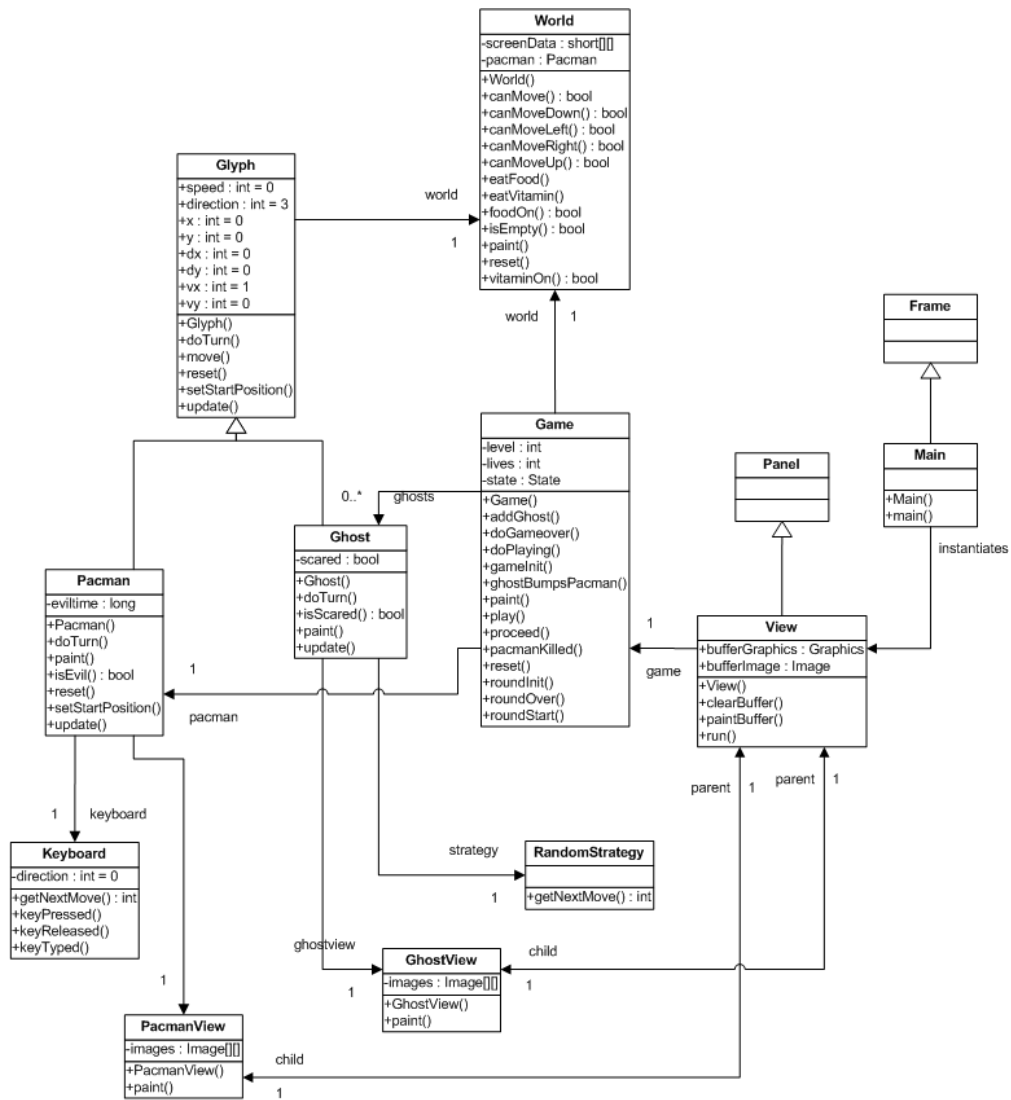


Figure 2.2: Class diagram of the object-oriented Pacman game

```

1 concern DynamicScoring in Pacman {
2   filtermodule dynamicscoring {
3     externals
4     score : pacman.Score = pacman.Score.instance();
5     inputfilters
6     score_filter : Meta = {[*].eatFood] score.eatFood,
7                           [*].eatGhost] score.eatGhost,
8                           [*].eatVitamin] score.eatVitamin,
9                           [*].gameInit] score.initScore,
10                          [*].setForeground] score.setupLabel}
11  }
12  superimposition {
13    selectors
14    scoring = { C | isClassWithNameInList(C, ['pacman.World',
15                                           'pacman.Game', 'pacman.Main']) };
16    filtermodules
17    scoring <- dynamicscoring;
18  }
19 }

```

Listing 2.2: DynamicScoring concern in Compose★

2.3.2 Completing the Pacman Example

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

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

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

2.3.2.1 Implementation of Scoring

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

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

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

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

2.3.2.2 Implementation of Dynamic Strategy

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

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

2.4 Compose★ Architecture

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

2.4.1 Integrated Development Environment

Some of the purposes of the Integrated Development Environment (IDE) layer are to interface with the native IDE and to create a build configuration. In the build configuration it is specified which source files and settings are required to build a Compose★ application. After creating the build configuration the compile time is started.

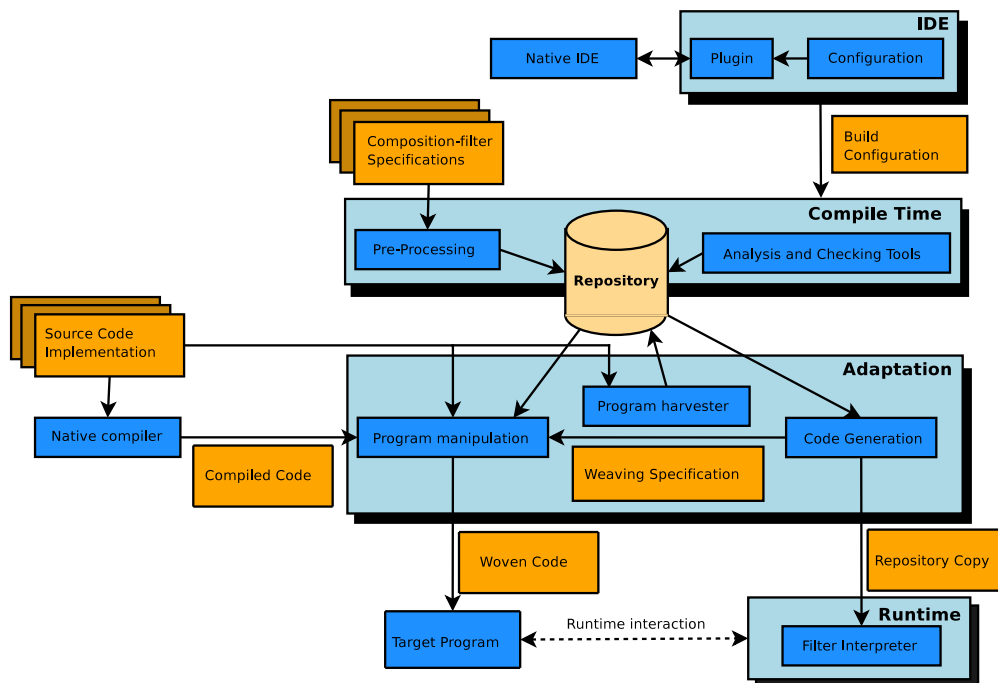

```
1 public class Score
2 {
3     private int score = -100;
4     private static Score theScore = null;
5     private Label label = new java.awt.Label("Score: 0");
6
7     private Score() {}
8
9     public static Score instance() {
10         if(theScore == null) {
11             theScore = new Score();
12         }
13         return theScore;
14     }
15
16     public void initScore(ReifiedMessage rm) {
17         this.score = 0;
18         label.setText("Score: "+score);
19     }
20
21     public void eatGhost(ReifiedMessage rm) {
22         score += 25;
23         label.setText("Score: "+score);
24     }
25
26     public void eatVitamin(ReifiedMessage rm) {
27         score += 15;
28         label.setText("Score: "+score);
29     }
30
31     public void eatFood(ReifiedMessage rm) {
32         score += 5;
33         label.setText("Score: "+score);
34     }
35
36     public void setupLabel(ReifiedMessage rm) {
37         rm.proceed();
38         label = new Label("Score: 0");
39         label.setSize(15*View.BLOCKSIZE+20,15*View.BLOCKSIZE);
40         Main main = (Main)Composestar.Runtime.FLIRT.message.MessageInfo.
41             getMessageInfo().getTarget();
42         main.add(label, BorderLayout.SOUTH);
43     }
44 }
```

Listing 2.3: Implementation of class Score

```

1 concern DynamicStrategy in Pacman {
2   filtermodule dynamicstrategy {
3     internals
4     stalk_strategy : pacman.Strategies.StalkerStrategy;
5     flee_strategy : pacman.Strategies.FleeStrategy;
6     conditions
7     pacmanIsEvil : pacman.Pacman.isEvil();
8     inputfilters
9     stalker_filter : Dispatch = {!pacmanIsEvil =>
10      [*getNextMove] stalk_strategy.getNextMove};
11    flee_filter : Dispatch = {
12      [*getNextMove] flee_strategy.getNextMove}
13  }
14  superimposition {
15    selectors
16    random = { C | isClassWithName(C,
17      'pacman.Strategies.RandomStrategy') };
18    filtermodules
19    random <- dynamicstrategy;
20  }
21 }

```

Listing 2.4: DynamicStrategy concern in Compose[★]Figure 2.3: Overview of the Compose[★] architecture

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

2.4.2 Compile Time

The compile time layer is platform independent and reasons about the correctness of the composition filter implementation with respect to the program which allows the target program to be build by the adaptation.

The compile time ‘pre-processes’ the composition filter specifications by parsing the specification, resolving the references, and checking its consistency. To provide an extensible architecture to facilitate this process a blackboard architecture is chosen. This means that the compile time uses a general knowledgebase that is called the ‘repository’. This knowledgebase contains the structure and metadata of the program which different modules can execute their activities on. Examples of modules within analysis and validation are the three modules SANE, LOLA and FILTH. These three modules are responsible for (some) of the analysis and validation of the super imposition and its selectors.

2.4.3 Adaptation

The adaptation layer consists of the program manipulation, harvester, and code generator. These components connect the platform independent compile time to the target platform. The harvester is responsible for gathering the structure and the annotations within the source program and adding this information to the knowledgebase. The code generation generates a reduced copy of the knowledgebase and the weaving specification. This weaving specification is then used by the weaver contained by the program manipulation to weave in the calls to the runtime into the target program. The end result of the adaptation the target program which interfaces wit the runtime.

2.4.4 Runtime

The runtime layer is responsible for executing the concern code at the joinpoints. It is activated at the joinpoints by function calls that are woven in by the weaver. A reduced copy of the knowledgebase containing the necessary information for filter evaluation and execution is enclosed with the runtime. When the function is filtered the filter is evaluated. Depending on if the the condition part evaluates to true, and the matching part matches the accept or reject behavior of the filter is executed. The runtime also facilitates the debugging of the composition filter implementations.

2.5 Platforms

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

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

2.6 Features Specific to Compose★

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

Ordering of filter modules

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

Filter consistency checking

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

Reason about semantic problems

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

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

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

Integrated Development Environment support

The Compose★ implementations all have a IDE plug-in; Compose★/.NET for Visual Studio, Compose★/J and Compose★/C for Eclipse;

Debugging support

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

Incremental building process

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

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

Language independent concerns

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

Reusable concerns

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

Expressive selector language

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

Support for annotations

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

Chapter 3

Introduction to the .NET Framework

*“The best way to prepare [to be a programmer] is to write programs,
and to study great programs that other people have written.
In my case, I went to the garbage cans at the Computer Science Center
and fished out listings of their operating system.”*
William Henry Gates III

This chapter gives an introduction to the .NET Framework of Microsoft. First, the architecture of the .NET Framework is introduced. This section includes terms like the Common Language Runtime, the .NET Class Library, the Common Language Infrastructure and the Intermediate Language. These are discussed in more detail in the sections following the architecture.

3.1 Introduction

Microsoft defines [Mic05] .NET as follows; “.NET is the Microsoft Web services strategy to connect information, people, systems, and devices through software.”. There are different .NET technologies in various Microsoft products providing the capabilities to create solutions using web services. Web services are small, reusable applications that help computers from many different operating system platforms work together by exchanging messages. Based on industry standards like XML (Extensible Markup Language), SOAP (Simple Object Access Protocol), and WSDL (Web Services Description Language) they provide a platform and language independent way to communicate.

Microsoft products, such as Windows Server System (providing web services) or Office System (using web services) are some of the .NET technologies. The technology described in this chapter is the .NET Framework. Together with Visual Studio, an inte-

grated development environment, they provide the developer tools to create programs for .NET.

Many companies are largely dependent on the .NET Framework, but need or want to use AOP. Currently there is no direct support for this in the Framework. The Compose★/.NET project is addressing these needs with its implementation of the Composition Filters approach for the .NET Framework.

This specific Compose★ version for .NET has two main goals. First, it combines the .NET Framework with AOP through Composition Filters. Second, Compose★ offers superimposition in a language independent manner. The .NET Framework supports multiple languages and is, as such, suitable for this purpose. Composition Filters are an extension of the object-oriented mechanism as offered by .NET, hence the implementation is not restricted to any specific object-oriented language.

3.2 Architecture of the .NET Framework

The .NET Framework is Microsoft's platform for building, deploying, and running Web Services and applications. It is designed from scratch and has a consistent API providing support for component-based programs and Internet programming. This new Application Programming Interface (API) has become an integral component of Windows. The .NET Framework was designed to fulfill the following objectives [Mic03b]:

Consistency

Allow object code to be stored and executed locally, executed locally but Internet-distributed, or executed remotely and to make the developer experience consistent across a wide variety of types of applications, such as Windows-based applications and Web-based applications;

Operability

The ease of operation is enhanced by minimizing version conflicts and providing better software deployment support;

Security

All the code is executed safely, including code created by an unknown or semi-trusted third party;

Efficiency

The .NET Framework compiles applications to machine code before running thus eliminating the performance problems of scripted or interpreted environments;

Interoperability

Code based on the .NET Framework can integrate with other code because all communication is built on industry standards.

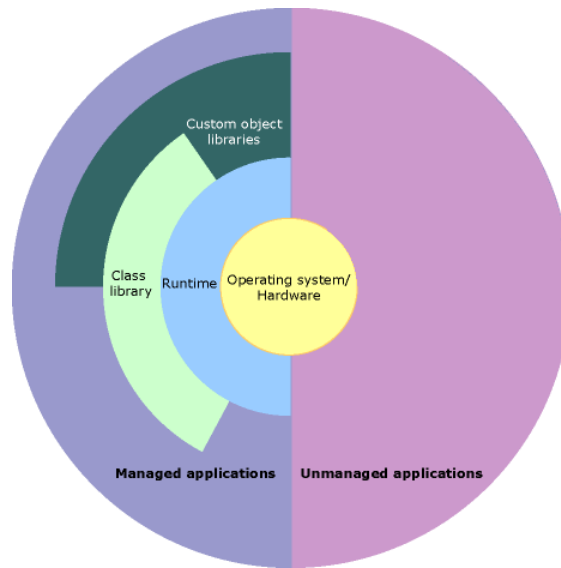


Figure 3.1: Context of the .NET Framework (Modified) [Mic03b]

The .NET Framework consists of two main components [Mic03b]: the Common Language Runtime (CLR, simply called the .NET Runtime or Runtime for short) and the .NET Framework Class Library (FCL). The CLR is the foundation of the .NET Framework, executing the code and providing the core services such as memory management, thread management and exception handling. The CLR is described in more detail in [Section 3.3](#). The class library, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that can be used to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to applications such as Web Forms and XML Web services. [Section 3.5](#) describes the class libraries in more detail.

The code run by the runtime is in a format called Common Intermediate Language (CIL), further explained in [Section 3.6](#). The Common Language Infrastructure (CLI) is an open specification that describes the executable code and runtime environment that form the core of the Microsoft .NET Framework. [Section 3.4](#) tells more about this specification.

[Figure 3.1](#) shows the relationship of the .NET Framework to other applications and to the complete system. The two parts, the class library and the runtime, are managed, i.e., applications managed during execution. The operating system is in the core, managed and unmanaged applications operate on the hardware. The runtime can use other object libraries and the class library, but the other libraries can use the same class library themselves.

Besides the Framework, Microsoft also provides a developer tool called the Visual Studio. This is an IDE with functionality across a wide range of areas allowing developers to build applications with decreased development time in comparison with developing applications using command line compilers.

3.2.1 Version 2.0 of .NET

In November 2005, Microsoft released a successor of the .NET Framework. Major changes are the support for generics, the addition of nullable types, 64 bit support, improvements in the garbage collector, new security features and more network functionality.

Generics make it possible to declare and define classes, structures, interfaces, methods and delegates with unspecified or generic type parameters instead of specific types. When the generic is used, the actual type is specified. This allows for type-safety at compile-time. Without generics, the use of casting or boxing and unboxing decreases performance. By using a generic type, the risks and costs of these operations is reduced.

Nullable types allow a value type to have a normal value or a null value. This null value can be useful for indicating that a variable has no defined value because the information is not currently available.

Besides changes in the Framework, there are also improvements in the four main Microsoft .NET programming languages (C#, VB.NET, J# and C++). The language elements are now almost equal for all languages. For instance, additions to the Visual Basic language are the support for unsigned values and new operators and additions to the C# language include the ability to define anonymous methods thus eliminating the need to create a separate method.

A new Visual Studio 2005 edition was released to support the new Framework and functionalities to create various types of applications.

3.3 Common Language Runtime

The Common Language Runtime executes code and provides core services. These core services are memory management, thread execution, code safety verification and compilation. Apart from providing services, the CLR also enforces code access security and code robustness. Code access security is enforced by providing varying degrees of trust to components, based on a number of factors, e.g., the origin of a component. This way, a managed component might or might not be able to perform sensitive functions, like file-access or registry-access. By implementing a strict type-and-code-verification infrastructure, called the Common Type System (CTS), the CLR enforces code robustness. Basically there are two types of code;

Managed

Managed code is code, which has its memory handled and its types validated at execution by the CLR. It has to conform to the Common Type Specification (CTS [Section 3.4](#)). If interoperability with components written in other languages is required, managed code has to conform to an even more strict set

of specifications, the Common Language Specification (CLS). The code is run by the CLR and is typically stored in an intermediate language format. This platform independent intermediate language is officially known as Common Intermediate Language (CIL Section 3.6) [Wat00].

Unmanaged

Unmanaged code is not managed by the CLR. It is stored in the native machine language and is not run by the runtime but directly by the processor.

All language compilers (targeting the CLR) generate managed code (CIL) that conforms to the CTS.

At runtime, the CLR is responsible for generating platform specific code, which can actually be executed on the target platform. Compiling from CIL to the native machine language of the platform is executed by the just-in-time (JIT) compiler. Because of this language independent layer it allows the development of CLR's for any platform, creating a true interoperability infrastructure [Wat00]. The .NET Runtime from Microsoft is actually a specific CLR implementation for the Windows platform. Microsoft has released the *.NET Compact Framework* especially for devices such as personal digital assistants (PDAs) and mobile phones. The .NET Compact Framework contains a subset of the normal .NET Framework and allows .NET developer to write mobile applications. Components can be exchanged and web services can be used so an easier interoperability between mobile devices and workstations/servers can be implemented [Mic03a].

At the time of writing, the .NET Framework is the only advanced Common Language Infrastructure (CLI) implementation available. A shared-source¹ implementation of the CLI for research and teaching purposes was made available by Microsoft in 2002 under the name Rotor [Stu02]. In 2006 Microsoft released an updated version of Rotor for the .NET platform version two. Also Ximian is working on an open source implementation of the CLI under the name Mono², targeting both Unix/Linux and Windows platforms. Another, somewhat different approach, is called Plataforma.NET³ and aims to be a hardware implementation of the CLR, so that CIL code can be run natively.

3.3.1 Java VM vs .NET CLR

There are many similarities between Java and .NET technology. This is not strange, because both products serve the same market.

Both Java and .NET are based on a runtime environment and an extensive development framework. These development frameworks provide largely the same functionality for both Java and .NET. The most obvious difference between them is lack of language independence in Java. While Java's strategy is 'One language for all platforms' the

¹Only non-commercial purposes are allowed.

²<http://www.go-mono.com/>

³<http://personals.ac.upc.edu/enric/PFC/Plataforma.NET/p.net.html>

.NET philosophy is ‘All languages on one platform’. However these philosophies are not as strict as they seem. As noted in [Section 3.5](#) there is no technical obstacle for other platforms to implement the .NET Framework. There are compilers for non-Java languages like Jython (Python) [Jyt] and WebADA [Ada96] available for the JVM. Thus, the JVM in its current state, has difficulties supporting such a vast array of languages as the CLR. However, the multiple language support in .NET is not optimal and has been the target of some criticism.

Although the JVM and the CLR provide the same basic features they differ in some ways. While both CLR and the modern JVM use JIT (Just In Time) compilation the CLR can directly access native functions. This means that with the JVM an indirect mapping is needed to interface directly with the operating system.

3.4 Common Language Infrastructure

The entire CLI has been documented, standardized and approved [Int02] by the European association for standardizing information and communication systems, Ecma International¹. Benefits of this CLI for developers and end-users are:

- Most high level programming languages can easily be mapped onto the Common Type System (CTS);
- The same application will run on different CLI implementations;
- Cross-programming language integration, if the code strictly conforms to the Common Language Specification (CLS);
- Different CLI implementations can communicate with each other, providing applications with easy cross-platform communication means.

This interoperability and portability is, for instance, achieved by using a standardized meta data and intermediate language (CIL) scheme as the storage and distribution format for applications. In other words, (almost) any programming language can be mapped to CIL, which in turn can be mapped to any native machine language.

The Common Language Specification is a subset of the Common Type System, and defines the basic set of language features that all .NET languages should adhere to. In this way, the CLS helps to enhance and ensure language interoperability by defining a set of features that are available in a wide variety of languages. The CLS was designed to include all the language constructs that are commonly needed by developers (e.g., naming conventions, common primitive types), but no more than most languages are able to support [Mic03c]. [Figure 3.2](#) shows the relationships between the CTS, the CLS, and the types available in C++ and C#. In this way the standardized CLI pro-

¹An European industry association founded in 1961 and dedicated to the standardization of Information and Communication Technology (ICT) Systems. Their website can be found at <http://www.ecma-international.org/>.

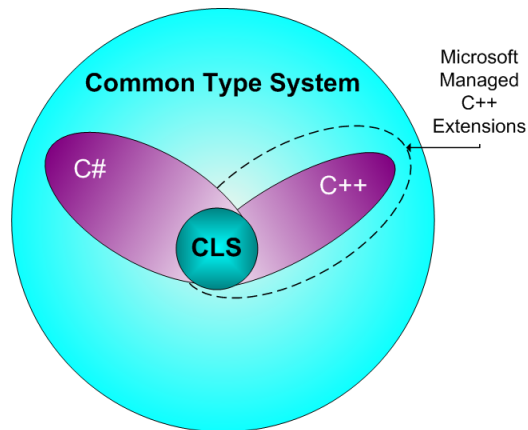


Figure 3.2: Relationships in the CTS

vides, in theory¹, a true cross-language and cross-platform development and runtime environment.

To attract a large number of developers for the .NET Framework, Microsoft has released CIL compilers for C++, C#, J#, and VB.NET. In addition, third-party vendors and open-source projects also released compilers targeting the .NET Framework, such as Delphi.NET, Perl.NET, IronPython, and Eiffel.NET. These programming languages cover a wide-range of different programming paradigms, such as classic imperative, object-oriented, scripting, and declarative languages. This wide coverage demonstrates the power of the standardized CLI.

Figure 3.3 shows the relationships between all the main components of the CLI. The top of the figure shows the different programming languages with compiler support for the CLI. Because the compiled code is stored and distributed in the Common Intermediate Language format, the code can run on any CLR. For cross-language usage this code has to comply with the CLS. Any application can use the class library (the FCL) for common and specialized programming tasks.

3.5 Framework Class Library

The .NET Framework class library is a comprehensive collection of object-oriented reusable types for the CLR. This library is the foundation on which all the .NET applications are built. It is object oriented and provides integration of third-party components with the classes in the .NET Framework. Developers can use components provided by the .NET Framework, other developers and their own components. A wide range of common programming tasks (e.g., string management, data collection, reflection, graphics, database connectivity or file access) can be accomplished easily

¹Unfortunately Microsoft did not submit all the framework classes for approval and at the time of writing only the .NET Framework implementation is stable.

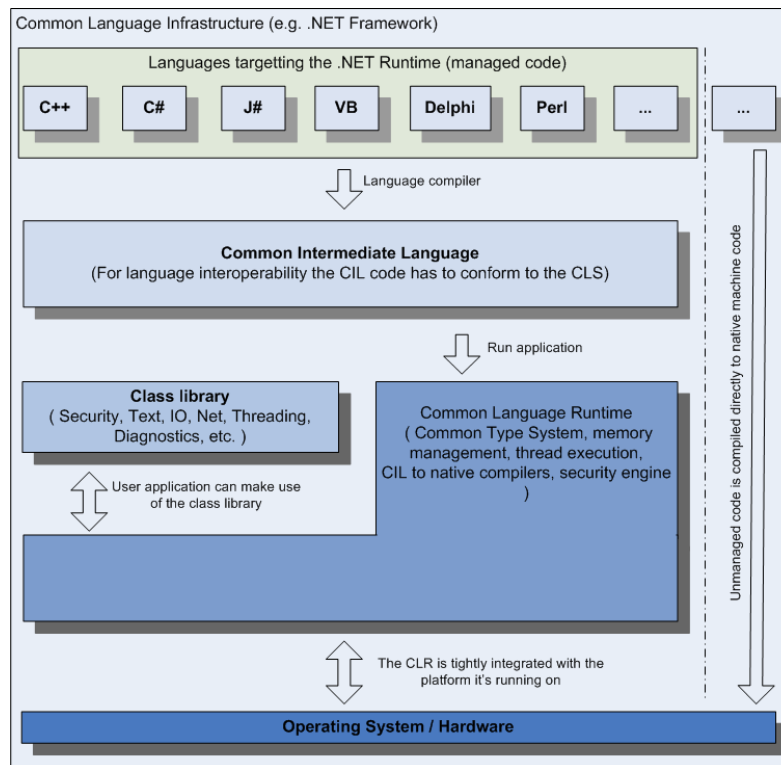


Figure 3.3: Main components of the CLI and their relationships. The right hand side of the figure shows the difference between managed code and unmanaged code.

by using the class library. Also a great number of specialized development tasks are extensively supported, like:

- Console applications;
- Windows GUI applications (Windows Forms);
- Web applications (Web Forms);
- XML Web services;
- Windows services.

All the types in this framework are CLS compliant and can therefore be used from any programming language whose compiler conforms to the Common Language Specification (CLS).

3.6 Common Intermediate Language

The Common Intermediate Language (CIL) has already been mentioned briefly in the sections before, but this section will describe the IL in more detail. All the languages targeting the .NET Framework compile to this CIL (see Figure 3.4).

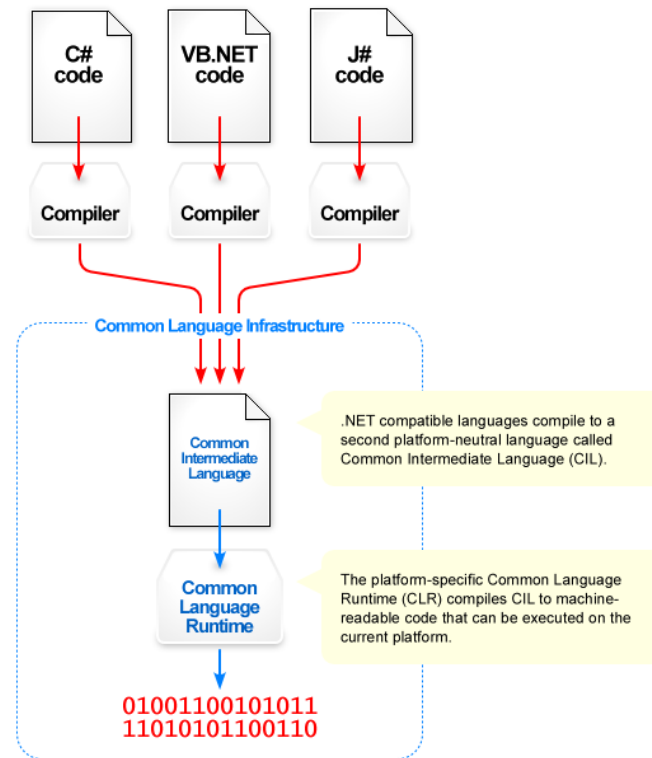


Figure 3.4: From source code to machine code

A .NET compiler generates a *managed module* which is an executable designed to be run by the CLR [Pro02]. There are four main elements inside a managed module:

- A Windows Portable Executable (PE) file header;
- A CLR header containing important information about the module, such as the location of its CIL and metadata;
- Metadata describing everything inside the module and its external dependencies;
- The CIL instructions generated from the source code.

The Portable Executable file header allows the user to start the executable. This small piece of code will initiate the just-in-time compiler which compiles the CIL instructions to native code when needed, while using the metadata for extra information about the program. This native code is machine dependent while the original IL code is still machine independent. This way the same IL code can be JIT-compiled and executed on any supported architecture. The CLR cannot use the managed module directly but needs an assembly.

An assembly is the fundamental unit of security, versioning, and deployment in the .NET Framework and is a collection of one or more files grouped together to form a logical unit [Pro02]. Besides managed modules inside an assembly, it is also possible to include resources like images or text. A manifest file is contained in the assem-

bly describing not only the name, culture and version of the assembly but also the references to other files in the assembly and security requests.

The CIL is an object oriented assembly language with around 100 different instructions called OpCodes. It is stack-based, meaning objects are placed on an evaluation stack before the execution of an operation, and when applicable, the result can be found on the stack after the operation. For instance, when adding two numbers, first those numbers have to be placed onto the stack, second the add operation is called and finally the result can be retrieved from the stack.

```
1  .assembly AddExample {}
2
3  .method static public void main() il managed
4  {
5      .entrypoint           // entry point of the application
6      .maxstack 2
7
8      ldc.i4 3               // Place a 32-bit (i4) 3 onto the stack
9      ldc.i4 7               // Place a 32-bit (i4) 7 onto the stack
10
11     add                    // Add the two and
12                               // leave the sum on the stack
13
14     // Call static System.Console.WriteLine function
15     // (function pops integer from the stack)
16     call void [mscorlib]System.Console::WriteLine(int32)
17
18     ret
19 }
```

Listing 3.1: Adding example in IL code

To illustrate how to create a .NET program in IL code we use the previous example of adding two numbers and show the result. In Listing 3.1 a new assembly is created with the name AddExample. In this assembly a function main is declared as the starting point (entrypoint) of this assembly. The maxstack command indicates there can be a maximum of two objects on the stack and this is enough for the example method. Next, the values 3 and 7 are placed onto the stack. The add operation is called and the results stays on the stack. The method WriteLine from the .NET Framework Class Library is called. This method resides inside the Console class placed in the System assembly. It expects one parameter with a int32 as its type that will be retrieved from the stack. The call operation will transfer the control flow to this method passing along the parameters as objects on the stack. The WriteLine method does not return a value. The ret operation returns the control flow from the main method to the calling method, in this case the runtime. This will exit the program.

To be able to run this example, we need to compile the IL code to bytecode where each OpCode is represented as one byte. To compile this example, save it as a text file and run the ILASM compiler with as parameter the filename. This will produce an executable runnable on all the platforms where the .NET Framework is installed.

This example was written directly in IL code, but we could have used a higher level

language such as C# or VB.NET. For instance, the same example in C# code is shown in Listing 3.2 and the VB.NET version is listed in Listing 3.3. When this code is compiled to IL, it will look like the code in Listing 3.1.

```
1 public static void main()  
2 {  
3     Console.WriteLine((int) (3 + 7));  
4 }
```

Listing 3.2: Adding example in the C# language

```
1 Public Shared Sub main()  
2     Console.WriteLine(CType((3 + 7), Integer))  
3 End Sub
```

Listing 3.3: Adding example in the VB.NET language

Chapter 4

Problem Statement

*“If debugging is the process of removing bugs,
then programming must be the process of putting them in.”
Edsger Dijkstra*

This chapter explains the problems associated with the debugging of Aspect-oriented programs which this thesis addresses. First, the concept of debugging is introduced, which includes terms like bug, fault and failure. Second, the difficulties and approaches in debugging software programs are addressed. Third, the difficulties in debugging Aspect-oriented programs are highlighted. Finally, the problem addressed in this thesis is formulated.

4.1 Bug Anatomy

Computer programs are build by compilers, from source code into a *target program*. This target program is then deployed and executed. Because computer programs are complex, developers make programming errors in the source code, called *bugs*. We distinguish two kinds of bugs; syntactic and semantic.

Syntactic bugs are a violation of the syntax of a language. An example of a syntactic bug in the English language would be;

I are going to my mother.

The *meaning* of such a sentence can be understood by an English speaker, but it is not grammatically correct and thus violates the syntax of the English language. Because computer languages are strict in their syntax, compilers will not compile source files

containing syntactic bugs. This means that detection of syntactic bugs can be done by using the compiler.

A semantic bug is an incorrect or unintended meaning of the specification. An example of a semantic bug in the English language is;

My hamster just drove a car to Jupiter.

While most English speakers would agree that the sentence is syntactical correct, its *meaning* could be questioned. Since a semantic bug does not violate the syntax of a language it can therefore not be found by just validating the syntax. Therefore a more detailed analysis called type checking is commonly used. Type checking validates the type system used in a program. Many semantic bugs however do not invalidate the type system and can therefore not be found using type checking. When a program has a correct syntax and a correct type system, the compiler can not recognize that the source code is incorrect and compiles the source files into a target program.

When a piece of code which contains a semantic bug is executed, it causes an incorrectness in the execution called a *fault* [Org02]. This fault could then cause the target program to misbehave which is then called a *failure* [Org02]. A fault which does not result in the program misbehaving therefore does not lead to a failure. This happens in fault tolerant systems.

Because a failure is unintended behavior caused by an incorrect implementation, a program which behaves correctly does not have failures during execution. Since bugs are the cause of the faults which are responsible for the failures, we want to remove the bugs from the source code to get the intended implementation.

4.2 Difficulties in debugging software programs

The process of removing bugs is called *debugging* which generally consists of five steps [ARF02]:

Recognize the failure

Determine what the misbehavior of the computer program is;

Isolation of the fault

Determine which fault is causing the failure;

Identify the bug

Determine what bug is causing the fault;

Repair the bug

Replacing the incorrect pieces of source code by the correct ones;

Validation

Check if the modification of the source code resulted in the desired behavior and whether it did not result in the same or different misbehavior.

The main difficulty of debugging is the isolation of the fault. One reason of this is the delay between the failure and the causing fault, called the fault delay. Another reason is that the complexity of a program increases when it is executed. An example of this

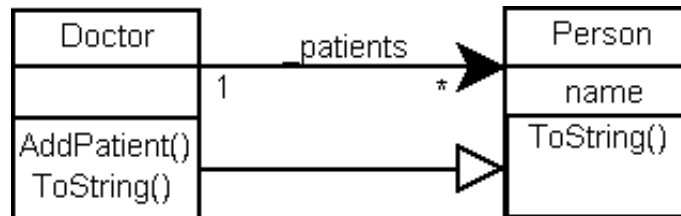


Figure 4.1: UML model of the Doctor example

increase in complexity is the small program which purpose it is to print the relations between doctors and their patients. The model of this program seen in Figure 4.1 is not complex. It consists of two classes representing the doctor and its patient. There is one reference from the doctor to the patient. The implementation of this model can be seen in Listing 4.1.

```

1 public class Person{
2     protected string _name;
3
4     public Person(string name) {
5         this._name = name;
6     }
7
8     public override string ToString() {
9         return "Patient: " + this._name;
10    }
11 }
12
13 public class Doctor : Person{
14     public Doctor(string name) : base(name) {}
15
16     public override string ToString() {
17         string result = "Doctor: "+this._name+"\nPatients:\n";
18         foreach (Person patient in patients){
19             result += patient.ToString() + '\n';
20         }
21         return result;
22     }
23
24     private List<Person> patients = new List<Person>();
25
26     public void addPatient(Person patient) {
27         patients.Add(patient);
28     }
29 }
  
```

Listing 4.1: Implementation of the doctor class in the C# language

When the program is executed by using the bootstrap code as can be seen in Listing 4.2, two objects are created; a doctor called piet and his patient hans.

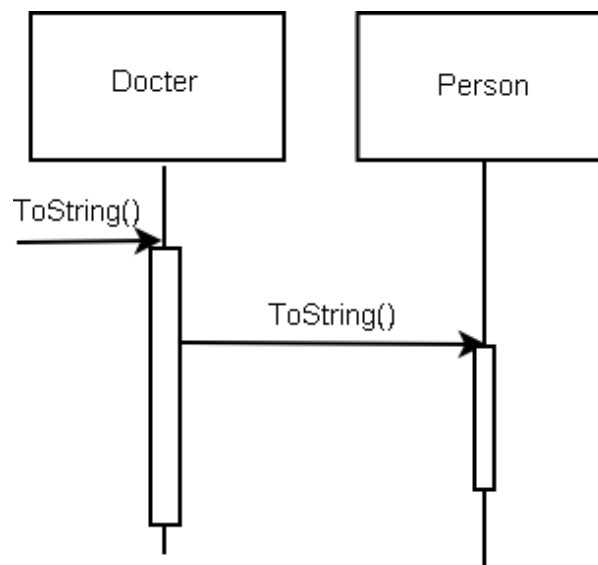
```

1  Doctor piet = new Doctor("Piet");
2  Person hans = new Person("Hans");
3  piet.addPatient(hans);
4  Console.WriteLine(piet.ToString());

```

Listing 4.2: Bootstrap code in the C# language

As can be seen in Figure 4.2, during execution the `ToString` function of the `piet` object will be executed which in its turn calls the `ToString` function of its patient `hans` because of the patient reference.

Figure 4.2: Execution of the `ToString` of Listing 4.2

While this small execution is not that complex, other executions are possible which were not intended. In Listing 4.3 can be seen an example of where `hans` is also a doctor with `piet` as his patient. This situation results in an endless loop as shown in Figure 4.3 and will eventually lead to a stack overflow. This execution is the result of both `piet` and `hans` being doctors which are cross-referenced to each other. This cross-reference makes a call to the method `ToString` of the object `piet` result in a call from object `piet` to the `ToString` method object `hans`. This call to `ToString` method of object `hans` results in a call from the object `hans` back to the method `ToString` of the object `piet` again.

```

1  Doctor piet = new Doctor("Piet");
2  Person hans = new Doctor("Hans");
3  piet.addPatient(hans);
4  hans.addPatient(piet);
5  Console.WriteLine(piet.ToString());

```

Listing 4.3: Bug revealing bootstrap code in the C# language

While the model and the static source file are not so complex, the execution can create complex situations. The more complex the relations are, the more difficult the isolation

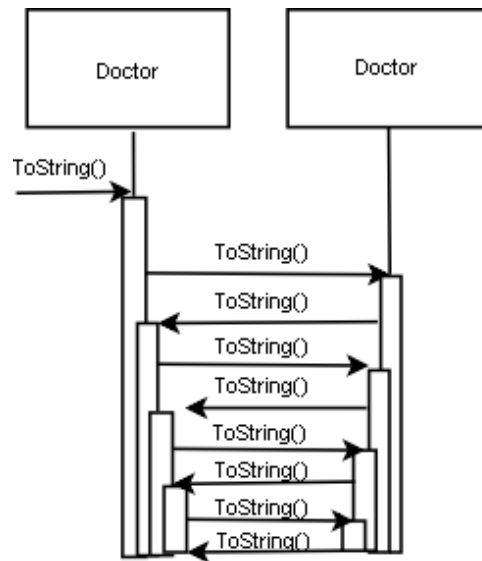


Figure 4.3: Execution of the `ToString` of Listing 4.3

of the fault becomes. In this example the number of instances is lower and the model was smaller compared to a real system which also has references that change during executing and concurrency. In a real system the increase of the complexity at execution is therefore bigger than in this example.

Because most programming languages in use today are Turing complete, the halting theorem implies that it is impossible to find all bugs using automated analysis. Since computer programs are still unable to grasp what we intended and automated analysis are insufficient it is required to provide the developer the insight into the implemented behavior and how that behavior is related to the source code. To achieve this kind of insight a computer program called a *debugger* is used. The most commonly used debuggers today are execution steppers. These allow stepping through the executing operations of the system that is being debugged. This gives the developer insight in the implemented behavior of the system. By using this insight a developer can isolate the fault and then fix the bug.

4.2.1 Breakpoints

Since stepping through all the operations of a program is too much for a programmer, debuggers commonly use breakpoints. Breakpoints limit the amount of operations to be stepped through by specifying limitations on the operations. At such an operation the execution of the program is suspended which is called *breaking*.

The most commonly used breakpoints are *Statement breakpoints*. Statement breakpoints are annotations on a statement of the programming source code. They break when the annotated statement is executed and are the usual type of breakpoints found in today's Integrated Development Environment (IDE). An example of this kind of

breakpoint can be seen in Figure 4.4. The biggest issue with Statement breakpoints

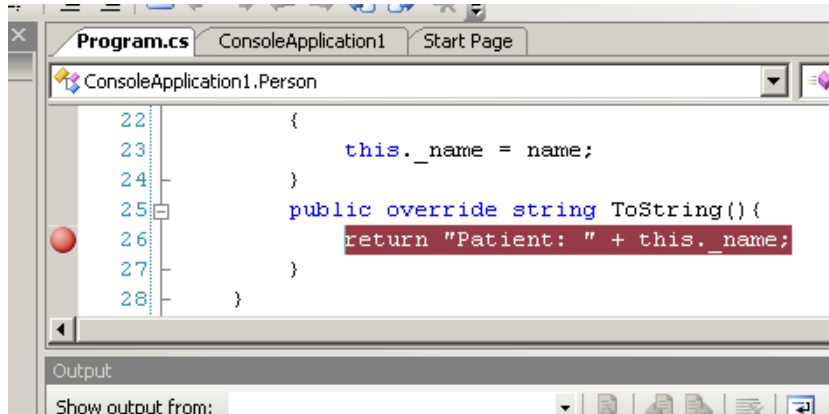


Figure 4.4: A conventional breakpoint in Visual Studio 2005

is that when an statement is executed too many times it overloads the developer. To improve it usefulness, programmer's insert a condition as can be seen in Listing 4.4.

```

1     if(this._name == null){
2         Console.WriteLine(); //statement breakpoint
3     }
  
```

Listing 4.4: Emulating a conditional breakpoint using a conventional one in the C# language

In this case if the this._name variable contains the value null the Console.WriteLine is executed which triggers the breakpoint. This use of an if-statement evolved in the Conditional breakpoint as can be seen in Figure 4.5.

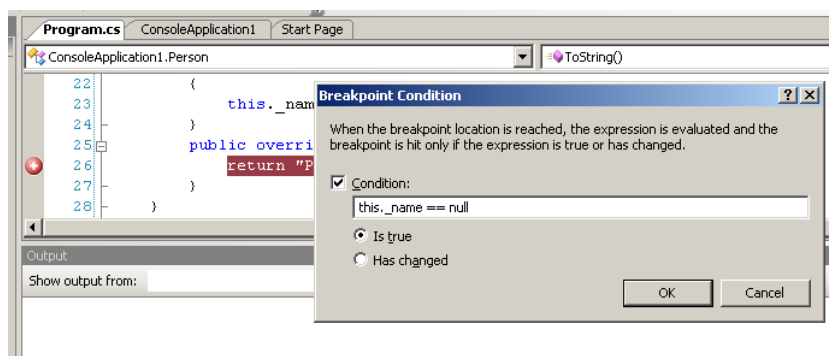


Figure 4.5: A conditional breakpoint in Visual Studio 2005

4.3 Added difficulties in debugging Aspect-oriented programs

In Figure 4.6 can be seen the relation between the instantiation and the implementation. This relation between the instance and its implementation is the implementation relation. In object-orientation a function is implemented in one class and multiple objects can be an instance of that class. In the case of object-orientation the implementation relation is therefore a one to many relation.

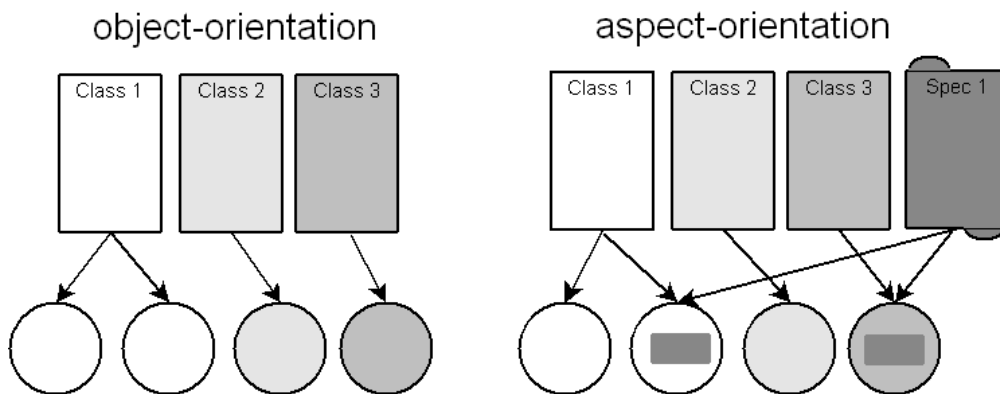


Figure 4.6: Difference between OO definitions and AOP definitions

Because crosscutting consists of spreading and tangling, isolation of crosscutted advice therefore always results in a many to many implementation relation. Because aspect-orientation allows for modularization of the crosscutting concerns, by default it also has a many to many implementation relation. This many to many relation enlarges the increase in complexity at runtime even more because the influence of the advice becomes another factor to take into account. When dynamic weaving is used, the dynamic behavior is another factor and thus the execution becomes even more complex.

Because advice can be executed in many places within the program, using a conventional breakpoint on advice, causes the amount of breaks to increase dramatically. This increase in breaks, overloads the developer. The added complexity of aspect-oriented programs and reduced usefulness of the conventional breakpoint at runtime, make aspect-oriented programs hard to debug.

Composition filters are not that different from other aspect oriented languages in this facet. In Listing 4.5 it can be seen that both the `logAddingPatient` and `rights` filtermodules are superimposed on the same class `Doctor`. This means that composition filters can add multiple advices to a single class. In the same example the `rights` filtermodule is superimposed onto multiple classes. Composition filters therefore have the same many to many implementation relation, which result in a high execution complexity. This complexity makes it more difficult to debug than when not using aspect oriented source code. While the `Compose*` compiler does some static semantic analysis which allow reasoning about the target program and composition filter interferences, the detection of semantic bugs is limited.

```
1 concern Logging{
2   filtermodule logAddingPatient {
3     inputfilters
4     logging : Meta = {[*.Add] logger.LogAdd}
5   }
6   superimposition {
7     selectors
8     persons = { C | isClassWithName(C, 'Doctor')};
9     filtermodules
10    persons <- recursion_fix;
11  }
12 }
13
14 concern Rights{
15   filtermodule rights {
16     conditions
17     access : Context.Access()
18     inputfilters
19     rights : Error = {!access -> [*.]}
20   }
21   superimposition {
22     selectors
23     persons = { C | isClassWithNameInList(C, ['Doctor', '
24       AdminConsole'])};
25     filtermodules
26     persons <- rights;
27   }
28 }
```

Listing 4.5: Concerns in on Doctor Compose[★]

This report will concentrate on improving the representation, editing, compiling, and debugging of composition filter programs while remaining extensible and programming language independent. The primary focus will be on the debugging of composition filter programs within the Compose[★] framework.

Chapter 5

Conceptual Solution

*“Everything should be made as simple as possible,
but no simpler.”
Albert Einstein*

To assist the developer in debugging applications using Composition Filters, we need to know what kinds of bugs are made. To identify these bugs, we observed the bugs commonly made in Compose* programs. At the time of writing the Compose*/.NET was the commonly used implementation of Compose*. We observed which bugs are commonly made by a total of 14 MSc. students while using Compose*/.NET. We conclude that if a program that uses Composition Filters has a fault, the bug which caused the fault is located in or caused by:

The original class

The implementation or use of the class is incorrect;

One of the filters

The filter is used or implemented incorrectly;

Combination of filters

The filters interfere with each other or with other non-composition filter source code;

Incorrect Superimposition

The wrong joinpoints are selected;

Combination of all

Combination of all problems mentioned above.

Other effects

The system may experience problems beyond the programs control e.g., hardware malfunction, bugs in the framework implementation.

These bugs except the latter are caused by insufficient comprehension of the program's execution behavior by the programmer. The programmer did not anticipate the unin-

tended behavior of the program when he/she developed the source code. These bugs are semantic bugs; they are not a violation of the syntax, but a violation of the program's meaning or intention. In essence the programmer intended to write a different program.

5.1 Execution behavior

When describing a program by a Deterministic Finite Automata (DFA), the execution behavior of an implemented program is all the possible paths in the automata. One of these paths is one possible execution path, of a program, for a given input.

The intended path is an execution path that the programmer originally envisioned for a given input. When a failure occurs, the execution path of the implemented program differs from the intended path. Isolating the fault is finding the first fault in the path where the implemented execution path deviates from the intended execution path. The transition pointing to the first fault is related to a transition in the implemented automata, which is the bug we are looking for.



Figure 5.1: Intended Path

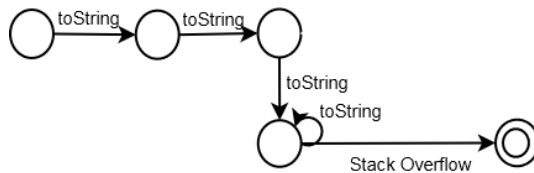


Figure 5.2: Implemented Path

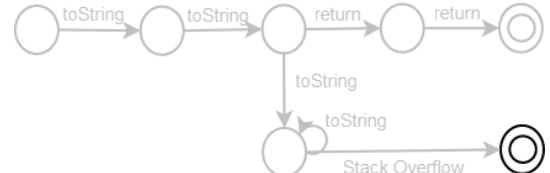


Figure 5.3: Failure

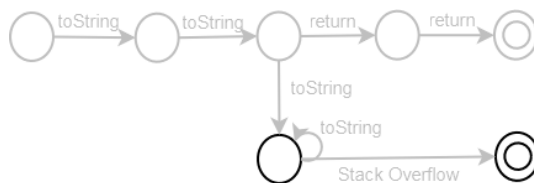


Figure 5.4: Faults

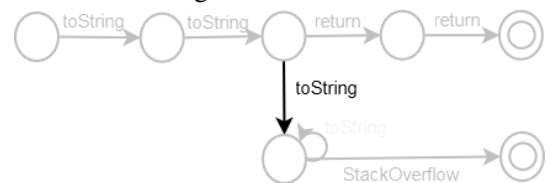


Figure 5.5: Bug

In the case of the example as shown in Listing 4.1 a possible execution path of the implemented automata is as shown in Figure 5.2. This execution path of *toString*, *toString*, ..., *StackOverflow* is not the intended behavior. The programmer experiences this as the program having a different output¹ than what he expected. The expected behavior is *toString*, *toString*, *return*, *return* which is our intended execution path as shown in Figure 5.1. As can be seen in Figure 5.3 the failure is the

¹This includes the absence of output in case of e.g. a deadlock

difference in *accepting state* between the intended and implemented path. Because the implemented path ends with a *StackOverflow* instead of a *return* the last state is a failure.

As can be seen in Figure 5.4, the faults are the differences in *states* between the intended and implemented path. While debugging, the programmer will try to find out where the program starts to behave differently. The programmer is searching for the first occurrence of a fault in the execution path. As can be seen in Figure 5.5 the bug is defined as a transition going from the correct state to a faulty state. In this example the bug is therefore the call of another *toString*.

Besides the situation that *toString* is considered the bug, another possibility is that the developer is of the opinion, that the incorrect use of these classes is the bug causing the fault. The failure in this case, is the state which caused the third *toString* to occur. Therefore the state before the first fault, the state between the second and third *toString*, has now become our failure, as can be seen in Figure 5.7. The programmer then tries to isolate the fault again.

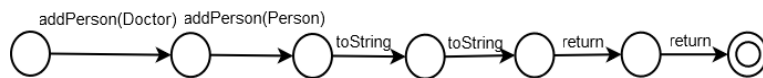


Figure 5.6: Intended path with the correct use

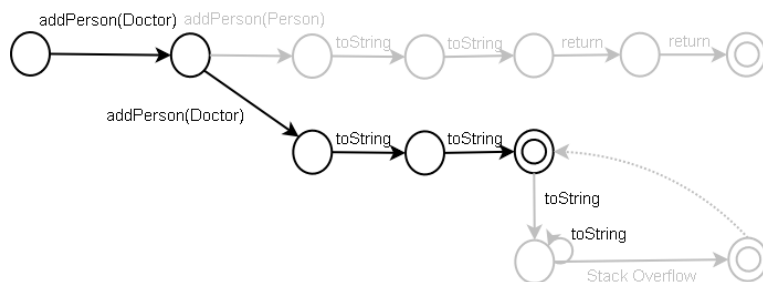


Figure 5.7: Implemented path incorrect use

This difference in opinion between *toString* being the bug or the *addPerson* being the bug, is a difference in the intended path, which results in a different transition being a bug. In practice, the intended path is implicit in the programmer's brain. While debugging, the programmer thinks the intended path is like the one shown in Figure 5.1. The programmer will generally restate the isolated fault as a failure, and perform the whole procedure again. This restating of the failure result in the programmer gaining the insight that the intended path is as shown in Figure 5.6.

This recursive isolation of the failure therefore refines the intended path to resemble what the programmer really intended. This is a recursive step of refining the intended behavior of a system which is fundamental to fault isolation. Essentially the programmer will recognize incorrect behavior when the programmer notices it, but does not know exactly how it should be. This is different from the common opinion. The cur-

rent common opinion is that the programmer always knows how its program should behave.

5.2 Breakpoints

Conventional breakpoints are used to assist the programmer in isolating the fault. The breakpoints are placed on the (line number of the) instruction. Because the instructions of the system are represented by a DFA as transitions, in this model conventional breakpoints are therefore placed on a transition. This means that the debugger will break the program, if that transition is about to be executed. Because conventional breakpoints are less suitable to gain insight in aspect oriented programs, our solution is to provide better suited breakpoints.

The specification of the location of a breakpoint can be seen as a fine-grained joinpoint specification where the advice results in a break. Since the specification of conventional breakpoint, is not expressive enough to specify a fine grained joinpoint in an aspect oriented program, we need to increase the expressiveness of the breakpoint specification. This is accomplished by moving from a structural to a behavioral joinpoint specification. Since we are specifying the location of a breakpoint, it is essential that the specification that we use is suitable for the finding bugs.

With a behavioral specification we can better describe invariants over the behavior of a program. It is then possible to validate the execution behavior of the program to this behavioral specification and find the misconceptions between the implemented program and what the programmer intended. This helps the programmer to comprehend the execution behavior of the implemented program with respect to its source code. Because we do not know where the bug is, we need a joinpoint specification that does not require specifying the bug itself. This is necessary because if we know where the bug is, we usually do not need to debug.

This requirement for the specification language may be obvious, but it results in many commonly used specification formalisms like; regular expressions, push down automata, and the specification language proposed by Conradi [Con06] for specifying within a Fine-grained Joinpoint Model, to be unpractical. They rely on the programmer, specifying an exact location within the program or behavior.

The join point specification language, we therefore propose to use, is Linear Temporal Logic (LTL) because of its describing power based on behavior instead of the structure of the implemented program. LTL was published by Pnueli [Pnu77] as a modal logic over infinite sequences. It is typically used in model checking. Model checking is an automatic verification technique for finite state systems. Specifications are written in propositional temporal logic as a formula, and verification is done by an exhaustive search of all states in the model.

LTL breakpoints contain such a LTL formula. This LTL formula is an invariant over the

execution behavior. When the LTL formula does not hold, the breakpoint is activated and the program breaks. With the use of an LTL breakpoint, the programmer can describe the intended behavior. The breakpoint will then halt when the behavior is about to deviate.

In the case of the doctor example as shown in Listing 4.1 the LTL formula the failure could be describing as $\Box(\neg StackOverflow)$. This means that in the whole execution we do not want the *StackOverflow* to occur. The breakpoint will then cause the program to break when *StackOverflow* is about to occur. This will allow the programmer to inspect the state which resulted in the formula not to hold.

Because the evaluation of an LTL formula, depends on the propositions used, the formula needs to be checked if it holds for every transition. LTL breakpoints are therefore more expensive in computation than normal statement or conditional breakpoints. When using LTL breakpoints for all instructions of a system, the overhead can reduce its usefulness. It is therefore advised to only validate the propositions for each advice and use conventional means to find bugs in non-aspect oriented code.

Because composition filters only have side effects on the message when the composition filters are evaluated, composition filters can be considered stuttering equivalent [MCBG88]. This means that while propositions that are based on the message can influence the holding of a formula, other propositions do not need to be evaluated. By using the stutter equivalent behavior of the filter evaluation we can reduce the amount the overhead of evaluating LTL breakpoints.

5.3 Behavior prediction

Since the most commonly used debuggers today are execution steppers, a naive debugger would step through the filter evaluation as shown in Figure 5.8. In this representation, the evaluation at a joinpoint would span multiple screens, because it only represents one moment in the complete filter evaluation. The programmer would then step through the filter evaluation, one filter at the time.

If the side effects of a filter evaluation is isolated to the messages, it is possible to evaluate a full filter evaluation on a joinpoint as shown in Figure 5.9. This representation provides the programmer with more insight into the filter evaluation than stepping through the evaluation of the individual filters, because he can directly see the influences between the filters within the filter set.

When dealing with evaluations that have side effects, beyond the messages, a simulation of the evaluation of the filter set can be done. This simulation uses the assumption that the side-effect, do not result in any effects on the filter evaluation. By using this simulation, a prediction of the behavior can be represented as shown on the left side in Figure 5.10.

After this prediction, the side effect can really be executed as is shown on the right side

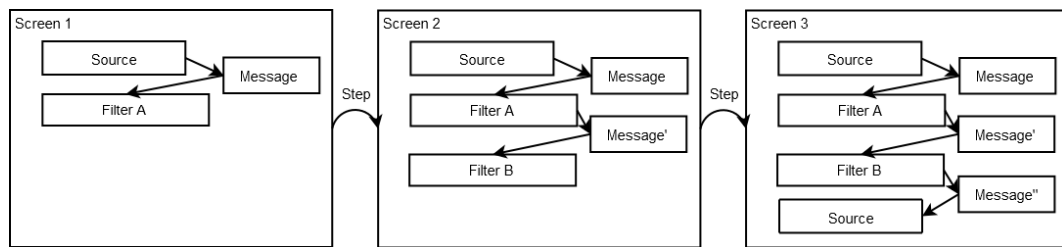


Figure 5.8: Stepping through filters at a joinpoint

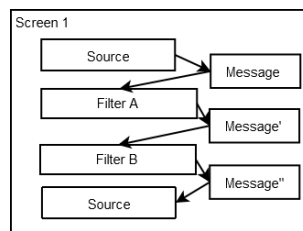


Figure 5.9: Representing the full joinpoint

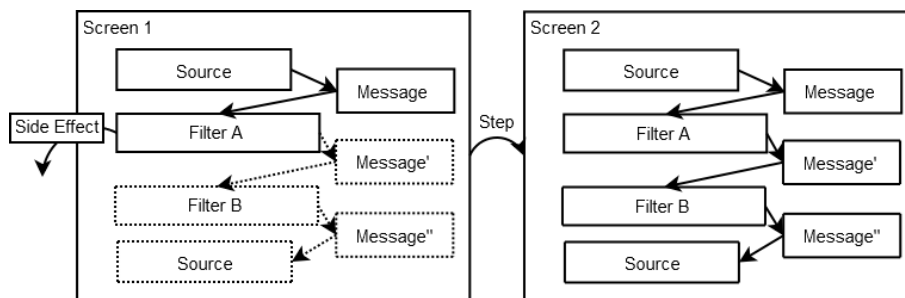


Figure 5.10: Predicting the rest of the execution behavior

in Figure 5.10. If this side effect results in an effect on the filter evaluation, the real behavior would be different than the predicted behavior. By representing the predicted and the real behavior, the programmer can comprehend the result of the side effects.

Conceptual Design

*“Part of the inhumanity of the computer is that,
once it is competently programmed and working smoothly,
it is completely honest.”*
Isaac Asimov

In this chapter we will make the conceptual solution more concrete. Firstly, we describe the LTL formula of the LTL breakpoint. This is used for expressing the moment in the execution we want to inspect. Secondly, we describe the different representations for the Composition Filters. Thirdly, we describe the different navigations for Composition Filters to advance to another moment in the execution when the program is broken. Fourthly, we show how to use this approach to debug an example program.

6.1 LTL Propositions

An LTL breakpoint consists of two elements: a LTL formula and the break behavior. Linear temporal logic (LTL) is a modal linear time logic over infinite traces. It was introduced by Pnueli [Pnu77]. The syntax of a LTL formula is as follows:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \psi \mid \bigcirc\phi \mid \phi\mathcal{U}\psi$$

- p is an atomic proposition. This proposition can hold or not hold.
- $\neg\phi$ is the negation. A negation holds, if ϕ does not hold. A negation is commonly written as $!\phi$.
- $\phi \vee \psi$ is the disjunction. A disjunction holds, if either ϕ holds or ψ holds. The disjunction is commonly written as $\phi|\psi$.

- $\bigcirc \phi$ is the next. The next holds in a state, if ϕ holds in the successive state. It is commonly written as $X\phi$.
- $\phi \mathcal{U} \psi$ is the until. The until holds if ψ eventually will hold for some state and ϕ continuously holds until that state. The until is commonly written as $\phi \bar{\mathcal{U}} \psi$

By using these elements the following operators can be derived:

$$\begin{array}{ll}
 \text{true} \equiv p \vee \neg p & \phi \Leftrightarrow \psi \equiv (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi) \\
 \text{false} \equiv \neg \text{true} & \diamond \phi \equiv \text{true} \mathcal{U} \phi \\
 \phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi) & \square \phi \equiv \neg \diamond(\neg\phi) \\
 \phi \Rightarrow \psi \equiv \neg\phi \vee \psi & \mathcal{W}\phi \equiv \square \phi \vee (\phi \mathcal{U} \psi)
 \end{array}$$

The first five definitions (true, false, conjunction, implication, and equivalence) are standard propositional logic. The conjunction is commonly written as $\phi \& \psi$ and the implication as $\phi \rightarrow \psi$. The last three operators are future (eventually ϕ), globally (always ϕ), and unless (weak until ϕ).

- Formula $\diamond \phi$ holds if either ϕ is true now, or it will become true in some state in the future. This formula is commonly written as $F\phi$.
- Formula $\square \phi$ holds if ϕ holds in the current state and it holds in every state in the future. This formula is commonly written as $G\phi$.
- Formula $\mathcal{W}\phi$ holds if $\phi \mathcal{U} \psi$ holds, without guarantee that ψ will ever hold, meaning that ϕ continuously holds if ψ never holds. This formula is commonly written as $W\phi$.

Because the expressiveness of a LTL formula is based on the expressiveness of the atomic proposition, we are going to define what propositions can be used. In order to find the necessary expressiveness of the proposition, we need to analyze the basic elements within the composition filter model. By being able to express these basic elements, a suitable expressive specification can be found.

As noted in Section 2.2 within Compose[★] we filter on messages. According to the language analysis of Doornenbal [Doo06], concerns are the distinctive building blocks of a Compose[★] application. For these basic model elements we now are going to determine what the expressiveness is, that we need.

6.1.1 Concerns

A concern has a name and an optional namespace. A concern consists of three elements: zero or more filter modules, an optional superimposition part, and an optional implementation part. The filter modules are superimposed on joinpoints by the filter module binding field of the superimposition. The implementation part contains language dependent code of the concern.

The statements which therefore can be used to address a concern are:

Concern.Name

Concern.Name is a `String` which is the name of the concern. This name can then be used in an evaluation.

Concern.Namespace

Concern.Namespace is a `String` which is the namespace the concern belongs to. The name of the namespace can then be used in an evaluation.

6.1.2 Superimposition

Superimposition is a pointcut specification. Because the pointcut specification applies filter modules onto joinpoints, a breakpoint specification which allows to specify filtermodules has enough expressive power.

6.1.3 FilterModules

A filter module has a name. A filter module can contain conditions, internals, externals, and two sets of filters: the input filters and the output filters.

The statements which therefore can be used to address a filtermodule are:

FilterModule.Name

FilterModule.Name is a `String` which is the name of the filtermodule. This name can be used in evaluations.

FilterModule.Condition

FilterModule.Condition is a list of conditions the filtermodule contains. We address individual conditions by their references. When addressing one condition, it's evaluation can be used as a `Propositions`.

FilterModule.Internal

FilterModule.Internal is a list of internals the filtermodule contains. We address individual internals by their references. When addressing a particular internal, the internal can be used in an evaluation.

FilterModule.External

FilterModule.External is a list of externals the filtermodule contains. We address individual externals by their references. When addressing a particular external, the external can be used in an evaluation.

6.1.4 Filters

A filter has three parts: a filter identifier, a filter type, and one or more filter elements. A filter element exists out of an optional condition part, a matching part, and an optional substitution part.

If the condition part is true and the matching part matches, the accept action of a filter is executed. Otherwise the reject action of a filter is executed. The type of the filter determines which kind of accept or reject action is executed. Since the matching part matches on the selector, the matching part itself can be described by allowing the description of the message. The behavior of the matching itself is something we like to describe at the filter. The condition part is a condition which can hold, or not hold. We therefore allow the usage of the condition within the proposition. Since a filter can accept or reject, we are interested when this happens, combined with the kind of filter action that is executed.

The statements which therefore can be used to address a filter are:

Filter.Name

Filter.Name is a `String` which is the name of the filter. This name can be used in evaluations.

Filter.Type

Filter.Type is a `Type` which is the type of a filter. Examples of such types are: Meta, Dispatch, Error, Send, Substitute. This type can then be used in an evaluation.

Filter.Action

Filter.Action is the filteraction a filter executes. Examples of such a filteraction are: ContinueAction, ErrorAction, DispatchAction. The filteraction can then be used in an evaluation.

Filter.Accept

Filter.Accept is a proposition which holds if the a filter accepts a message.

Filter.Reject

Filter.Reject is a proposition which holds if the a filter rejects a message.

6.1.5 Message

Since the messages are the most atomic communication within the composition filter model, matching a unique message should be expressible within the atomic proposition. A message contains five elements of data; Sender, Selector, Server, Target, and Arguments.

The statements which therefore can be used to address a message are:

Message.Sender

Message.Sender is the sender of the message. The sender can be used in evaluations.

Message.Selector

Message.Selector is a `String` which is the selector of the message. This selector can then be used in an evaluation.

Message.Arguments

Message.Arguments is a list of arguments the message contains. We address

individual arguments by their order. When addressing a particular argument, the argument can be used in an evaluation.

Message.Target

Message.Target is the target of the message. The target can be used in evaluations.

6.1.6 Operations

Because many of the data elements within the Composition Filter model are not propositions or conditions, we need to compare the data to a given value. An example of this is “the selector of the message is *ToString*”. This approach however does not work for the sender and target, since these can be objects and thus are very difficult to specify by a value. We therefore also allow the comparison of types. A proposition like: “the sender of the message is of a certain type” should therefore be possible.

The statements which therefore can be used to compare to are:

Basic Values

Basic values are values which a `String`, `Type`, or other value can be compared to. Examples of basic values are: 1, 2, 3, .., "", "a", "aa", .., 'a', 'b', ..., etc.

Value Operators

Value operators are operators to compare values. Examples of value operators are: Equals, Not Equals, Bigger Then, Smaller Then, etc.

Type Operators

Type Operators are operators which compare types. examples of type operators are: instanceof, .class, .type, etc.

6.1.7 Implementation Part And Other Language Dependent Source Code

The implementation part and the source code the Composition Filters are applied on, are both language dependent. If we would allow the specification of the basic elements within these language dependend source code, the expressive power of the breakpoint specification would be very large. The problem however would be, that by specifying elements beyond the compositional model within a proposition, we need to put restrictions on the programming language that the breakpoint specification is applied on. Since Composition Filters are (and need to be) language independent we cannot do that.

However, in the implementation the programming language is much more restricted. It is therefore possible to add these elements as propositions. It is however paramount, that the propositions do not have any side effects on the system itself.

6.2 Filter Representation

If an LTL breakpoint breaks a system at a joinpoint, the system is in a state which we want to inspect. We therefore need to focus on how to represent what we want to inspect. Since we want to inspect Composition Filters, we look at the different representations of Composition Filters. These representations are then compared to each other, in order to find the representation best suited for debugging.

6.2.1 8Ball representation

The 8Ball representation, as shown in Figure 6.1, shows a joinpoint of the non-compositional filter part of the source code on a sphere. By rotating the sphere different filters get superimposed on the shown joinpoint. By rotating in a different direction, combinations of filters can be tried on the model. If the programmer rotates the sphere to the top-right: Filter A gets superimposed as shown in Figure 6.2.1. If the programmer would rotate the sphere down: Filter B gets superimposed instead of Filter A. If the programmer would rotate the sphere to the down-right both Filter A and Filter B will be superimposed.

This sphere can be in a larger sphere which rotation select Filter Modules. This larger sphere can then be part of an even larger sphere. The rotation of this largest sphere selects Concerns.

By using this representation, the programmer can see the different behavior of the statically defined superimposition when using different filters, filter modules, and concerns.

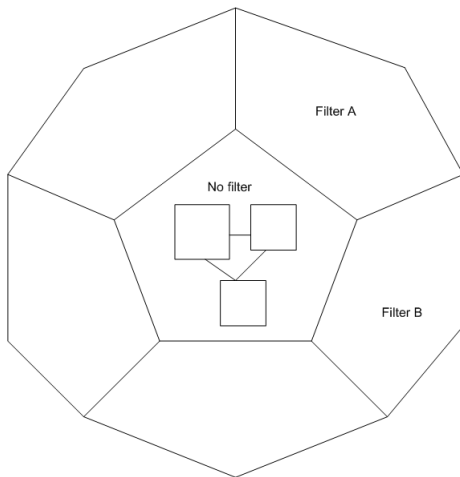


Figure 6.1: The 8Ball representation

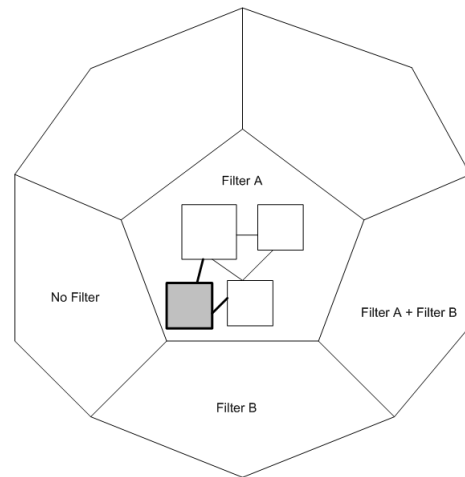


Figure 6.2: The 8Ball rotated view

6.2.2 3DBox representation

The 3DBox representation shown in Figure 6.2.2 is a filter representation based on the hyper slice approach [OT01]. In this representation the front of the cube displays the source model. This cube is intersected with filters represented as cards. In this cube an individual card can be selected. If a card is selected, the affected classes by the associated filter will be highlighted as shown in Figure 6.4. The cube can also be rotated. In the case of a rotated cube, the front will show the filter modules as shown in Figure 6.2.2. One can then select a class and then the filter modules that are superimposed on that class will be highlighted.

By using this representation, the programmer can see the different associations between the filter model elements and the non-composition filter elements.

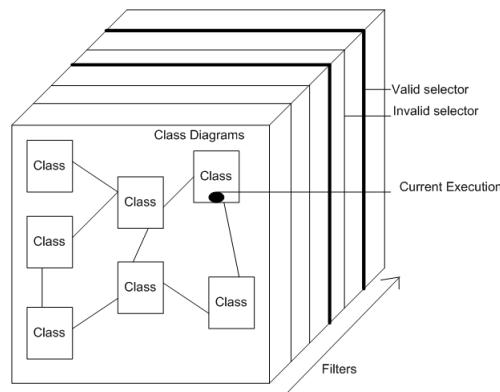


Figure 6.3: Default view

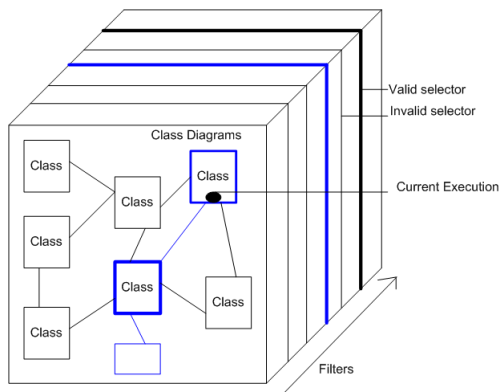


Figure 6.4: Selected view

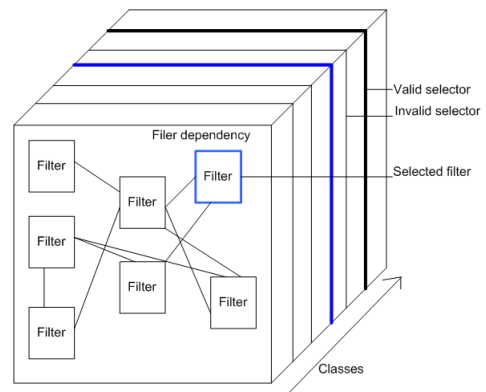
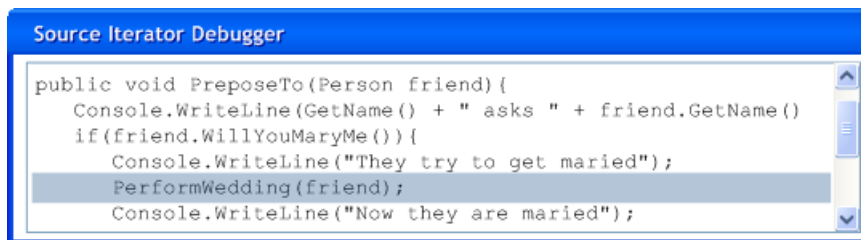


Figure 6.5: Rotated view

6.2.3 Source iterator representation

Since the most commonly used debuggers today are execution steppers, a naive debugger would therefore step through the source code. As shown in Figure 6.6, the source iterator approach is a source iterator which steps through the functions being executed. It consists of an accent or current execution point to show which function will be executed. Combined with the options to go down into the stack, up in the stack, pause/resume execution and terminate; it is a very low level tool.



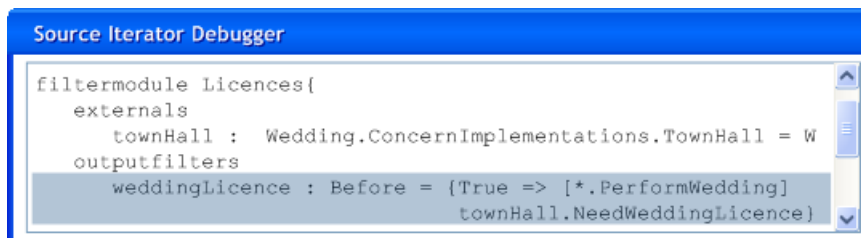
```

Source Iterator Debugger
public void PreposeTo(Person friend){
    Console.WriteLine(GetName() + " asks " + friend.GetName())
    if(friend.WillYouMaryMe()){
        Console.WriteLine("They try to get married");
        PerformWedding(friend);
        Console.WriteLine("Now they are married");
    }
}

```

Figure 6.6: The source iterator representation showing non-Composition Filter code

Composition filters on this level could be supported by treating them like the functions in non-composition filter source code. At a joinpoint the evaluating filters source code would be stepped trough as shown in Figure 6.7. The highlighted line is in this representation, the currently evaluating filter.



```

Source Iterator Debugger
filtermodule Licences{
    externals
        townHall : Wedding.ConcernImplementations.TownHall = W
    outputfilters
        weddingLicence : Before = {True => [*.PerformWedding]
                                     townHall.NeedWeddingLicence}
}

```

Figure 6.7: The source iterator representation showing Composition Filter code

The source iteration representation is commonly used in combination with a stack trace as shown in Figure 6.8. The stack trace consists of the functions which are on the execution stack. Because we treat a Composition Filter as we would treat a function, Composition Filter evaluations are just execution stack elements like the functions on the call stack.

By using this representation, the programmer can see the behavior of the composition filters and the source code.

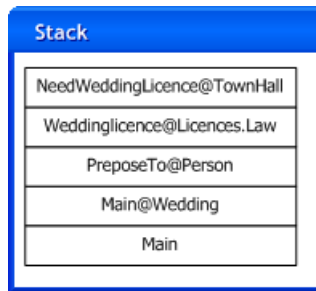


Figure 6.8: The call stack

6.2.4 Visual Composition Filters representation

The visual composition filter representation shown in Figure 6.9, is a visual language for Composition Filters as proposed by Binnema [Bin98]. The language focuses on visualizing the class interface and the effects Composition Filters have on the interface. Because of the language focus, according to Binnema, object instantiations are not supported.

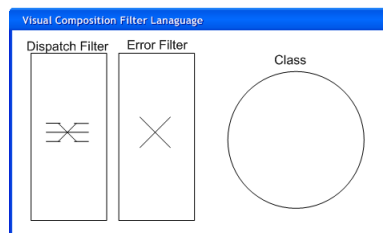


Figure 6.9: The visual composition filter representation from Binnema

By changing the symbol of a class to a square, an object to a circle, and a message to an arrow, as can be seen in Figure 6.10, we can reuse the representation in a language which does support instances. In this representation the sender object is left and the target is right. The sent messages are represented by arrows. The filters are represented by boxes.

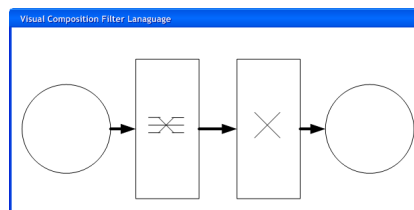


Figure 6.10: The modified visual composition filter representation

If a filter changes a message, different evaluation paths can be displayed as can be seen in Figure 6.11. This way, filter evaluation behavior can be visualized in an intuitive way.

By using this representation, the programmer can see the behavior of the composition filters evaluations.

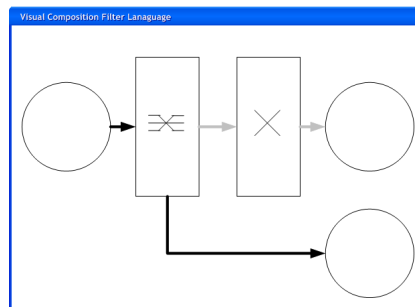


Figure 6.11: Dispatch action in visual composition filter representation

6.2.5 Tree representation

The tree representation as shown in Figure 6.12, uses a single root with the concerns as child elements. The child elements of the concern are filter modules. The child elements of the filter modules are the; internals, externals, and filter elements. When executing, the highlighted node, is the filter which is being evaluated.

By using this representation, the programmer can see how the filters are composed.

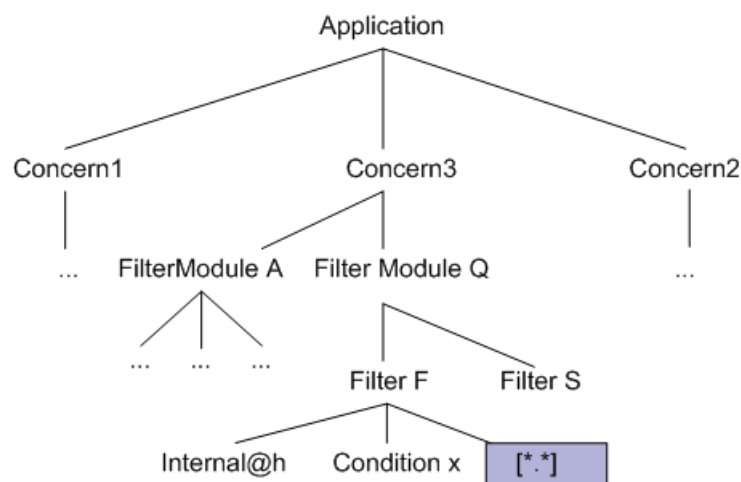


Figure 6.12: The tree representation

6.2.6 Vortex representation

The vortex representation as shown in Figure 6.13 is inspired on the whirling of the wind. The path of a message is represented by the line, with the boxes being filters. The partition the filters are in, represent the concerns the filter belong to.

To aid the programmer, to better comprehend the behavior, we use color. Green is an accepting filter within the filter module. A red filter is a rejecting one for a given message. The currently evaluating message is represented by a black dot. The ordering of the filters is represented by where they are on the line.

By using this representation, the programmer can see the behavior of the composition filters.

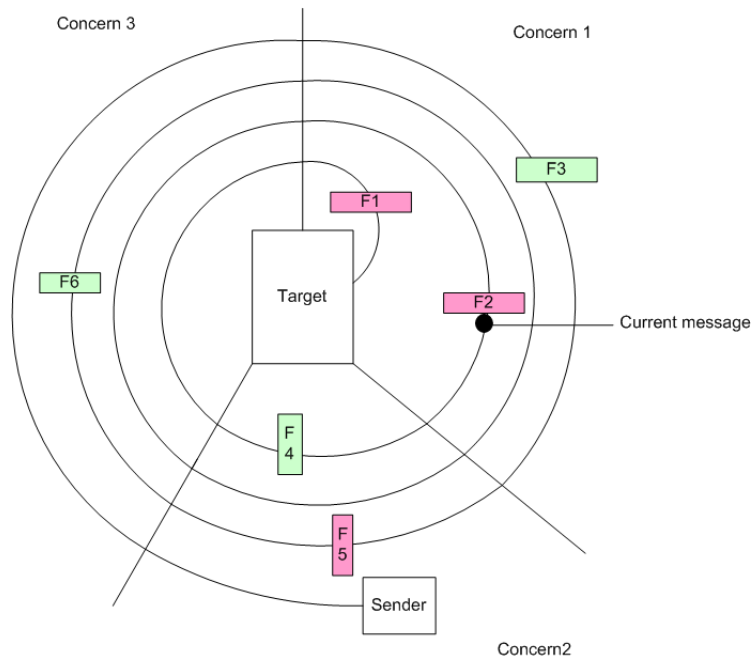


Figure 6.13: The vortex representation

6.2.7 Message box representation

The message box representation as shown in [Figure 6.14](#), represents a joinpoint. The screen is divided in two sides.

The left side is the source code side. It contains the written source code with the sender at the top, the target at the bottom, and the superimposed filters in between.

The right side is the message side. It contains the values of the messages. The message at the top is the original message sent by the sender. The message below the top is the message which resulted by the behavior of the filter actions belonging to the filter left of it. With this representation it is therefore very clear, what the behavior of a filter is, and how it relates to the source code. By using this representation, the programmer can see the behavior of the composition filters in combination with the source code.

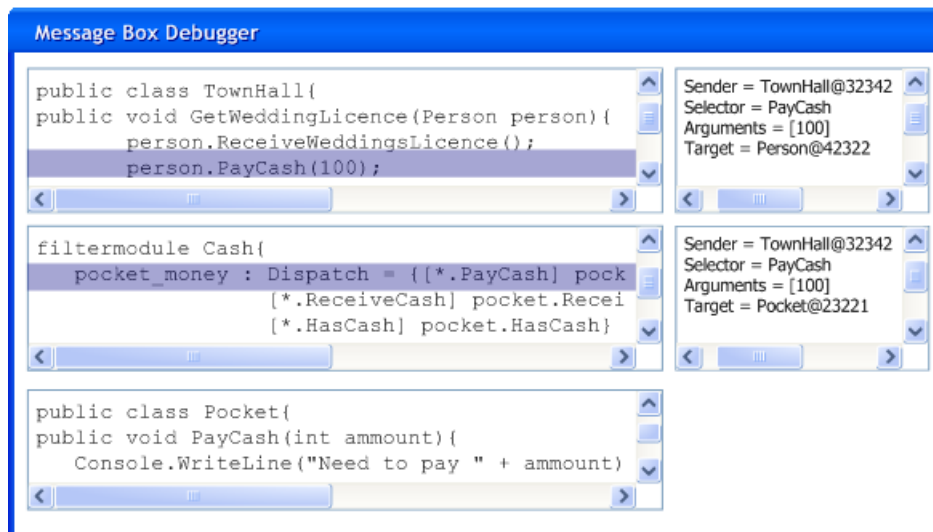


Figure 6.14: The message box representation

6.2.8 Action list representation

The action list representation shown in [Figure 6.15](#) focuses on the execution of filter actions. The top box is the source of the sender. The box under the sender source code is divided in two sides, with the left side for the accept actions and the right for reject actions. The actions which are executed by the message from the sender are highlighted. The final target is then shown in the bottom box.

By using this representation, the programmer can see the actions of the filters being executed.

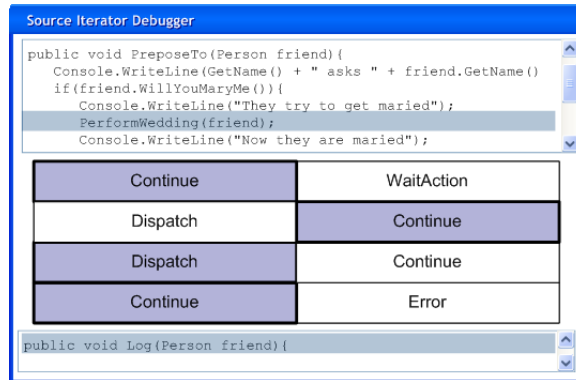


Figure 6.15: The action list representation

6.2.9 Face the music representation

The Face the music representation as shown in Figure 6.16, is based on Tablature music notation. On the left, all filters of the program represent one line (or note) in the Tablature. The left represents the sender and the right the target. Individual filters are represented as notes. The current location of the execution is a line which is moving from left to right.

By using this representation, the programmer can see the behavior of the composition filters over an amount of time.

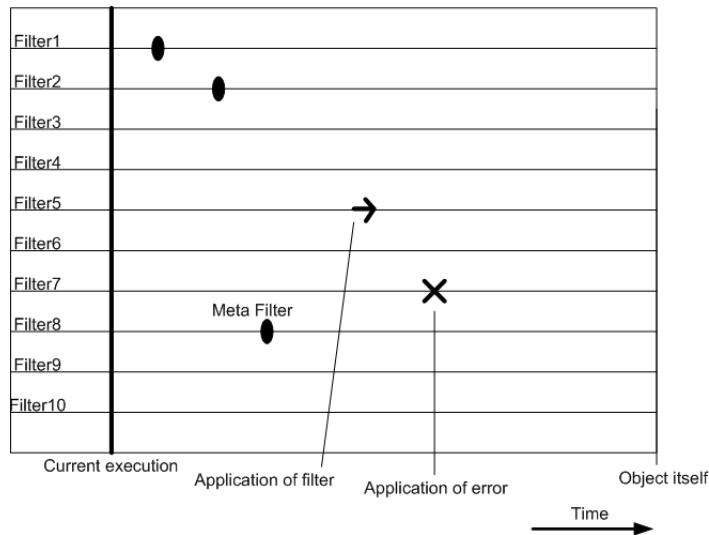


Figure 6.16: The face the music representation

6.2.10 Conclusion

Our opinion is, that considering the program size of many projects, most of the representations are unpractical. The program size causes those representation to be too large for a computer screen. However, not all the representations that are cheap in screen real estate are usable. We want the programmer to not only understand what the behavior is, but also comprehend why this behavior is as it is. This means, that the representation needs to reflect the behavior of the program back to the programs source code.

The commonly used source iteration is suited as a debugging representation. Because the representation is cheap in screen real estate, reflects the behavior back to the source code, and is commonly used in other programming languages. But we found that the Message Box representation is, in our opinion, a much more suitable representation. This is because the Message Box representation represents a complete filter evaluation at a joinpoint, while the source iterator only represents one moment in the filter evaluation. While the source iterator only shows the executing location in the source code, the message box representation shows the message changes in combination with the source code. Therefore in our opinion the Message Box representation is best suited for the programmer to comprehend the behavior.

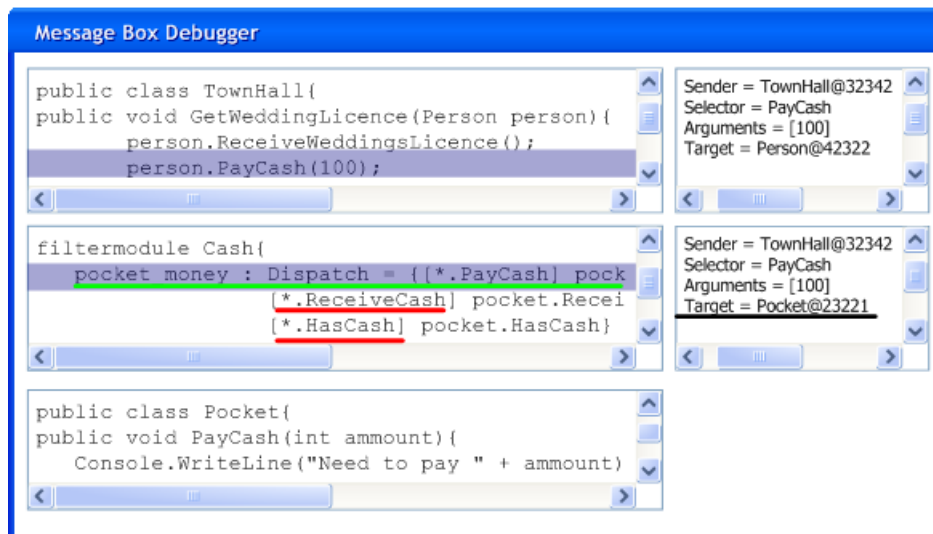


Figure 6.17: The message box representation with added colour

To make this representation more concrete, we add additional aids to assist the programmer within this representation. As described previously, the screen represents a single joinpoint and is divided in two sides. The left side is the source code side, the right side is the message side. It contains the written source code with the sender of the message at the top, the target of the message at the bottom, and the superimposed filters in between. To further assist the programmer in comprehending the behavior,

we annotate the individual parts of the filter elements with color. Green for accepting, red for rejecting as shown in Figure 6.17. We also annotate the changed properties of a message. This way, a programmer can see the behavior at a joinpoint in one blink of an eye.

Because this representation does not give information about the internal state of the language dependent elements, we allow the inspection of those elements, using language dependent inspectors.

6.3 Break Navigation

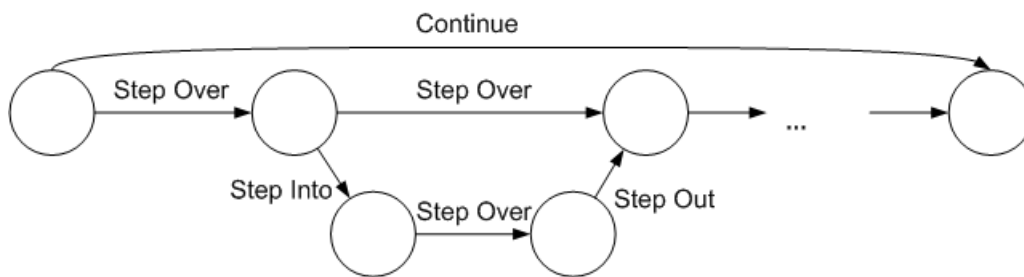


Figure 6.18: The effects of different navigations on the execution flow

If the program is broken, the state is represented. Because the represented state is a snapshot in the execution and the state might not be the state we are interested in, we want to navigate through the execution. As can be seen in Figure 6.18, within a debugger there are four basic navigations: continue, step over, step into, and step out. We need to validate if Composition Filters and the LTL breakpoints are suitable for using these navigations.

6.3.1 Continue

Continue resumes the execution till another break occurs. Since the continue just waits for a new break to occur, it does not conflict with the composition filters model. However, it is likely that when a LTL formula does not hold for a state, it does not hold for the next state. This next state is then broken as well, causing an overload to the programmer.

A solution to fix this problem, would be to invert the holding of the LTL formula for the next state. This would mean that the LTL breakpoint will hold in the next state if the LTL formula holds in that state. This solution however has the drawback, that if the breaking of the next state was intended, it becomes difficult to describe.

An in our opinion better solution, is to leave it to the programmer to specify a correct LTL formula. The programmer can easily solve this issue by specifying a fairness

like: $\Box \diamond \phi \Rightarrow \diamond \Box \phi$. Such a fairness would result in a halt when the proposition fails the next time. Since the programmer can easily make a mistake in specifying an LTL formula, we need to allow the modification of the LTL formula while debugging.

6.3.2 Step Over

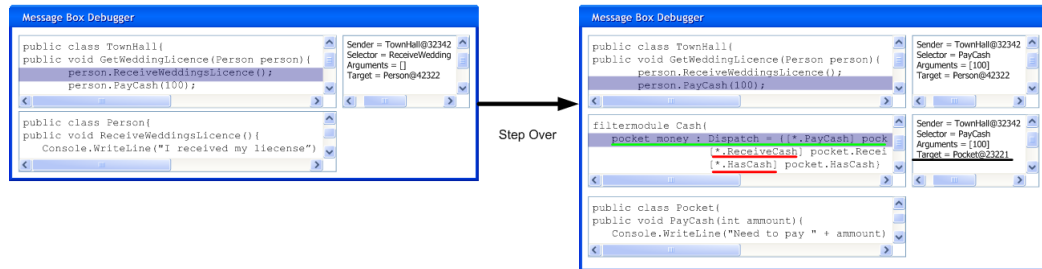


Figure 6.19: Stepping Over

Step over resumes the execution and inspects the next instruction or function. Within the representation of the complete joinpoint evaluation at once, there is not something like a next operation or function. Essentially a step over is to continue execution using the same calling context. As can be seen in Figure 6.19, because the calling context in the composition filter model is the sender, the next message from the same sender will halt the system. This means that when there is a return message, the return message will be halted.

6.3.3 Step Into

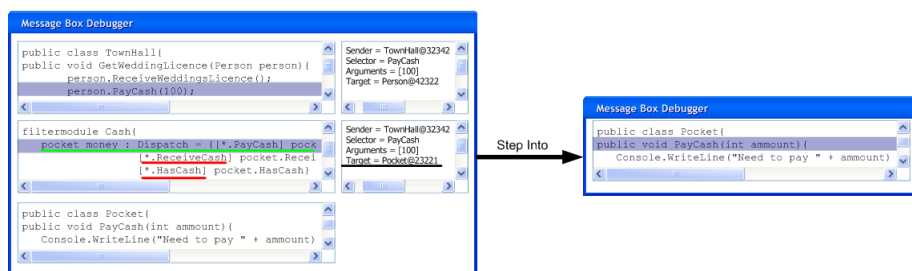


Figure 6.20: Stepping Into

Step into results in an inspection of the same state in a more fine grained execution flow. Essentially a step into is to continue execution using the called context. The step into when using custom and Meta filters can therefore be used to see the behavior within those filters showing the source code. As can be seen in Figure 6.20, because the called context in the composition filter model is the target, the step into navigates to executing code of the target.

6.3.4 Step Outside

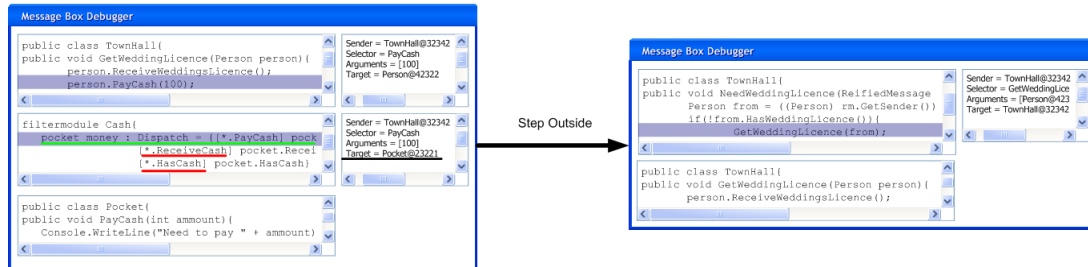


Figure 6.21: Stepping Outside

Step outside, results in an inspection of the same state in a less fine grained execution flow. Essentially a step outside is to continue execution using the context of the callers caller. As can be seen in Figure 6.21, because the callers caller context in the composition filter model is the joinpoint resulting in the sender sending a message, the step outside navigates, to the joinpoint later in the execution, where the target is what currently is the sender.

6.4 Debugging Example

We are now going to use the LTL breakpoints on an example. In this section we disclose the details of the example as they would be found when the programmer is debugging. The full source code of the example can be found in Appendix A.

6.4.1 Non-Compositional Filter Behavior

The example we are going to use is about a man proposing to a woman to marry him. The model consist of just one class; Person. This class has two instances; the male called Linus Gates, and the Female called Belinda Torvalts.

As can be seen in Listing 6.1, the execution without Composition Filters applied is; the man proposes to the woman, the woman accepts his proposal, and they then they are married.

```

1 Linus Gates
2 Belinda Torvalts
3 Linus Gates asks Belinda Torvalts: "Will you marry me?".
4 Belinda Torvalts answers: "I do".
5 So in the end
6 They are married

```

Listing 6.1: Execution wedding without Composition Filters

```

1  private Person partner = null;
2  public bool IsMarried()
3  {
4      return partner != null;
5  }

```

Listing 6.3: Source code responsible for checking for marriage

6.4.2 Compositional Filter Behavior

We now implement some concerns that could happen in the real world by using the Composition Filter language. After the implementation of these concerns the behavior has changed as can be seen in Listing 6.2. In this case; the man proposes to the woman, the woman accepts his proposal, but they are not married. So why are they not married?

```

1  Linus Gates
2  Belinda Torvalts
3  Linus Gates asks Belinda Torvalts: "Will you marry me?".
4  Belinda Torvalts answers: "I do".
5  So in the end
6  They are not married

```

Listing 6.2: Execution wedding with Composition Filters

The source code which is responsible for the marriage is in the Person class as shown in Listing 6.3. As can be seen, the marriage status dependent on the *partner* variable.

Because the *partner* variable resides outside composition filters, we use a conventional conditional breakpoint on the Person class which triggers when the *partner* variable is changed. This results in a break, when the *Divorce* function is called as shown in Figure 6.22. Because there is no reference within the Person class to the divorce function, it is unclear why the divorce function is called.

6.4.3 Origin Of The Divorce Function Call

By using conventional techniques we isolated the problem to the *Divorce* function. We now are going to use the composition filter debugging techniques to find out why this happens. In this manner we can use conventional techniques to debug and locate the responsible code within the non-composition filter source code and the composition filter techniques for the compositional filter source code.

To find out why the *Divorce* function is called, we formulate the LTL formula:

$$\square \text{Message.Selector} \neq \text{"Divorce"}$$

This LTL formula means: “*Divorce* should not be the selector of the message”. We

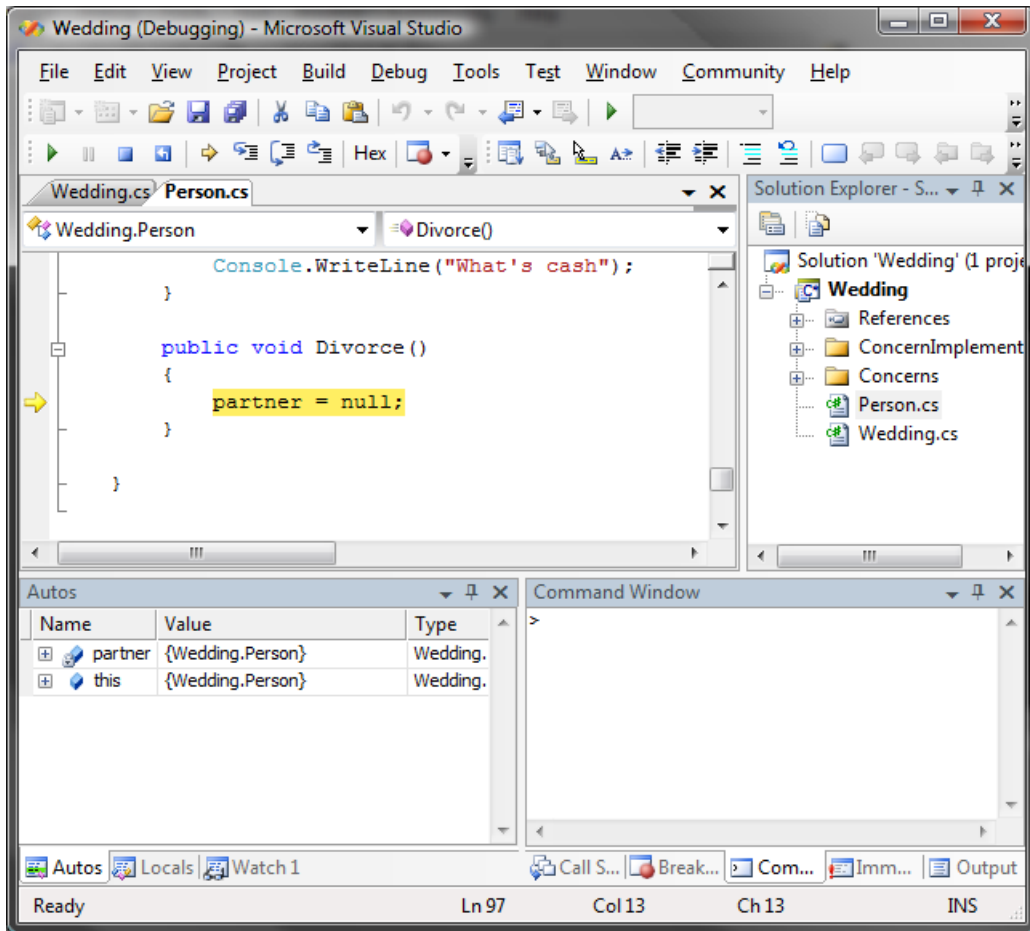


Figure 6.22: A conventional debugger inspecting the divorce function

use this LTL formula as a breakpoint. The breakpoint results in the program breaking at the joinpoint shown in Figure 6.23.

But, the debugger displays source code within the `TownHall` class. As we stated previously, the model only consisted out of one class `Person`. So where did this class come from?

The class itself belongs to the classes used by the implemented filters. So therefore it does not belong to the original model, and does the model consists of just one class. But which filter is causing the usage of the `TownHall` class ?

In order to find this filter we formulate the LTL formula:

$$\square!(Message.Target \text{ instanceof } TownHall)$$

This LTL formula means informally: “The `TownHall` class should not be used”. We

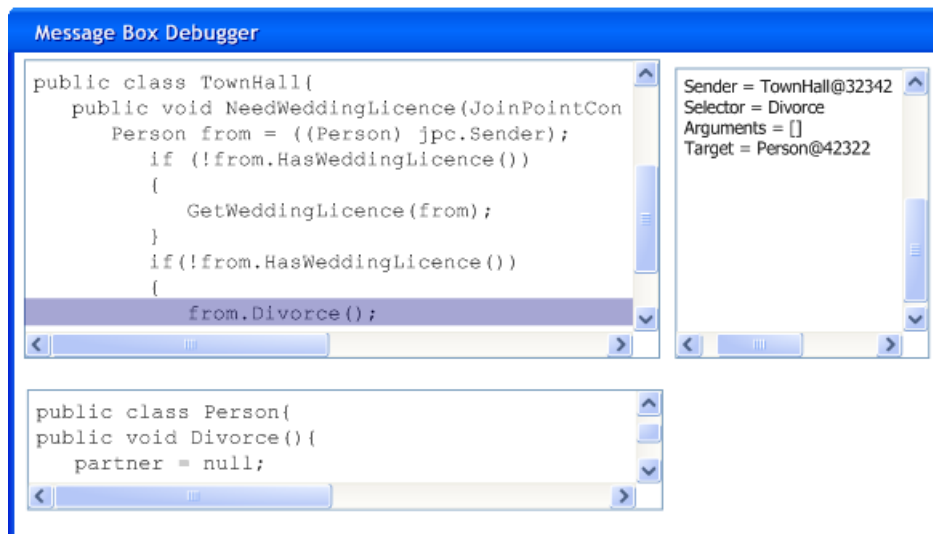


Figure 6.23: Composition Debugger inspecting the Divorce joinpoint

use this LTL formula as a new breakpoint. In this manner we can use the LTL break-points to find out which filters are responsible for certain behavior.

This breakpoint results in the program breaking at the joinpoint shown in Figure 6.24. As can be seen, the law requires to have a wedding license before you get married. We therefore need to find out, why Linus does not get a wedding license.

6.4.4 Arranging a marriage license

Because the rest of the filter evaluation is simulated, we see on closer inspection that while a wedding deserves a party, the party is before the wedding itself. This illustrates the added value of being able to see the complete filter evaluation.

Because we find this strange, we want to see what kind of consequences this has. We therefore use the navigation *step into*, to see what happens in the execution. This results in the program breaking at the joinpoint shown in Figure 6.25. As can be seen, this results in a band playing, and after that Linus pays the band. We wonder if this has any consequences so we change the LTL formula to;

$$\square \text{Message.Selector} \neq \text{"Pay"}$$

The we resume the execution. This results in the system breaking at the joinpoint shown in Figure 6.26.

We can see that this behavior probably happens because Linus does not have enough money. We now change the implementation of the Pocket, to sell his car when he does



Figure 6.24: Composition Debugger inspecting the PreformWedding joinpoint

not have enough money. This results in the behavior as shown in Listing 6.4; So now he is married again.

```

1 Linus Gates
2 Belinda Torvalts
3 Linus Gates asks Belinda Torvalts: "Will you marry me?".
4 Belinda Torvalts answers: "I do".
5 Linus does not have enough money.
6 So he sells his car.
7 So in the end
8 They are married

```

Listing 6.4: Execution wedding with Selling car

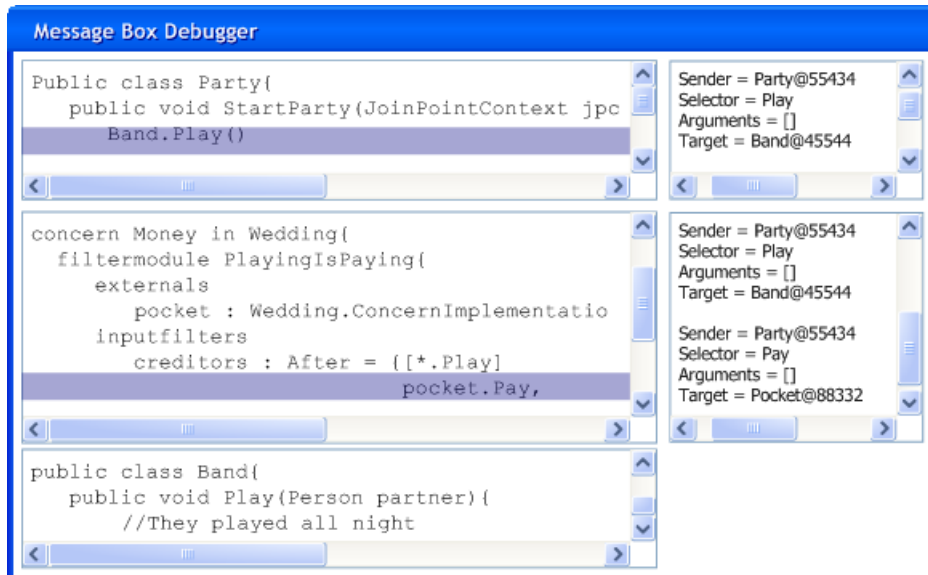


Figure 6.25: Step Into by the Composition Debugger

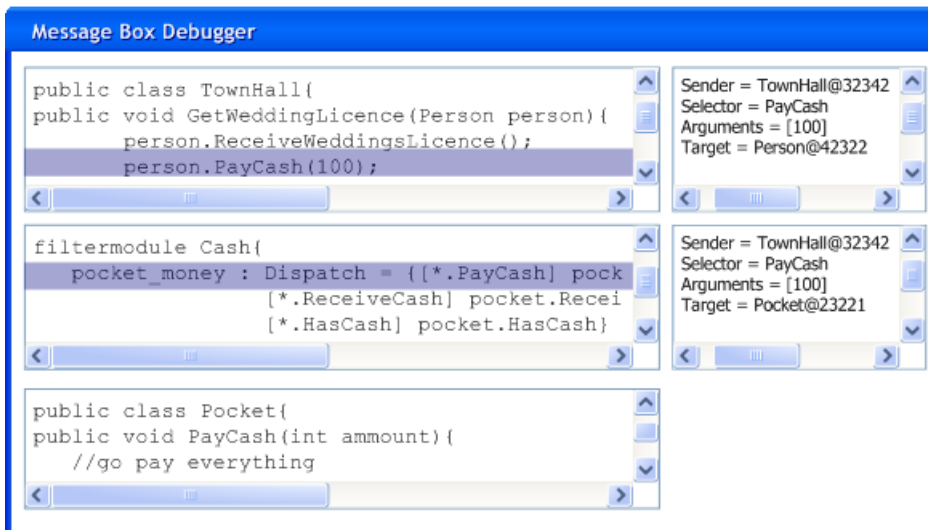


Figure 6.26: Break after the modified breakpoint Composition Debugger

Chapter 7

Implementation Design

*“You can put the combustion chamber of a car in the tire,
but that kind of car would be very difficult to build.”*
Mehtmet Akşit

In this chapter we implement the in [Chapter 6](#) described debugger into a software design which can be integrated into the Compose[★] framework. This chapter uses the *Not Yet Another Method(NYAM)* design method as described in ‘Design Methods for Reactive Systems’ by J.R.Wieringa [[Wie03](#)]. This method begins with the description of the business context of the project. This context consists of the business goals and the functional decomposition. The functional decomposition is followed by the specification of the functional requirements. The method then continues by classifying the type of the project as a reactive system. Depending on the type of the project, different analysis and specification methods need to be used. By using these specification methods, the technical requirements are specified. Then an information analysis of the environment is performed. The information analysis is supported by the designed architecture. This architecture is then implemented into a software design. This software design is finally implemented into a real debugger.

7.1 Business Context

The debugger is called *CODER* (COmposition filters DebuggER).

The general goal of the debugger is, to assist in the debugging process of composition filter programs within the Compose[★] framework. The responsibilities of the debugger include:

- Debugging the behavior of the composition filter source code.

- Debugging the behavior at the joinpoint where composition filter code and non-composition filter code connect.

The responsibilities of the debugger exclude:

- Debugging the behavior of non-composition filter source code.
- Debugging the behavior of the Compose[★] framework.

According to the insights described in this thesis, the goal can be achieved by supporting halting the execution and filter representations. As can be seen in [Figure 7.1](#), for supporting execution halting we require LTL Breakpoints, where the LTL Breakpoints require the validation of LTL formulas and the halting of the program execution. For the representation of the filters we need to be able to inspect the halted state. In order to be able to validate LTL formulas, we need to be able to evaluate the propositions. To be able to halt a program, we need to be able to stop and to resume a program.

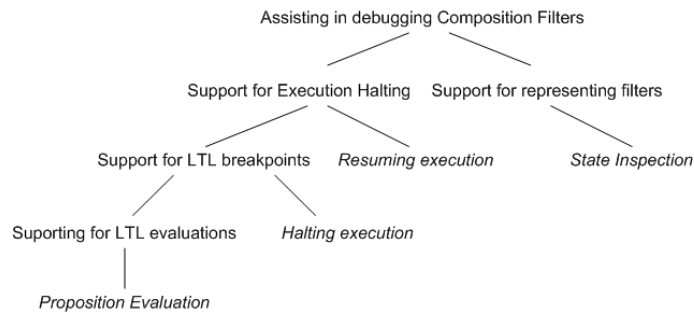


Figure 7.1: Goal tree of the CODER business context.

7.2 Functional Requirements

Because of the business context, the debugger needs to comply to the following functional requirements:

- The debugger needs to support the specification and handling of LTL breakpoints
- The debugger needs to support the execution navigation of filters as described in [Section 6.3](#)
 - The debugger needs to support the halting of a programs execution
 - The debugger needs to support for resuming the programs execution
- The debugger needs to support the represent of filters as described in [Section 6.2.10](#)

7.3 Project Classification

We are now going to classify the debugger as a reactive or transformational system. This classification is important because reactive systems differ from transformational systems in terms of complexity and behavior and thus require different design methodologies. Examples of reactive systems are: Real-time, embedded and control systems, ERP, and informational systems. In these kinds of systems the environment is analyzed. Examples of transformational systems are: Compilers, translators, generators, and Routines in mathematical libraries.

7.3.1 Interaction with the environment

Reactive systems are systems which are in constant interaction with its environment. Transformational systems interact with its environment, but they just acquire sufficient information to execute its tasks. A transformational system would therefore not maintain a state of affairs in its environments while a reactive system would

A debuggers job is represent the state of the subject. This means that the debugger needs to monitor the state of affairs within the subject. The classification of the interaction with the environment is therefore defined as transformational system like.

7.3.2 Termination of the process

The process of a reactive system is normally nonterminating. If a reactive system terminates it is considered a failure. A transformational system should produce its output and then terminate. If it does not terminate it is considered a failure.

The debugger should terminate after the client program terminates. A debugger should not keep on running. The classification of the process termination is therefore defined as transformational system like.

7.3.3 Interrupt driven

Reactive systems are systems which are in constant interaction with its environment. It therefore needs to act on stimuli regardless of what it's doing. Transformational systems react on sequential oriented input. Transformational systems, therefore allow the queuing of the input.

The hitting of a breakpoint could be seen as a stimuli. Such a stimuli however does not require an action regardless of what the debugger is doing.

The classification of the behavior being interrupt driven is therefore defined as transformational system like.

7.3.4 State-dependent response

The response of a reactive system depends on the internal state and the external event that it reacts to. This response can leave the system in another state than it was. A transitional system has no state that is dependent on external entities. And thus a transitional system will just deliver output that depends on the input given.

LTL is evaluated in computer programs by converting it to a buchi automata and then check if the final state is accepting. the behavior of this automata can be seen as a response depending on the internal state like a reactive system. The classification of the state-dependent response is therefore defined as reactive system like.

7.3.5 Environment-oriented response

The response of a reactive system, results in the modification of the behavior of the environment. The output of a transformational system does not have direct influence on its environment.

A goal within the implementation of a debugger is not to influence the observed behavior of the subject, unless the programmer wants it to. If the subject's observed behavior is influenced by side effects of the debugger, the results can be contaminated. This would make the results less valuable for debugging.

The programmer may also choose to intentionally change the behavior of the subject during debugging. This change can be the breaking of a program or even changing its source code during execution. The classification of the environment-oriented response is therefore defined as reactive system like.

7.3.6 Parallel processes

Reactive systems usually have many parallel processing interactions. Transformational systems usually have sequential processing interactions.

The behavior of many filter actions are sequential interactions. There are however also Meta and custom filters which require parallel interaction and the client programs could be parallel as well. The classification of the behavior as having parallel processes is therefore defined as reactive system like.

7.3.7 Real-Time constraints

Reactive systems usually have real-time constraints. Transformational systems usually do not have real-time constraints.

The program is needed to perform within human acceptable time limits but it has not got “hard” real-time constraints. The classification of the real-time constraints is therefore defined as transformational system like.

7.3.8 Conclusion

The debugger has many characteristics of a reactive system. It is therefore wise to treat it as a reactive system.

Because the debugger is classified as a reactive system it is needed to use strict data analyzing and design methodologies instead of agile ones to achieve quality of the end product. Functional decomposition in these kinds of systems is not the only important design approach; also the structure of the environment in conjunction with the end product must be taken into account because a reactive system is highly dependent on its environment.

It should however be noted that the compile time of the Compose★ framework is a transformational system and should be treated as such. The methods that are being used in this thesis can possibly therefore not be suitable for the compile time.

7.4 Technical Requirements

The debugger needs to comply to the following technical requirements:

- The debugger should run on the Microsoft .NET CLR and Sun Java VM. The debugger should be able to run on most .NET CLR’s (Mono) and Java Virtual Machines (IBM, BlackRock).
- The debugger should integrate with Compose★
- The debugger should be able to later integrate with StarLight

7.5 Information Analysis

For the information analysis of the environment we use the *Nijssens Informatie Analyse Methode(FCO-NIAM)* method. FCO-NIAM is a method based on the structuring of natural language. It is a method best suited for analyzing the static structure of dynamic and static information. It is therefore used for the analysis of the structure that make up the internal state.

7.5.1 Step 1: Assumptions

This step is to document any assumptions which the domain expert can validate. Because the derived model is based on these assumptions, they need to be correct.

We use the assumption; that the concepts used in the system are the same as the concepts of Compose★, as described in [Section 2.2](#).

7.5.2 Step 2: Describing

This step is to describe the domain in a language which is natural for the domain expert. Since this text is written by a domain expert it can be deemed correct. Read [Section 2.2](#) for a description of composition filters in Compose★.

7.5.3 Step 3: Generalization

This step is the transformation of the natural language in generic sentences which allow for a generic representation of the information.

- Application A has Concern C
- Concern C is identified by Name N
- Concern C has Filter module FM
- Concern C has Implementation I
- Thread T send Message M to Joinpoint JP
- Message List ML contains Message M
- Object O is sender Message M
- Message M target Object O
- Message M has Selector S
- Filter Module is Name N
- Filter Module FM contains Filter F
- Filter F is Name N
- Filter F is of Filter Type FT
- Filter F contains FilterElement FE
- Filter Type FT has RejectAction AC

- Filter Type FT has AcceptAction AC
- Filter Element FE has ConditionPart CP
- Filter Element FE has MatchingPart MP
- Filter Element FE is composed with Filter Element FE1
- ConditionPart CP has a Condition CO
- ConditionPart CP has a ConditionLiteral CL
- Super Imposition SI superimposes FilterModule FM
- Super Imposition SI superimposes on Joinpoint JP

7.5.4 Step 4: Cardinality

This step is done to analyse the relations and cardinality of those relations. Because this step is in natural language, these cardinalities can be checked by the domain expert.

- an Application can have multiple Concerns
- multiple Applications can have the same Concern
- a Concern is identified by one Name
- multiple Concerns cannot be identified by the same Name
- a Concern can have multiple Filter modules
- multiple Concerns can have the same Filter module
- a Concern can have multiple Implementations
- multiple Concerns can have the same Implementation
- a Thread contains multiple Joinpoints
- multiple Threads contain the same Joinpoint
- a Thread cannot send the same Message to multiple Joinpoints
- a Thread can send multiple Messages to the same Joinpoint
- multiple Threads cannot send the same Message to the same Joinpoint
- multiple Threads cannot send the same Message to multiple Joinpoints
- multiple Threads can send multiple Messages to the same Joinpoint
- multiple Threads can send multiple Messages to multiple Joinpoint

- a Message List contains multiple Messages
- multiple Message Lists cannot contain the same Message
- an Object is sender multiple Messages
- multiple Objects cannot be the sender of the same Message
- a Message cannot target multiple Objects
- multiple Messages can target the same Object
- a Message cannot have multiple Selectors
- multiple messages can have the same Selector
- a Filter Module cannot have multiple Names
- multiple Filter Modules cannot have the same Name
- a Filter Module contains multiple Filters
- multiple Filter Modules cannot contain the same Filter
- a Filter cannot have multiple Names
- multiple Filters cannot have the same Name
- a Filter contains multiple FilterElements
- multiple Filters cannot contain the same FilterElement
- a Filter cannot have multiple FilterTypes
- multiple Filters can be of the same Type
- a FilterType cannot have multiple RejectActions
- multiple FilterTypes can have the same RejectAction
- a FilterType cannot have multiple AcceptActions
- multiple FilterTypes can have the same AcceptAction
- a Filter Element accepts multiple Message Lists
- multiple Filter Elements accept the same Message List
- a Filter Element cannot have multiple ConditionParts
- multiple Filter Elements can have the same ConditionPart
- a Filter Element cannot have multiple MatchingParts
- multiple Filter Elements can have the same MatchingPart
- a Filter Element cannot be composed with multiple Filter Elements

- multiple Filter elements cannot be composed with the same Filter Element
- a Filter Element is not composed with itself
- a ConditionPart cannot have multiple Conditions
- many ConditionParts can have the same Condition
- a ConditionPart cannot have multiple ConditionLiterals
- many ConditionParts has the same ConditionLiteral
- a Super Imposition can superimpose multiple FilterModules
- multiple Super Imposition can superimpose the same FilterModule
- a Super Imposition can superimpose on multiple Joinpoints
- multiple Super Impositions can superimpose the same Joinpoint

7.5.5 Step 5: Transformations

This step is a model transformation of the NIAM generic model to the intended target model to allow the representation of the information. In this case we define the NIAM

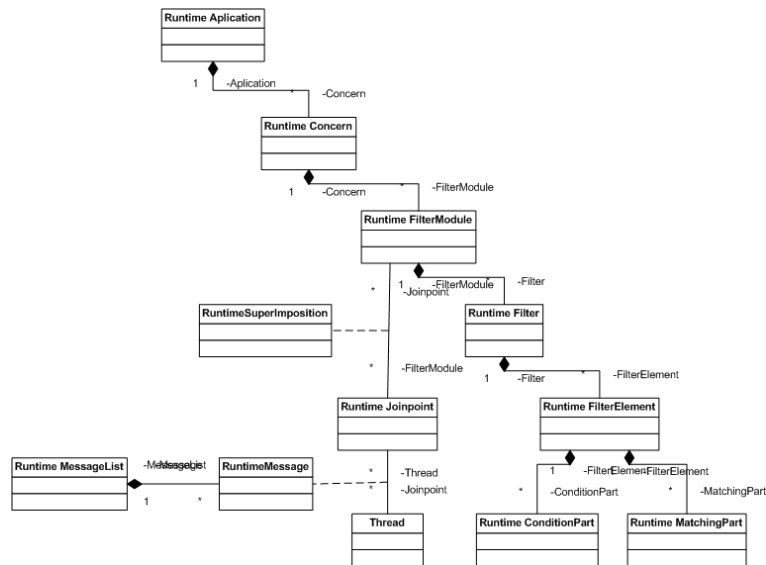


Figure 7.2: Components of the composition filters model

language elements as classes. We define the rules composed of the language elements as the associations between the classes. The end nodes will become the properties of the classes. Since the runtime is running the application to which the concerns are referenced the concept of multiple Applications is dropped in the transformation. The result of this transformation is a class diagram of the model structure as seen in Figure 7.2.

7.6 Use Cases

The use cases are divided in two perspectives; the use cases from the programmers and the programs perspective.

7.6.1 Programmers perspective

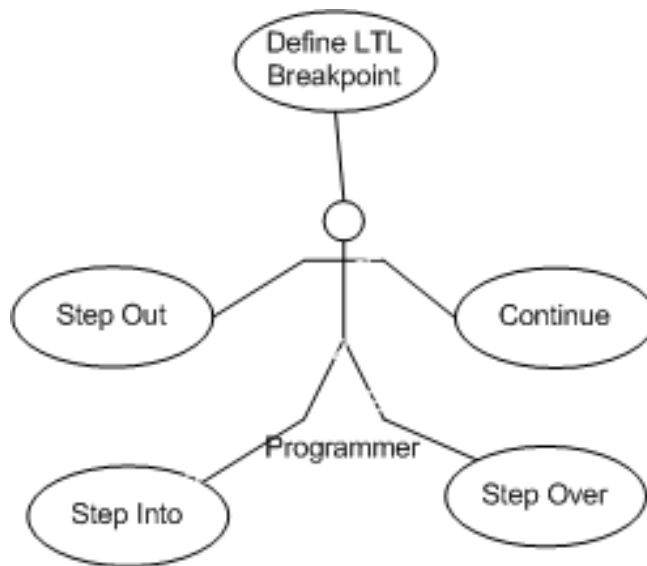


Figure 7.3: Use Cases of the debugger from the programmers perspective

From the perspective of the programmer the debugger has, as can be seen in [Figure 7.3](#), five actions from the functional requirements:

Define LTL Breakpoint

Is the action which is performed when changing the LTL formula of the breakpoint.

Continue

Is the action which is performed when the programmer wants to continue to a next break in the execution.

Step Over

Is the action which is performed when the programmer wants to continue to the next message from the same sender

Step Into

Is the action which is performed when the programmer wants to continue to the inspection of executing code of the target

Step Out

Is the action which is performed when the programmer wants to continue to the joinpoint where the target is currently the sender

7.6.2 Program perspective

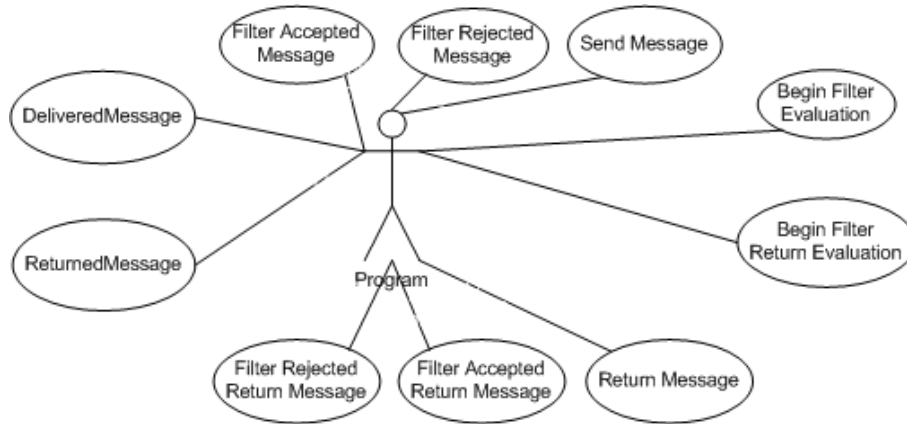


Figure 7.4: Use Cases of the debugger from the program perspective

From the perspective of the program the filter evaluation, as can be seen in Figure 7.4, the use cases it consists of ten actions;

Sending Message

Is the action the sender performs to send a message to the target.

Filter Evaluation

Is the action where the filter evaluates its condition and matching part.

Filter Accepts Message

Is the action where the filter has accepted the message and now executes his accept filter action.

Filter Rejects Message

Is the action where the filter has accepted the message and now executes his reject filter action.

Message delivered

The message finally gets to its target.

Returning Message

Is the action the target (implicit) return of the message to the sender.

Filter Evaluation Returning

Is the action where the filter evaluates its condition and matching part.

Filter Accepts Returning Message

Is the action where the filter has accepted the (implicit) returning message and now executes his accept filter action.

Filter Rejects Message

Is the action where the filter has accepted the (implicit) message and now executes his reject filter action.

Message returned

The message has finally returned to its sender.

7.7 Architecture

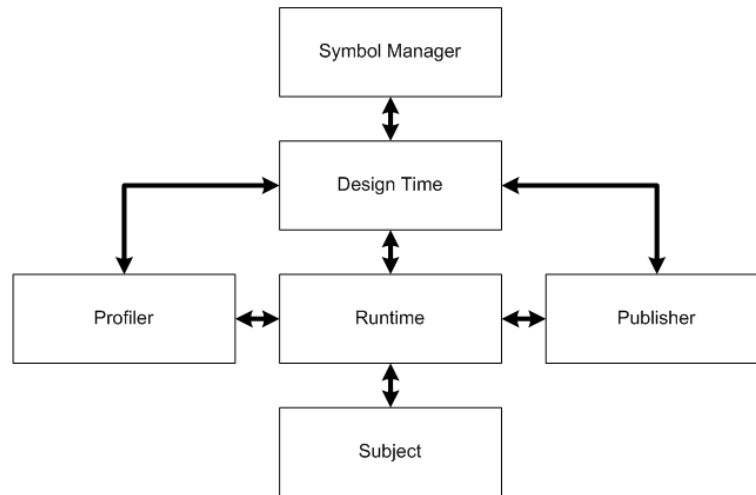


Figure 7.5: General architecture of a debugger

As can be seen in [Figure 7.5](#), the general architecture of a debugger consists of 6 elements [ARF02]. These elements work together to provide the representation of the current system state and the state transitions.

Subject

The system in which the insight of its execution is desired is called the subject. In most software development cases the subject is the program being debugged.

Runtime

The runtime facilitates the execution of the subject. In most software development cases the runtime subject is usually the operating system or virtual machine. The runtime offers a debugging interface which can be used to monitor and influence the subject.

Profiler

The profiler monitors the subject for changes of the internal state of the subject. This monitoring is done with the use of the runtime. The most common representation in use today for a state in computer programs, is the current execution location and the current values of the stack within the software program. These changes in the programs state are combined to form an execution trace. Execution traces can be built up on the fly, in case of real time monitoring, or at once, in the case of execution signature dumping. The most important criterion of the profiler is, that it monitors the subject without changing its behavior. If a profiler would change the behavior of the subject, the insight of the subjects behavior with respect to the source code will be reduced.

Symbol manager

The execution traces produced by the profiler contain states. These states are based on executing code. Often the executing code is not the source code but the compiled code. Because the compiled code is a translation of the source

code there is a relation between them. The relation between the lines of the compiled code and the corresponding lines within the source code is called a symbol. Multiple lines of source code can be compiled into one line of compiled code, and one line of source code can compile to multiple lines of compiled code. Therefore a symbol is an many to many relation. A compiler usually creates more then one symbol. these symbols are managed by the symbol manager.

Design Time

The design time combines the execution state with the symbols from the symbols manager to provide reasoning about a state. The design time can then use the information from the reasoning to influence the subject by using the runtime. Possibilities of such an influence are to halt the execution of the subject, modify the subject's current state (variable editing), and to modify the subject's source code ("stop, change and go").

The Publisher

The publisher is the user interface for the developer. This interface displays a representation of the system state by using information from the design time.

7.8 Sequences

We no are going to highlight how the architecture components interact with each other.

7.8.1 Define LTL Breakpoint

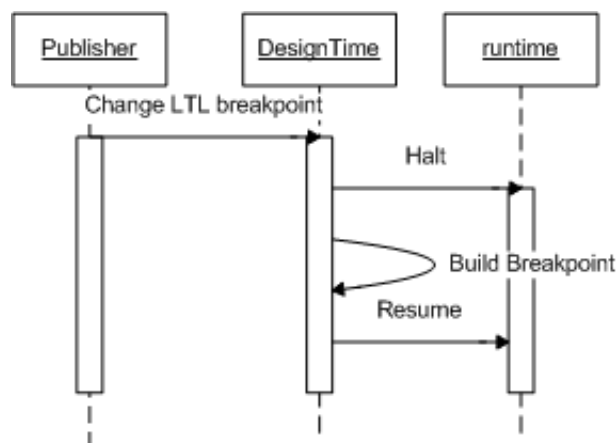


Figure 7.6: Sequence diagram of the LTL breakpoint change

As can be seen in [Figure 7.6](#), when the programmer defines a LTL breakpoint the publisher notifies the design time. The design time will then notify the runtime to halt the execution of the subject. The design time will then parse the specification to build a breakpoint. The design time will then notify the runtime to resume the execution.

7.8.2 Program actions

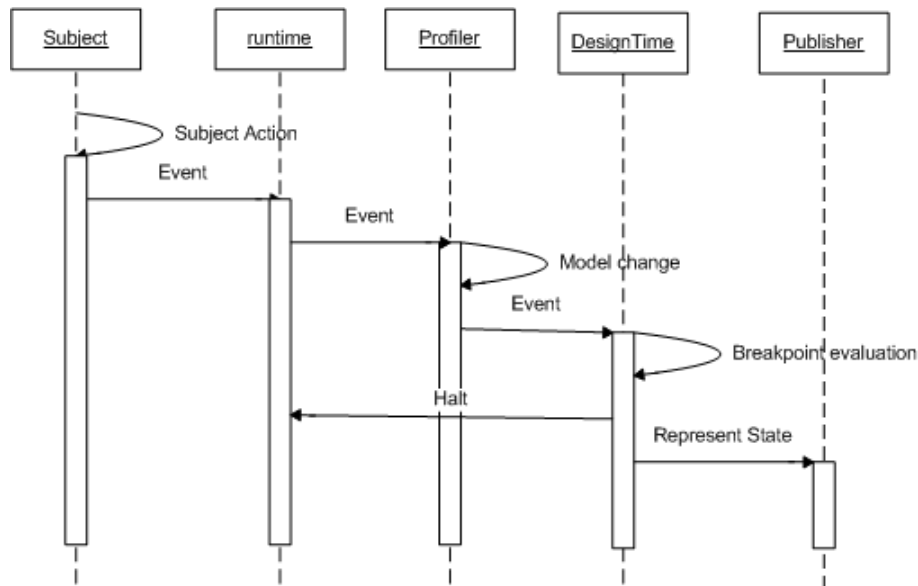


Figure 7.7: Sequence diagram of the filter evaluation actions

As can be seen in Figure 7.7, the ten actions of the programs filter evaluation are treated equally. They all raise an event on the runtime, where the runtime notifies the profiler of a model change. This model change raises an event at the design time. In the design time the breakpoint conditions are evaluated. If the evaluation of the breakpoint condition prompt a halt, the design time notifies the runtime to halt the subject. After the subject is halted, the publisher is activated to represent the state of the subject.

7.8.3 Continue, Step Over, Step Into, Step Out

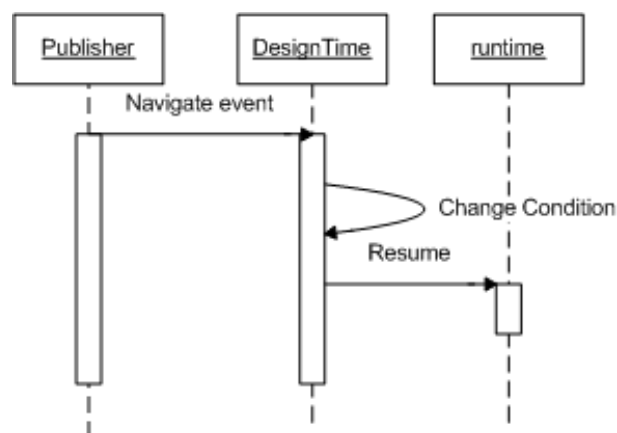


Figure 7.8: Sequence diagram of the execution navigation actions

The four execution navigations: Continue, Step Over, Step Into, and Step Out, get treated the same way. As can be seen in [Figure 7.8](#), when the programmer selects a navigation, the publisher notifies the design time. The design time will then change the conditions to reflect the navigation. The design time then will notify the runtime to resume the execution.

7.9 Component Design

We now are going to implement the 6 components of the architecture into a design. The Compose★ framework has a version with a filter interpreting runtime and a version which inlines filters.

The Compose★ version that uses an interpreting runtime to execute the filter behavior is the oldest implementation of the framework. During the compilation of a Compose★ program, function calls to the runtime are weaved into the target program. These function calls are called hooks. The runtime then executes the filter behavior at the appropriate moments.

7.9.1 Subject

The subject is the system in which the insight of its execution is wanted. Since we do not have any control over the subject we do not have a design for it.

7.9.2 Runtime

The runtime facilitates the execution of the subject. In the case of Compose★/.NET and Compose★/Java the execution is within a virtual machine. We can use the reflection API from the virtual machine to inspect and influence the subject.

To halt the execution of the subject is a different matter. While it is technically possible to halt a virtual machine, there are however some practical issues. Because the technical requirements requires a software design which does not require a specific virtual machine debugging interface, the debugger runs in the same virtual machine as the subject. If the virtual machine would be halted, the debugger would halt as well. We therefore halt the program by halting the threads of the program. Because we can only influence the program at the joinpoints, we halt the program by halting the threads that hit a joinpoint.

The halting behavior is done by the Halter class. This class contains a mutex which can be set or unset by the debugger and only read by the program under debug. Every time the program hits a join point the `halting` function is called. If the mutex is not set, the program then continues. When the debugger sets the mutex, the program waits until the mutex is unset.

7.9.3 Profiler

The profiler monitors the subject for changes of the internal state of the subject. Profiling is done by the DebugInterface class. This class is a facade for the model. The functions of this class are the possible changes within the evaluation of the messages.

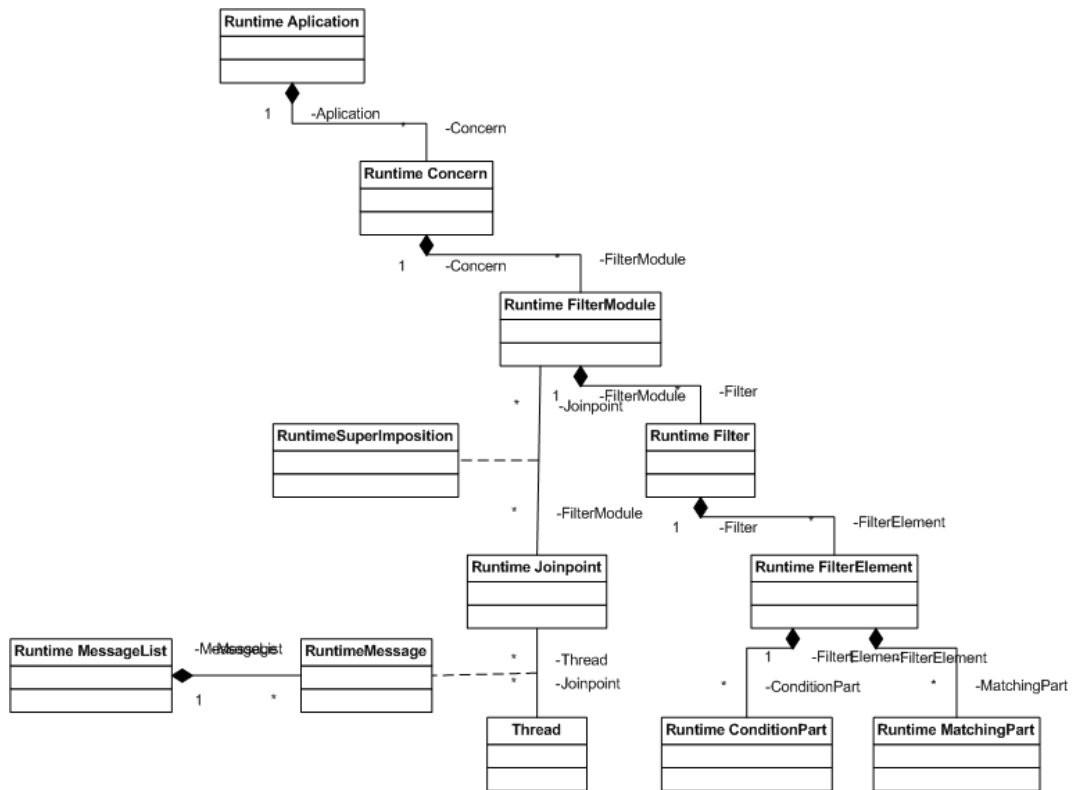


Figure 7.9: Components of the composition filters model

Shown in Figure 7.9 is the model which is already described in the information analysis. Because the evaluation is a change over an amount of time we add history sensitivity to the model. As can be seen in Figure 7.10 this is done by using the memento pattern.

7.9.4 Symbol manager

As written earlier, the symbol manager manages symbols. However within Compose*/.NET and Compose*/Java the virtual machine already maps the symbols for us. We therefore do not need a symbol manager to read the symbols.

However in the case of aspect oriented software the compiler weaves advice into the target program. Because the weaver Compose*/.NET uses IL manipulation the symbols need to be updated within the weaving process.

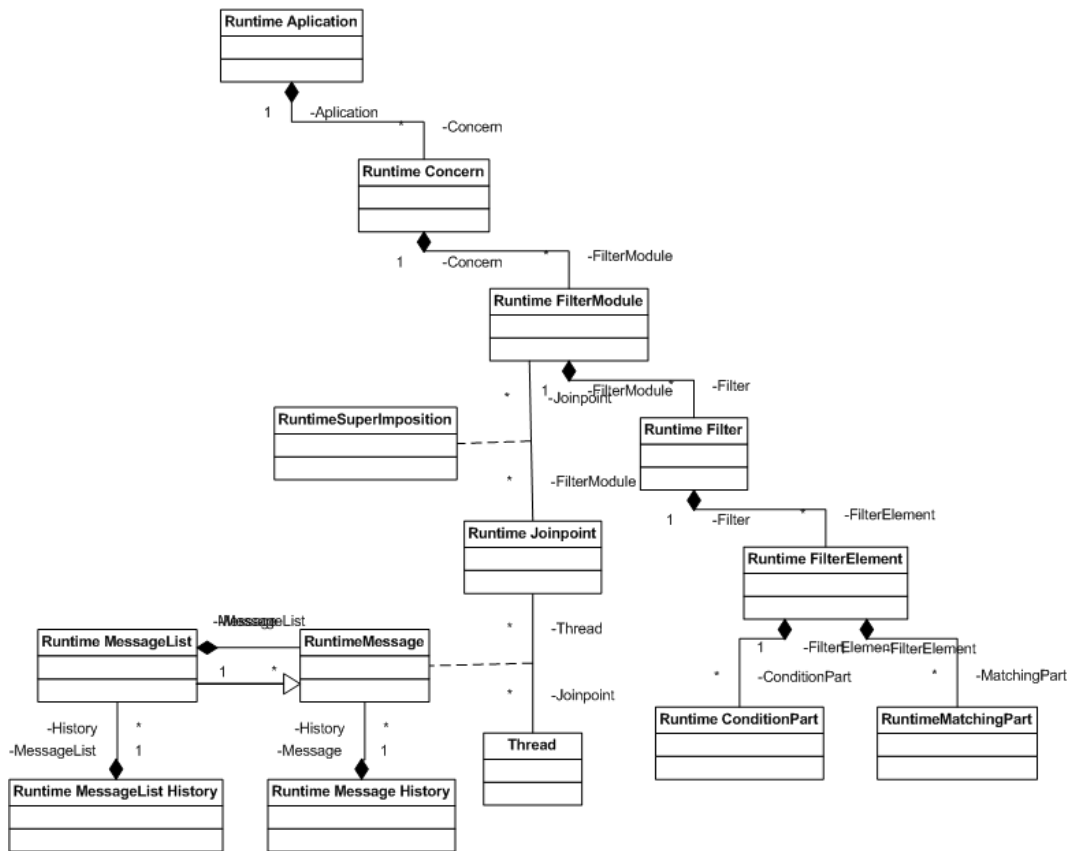


Figure 7.10: Memento applied on the components of the composition filters model

In the runtime version of Compose^{*}, only hooks to a runtime are woven into the compiled code. The hooks in the compiled code therefore need to be translated back to the source code. This translation is then performed by a symbol manager.

In 2006 a filter inlining version of Compose^{*} was built with the codename *StarLight*. Filter inlining means that the code needed for the filter behavior is weaved into the program instead of hooks which call a runtime. The advantage of inlining filters is mostly speed because it does not have the overhead of a runtime.

In the inlined filter version of Compose^{*}, when the filters are inlined, in the resulted compiled code lines need to correspond to the correct source code lines in the symbols. For the inlined filter version we therefore do not need a special symbol manager.

7.9.5 Design Time

The design time combines the execution state with the symbols from the symbols manager to provide reasoning about a state. Because our LTL breakpoints are behavioral breakpoints, we need the reasoning of the design time to validate them.

The way we support LTL breakpoints is by interpreting them as a buchi automata. The behavior of the program is then executed on the automata to see if a final state is reached. If a final state is reached, the subject is halted. The buchi automata is encapsulated by the Breakpoint class. By using the composite pattern the different kinds of breakpoints can be implemented. Every time the program hits a joinpoint the `checkBreak` function is called on the breakpoint. If the breakpoint is in a state that the subject needs to be halted, the function returns true. The design time will then halt the subject by using the runtime as described in [Section 7.9.2](#).

7.9.6 The Publisher

The publisher is the user interface for the developer. The publisher uses the Model View Controller architecture.

Model

The model within the publisher is the design time.

View

The view within the publisher displays the representation of the filter evaluation.

Controller

The controller within the publisher are the buttons for going into the filter evaluation or to the next break. The controller uses the runtime to achieve this.

Conclusion and Future Work

*“We can only see a short distance ahead,
but we can see plenty there that needs to be done.”*
Alan Turing

This chapter concludes this thesis. It discusses related work, contains conclusions, and suggests directions for future work.

8.1 Related work

In this section we discuss certain projects and implementations that are related to the subject of this thesis. While there is much more work which can be related, there is not enough space and time to discuss them all. Therefore we only discuss a selection.

8.1.1 Aspect Slicing

Aspect slicing is a debugging technique devised by Takashi Ishio, Shinji Kusumoto, and Katsuro Inoue [IKI04]. It makes use of call graphs to isolate incorrect behavior in aspect oriented programs. By limiting the debugging to the isolated behavior the bugs can easier be found. Aspect slicing is related, because it is about debugging aspects.

It is different from the technologies discussed in our approach because aspect slicing is a debugging technique using a static analysis. Because the program being debugged is a Turing complete program, slicing aspects may help in the isolation of the fault, but it is not able to find all faults. This is a limitation on any static analysis posed by the halting theorem. Also the question of discriminating intended from unintended behavior is not answered. Aspect slicing however can be used in conjunction with

LTL breakpoints to reduce the amount of overhead caused by the LTL breakpoints. This would make LTL breakpoints more usable for aspect and non-aspect oriented languages.

8.1.2 NAspect

NAspect [NAs05] is an aspect oriented framework that uses dynamic proxies. The debugging functionality of NAspect is the visualization of the dynamic proxies. Its visualization is based on abstract aspect representations. NAspect is related, because instead of a source iterator it has, like the composition filters debugger, an abstract aspect representation.

The difference is that the focus of debugging within NAspect is on the values within the objects instead of the evaluation behavior. They focus on the values, because NAspect uses a different conceptual model which only has proxy redirects on the basis of these values. Within the debugging of NASpects they therefore only represent the internal values. This representation makes it difficult to see why the source code has the dynamic behavior. Because NASpects are not stutter equivalent, as defined by Browne et al [MCBG88], debugging needs to be on a statement for statement basis, which can cause an overload to the programmer.

8.1.3 SPIN

SPIN [Hol03] is an open-source model checker that can be used for the formal verification of distributed software systems. It is in use today by thousands of people worldwide. The tool was developed in 1980 at Bell Labs in the original UNIX group of the Computing Sciences Research Center and has been supported ever since.

SPIN is related because it also uses LTL for the specification of intended behavior. SPIN uses state pruning to verify that the program complies with this intended behavior. SPIN also makes use of stutter equivalence to reduce the amount of states to be validated.

The difference with the composition debugger is that the LTL formula is used for isolating a fault in a given execution, while model checkers validate all paths, thus validating all possible executions. Because real programs are bigger than models of the software, validation of real source code is difficult and time consuming. This makes model validation good for debugging models and in a lesser extent the debugging of the real source code. Since the debugger uses LTL formulas, the formulas can be reused to validate an execution of the implementation against a model. By validating the differences between the model and the implementation, an incorrectness within the model or implementation can be found.

8.1.4 AspectJ Debugger

AspectJ [CC05] is an aspect-oriented programming language based on the Java programming language. AspectJ(tm), which is part of the eclipse project, is the most recent implementation of the AspectJ language. AspectJ(tm) includes a compiler (ajc), a debugger (ajdb), a documentation generator (ajdoc), a program structure browser (ajbrowser), and integration with Eclipse, Sun-ONE/Netbeans, GNU Emacs/XEmacs, JBuilder, and Ant. The debugger of AspectJ(tm) is related because it is, like our debugger, a debugger for an aspect oriented language.

The difference is that the AspectJ language is not stutter equivalent, as defined by Browne et al [MCBG88]. Because the language is not stutter equivalent debugging needs to be on a statement for statement basis. The ajdb of AspectJ(tm) is therefore a source iterator debugger like the representation shown in Section 6.2.3. LTL breakpoints could be used to debug AspectJ programs. When using LTL breakpoints to debug AspectJ programs, the elements used in the LTL formula need to be adapted to support the AspectJ language.

8.1.5 Control-flow Breakpoints

Control-flow Breakpoints [CV07] is another method for setting breakpoints on the behavior of a program. Control-flow breakpoints are related because LTL Breakpoints are a type of Control-flow breakpoints.

The difference is that the proposed breakpoint specification language by Rick Chern and De Volder is based on the sequence of joinpoints. When a certain sequence occurs in the execution the breakpoint is halted. The drawback of defining a breakpoint in such a manner, is that you already need to know the source location where you want to stop. Essentially you already suspect where your bug is which reduces its usefulness as described in Chapter 5. By using use cases they come to the conclusion, that their specification languages needs to be modified. LTL breakpoints are devised from the conception that the programmer does not know where to stop in the program. LTL breakpoints are therefore a sequence of propositions instead of specified joinpoints. This results in more usable joinpoints.

8.2 Conclusion

In this thesis we searched for ways to debug programs which are implemented using Composition Filters. In Chapter 4, we described debugging as the process of removing bugs. While debugging, the difficulty in removing the bugs, was caused by an insufficient comprehension of the programs incorrect execution behavior by the programmer. In essence, the programmer does not understand why the program behaves in a incorrect way, because its behavior at execution is too complex in reflection with the source

code.

It is quite interesting that by specifying the crosscutting concerns in an isolated manner, the execution behavior becomes more difficult to understand. In essence, the most important reason aspect-oriented programs are harder to debug than non-aspect-oriented programs, is a direct consequence of solving crosscutting by using aspect-orientation.

Besides the increased difficulty we also found that the conventional debugging tools and techniques are less suitable for use within aspect oriented programming. An example is the use of a conventional breakpoint. When using a conventional breakpoint within advice, the scattering of the advice may cause the breakpoint to hit very often, thereby causing an overload to the programmer.

In [Chapter 5](#) we stated that not only does the programmer not know why his program behaves in an incorrect way, the programmer also does not even know how his program should be implemented to behave correctly. This view is different from the common opinion that the programmer always knows how his program should be implemented to behave certain way.

We approached this issue by allowing the programmer to set breakpoints on the behavior of a program, to allow the programmer to see how the execution of the implementation behaves. This behavior of the program can be specified with use of the proposed LTL breakpoints. By using LTL breakpoints the amount of program halts can be reduced.

Because of the sound conceptual model of Composition Filters, if the side effects of a filter are limited to a message, the behavior becomes stutter equivalent. This stutter equivalent behavior can be used to reduce validations and allows a representation of a complete filters evaluation at a joinpoint. This representation of the complete filter evaluation, is a representation over an amount of time instead of a just snapshot. This increased time window of the representation improves comprehensiveness of the filter executions, because it allows the programmer to instantly see the effects of the filter actions. When a side effect is not limited to the message, the difference between the simulation and the real filter evaluation can help the programmer comprehend the behavior.

It must however be concluded that being stutter equivalent is a property not many other aspect oriented languages hold. This implies that the debugging representation of other aspect oriented languages probably is limited to step by step debugging. This step by step debugging is also known as the conventional, source iteration trace debugging. These aspect oriented languages however can make use of the proposed LTL breakpoints to reduce breaks and isolate the fault.

While much money is spent on debugging, it is strange that there is not so much research concentrated on debugging. Most research is concentrated on making the programming languages consistent and easier to express certain solutions. Only in the field of model validation and testing is the avoidance of bugs the ultimate goal. Most debuggers are therefore based on the experience of developers using the programming

languages. While it can be concluded that with the usage of good breakpoints and representations the debugging of aspect oriented programs becomes easier. However, debugging is still limited by the capabilities of the developer using them. Since the developer only knows how he intended the program to behave, this limitation is impossible to circumvent.

8.3 Future Research

Because the side effects of some filter actions remain limited to the message, the behavior of the filter actions can therefore be described as being stutter equivalent. This stutter equivalent behavior can be used to reduce the overhead in the reasoning FIRE does within Compose[★] about filter evaluations. For instance, the method of partial ordering reduction can be used. This reduction of reasoning overhead, may allow us to do more reasoning about the program and therefore find more semantic bugs. It can also be used in model validations of applications where filters have been superimposed.

Because of the added value of LTL breakpoints it is advisable to see if the usage of LTL breakpoints can also have positive debugging capabilities in other aspect and non-aspect oriented programming languages. Since the evaluation of the LTL breakpoints can be seen as a crosscutting concern it may be possible to enhance the debugging capabilities by implementing these invariants as filters. In this way debugging filters can be used to debug composition filters and non-composition filter source code.

Even with the most advanced debuggers today, the isolation of the fault is still the most daunting task. More fundamental research needs to be done on ways to easier isolate a fault. The focus of such research, can be on debuggers which represent *why* certain behavior is happening, instead of *what* is happening. While the technologies we propose, are a step in the right direction, much more research needs to be done in order to create the perfect debugger.

Appendix **A**

Source Code Wedding Example

Appendix B

Non serious representations

Pacman representation

Since the most used example to illustrate features in Compose* was the Pacman example we tried this as a representation of filters (Figure:B.1). Things like meta filters were hard to represent in this way so we dropped it.

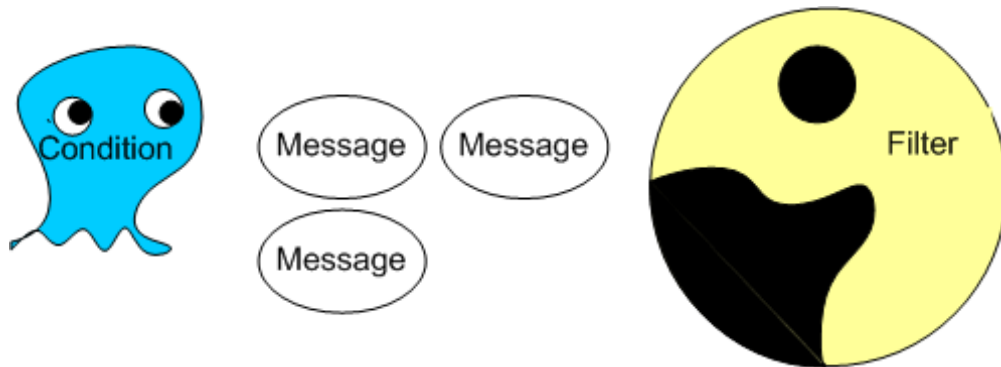


Figure B.1: The pacman representation

Coffee representation

When we asked some software engineers: "What is the first thing that comes up into your mind when you say filters?". The answer was short and powerful: "Coffee"! Therefore this coffee representation (Figure:B.2) was noted, but it was dropped because it lacked expression power.

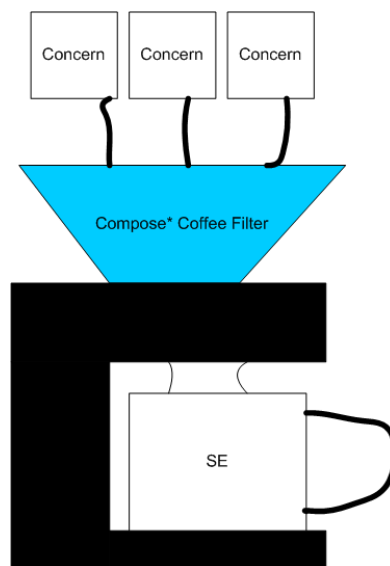


Figure B.2: The Compose* coffee filter representation

Bibliography

- [Ada96] Ada for the web, 1996.
- [ARF02] Toby J. Theorey Ann R. Ford. *Practical Debugging in C++*. Pearson Education, 2002.
- [BA01] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, October 2001.
- [Ber94] L. Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994.
- [Bin98] D.C. Binnema. Visual composition filters. Master’s thesis, University of Twente, 1998.
- [Bos04] Raymond Bosman. Automated reasoning about Composition Filters. Master’s thesis, University of Twente, The Netherlands, November 2004.
- [Bos06] Sverre R. Boschman. Performing transformations on .NET intermediate language code. Master’s thesis, University of Twente, The Netherlands, August 2006.
- [CC05] A. Colyer and A. Clement. Aspect-oriented programming with aspectj. *IBM Syst. J.*, 44(2):301–308, 2005.
- [Con06] Olaf Conradi. Fine-grained join point model in Compose*. Master’s thesis, University of Twente, The Netherlands, August 2006.
- [CV07] Rick Chern and Kris De Volder. Debugging with control-flow breakpoints. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 96–106, New York, NY, USA, 2007. ACM Press.
- [Doo06] Dirk Doornenbal. Analysis and redesign of the Compose* language. Master’s thesis, University of Twente, The Netherlands, October 2006.
- [Dür04] Pascal E. A. Dürr. Detecting semantic conflicts between aspects (in Compose*). Master’s thesis, University of Twente, The Netherlands, April 2004.

- [EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Comm. ACM*, 44(10):29–32, October 2001.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison Wesley, 1995.
- [GL03] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley and Sons, 2003.
- [Gla95] M. Glandrup. Extending C++ using the concepts of composition filters. Master’s thesis, University of Twente, 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.
- [Hol03] Gerard J. Holzmann. *The SPIN MODEL CHECKER, Primer and Reference Manual*. Addison Wesley, 2003.
- [Hol04] Frederik Jacob Bouwhof Holljen. Compilation and type-safety in the Compose* .NET environment. Master’s thesis, University of Twente, The Netherlands, May 2004.
- [Hui07] Rolf L. R. Huisman. Debugging Composition Filters. Master’s thesis, University of Twente, The Netherlands, February 2007.
- [Hut06] Stephan H. G. Huttenhuis. Using aspect-oriented programming to solve problems of design pattern implementations. Master’s thesis, University of Twente, The Netherlands, 2006. November.
- [IKI04] Takashi Ishio, Shinji Kusumoto, and Katsuro Inoue. Aspect debugging through slicing. Technical report, Osaka University, 2004.
- [Int02] ECMA International. Common language infrastructure (CLI). Standard ECMA-335, ECMA International, 2002.
- [Jyt] Jython homepage.
- [KHH⁺01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.
- [Koo95] P. Koopmans. Sina user’s guide and reference manual. Technical report, Dept. of Computer Science, University of Twente, 1995.
- [MCBG88] E. M. Clarke M. C. Browne and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theor. Comput. Sci.*, 59(1-2):115–131, 1988.
- [Mic03a] Microsoft Corporation. .NET compact framework - technology overview. Technical report, Microsoft Corporation, 2003.

- [Mic03b] Microsoft Corporation. Overview of the .NET framework. Technical report, Microsoft Corporation, 2003.
- [Mic03c] Microsoft Corporation. What is the common language specification. Technical report, Microsoft Corporation, 2003.
- [Mic05] Microsoft Corporation. What's is .NET? Technical report, Microsoft Corporation, 2005.
- [Nag06] István Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. PhD thesis, University of Twente, The Netherlands, June 2006.
- [NAs05] Naspect framework, 2005.
- [Org02] International Standards Organization. Iso/cd 10303-226. Technical report, International Standards Organization, 2002.
- [OT01] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the Hyperspace approach. In M. Akşit, editor, *Software Architectures and Component Technology*. Kluwer Academic Publishers, 2001.
- [PAG03] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just in time aspects. In Mehmet Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 100–109. ACM Press, March 2003.
- [PGA02] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In Gregor Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 141–147. ACM Press, April 2002.
- [Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.
- [Pro02] Jeff Prorise. *Programming Microsoft .NET*. Microsoft Press, Redmond, WA, USA, 2002.
- [Sal01] P. Salinas. Adding systemic crosscutting and super-imposition to Composition Filters. Master's thesis, Vrije Universiteit Brussel, August 2001.
- [Spe06] Dennis R. Spenkelink. Compose* incremental. Master's thesis, University of Twente, The Netherlands, December 2006.
- [Sta05] Tom Staijen. Towards safe advice: Semantic analysis of advice types in Compose*. Master's thesis, University of Twente, April 2005.
- [Stu02] David Stutz. The Microsoft shared source CLI implementation. *MSDN Magazine*, 17(7), July 2002.

- [TOSH05] Peri Tarr, Harold Ossher, Stanley M. Sutton, Jr., and William Harrison. N degrees of separation: Multi-dimensional separation of concerns. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 37–61. Addison-Wesley, Boston, 2005.
- [tW06] Johan W. te Winkel. Bringing Composition Filters to C. Master’s thesis, University of Twente, The Netherlands, December 2006.
- [Vin04] Christian Vinkes. Superimposition in the Composition Filters model. Master’s thesis, University of Twente, The Netherlands, October 2004.
- [vO06] Michiel D. W. van Oudheusden. Automatic derivation of semantic properties in .NET. Master’s thesis, University of Twente, The Netherlands, August 2006.
- [Wat90] David A. Watt. *Programming language concepts and paradigms*. Prentice Hall, 1990.
- [Wat00] Damien Watkins. Handling language interoperability with the Microsoft .NET framework. Technical report, Monash Univeristy, October 2000.
- [Wic99] J. C. Wichman. The development of a preprocessor to facilitate composition filters in the Java language. Master’s thesis, University of Twente, 1999.
- [Wie03] R.J. Wieringa. *Design Methods for Reactive Systems*. Morgan Kaufmann, 2003.