Porting Compose* to the Java Platform

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

Roy David Spenkelink

Enschede, June 06, 2007

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

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

Abstract

Compose⋆ is a project that aims at enhancing the power of component- and object-based programming, so that software becomes easier to structure and modularize, hence easier to develop, maintain and extend. In particular, Compose⋆ offers aspect-oriented programming through the composition filters model.

One goal of the Compose⋆ project is to familiarize a large audience with the concept of the composition filters model. Currently, Compose* runs on the .NET platform and C platform. This thesis describes in detail the process of porting Compose⋆ to the Java platform, resulting in Compose⋆/J.

Since Compose⋆ is a language and platform independent solution, some of the key Java features might not be supported by the composition filters model. Therefore, this thesis also investigates the possibility of supporting specific Java features in Compose⋆/J. First, it discusses the possibilities for modularizing exception handling with composition filters. Second, it discusses the possibilities for expressing crosscutting concerns on inner classes. Finally, it discusses the possibilities and benefits for weaving on Java interfaces.

Acknowledgements

My graduation time was a long but exciting process and I would not have missed it for the world. Many people contributed to the completion of this thesis, for which I am grateful. In particular, I would like to express my appreciation to the following people.

First, I would like to thank the members of my graduation committee. I would like to thank my supervisor Lodewijk Bergmans. Although he sometimes responded late to my e-mails, that did not weight up against his enthusiastic and expert guiding. I also would like to thank Wilke Havinga. His remarks and suggestions helped me a lot.

In addition, I am thankful to Pascal Durr. Although, not a member of my committee, he provided me with valuable tips. Finally, many thanks go to my family for encouraging me to do my best and supporting me all the way. A special thanks goes out to my brother Dennis for sharing his thoughts about my project day in day out.

Contents

List of Figures

Listings

Chapter 1

Introduction to AOSD

The first two chapters have originally been written by seven M. Sc. students [6, 7, 12, 19, 20, 35, 41] at the University of Twente. The chapters have been rewritten for use in the following theses: [9, 10, 11, 21, 22, 34, 38, 40] and this thesis. 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 program-

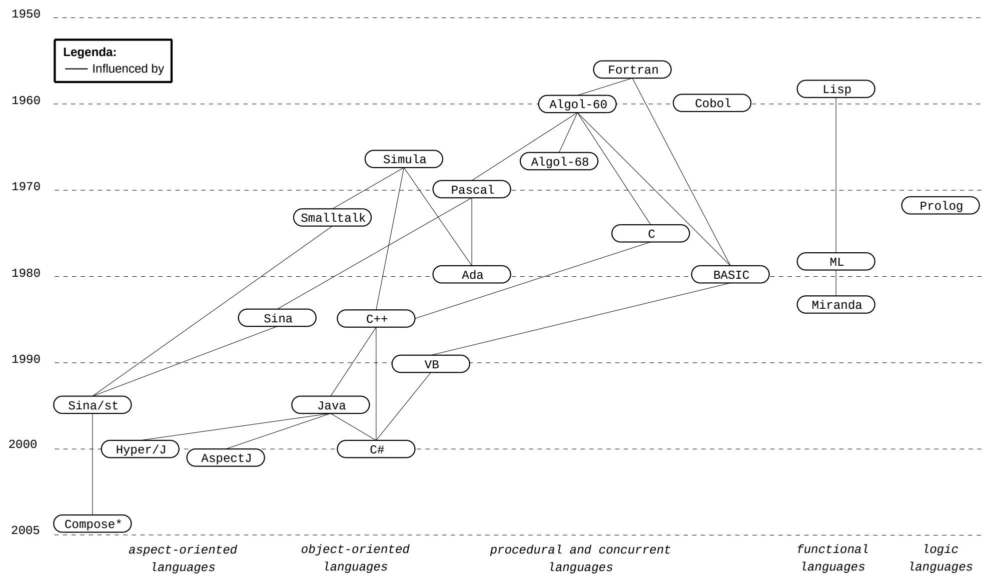


Figure 1.1: Dates and ancestry of several important languages

ming. 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 [42]. Every paradigm uses a different modularization mechanism for separating concerns into modules.

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

A logic language is based on a subset of mathematical logic. The computer is programmed to infer relationships between values, rather than to compute output values from input values. Prolog is currently the most used logic language [42].

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 [42]. 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 [15].

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 [37]. A specific decomposition limits the ability to implement other concerns in a modular way. For example, OOP modularizes concerns in classes and only fixed relations are possible. Implementing a concern in a class might prevent another concern from being implemented as a class.

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

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

1.2 Traditional Approach

Consider an application containing an object `Add` and an object `CalcDisplay`. `Add` inherits from the abstract class `Calculation` and implements its method `execute(a, b)`. It performs the addition of two integers. `CalcDisplay` receives an update from `Add` if a calculation is finished and prints the result to screen. Suppose all method calls need to be traced. The objects use a `Tracer` object to write messages about the program execution to screen. This is implemented

```java
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     result = a + b;
16     calcDisplay.update(result);
17   }
18
19   public int getLastResult() {
20     trace.write("int Add.getLastResult()");
21     return result;
22   }
23 }
```

(a) Addition

```java
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(int)");
10     System.out.println("Printing new value of calcula
11   }
12 }
```

(b) CalcDisplay

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

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

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

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

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

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

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

(a) Addition concern

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

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

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

(b) Tracing concern

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

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 [18]: first to provide a mechanism to express concerns that crosscut other components. Second to use this description to allow for the separation of concerns.

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

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

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

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

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

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

Aspect pluggability
Enabling or disabling concerns becomes possible.

1.3.1 AOP Composition

AOP composition can be either symmetric or asymmetric. In the symmetric approach, every component can be composed with any other component. For instance, Hyper/J follows this approach.

In the asymmetric approach, the base program and aspects are distinguished. The base program is composed with the aspects. For instance, AspectJ (covered in more detail in the next section) follows this approach.

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 opti-

mization 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 cannot 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 [31, 32].

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 spent 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 [14] 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.

In the next sections, we introduce AspectJ [25], Hyperspaces [30] and Composition Filters [5], which are three main AOP approaches.

1.4.1 AspectJ Approach

AspectJ [25] is an aspect-oriented extension to the Java programming language. It is probably the most popular approach to AOP at the moment, and it is finding its way into the industrial software development. AspectJ has been developed by Gregor Kiczales at Xerox's PARC (Palo Alto Research Center). To encourage the growth of the AspectJ technology and community, PARC transferred AspectJ to an open Eclipse project. The popularity of AspectJ comes partly from the various extensions based on it. Various projects 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:

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

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

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

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

Listing 1.3: Example of dynamic crosscutting in AspectJ

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 *join points*. A *pointcut* has a set of join points. In Listing 1.3 is `traceMethods` an example of a pointcut definition. The pointcut includes all executions of any method that is in a class contained by package `edu.utwente.trese`.

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

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

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

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

Listing 1.4: Example of static crosscutting in AspectJ

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 [30], which involves:

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

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

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

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

The first line indicates that, by default, all of the units contained within the package `edu.utwente.trese.pacman` address the kernel concern of the feature dimension. The other map-

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

Listing 1.5: Creation of a hyperspace

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

Listing 1.6: Specification of concern mappings

pings 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 remodularisation: the ability to extract hyperslices to encapsulate concerns that were not separated in the original code. This makes hyperspaces especially useful for evolution of existing software.

1.4.3 Composition Filters

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

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

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

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

Listing 1.7: Defining a hypermodule

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

Chapter 2

Compose⋆

Compose⋆ is an implementation of the composition filters approach. There are three target environments: 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 [26].

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 dispatch filter replaces the interface predicates, 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 [4].

1995 The Sina language with Composition Filters is implemented using Smalltalk [26]. The implementation supports most of the filter types. In the same year, a preprocessor providing C++ with support for Composition Filters is implemented [17].

1999 The composition filters language ComposeJ [43] 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 [33]. ConcernJ adds the notion of superimposition to Composition Filters. This allows for reuse of the filter modules

	and facilitation of 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.
2006	Compose⋆ is ported to the C platform.

2.2 Composition Filters in Compose⋆

A Compose⋆ application consists of concerns that can be divided in three parts: filter module specifications, superimposition, and implementation. A filter module contains the filter logic to filter on incoming or outgoing messages on 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.

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

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

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

  superimposition {
    selectors
    filtermodules
    annotations
    constraints
  }

  implementation
}
```

Listing 2.1: Abstract concern template

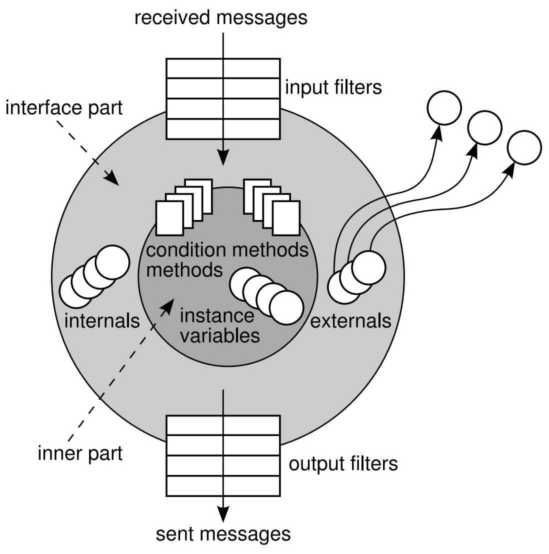


Figure 2.1: Components of the composition filters model

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

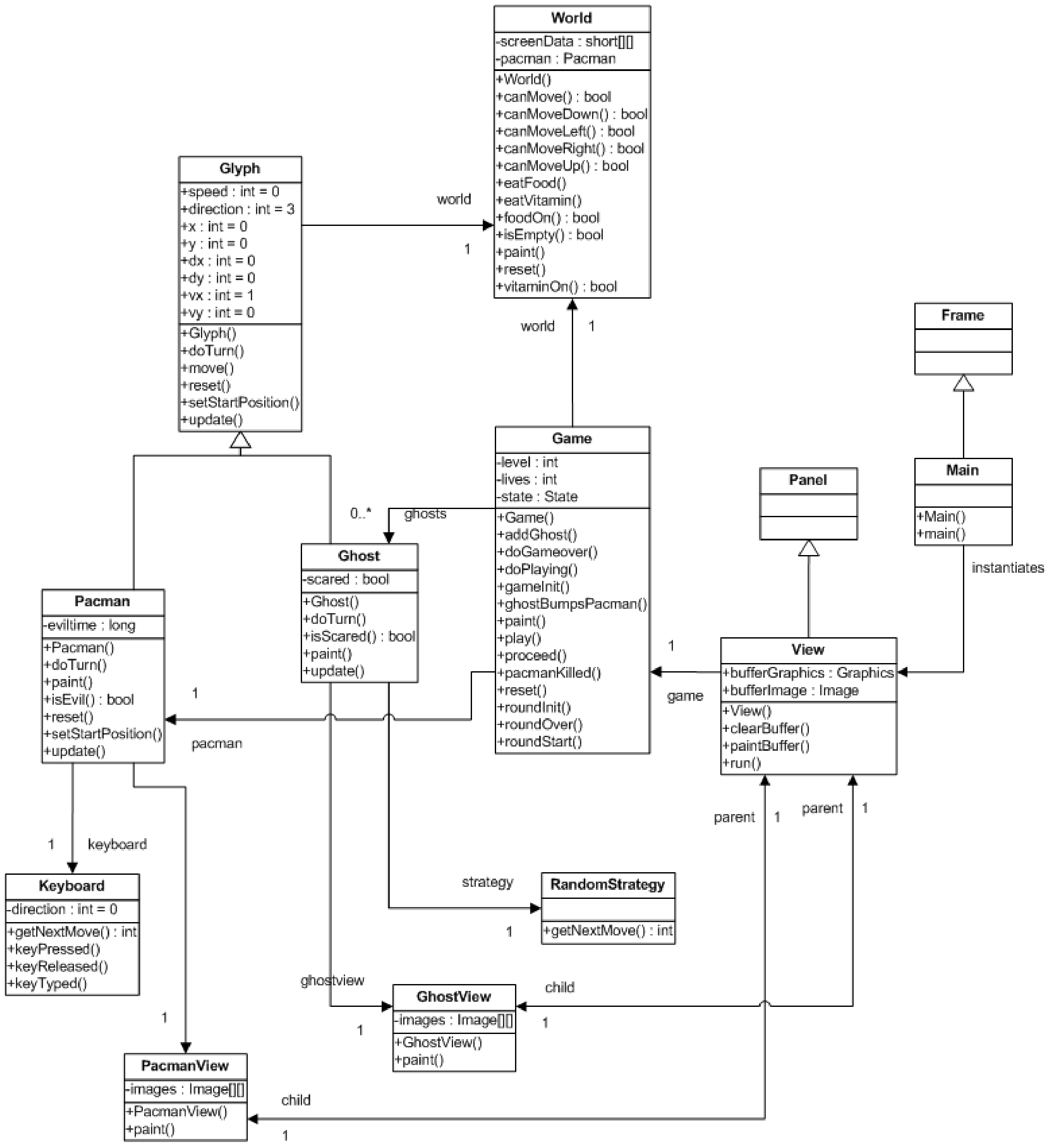


Figure 2.2: UML class diagram of the object-oriented Pacman game

PacmanView
: This class is responsible for painting pacman;

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

View
: This class is responsible for painting a maze;

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

2.3.2 Completing the Pacman Example

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

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

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

2.3.2.1 Implementation of Scoring

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

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

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

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

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

Listing 2.2: `DynamicScoring` concern in Compose⋆

2.3.2.2 Implementation of Dynamic Strategy

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

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

2.4 Compose⋆ Architecture

An overview of the Compose⋆ architecture is illustrated in Figure 2.3. The Compose⋆ architecture can be divided in four layers [28]: 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.

The creation of a build configuration can be done manually or by using a plug-in. Examples

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

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

  private Score() {}

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

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

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

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

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

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

Listing 2.3: Implementation of class Score

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

Listing 2.4: DynamicStrategy concern in Compose*

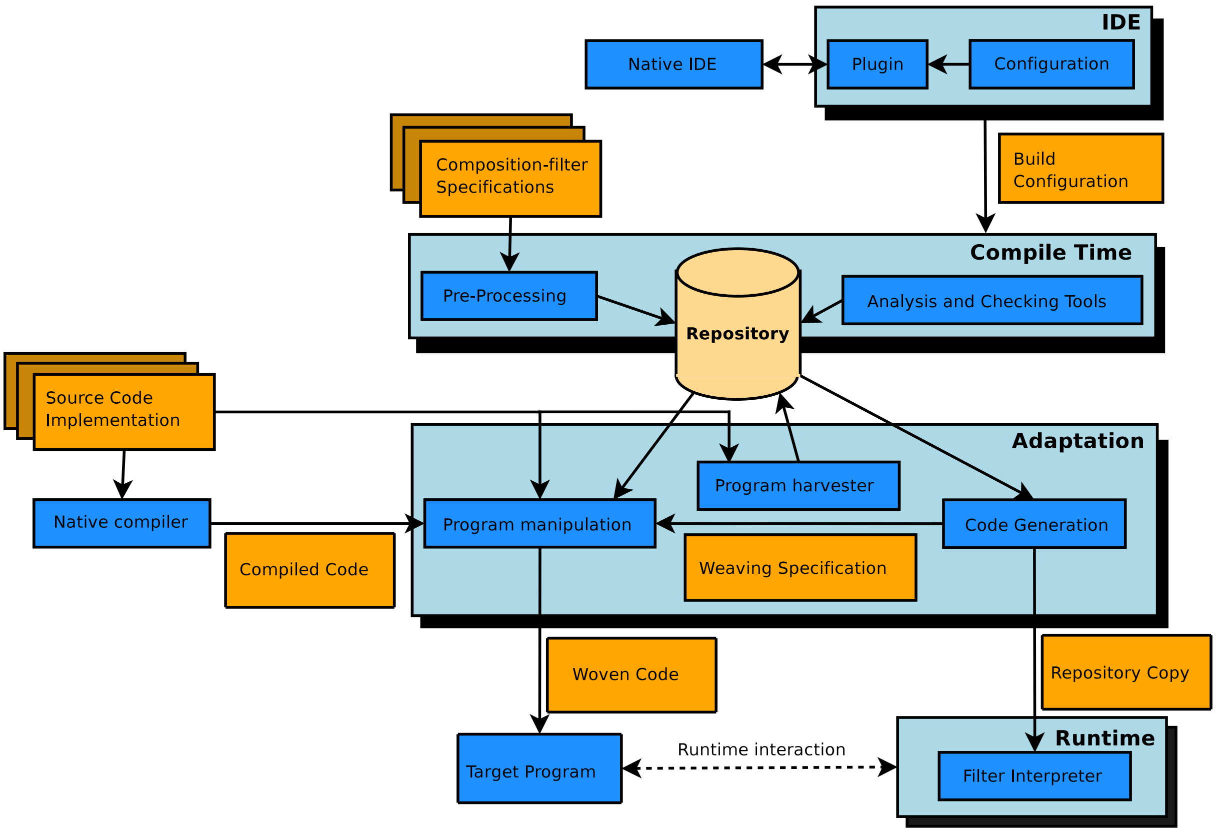


Figure 2.3: Overview of the Compose* architecture

of these plug-ins are the Visual Studio add-in for Compose⋆/.NET and the Eclipse plug-in for 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 result of the adaptation is the target program that interfaces with the runtime.

2.4.4 Runtime

The runtime layer is responsible for executing the concern code at the join points. It is activated at the join points 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 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 two platforms: .NET 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⋆/C contains support for the C programming language. The implementation is different from the .NET counterpart, 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. 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 join point, partial ordering constraints can be applied to reduce this number. These constraints can be declared in the concern definition;

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

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

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

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

Integrated Development Environment support
The Compose⋆ implementations all have a IDE plug-in; Compose⋆/.NET for Visual Studio, Compose⋆/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
: Incremental rebuilding re-uses the compilation results of previous buildings to safe compilation 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

Problem Identification

This chapter identifies the challenges that exist in porting Compose⋆ to the Java platform.

3.1 Background

One goal of the Compose⋆ project is to familiarize a large audience with the concept of the composition filters model. Currently, Compose⋆ runs on the .NET platform [6, 16, 20, 40] and C platform [38]. Adding a third platform increases our audience. Furthermore, it helps us in proving the language and platform independent aspect of the composition filters model.

Our choice for Java as the third platform is straightforward. First of all, Java is the most popular programming language according to the TIOBE index published on the internet [39]. The TIOBE index gives an indication of the popularity of programming languages. The ratings are based on the world-wide availability of skilled engineers, courses and third party vendors. Java is especially popular in the Software Engineering research community. Secondly, since the introduction of AspectJ [25] in 2001, AOP has become increasingly popular on the Java platform. This increases the possibility of using or working with existing third-party tools, frameworks and libraries. Finally, previous efforts were made in the past to implement the composition filters model on the Java platform [33, 43]. We can use this experience for the development of Compose⋆/J.

3.2 Designing Compose⋆/J

Our first challenge is to design and implement Compose⋆/J.

As described in Chapter 2, the Compose⋆ architecture is divided in four layers [28]: IDE, compile-time, adaptation and runtime. The compile-time layer reasons about the composition filters model, and thus is considered language independent. All implementations of Compose⋆ share the analysis tools that exist in this layer.

The other three layers are partially language dependent. Thus we need to design and implement these layers in Compose⋆/J. More specific, Compose⋆/J is based on the following technologies:

IDE integration. Integration into an existing Integrated Development Environment (IDE) in the form of a plug-in. One of the purposes of the plug-in is to create a build configuration. In the build configuration, it is specified which source files and settings are required to build a Compose⋆ application.

Type Collecting. In order to reason about concerns, the program structure and annotations within the source code are collected and stored in a knowledgebase. This technology is part of the adaptation layer.

Weaver. The weaver is responsible for weaving advice code into the target program. The weaver is also part of the adaptation layer.

Interpreter. The interpreter is responsible for executing concern code at the joinpoints. It is triggered by function calls that are woven in by the weaver. It is part of the runtime layer.

3.3 Supporting specific Java features

Since Compose⋆ is a language independent solution, some of the Java features might not be supported by the composition filters model or even be supported in the future. Our second challenge is to investigate the possibility of supporting specific Java features in Compose⋆/J. We focus on the following Java features:

Exception Handling. Mechanism that handles the occurrence of an exceptional condition that changes the control flow of the normal execution of a program. Currently, Compose⋆ generates exceptions, but it cannot handle them. Chapter 4 discusses the possibilities for modularizing exception handling with composition filters.

Inner Classes. Inner classes are classes defined inside the definition of another class. Currently, Compose⋆ cannot handle inner classes. In Chapter 5, we discuss the possibilities for expressing crosscutting concerns on inner classes.

Java Interfaces. An interface in Java is a group of related methods with empty bodies. They form a contract between classes and the outside world. Currently, Compose⋆ only supports class-based weaving. Chapter 6 discusses the possibilities and benefits for weaving on Java interfaces.

In our approach, we do not consider the language independence of Compose⋆. We assume that Compose⋆ is solely meant for the Java platform. Therefore, solutions presented in this thesis should be further investigated with the aspect of language independence in mind.

3.4 Summary

Since the introduction of AspectJ, a lot of AOP tools have been on the Java platform. This fact combined with our intention to prove the language and platform independence of Compose⋆, motivates us to port Compose⋆ to the Java platform, yielding in Compose⋆/J.

Implementing Compose⋆/J comes with challenges. Our first challenge is to implement the language dependent technologies in Compose⋆/J. Chapter 7 and Chapter 8 present the design and implementation of Compose⋆/J.

Our second challenge is to investigate the possibility of supporting specific Java features in Compose⋆/J. The next three chapters elaborate on this.

Chapter 4

Exception Handling in Compose*

This chapter focuses on the possibility of supporting exception handling in Compose⋆. First, it presents some background information. Then, it describes our motivation for modularizing exception handling with composition filters. After that, it defines a set of comparison criteria for composition filters models. Finally, it presents three possible composition filters models that support exception handling, applies the criteria on the models and evaluates them.

4.1 Background

Exception handling is a mechanism to handle the occurrence of an exceptional condition that changes the control flow of the normal execution of a program. Such a condition is called an *exception*. When an exception occurs, a *handler* temporarily interrupts the normal execution. Nowadays, many computer languages have built-in support for exception detection and exception handling. One of these languages is Java. Section 4.1.1 describes the exception handling mechanism in Java. Section 4.1.2 describes the error filter, which is currently the only built-in support in Compose⋆ that deals with exceptions.

4.1.1 Java Exception Handling

When an exceptional behavior causes an exception to be thrown, that exception is represented by an object. The Java exception hierarchy is shown in Figure 4.1.

All exception objects are instantiated from a class named `Throwable` or one of its subclasses. Sun [36] states the following about the class `Throwable`:

The Throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement. Similarly, only this class or one of its subclasses can be the argument type in a catch clause.

As we can see in Figure 4.1 the class `Throwable` has two subclasses. The difference between these two can be best explained by looking again at their definition:

An Error is a subclass of Throwable that indicates serious problems that are external to an application. A reasonable application should not try to catch these errors. Most such errors are abnormal conditions

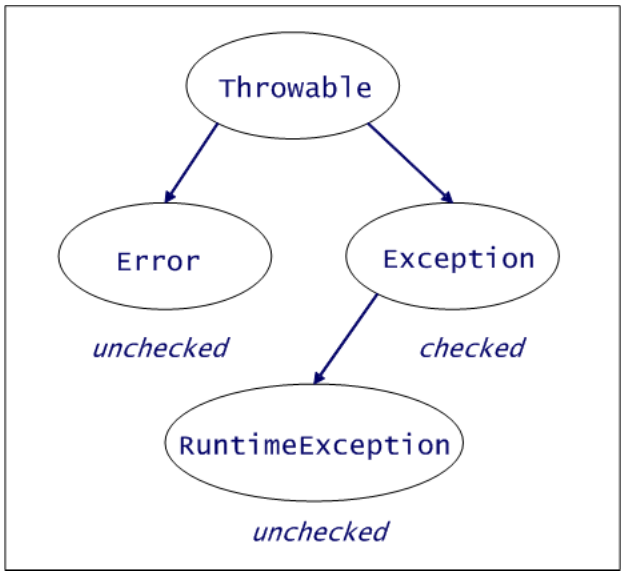


Figure 4.1: Java Exception Hierarchy

that the application cannot anticipate or recover from. These errors can be caught to notify the user about the problem, but it also makes sense to print the stack trace and exit the application.

The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.

Unlike many programming languages that support exception handling, Java distinguishes between two types of exceptions:

Unchecked exceptions. Unchecked exceptions are types of exceptions that you can optionally handle, or ignore. If you elect to ignore the possibility of an unchecked exception (i.e. exception is never caught in the program), and one occurs, your program will terminate as a result. If you elect to handle an unchecked exception and one occurs, the result will depend on the code that you have written to handle the exception. Unchecked exceptions are exceptions instantiated from `RuntimeException` and its subclasses.

Checked exceptions. Checked exceptions are all exceptions instantiated from the class `Exception`, or from subclasses other than `RuntimeException` and its subclasses. These exceptions cannot be ignored while writing code. So checked exceptions represent abnormal conditions that should be anticipated and caught to prevent program from termination. The Java compiler raises an error when checked exceptions are not handled by a try/catch block or throws definition.

How to catch or throw an exception?

Listing 4.1 shows an example of catching exceptions in Java. An exception can be caught by declaring a **try/catch** block. The parameter of the catch block indicates which exception objects are caught. In this case, the catch block catches instances of the class `Exception` and of its subclasses.

```java
public class myClass
{
    public void myMethod()
    {
        try
        {
            ...
        }
        catch(Exception e)
        {
            ...
        }
    }
}
```

Listing 4.1: Catching exceptions in Java.

Listing 4.2 shows an example of throwing exceptions in Java. A method can either throw an existing exception object (i.e. generated elsewhere) or throw a new exception. A new exception can be thrown by using the keywords **throw** and **new**. A **throws**-clause in the method declaration indicates that a method throws the declared exceptions. In this case `methodB` throws a new exception, `methodA` opts to ignore the exception and throws it again.

```java
public class myClass
{
    public void methodA() throws Exception
    {
        methodB();
    }

    public void methodB() throws Exception
    {
        throw new Exception("my own exception");
    }
}
```

Listing 4.2: Throwing exceptions in Java.

4.1.2 Error Filter

Currently, the error filter is the only filter in Compose⋆ that deals with exceptions. The definition of the error-filter is as follows:

An error filter raises an exception when it rejects a message. The substitution part is ignored. When it accepts a message, the message continues through the next filter.

The error filter was first used to express multiple views on objects. It enables us to restrict access to particular methods.

4.2 Motivation

In the world of AOP, exception handling is often mentioned as an example of a crosscutting concern. The following properties of crosscutting concerns explain this:

Code tangling. There is tangling between code for what the program should do (i.e. its normal behavior) and code for detecting and handling exceptions.

Code replication. Often responses to exceptions are similar. This leads to replication of code. Examples of such responses are: *"log and ignore"*, *"set the return to a default value"* and *"throw an exception of a different kind"*. [27]

Code scattering. Code scattered among different objects can sometimes be identified as one concern. E.g. *All methods that call a server should be prepared for network failures and retry calling it 3 times before giving up*.

Since Compose⋆ is an AOP solution, our motivation for supporting exception handling in Compose⋆ becomes clear. We would like to support the following things:

Modularize exceptional/normal behavior. We want to split up the exceptional behavior from the normal behavior. Specificly, we want to express code belonging to exception detection and handling with composition filter code.

Eliminate scattering of exception declarations The use of checked exceptions forces programmers to declare throws clauses at places in the program where they do not want to handle exceptions. This leads to scattering of exception declarations (i.e. numerous throws clauses). This scattering hampers the process of pinpointing the exact locations of where the exceptions are handled. We want to eliminate this redundancy by specifying in the concern code where exceptions are handled.

4.2.1 Demonstrating Example

As described above, exception handling can often lead to redundant code due to replication of code. Assume we write a data access object (DAO) `PersonDAO` for retrieving and updating data of a `Person` in a database, as shown in Listing 4.3.

The two methods *getPerson* and *updatePerson* both have the same response to handling a `SQLException`. They simply log the exception.

We can say that lines 12, 14-17, 21 and 23-26 belong to a single concern *"Simple logging of exceptions"*. It should be possible in Compose⋆ to leave out these lines from the code and express them with composition filters code. Currently, Compose⋆ does not support this. The result could be a clear reduction of source code, no tangled code and modularization between normal and exceptional behavior. The reduced source code is shown in Listing 4.4.

Section 4.3 presents three possible composition filters models for expressing this concern with composition filters.

```java
package apackage;

public class PersonDAO
{
  private Person person;

  public PersonDAO() {
        //... setup database connection
  }

  public void getPerson() {
    try {
        // ... execute query for retrieving Person data.
    }
    catch{SQLException e} {
        System.out.println(e.getMessage());
    }
  }

  public void updatePerson() {
    try {
        // ... execute qeury for updating Person data.
    }
    catch{SQLException e} {
        System.out.println(e.getMessage());
    }
  }
}
```

Listing 4.3: Snapshot implementation of class `PersonDAO`.

```java
package apackage;

public class PersonDAO
{
  private Person person;

  public PersonDAO() {
        //... setup database connection
  }

  public void getPerson() {
    // ... execute query for retrieving Person data.
  }

  public void updatePerson() {
    // ... execute qeury for updating Person data.
  }
}
```

Listing 4.4: Snapshot partial implementation of class `PersonDAO` (i.e. without concerns) in Compose⋆.

4.3 Solution Models

This section describes three possible composition filters models that enable us to modularize exception handling in Compose⋆. The next section compares these models based on three criteria, which we present below. Since these criteria are hard to measure, we evaluate them based on a qualitative study.

Intuitive semantics. Using intuitive semantics facilitates reasoning about the composition filters models and the filters used in the models. We define intuitive as: are things executed in the order in which they are written; are conditions evaluated at the moment a message passes a filter; is it clear what the filters do when they accept or reject a message; etc.

Amount of ordering. Different orderings of the filters superimposed on an object can lead to different results. This criteria evaluates in what degree it is possible to specify orderings between the filters in the different composition filters models. We evaluate this criteria by examining the possible orderings.

Reusability of concerns. This criteria evaluates the reusability of concerns in the different composition filters models. We define reusability of concerns as: 1) the ability to partially reuse concerns through appriorate modularization(s) and 2) the ability to adapt concerns. We evaluate this criteria by examining concerns that can potentially be used by other concerns, e.g. exception handling and profiling.

4.3.1 Preliminaries

First, we give preliminaries to better understand the composition filters models presented below. Consider the control flow of a message currently in Compose⋆, illustrated in Figure 4.2.

In the figure, we see three objects (A, B and C). Object A sends a message to object B. This message first runs through a set of outputfilters superimposed on object A. When a message runs through a filter, it first runs through the matching part of the filter. This matching part decides what filter action is going to be executed. Each filtertype defines four filter actions: *accept call*, *reject call*, *accept return* and *reject return*. If a message matches with the matching pattern, the *accept call* action of the specified filter is executed. If the message does not match with the matching pattern, the filter executes it *reject call* action. Furthermore, the call actions are tightly coupled with the return actions. This means that if a filter executed its *accept call* action, it executes its *accept return* action when it receives a *return* (i.e. normal method return). Likewise, a filter executes its *reject return* action when it rejected the message.

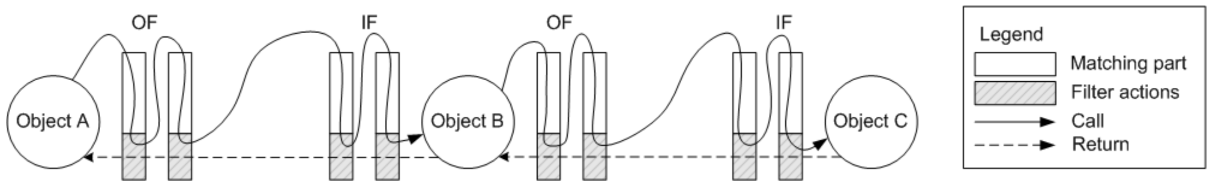


Figure 4.2: Control flow of a message in Compose⋆

In this case, all outputfilters superimposed on object A decide not to change the message and the message continues through the inputfilters superimposed on object B. The last filter in this set dispatches the message to object B. After receiving the message from object A, object B sends a message to object C. Again the message runs through a set of outputfilters and a set of inputfilters. Eventually the message is dispatched to object C and object C invokes the message.

After object C invokes the message, it sends a *return* to its caller. This *return* runs through the same filters that were passed by the message, but in opposite direction. In contrast to messages, *returns* bypass the matching pattern part and run straight through the filter action part of the filters (i.e. the return action has already been decided on the call). The filters either perform their *accept return* or *reject return* action, based on whether they accepted the call or not. In this case, no further action is taken on *returns* and the *return* returns to object A.

Table 4.1 shows the possible filter actions of the filtertypes currently supported in Compose⋆/J. For example, the dispatch filter only defines an *accept call* action (i.e. it dispatches a message). Furthermore, note that none of the current filters define any return actions.

Filtertype	accept call	reject call	accept return	reject return
Dispatch	dispatch	x	x	x
Send	dispatch	x	x	x
Error	x	raise	x	x
Meta	reify	x	x	x

Table 4.1: Filter actions of the filtertypes.

A more detailed figure of the composition filters model is shown in Figure 4.3.

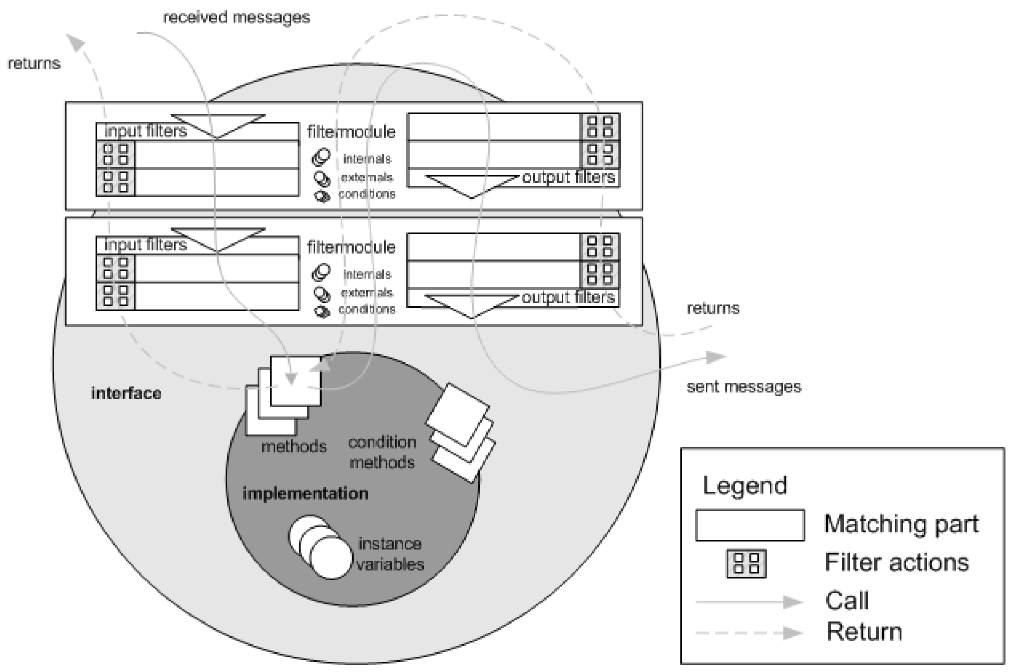


Figure 4.3: Current composition filters model in Compose⋆

It shows two filtermodules superimposed on an object. The implementation part of the object consists of instance variables, methods and condition methods (i.e. methods that return a boolean).

A filtermodule consists of conditions, internals and externals. It can also contain two filtersets: *inputfilters* and *outputfilters*. 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. As described above, the filtertypes in each filterset define four possible actions: it can either *accept* or *reject* a message or a return.

Furthermore, the ordering of the filtermodules is the same for incoming as for outgoing messages. Filters in each filterset are evaluated in the order they are written.

This model acts as the base for the solution models. In two of the three models, we propose slight adjustments to this model in order to support modularizing exception handling with composition filters, but first we look at the current model. Is this model suitable for the job?

4.3.2 Solution Model A: Exception handling in current composition filters model

There exists several alternatives to modularize exception handling using the current composition filters model. For example, consider the meta filter. Figure 4.4 illustrates the control flow of the meta filter.

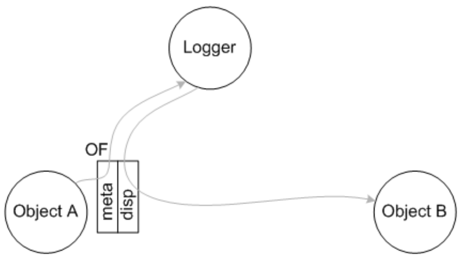


Figure 4.4: Control flow of a meta filter

In the figure, we see an example of an implementation of the logging concern. Object A sends a message to object B. This message runs through the outputfilters superimposed on object A. The first filter in the filterset is a meta filter. This filter accepts the message and sends a reified message to an instance of class `Logger`, which implements the logic for logging. After `Logger` finishes its logging procedure, the original message proceeds again through the filterset. Finally, the dispatch filter dispatches the message to object B.

In other words, if we use a filter we can leave a filterset, either temporarily or completely. This behavior allows us to support exception handling, as is illustrated in Figure 4.5.

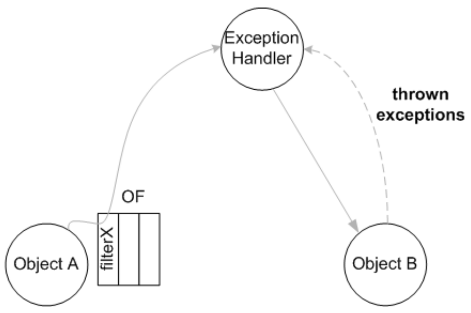


Figure 4.5: A possible way to support exception handling in current composition filters model.

In the figure, object A sends a message to object B. This message is intercepted by an unidentified filter, called filterX. The filter accepts the message and sends it (i.e. as an argument of a new message) to an instance of the class `ExceptionHandler`, which acts as a wrapper class for catching exceptions. `ExceptionHandler` executes the original message, and catches exceptions thrown by this message.

A suitable name for filterX is the *"catch"* filter. We can use this filter either as an inputfilter or as an outputfilter. The possible semantics of this catch filter is:

If the catch filter accepts a message, it adds the message as an argument of a new message. This new message is send to the object defined in the substitution part. This object executes the message and catches exceptions thrown during the execution. The type of exceptions caught is specified by the implementation part of the object.

A possible implementation of the concern described in Section 4.2.1 using the above semantics is shown below:

```
1  concern SimpleLogExceptionConcern in apackage {
2    filtermodule SimpleLogException {
3      externals
4        exch : apackage.ExceptionHandler = apackage.ExceptionHandler.instance();
5      outputfilters
6        catchfilter : Catch = {[*.getPerson] exch.log,
7                               [*.updatePerson] exch.log}
8    }
9    superimposition {
10     selectors
11       class = { C | isClassWithName(C, 'apackage.PersonDAO') };
12     filtermodules
13       class <- SimpleLogExceptionConcern;
14   }
15 }
```

Listing 4.5: *"Simple exception logging"* concern in current composition filters model.

The concern is called `SimpleLogExceptionConcern` (line 1) and it contains one filtermodule called `SimpleLogException` (lines 2–8). The filtermodule contains one outputfilter *catchfilter* (lines 6–7) and is superimposed on all instances of `PersonDAO` (lines 9–14).

The catch filter accepts all outgoing messages with selector *getPerson* or *updatePerson*. Upon acceptance, the message is sent as an argument of a new message to the `log()` method of an instance of the class `ExceptionHandler`. This method executes the original message and catches the exceptions thrown by the message.

Although the above semantics sound correct, there exist some drawbacks. For example, we cannot declare a filter after a catch filter, because the catch filter executes the message. This leads to unwanted restrictions. Suppose we want to log the execution of a method call and handle exceptions thrown by this method. We can only achieve this by declaring a meta filter before a catch filter. Such a relationship between filters is generally unwanted, since it hampers the programmers freedom.

Furthermore, notice that the type of exceptions that are handled is determined by the implementation part of the object defined in the substitution part. This could also be determined by the catch filter itself (i.e. by using filter parameters).

There exist a second alternative that solves these drawbacks. As described in the previous section, a filter has four filter actions. It can either accept or reject a message or a *return*. Now, we can consider a thrown exception as a *return*. In other words, we can create a filter that catches exceptions, as is illustrated in Figure 4.6.

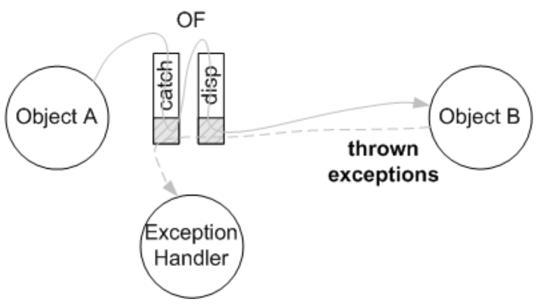


Figure 4.6: Another way to support exception handling in current composition filters model.

In the figure, object A sends a message to object B. A catch filter intercepts this message. Upon acceptance, the catch filter does nothing and the message continues through the filterset. The next filter is a dispatch filter, which dispatches the message to object B and the message is executed. Exceptions thrown during the execution of this message run back through the filters in opposite direction. The dispatch filter does not define any *return action*, so the *return* is sent to the catch filter. Now, the catch filter defines an *accept return* action. Since the catch filter accepted the call, this action is executed. The catch filter dispatches the exception to an instance of the class `ExceptionHandler` which handles the exception.

Table 4.2 shows again the possible filter actions of the filtertypes. The catch filter only defines the *accept return* action.

Filtertype	accept call	reject call	accept return	reject return
Dispatch	dispatch	x	x	x
Send	dispatch	x	x	x
Error	x	raise	x	x
Meta	reify	x	x	x
Catch	x	x	dispatch	x

Table 4.2: Filter actions of the filtertypes.

The semantics of the catch filter changes slightly:

If the catch filter accepts a message, it does nothing. If the catch filter accepts a return (i.e. an exception), it dispatches the exception to an instance of the object defined in the substitution part, which implements the logic for handling the exception. The type of exceptions that are caught in this way are specified with filter parameters.

A possible implementation of the concern described in Section 4.2.1, using these updated semantics of the catch filter, is shown in Listing 4.6. The code differs from the code in Listing 4.5 at only one place. The catch filter specifies which type of exceptions are caught (line 6). In the example, only exception objects of the type `SQLException` are handled.

To summarize, it is possible to modularize exception handling using the current composition filters model. This is accomplished by using a new filter, called the catch filter. We presented two different semantics and demonstrated the use of this catch filter. The next section proposes a model that supports exception handling based on intercepting *return messages*.

```
concern SimpleLogExceptionConcern in apackage {
  filtermodule SimpleLogException {
    externals
      exch : apackage.ExceptionHandler = apackage.ExceptionHandler.instance();
    outputfilters
      catchfilter : Catch ("SQLException") = {[*.getPerson] exch.log,
                                              [*.updatePerson] exch.log}
  }
  superimposition {
    selectors
      class = { C | isClassWithName(C, 'apackage.PersonDAO') };
    filtermodules
      class <- SimpleLogExceptionConcern;
  }
}
```

Listing 4.6: *"Simple exception logging"* concern in current composition filters model.

4.3.3 Solution Model B: Two-way composition filters model

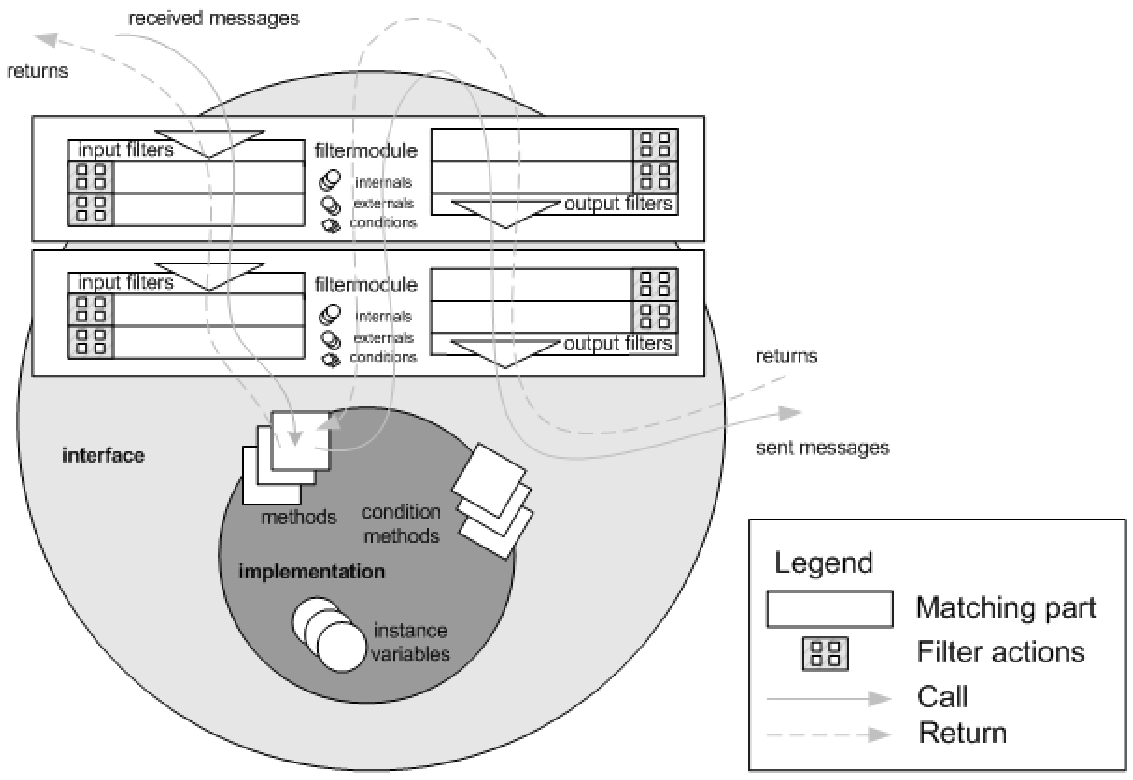


Figure 4.7: Two way composition filters model: filter on return messages

The second model is depicted in Figure 4.7. As we can see, it has has two filtersets: *inputfilters* and *outputfilters*. The difference with model A is, that in this model the filters in the filtersets can accept or reject two different types of messages: *call messages* (as in model A) and *return* *messages*. In other words, in this model we artificially consider *returns* as messages. This means that, similar to messages, *returns* run through the matching part of the filters. Since the filters in this model can accept two different types of messages, we refer to this model as the two-way composition filters model. To distinct between messages and returns, in this section we refer to messages as *call messages* and returns as *return messages*.

Return messages represent method returns (i.e. normal returns or exceptions). Similar to incoming and outgoing call messages, return messages can have a *target* and a *selector*. Together they represent the method that returned the message. This enables the filters to intercept return messages in a similar way that they intercept call messages.

In this model, the matching part of a specific filter is passed twice. First, a call message passes the filter after a method call is made. Then, after the method returns, a return message passes the same filter again. This means, that we have two options for evaluating the conditions of the filters:

Evaluate once. Conditions are evaluated each time a call message passes a filter.

Evaluate twice. Conditions are evaluated each time a message (i.e. a call or a return) passes a filter.

We opt for the second option in this model, since the state of an object can change during the execution of a method. Thus, conditions are evaluated twice. The change in effect, compared to model A, is that for instance the question whether or not to catch exceptions is determined at a later stage. In model A, this is determined before the execution of a method. In this model, the question is answered after the exception is thrown.

Furthermore, we can choose between two possible control flows for return messages:

Run back through evaluated filters. Return messages only run back through the filters that were evaluated on the call.

Run back through superimposed filters. Return messages run through the filters superimposed on an object, including the filters that were not evaluated on the call.

To choose between these two options, let's take a look at an example. Suppose a programmer wants to replace calls to method `bar()` with method `foo()` and he wants to catch any exception thrown by any method. He comes up with the following code:

```
concern SimpleLogExceptionConcern in apackage {
  filtermodule SimpleLogException {
    externals
      exch : apackage.ExceptionHandler = apackage.ExceptionHandler.instance();
    inputfilters
        disp : Dispatch = {[*.bar] *.foo}
        catch : Catch ("Exception") = {[*.*] exch.log}
    }
  superimposition {
    selector
        classes = ...
    filtermodules
      classes <- SimpleLogExceptionConcern;
  }
}
```

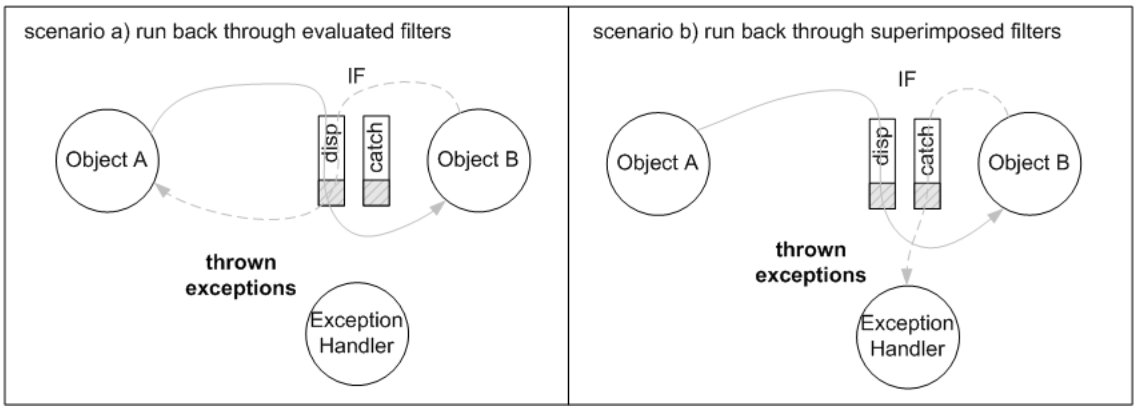


Figure 4.8: Different scenarios for the control flow of return messages.

Figure 4.8 shows what happens in both scenarios. In the first scenario, the exceptions thrown by `foo()` are never caught since the catch filter was not evaluated on the call. The second scenario results in the right behavior. Note that the programmer can also achieve the right behavior in the first scenario if he switches the two filters. In other words, both scenarios are valid options, but the first option hampers the programmers freedom, so we opt for the second option.

Now, the way exception handling works in this model is shown in Figure 4.9.

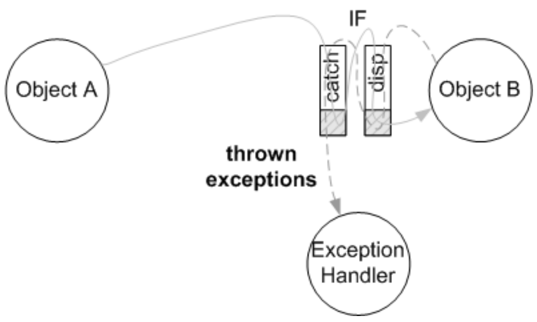


Figure 4.9: Exception handling in two-way composition filters model.

In the figure, object A sends a message to object B. The message runs through a set of inputfilters superimposed on object B. The first filter in the set, a catch filter, does not accept the call message and the message continues through the filterset. Then, a dispatch filter dispatches the message to object B and the message is executed. During the execution, return messages (i.e. thrown exceptions) run through the superimposed filters in opposite direction. The dispatch filter does not accept any return messages. The catch filter accepts the return message and sends a message to an instance of the class `ExceptionHandler` which handles the exception.

As described above, we can again use a catch filter to modularize exception handling. The filter actions of this catch filter are similar as the actions shown in Table 4.2. The semantics of the catch filter in this model is:

If a catch filter accepts a call message, it does nothing. If it accepts a return message, that contains a thrown exception, it dispatches the exception to an instance of the object defined in the substitution part, which implements the logic for handling the exception. The type of exceptions that are caught in this *way are specified with filter parameters.*

A possible implementation of the concern described in Section 4.2.1 in this model is shown in Listing 4.7.

```
concern SimpleLogExceptionConcern in apackage {
  filtermodule SimpleLogException {
    externals
      exch : apackage.ExceptionHandler = apackage.ExceptionHandler.instance();
    inputfilters
      catchfilter : Catch ("SQLException") = {[*.getPerson] exch.log,
                                              [*.updatePerson] exch.log}
  }
  superimposition {
    selectors
      class = { C | isClassWithName(C, 'apackage.PersonDAO') };
    filtermodules
      class <- SimpleLogExceptionConcern;
  }
}
```

Listing 4.7: *"Simple exception logging"* concern in model B.

The code is similar to the code in Listing 4.6, only this time we use an inputfilter. If we use an outputfilter the effect is the same.

To summarize, this model is based on intercepting *return messages*. Return messages represent method returns (i.e. normal returns or exceptions). Similar to *call messages*, they have a *target* and a *selector*. Together they represent the method that returned the message. This enables filters in this model to intercept *call messages* as well as *return messages*. Similar to the first model, we can use a catch filter to modularize exception handling. We presented the semantics and demonstrated the use of this catch filter. A difference between this model and the current composition filters model, is the different moments in time of condition evaluations, which can lead to different effects.

In the next section, we propose another way to modularize exception handling, namely a model based on using a separate filterset for return messages (i.e. *returnfilters*).

4.3.4 Solution Model C : Introducing return filters

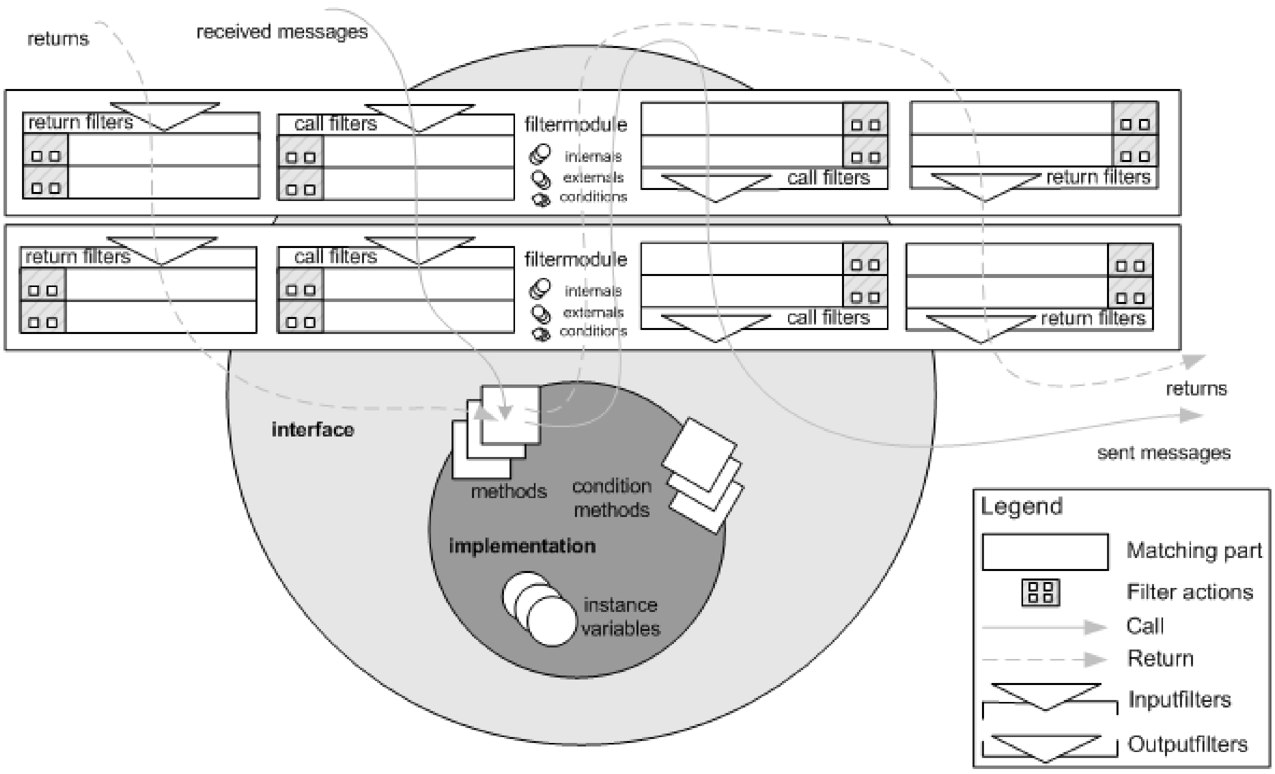


Figure 4.10: Model C: Introduction of return filters

The third model is depicted in Figure 4.10. This model introduces a new type of filterset: *returnfilters*. Similar to the original inputfilters and outputfilters, returnfilters can be placed on the incoming and outgoing side of an object, giving a total of four filtersets. All filters define two possible actions: it can *accept* or *reject* a message. The difference is that *returnfilters* can only accept or reject *return messages*. Similar, the original *inputfilters* and *outputfilters* can only accept or reject call messages. For clarity, we refer to these filters as *callfilters*.

The idea behind this model is to distinct between *calls* and *returns*. As described in the previous sections, most of the filters only define actions on one particular event (i.e. a call or return). This means, that a lot of the four possible filter actions are undefined. By introducing returnfilters, we can reduce the amount of filter actions to two, and subsequently reduce the amount of undefined actions, as is shown in Table 4.3. The context column tells us in what filterset the filter can be used (i.e. either as a callfilter or as a returnfilter)

Filtertype	accept	reject	context
Dispatch	dispatch	x	call
Send	dispatch	x	call
Error	x	raise	call
Meta	reify	x	call
Catch	dispatch	x	return

Table 4.3: Filter actions of the filtertypes.

Using this model it is again possible to introduce a *"catch filter"*. We use this filter as a returnfilter. The possible semantics of this catch filter is:

The catch filter only filters return messages. If it accepts a return message, that contains a thrown exception, it dispatches the exception to an instance of the object defined in the substitution part, which implements the logic for handling the exception. The type of exceptions that are caught in this way are specified with filter parameters.

The way exception handling works in this model is shown in Figure 4.11. In the figure, object A sends a message to object B. During the execution of the message, thrown exceptions are caught by a returnfilter (i.e. a catch filter). Exceptions within the return message are handled by an instance of the class `ExceptionHandler`.

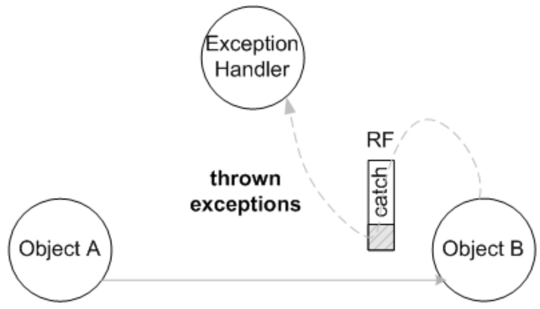


Figure 4.11: Exception handling in action using a return filter.

A possible implementation of the concern described in Section 4.2.1 in this model is shown in Listing 4.8. The keyword *return* (line 6) is introduced to make a distinction between the returnfilters and the regular incoming and outgoing filters. The concern is called `SimpleLogExceptionConcern` (line 1) and it contains one filtermodule called `SimpleLogException` (lines 2–9). The filtermodule contains one inputfilter *catchfilter* (lines 7–8) and is superimposed on all instances of `PersonDAO` (lines 10–15).

The catch filter accepts all outgoing returning messages with selector *getPerson* or *updatePerson* that contains an exception of type `SQLException` (line 7). Upon acceptance, the exception is handled by an instance of `ExceptionHandler`, which is used as an external object (line 4).

```
 1  concern SimpleLogExceptionConcern in apackage {
 2    filtermodule SimpleLogException {
 3      externals
 4        exch : apackage.ExceptionHandler = apackage.ExceptionHandler.instance();
 5      outputfilters
 6          return
 7              catchfilter : Catch ("SQLException") = {[*.getPerson] exch.log,
 8                                                     [*.updatePerson] exch.log}
 9    }
10    superimposition {
11      selectors
12        class = { C | isClassWithName(C, ’apackage.PersonDAO’) };
13      filtermodules
14        class <- SimpleLogExceptionConcern;
15    }
16  }
```

Listing 4.8: *"Simple exception logging"* concern in model C.

4.4 Comparison of Solution Models

Section 4.3 presents three possible solution models that enable us to modularize exception handling in Compose⋆. This section applies the criteria, described in Section 4.3, to the models and evaluates the results.

4.4.1 Criteria applied to Model A

Model A is depicted in Figure 4.3. Applying the criteria, results in the following evaluation:

Intuitive semantics : -/+

Model A exposes some unintuitive semantics according to the following arguments:

Firstly, the idea of using an inputfilter or an outputfilter for catching exceptions sounds awkward, since exceptions are caught after a return, not before a message is executed. Using a separate filterset for return messages (i.e. as in model C) sounds more intuitive.

Secondly, in Section 4.3.2 we mentioned two different semantics of the catch filter in this model. This proves that it might be unclear to the programmer how to use this filter.

Finally, sometimes the semantics of concerns might be unclear to the programmer. Suppose we declare a substitute filter after a catch filter, as is shown below.

```
concern SimpleLogExceptionConcern in apackage {
  filtermodule SimpleLogException {
    externals
      exch : apackage.ExceptionHandler = apackage.ExceptionHandler.instance();
    inputfilters
      catchfilter : Catch ("Exception") = {[*.getName] exch.log}
      substitute : Substitute = {[*.getName] *.getAddress}
  }
  superimposition {
    selectors
      class = { C | isClassWithName(C, 'apackage.PersonDAO') };
    filtermodules
      class <- SimpleLogExceptionConcern;
  }
}
```

Listing 4.9: Unclear semantics in current composition filters model.

Consider we use the second semantic of the catch filter described in Section 4.3.2. Intuitively, the programmer might come up with the following semantics:

1. Exceptions thrown by *getName* are caught. Since *getName* gets substituted by *getAddress*, no exceptions are caught.

2. Calls to *getName* are substituted by *getAddress*, so exceptions thrown by *getAddress* are caught.

Although the first semantic sounds plausible, the correct semantic is the second one. First, the catch filter accepts messages with selector *getName*. When it accepted the message, the filter also selected its *accept return* action. This action is triggered whenever *any* return (i.e. that is an exception) runs back again through the filters. *getName* gets substituted with *getAddress* by the substitute filter, thus the only exceptions that are caught are the exceptions thrown by *getAddress*.

A positive thing about model A is the fact that it uses a simpler syntax compared to model C. This improves readability and thus reasoning about the concerns.

Amount of ordering : -/+

In the current composition filters model, it is possible to specify an ordering between filtermodules superimposed on the same object. Likewise, it could be possible to specify an ordering between filters (i.e. execute filters of a particular type before filters of an other type), but this can conflict with one characteristic we defined for intuitive semantics, namely that things should be executed in the order in which they are written. In other words, an ordering specification on filter-level hampers the programmer to reason about the control flow. Thus, this is not supported in Compose⋆. However, there exists a simple workaround for this issue. An ordering on filter-level can be accomplished by defining each filter in a separate filtermodule.

Reusability : +

In Compose⋆, it is possible to partially reuse concerns by references or with filtermodule parameters. [11] identifies several drawbacks for reuse by references. For example, when a programmer reuses a filter, he needs to take care that the filtermodule uses the same identifiers for internals, externals and conditions as the "donor" filtermodule. This hampers the adaptability of concerns. It also mentiones the fact that we can rewrite the code that uses the *reuse-by-referencing* construct, into generic code with parameters. For these reasons, we only discuss reuse with filtermodule parameters.

As described in Section 4.3.2, we only need one filtertype (i.e. a catch filter) to express a concern like exception handling. Likewise, we can also use one filter to express a profiling concern, as is shown in Listing 4.10. The concern profiles the execution time of the method `bar()`. Both concerns are examples of reusable concerns. They can easily be used by other concerns. In other words, they can be made generic. For example, Listing 4.11 shows a generic profiling concern. In this case, the generic concern code uses one parameter: a string that identifies the selector of the message. This is just one of the several uses of filtermodule parameters. A complete list of the possibilities can be found in [11]. In general, we can say that the reusability in this model is good thanks to the use of filtermodule parameters.

```
concern ProfilingConcern in apackage {
  filtermodule Profiling {
    inputfilters
        profile : Profiling = {[*.bar]}
  }
  superimposition {
    selectors
      foo = { C | isClassWithName(C, 'apackage.Foo') };
    filtermodules
      foo <- Profiling;
  }
}
```

Listing 4.10: Profiling concern in current composition filters model.

```
concern GenericProfilingConcern in apackage {
  filtermodule Profiling(?method) {
    inputfilters
        profile : Profiling = {[*.?method]}
  }

  superimposition {
    selectors
      foo = { C | isClassWithName(C, 'apackage.Foo') };
    filtermodules
      foo <- Profiling("bar");
  }
}
```

Listing 4.11: Generic profiling concern in current composition filters model.

4.4.2 Criteria applied to Model B

Model B is depicted in Figure 4.7. Applying the criteria, results in the following evaluation:

Intuitive semantics : -/+

Model B exposes some unintuitive semantics according to the following arguments:

Firstly, the two-way composition filters model does not match our intuitive thoughts in the sense that the return messages follow the wrong direction. In the model, a return message first runs through a set of inputfilters superimposed on the *target*-object. Later on, when the return message reaches the original *caller*-object, it runs through the outputfilters superimposed on the caller-object. The other way around is more intuitive (i.e. outputfilters before inputfilters). Furthermore, the filters are not executed in the order they are written on returns.

Secondly, it might be unclear which type of messages the different filtertypes can accept. Filters can define actions on call messages and return messages. However, most of the filters have undefined actions on one of the two events (i.e. calls and returns). This makes reasoning about the concern a bit harder.

Thirdly, in Section 4.3.3 we described two different control flows for return messages. This might confuse the programmer.

Finally, the unclear semantics shown in Listing 4.9 also applies to this model. A programmer might think that the catch filter catches exceptions thrown by `getAddress`, but this is not the case. The matching pattern of the catch filter never evaluates to true, because the substitute filter substitutes `getName` with `getAddress`. Thus, the catch filter never accepts a message.

A positive thing about model B is the fact that it uses a simple syntax, similar to model A.

Amount of ordering : -/+

In the two-way composition filters model, it is possible to specify an ordering between filtermodules superimposed on the same object, similar to model A. The order in which the filters are evaluated inside a filtermodule is fixed for the same reason as explained in Section 4.4.1.

Reusability : +

The two-way composition filters model has the same syntax as the current composition filters model. Therefore, the reusability in both models is the same.

4.4.3 Criteria applied to Model C

Model C is depicted in Figure 4.10. Applying the criteria, results in the following evaluation:

Intuitive semantics : +

Model C exposes the most intuitive semantics of all three models. Unlike in model B, return messages follow the more intuitive direction. It also makes sense to dinstinct between filtersets that accept call messages and filtersets that accept return messages. Furthermore, the filters are executed in the order they are written on returns.

If we express the example concern from Listing 4.9 using this model, then we get the following code:

```
concern SimpleLogExceptionConcern in apackage {
  filtermodule SimpleLogException {
    externals
      exch : apackage.ExceptionHandler = apackage.ExceptionHandler.instance();
    inputfilters
      call
        substitute : Substitute = {[*.getName] *.getAddress}
    outputfilters
      return
        catchfilter : Catch ("Exception") = {[*.getName] exch.log}
  }
  superimposition {
    selectors
      class = { C | isClassWithName(C, 'apackage.PersonDAO') };
    filtermodules
      class <- SimpleLogExceptionConcern;
  }
}
```

Now, it is not likely that a programmer will come up with more than one semantic. The programmer undoubtedly knows that exceptions that are thrown by *getName* are caught . Whether or not *getName* gets executed is another question. This question does not interfere with the question what exceptions are caught, unlike in the other two models.

On the other hand, two extra filtersets and extra keywords (`return` and `call`) result in less concise files and mental overloading, which makes reasoning about concerns a bit harder.

Amount of ordering : +

In model C, it is possible to specify an ordering between filtermodules superimposed on the same object. Additionally, we can specify different filtermodule orderings for call messages and returning messages. The order in which the filters are evaluated inside a filtermodule is fixed.

Reusability : +

This model differs mainly from the other two models in the area of expressing concerns that do something on the call and on the way back (return). An example of such a concern is the profiling concern. Unlike in the other two models, we need two filters to express the profiling concern, as is shown in Listing 4.12. One filter that intercepts a call message (i.e. profile_before) and one filter that intercepts a return message (i.e. profile_after).

```
concern ProfilingConcern in apackage {
  filtermodule Profiling {
    inputfilters
        call
            profile : profile_before = {[*.bar]}
        return
            profile : profile_after = {[*.bar]}
  }
  superimposition {
    selectors
      foo = { C | isClassWithNameInList(C, 'apackage.Foo') };
    filtermodules
      foo <- Profiling;
  }
}
```

Listing 4.12: Profiling concern in model C.

The introduction of the extra filter results in less adaptability. However, when we try to reuse this concern by using the *reuse-by-filtermodule-parameters* construct, as is shown in Listing 4.13, we see that the syntax of the superimposition is the same as in the other two models. In other words, the reusability is as good as in the other two models.

```
concern GenericProfilingConcern in apackage {
  filtermodule Profiling(?method) {
    inputfilters
        call
            profile : profile_before = {[*.?method]}
        return
            profile : profile_after = {[*.?method]}
  }

  superimposition {
    selectors
      foo = { C | isClassWithName(C, 'apackage.Foo') };
    filtermodules
      foo <- Profiling("bar");
  }
}
```

Listing 4.13: Generic profiling concern in model C.

4.4.4 Evaluation

A summary of the evaluation results is shown in Table 4.4.

Table 4.4: Evaluation results composition filters models.

Model	Intuitive Semantics	Ordering	Reusability
A	-/+	-/+	+
B	-/+	-/+	+
C	+	+	+

Based on the results, the best possible solution model is model C. The semantics in model C matches our intuitive thoughts the most. Furthermore, it scores good on ordering. There is no difference between the reusability in the three models.

For conclusion, we here briefly present a list of the advantages and disadvantages of the models:

Model A: Current composition filters model

+ Simple syntax.
+ Only needs small changes to the interpreter to support exception handling.
- Unclear semantics.
- Conditions are not evaluated on returns. This means for instance, that the question whether or not to catch exceptions has to be decided before the execution of a method. In the other two models this can be decided after an exception has been thrown.
- Average amount of filter orderings.

Model B: Two-way composition filters model

+ Simple syntax.
+ Conditions are evaluated on returns.
- Reasoning about the control flow is hard / unclear semantics.
- Average amount of filter orderings.
- The interpreter must be changed to support exception handling.

Model C: Introducing return filters

+ Exposes the most intuitive semantics of the three models.
+ Conditions are evaluated on returns.
+ Highest amount of possible filter orderings.
- A bit more complex syntax.
- The interpreter must be changed to support exception handling.

Chapter 5

Composition filters & Inner classes

This chapter discusses the possibility of expressing crosscutting concerns on inner classes in Compose⋆/J. First, it presents some background information about inner classes. Then, it describes our motivation. Finally, it presents solutions to support this feature in Compose⋆/J.

5.1 Background

In Java, a class definition can contain other class definitions. These classes are called *inner classes*. Inner classes are primarily used for code readability. A program that contains inner classes can be transformed into a program that only contains top-level classes. Actually, this happens when the Java compiler compiles a program. Since the Java Virtual Machine knows nothing about the various types of inner classes, the Java compiler converts them into standard non-nested class files that the Java interpreter understands.

There exist four kinds of inner classes. We explain each of them in the following subsections.

5.1.1 Member classes

A *member class* is a non-static class defined inside another class definition. An object of the member class is internally linked to an object of the enclosing class.

The most important benefit of member classes has to do with accessing members of the enclosing classes. The methods of a member class have direct access to all members of the enclosing classes, including private members. Thus, the use of member classes eliminates the requirement to connect objects together via constructor parameters.

Listing 5.1 illustrates the use of a member class. In the example, class `B` is a member class of class `A`. As we can see, class `B` has access to the members of class `A`. In this case, method `printA` of class `B` prints out the value of the instance variable `aVar` of class `A`.

```java
public class A
{
    private int aVar;

    class B // member class
    {
        public void printA()
        {
            System.out.println("aVar has value "+aVar);
        }
    } // end class definition B

} // end class definition A
```

Listing 5.1: Example member class

5.1.2 Local classes

A *local class* is a class that is defined within a block of Java code. Local classes are most frequently defined within methods and constructors, but they can also be defined elsewhere (e.g. static initializers blocks or instance initializers). Similar to member classes, an object of a local class is internally linked to an object of the enclosing class.

Objects instantiated from local classes share many of the characteristics of objects instantiated from member classes (e.g. direct access to members of enclosing classes). However, a local class can be defined closer to its point of use than would be possible with a member class, leading to improved code readability.

Listing 5.2 illustrates the use of a local class. In the example, class `B` is a local class of class `A`. It is defined inside the method block `meth` of class `A` (lines 7–13). As with local variables, the class definition for a local class must appear before the code that attempts to instantiate the class (line 15).

```java
 1  public class A
 2  {
 3    private int aVar;
 4
 5      public void meth()
 6      {
 7          class B // local class
 8          {
 9              public void printA()
10              {
11                  System.out.println("aVar has value "+aVar);
12              }
13          } // end class definition B
14
15          B obj = new B();
16          obj.printA();
17      }
18
19  } // end class definition A
```

Listing 5.2: Example local class

5.1.3 Anonymous classes

An *anonymous class* is essentially a local class without a name. An anonymous class is defined and instantiated in a single expression using the new operator. While a local class definition is a statement in a block of Java code, an anonymous class definition is an expression, which means that it can be included as part of a larger expression, such as a method call. When a local class is used only once, it can be replaced by an anonymous class.

Listing 5.3 illustrates the use of an anonymous class. In the example, the anonymous class definition appears as an argument of a method call (lines 6–12). Note that the object is not instantiated from the class `WindowAdapter`, but from a subclass of `WindowAdapter` that does not have a name (line 5).

```
 1  public class GUI extends Frame
 2  {
 3      public GUI()
 4      {
 5          addWindowListener(new WindowAdapter()
 6              { // begin anonymous class definition
 7                  public void windowClosing(WindowEvent e)
 8                  {
 9                      System.out.println("Close button clicked");
10                      System.exit(0);
11                  }
12              } // end anonymous class definition
13          ); // end addWindowListener
14      } // end constructor
15
16  } // end class definition GUI
```

Listing 5.3: Example anonymous class

5.1.4 Nested top-level classes

A *nested top-level class* is a static class defined inside another class definition. Unlike the other kinds of inner classes, an object of a nested top-level class is not internally linked to an object of the enclosing class. A nested top-level class behaves the same as a normal top-level class. The difference is that the name of a nested top-level class includes the name of the class in which it is defined.

Nested top-level classes are typically used as a convenient way to group related classes.

Listing 5.4 illustrates the use of a nested top-level class. In the example, two nested top-level class definitions (`Rectangle` and `Circle`) appear in the class definition of class `Shape` (lines 4–9).

```java
public class Shape
{
    // two nested top-level class definitions
    public static Rectangle extends Shape {

    }
    public static Circle extends Shape {

    }
}
```

Listing 5.4: Example nested top-level class

5.2 Motivation & Demonstrating Example

As described in Section 5.1, inner classes are primarily used for code readability. During a compilation process, inner classes are transformed to top-level classes. This means that cross-cutting concerns (e.g. logging) that apply to top-level classes may also apply to inner classes. This motivates our idea to support advice code weaving on inner classes in Compose★/J.

Listing 5.5 presents a demonstrating example. In the example, we use an anonymous class (lines 5–13). Our goal is to add a logging concern to method `windowClosing` of the anonymous class (line 9).

```java
 1  public class GUI extends Frame
 2  {
 3      public GUI()
 4      {
 5          addWindowListener(new WindowAdapter()
 6              {
 7                  public void windowClosing(WindowEvent e)
 8                  {
 9                      // printing handled by concern...
10                      System.exit(0);
11                  }
12              }
13          );
14          setVisible(true);
15      }
16  }
```

Listing 5.5: Demonstrating example

Currently, the Compose* language possesses the tools to express the above concern with composition filters as is shown in Listing 5.6. An anonymous class does not have a name, but as described in Section 5.1.3, it is a subclass of an existing class (or a class that implements a particular interface). We can use this information in the selector definition. The problem of this approach is that the selector selects all subclasses of `WindowAdapter` (lines 10–11). In some scenarios, we may want to select one specific class or a specific group of classes (e.g. only anonymous classes). The next section presents a solution to this problem by extending the selector language.

```
concern WindowEventLogConcern {
    filtermodule WindowEventLog {
        externals
            logger : Logger = Logger.instance();
        inputfilters
            log : Meta = { [*.windowClosing] logger.log }
    }
    superimposition {
        selectors
            anonymous = { C | isClassWithName(Super, "java.awt.event.WindowAdapter")
                                    , isSuperClass(Super, C)  }
        filtermodules
            anonymous <- WindowEventLog;
    }
}
```

Listing 5.6: Applying concern on anonymous class with current Compose⋆ language toolset.

5.3 Extending selector language

The selector definition in Listing 5.6 gives all subclasses of `WindowAdapter`. We may want to select a more specific set of classes. In order to achieve this, we expand the selector language in Compose⋆/J with the following predicates:

```
isInnerClass(Class).
isDefinedWithin(InnerClass, DefinedInClass).
isDefinedWithin(InnerClass, DefinedInMethod).

isAnonymous(Class).
isLocal(Class).
isMember(Class).
isNested(Class).
```

The first predicate makes it possible to select all classes that are inner classes. The second and third predicate allow us to be even more specific. With these predicates, we can select all inner classes defined within a particular class or method.

The last four predicates are used to distinct between the four different kinds of inner classes.

Listing 5.7 demonstrates the use of the extra predicates. The selector `anonymous` only selects anonymous classes, enclosed in the class `GUI`, that extend the functionality of class `WindowAdapter`.

```
selectors
    anonymous = { C | isClassWithName(Super, "java.awt.event.WindowAdapter"),
            isSuperClass(Super, C), isAnonymous(C),
            isDefinedWithin(C, DefClass),   isClassWithName(DefClass, "aPackage.GUI") }
```

Listing 5.7: More specific selector that uses predicates specified for inner classes.

5.4 Implementation issues

The previous section describes the changes in the Compose⋆/J selector language to support expressing crosscutting concerns on inner classes. This section addresses some implementation issues that exist when implementing these changes.

5.4.1 Dummy generation

In order to let the selector language function properly, we need to gather the type information from our application. In Compose⋆/J this is done by using reflection, which means the program sources need to be compiled first. Since composition filters can create a mismatch between the class interfaces and real interfaces (signature mismatch) errors can occur while compiling the program sources. A solution was found in the form of using dummy sources [20].

Currently, dummy sources are created from program sources by replacing all method bodies with empty or default return statements. Since some class definitions can exist in the scope of method blocks (i.e. local and anonymous classes), this may lead to loss of type information. Thus, special care should be taken with local and anonymous classes. All class definitions must be preserved in the dummy sources.

5.4.2 Weaving technique : interpreter vs inlining

Two main alternatives exist to weave advice code into a target program.

Currently, Compose⋆/J uses an interpreter-based weaving approach. This means that calls inside the target program are replaced with interpreter calls. The interpreter uses filter information, stored in a central repository, to run the actual advice code.

The other approach is the inlining technique, which is based on inserting the actual advice code directly into the target program. The primary advantage of using inlining is performance gain due to the absence of an interpreter. Its implementation is a bit more complex than the interpreter-based approach.

The interpreter-based approach exposes a drawback when we try to weave on inner classes. Suppose we want to weave the code belonging to the concern described in Section 5.2. In the interpreter-based approach we replace the call to `windowClosing` with an interpreter call. In this case, the example is an example of a call-back and the call takes place in a system library. Weaving in a system library is not recommended, since another running program can use the same library. Besides that, we do not exactly know where the actual call is made inside the system library. This problem does not apply to inner classes only. It applies to call-back situations to a system library in general.

The inlining technique solves this problem partly. If we use an inputfilter to express the concern, than the actual advice code is woven in the beginning of method `windowClosing`. We do not have to weave in a system library. If we use an outputfilter than this problem still exists (i.e. we still need to replace a call).

We conclude that the interpreter-based approach is insufficient when dealing with call-back situations. The inlining technique is more suitable for this job.

5.5 Summary and Conclusion

In Java, there exist four kinds of inner classes. They are primarily used for code readability. At compile-time, inner classes are transformed to top-level classes, which means that crosscutting concerns that apply to top-level classes may also apply to inner classes. This motivates us to support expressing crosscutting concerns on inner classes.

To distinct between inner classes and standard top-level classes, we introduced some new predicates in the selector language of Compose⋆/J. Finally, we identified some implementation issues. We concluded that the inlining weaving approach is more suitable than the interpreter-based approach when dealing with call-back situations.

Chapter 6

Composition filters & Java interfaces

This chapter discusses the possibilities and benefits for weaving on Java interfaces in Compose⋆/J. First, it presents some background information about Java interfaces. Then, it describes our motivation for interface-based weaving. Finally, it proposes a solution to support this feature in Compose⋆/J.

6.1 Background

What is an Java interface?

An interface in Java is a group of related methods with empty bodies. They form a contract between classes and the outside world. It is a type that defines *what* should be done, but *now how* to do it. The actual implementation is done by the class that *implements* an interface.

A simple interface example

Listing 6.1 shows an example of an interface declaration. An interface is declared by using the keyword *interface*. Similar to classes, an interface can inherit functionality. In the example, we have declared the interface `List`. It inherits functionality from the interface `Collection`. Furthermore, the example shows two empty method declarations: `get()` and `isEmpty()`.

```java
public interface List extends Collection
{
    public Object get(int index);

    public boolean isEmpty();

    //...
}
```

Listing 6.1: Interface declaration

Implementing an interface

Listing 6.2 shows how a class implements an interface. A class implements an interface by using the keyword *implements*. A class which implements an interface must either implement all methods in the interface, or be an abstract class. In this case the class `ArrayList` is not abstract and it implements the interface `List`, which means it implements all methods declared in `List`.

```java
public class ArrayList implements List
{
    public Object get(int index) {
        //...
    }

    public boolean isEmpty() {
        //...
    }
}
```

Listing 6.2: Implementing an interface

Why using Java interfaces?

Java interfaces offer the following benefits:

They speed up the development process. By using Java interfaces, a team of developers can quickly establish integration among the application objects without knowing the exact implementations of these objects. This enables developers to work simutaneously and concentrate on their development tasks without having to worry about the integration.

They improve maintainability. Java interfaces improves the maintainability of an application in a couple of ways. Firstly, if a class implements an interface, the interface type can be used as the reference type for instances of that class. This means, that the actual implementation can be swapped out without breaking the code. Secondly, if we change an interface, the Java compiler automatically identifies which classes needs to be changed as well. This reduces the investigation of the possible impact of changes to an interface.

They improve readability. Interfaces improve readability, in the sense that they give programmers a second, concise location to overview what a class does.

Multiple inheritance

Java interfaces is Java's answer to multiple inheritance. In Java, a class can only extend from one superclass. Multiple inheritance of classes is not allowed. However, a Java class may implement any number of interfaces. Using this characteristic and Java's delegation technique, multiple inheritance can be simulated, as is shown in Listing 6.3.

The example shows three classes (`A`,`B` and `C`) and one interface (`interfaceX`). Class `C` extends from class `A`.

Now, class `C` can also extend from class `B` by using an interface and Java's delegation technique. First, class `C` and class `B` should both implement the interface `interfaceX`. To accomplish multiple inheritance, class `C` delegates all messages defined in the interface `interfaceX` to an instance of class `B`.

```java
public interface interfaceX
{
    public void foo();
}

public class A
{
    public void bar(){
        System.out.println("Inheriting from A!")
    }
}

public class B implements interfaceX
{
    public void foo(){
        System.out.println("Inheriting from B!");
    }
}

public class C extends A implements interfaceX
{
    private B otherParent;

    public C(){
        otherParent = new B();
    }

    public void foo(){
        otherParent.foo();
    }
}
```

Listing 6.3: Simulating multiple inheritance in Java

6.2 Motivation & Demonstrating example

Currently, Compose⋆/J only supports weaving on classes. This section presents our motivation for weaving on Java interfaces.

In general, suppose we superimpose filters on a class that implements an interface. These filters can create a signature mismatch in the form of a growing interface, which means that the filters can introduce new methods to the interface[1] of a class [20].

In the case of a growing interface, the new methods should be available to call on all instantiations of the superimposed class [20]. The Compose⋆/J compiler solves this issue by adding default implementations of the methods to the class definition. However, the method declarations are not added to the Java interface. This can result in unavoidable castings to the implementing class if we want to make a call to the new method. We cannot use any references to the interface type because that results in a compile error.

To demonstrate this, consider the concern in Listing 6.4. In the example, we *humanize* [2] all classes that implement the interface `List` by adding two new methods: `first()` and `last()`.

[1]The conceptual nature, not the concrete Java programming language construct.

[2]In general, there exist two types of interfaces: ”humane” interfaces and ”minimal” interfaces. The idea behind the ”minimal” interface is to design an API that allows the client to do everything they need to do, but boils down

To accomplish this, we use a dispatch filter (line 6) that dispatches calls to `first()` and `last()` to an instance of the class `HumanizedList`. In the selector we select all classes that implement the interface `List` (line 11). The implementations of the methods `first()` and `last()` use a static method call to the class `MessageInfo` to retrieve the inner object (line 30 and line 44).

```
 1 concern HumanizeListConcern {
 2     filtermodule HumanizeList {
 3         externals
 4             h : aPackage.HumanizedList = aPackage.HumanizedList.instance();
 5         inputfilters
 6             disp : Dispatch = { [*.first] h.first, [*.last] h.last }
 7     }
 8 
 9     superimposition {
10         selectors
11             classes = { C | isClass(C), isInterfaceWithName(I,"java.util.List"),
12                         classImplementsInterface(C,I) }
13         filtermodules
14             classes <- HumanizeList;
15     }
16 
17     implementation in Java by aPackage.HumanizedList as "HumanizedList.java"
18     {
19         package aPackage;
20 
21         import java.util.List;
22 
23         private static HumanizedList instance = null;
24 
25         public class HumanizedList
26         {
27             public Object first() {
28 
29                 List list = null;
30                 Object inner = Composestar.Runtime.FLIRT.message.MessageInfo
31                             .getMessageInfo().getInner();
32 
33                 if (inner instanceof List) {
34                     list = (List)inner;
35                     return list.get(0);
36                 }
37 
38                 return null;
39             }
40 
41             public Object last() {
42 
43                 List list = null;
44                 Object inner = Composestar.Runtime.FLIRT.message.MessageInfo
45                                 .getMessageInfo().getInner();
46 
47                 if (inner instanceof List) {
48                     list = (List)inner;
49                     return list.get(list.size()-1);
50                 }
```

the capabilities to the smallest reasonable set of methods that will do the job. A ”humane” interface, on the other hand, considers typical uses of the interface and provides convenience methods as a part of the interface itself.

```

                return null;
            }

            public static HumanizedList instance()
            {
                if(instance == null)
                {
                    instance = new HumanizedList();
                }
                return instance;
            }
        }
    }
}
```
Listing 6.4: Example concern that results in a growing interface (i.e. methods `first()` and `last()` are added to all classes that implement the `List` interface).

Now, Listing 6.5 shows a possible way of using this concern in practice. First we retrieve a `List` (line 1). Since, the Compose★/J weaver did not weave the new methods in `List`, we cannot use the reference to `List` (line 3). We explicitly need to cast to the implementing class (line 4). As you can imagine, these castings can lead to runtime errors. The method `getList()` can return any class that implements the interface `List`. In other words, supporting only class-based weaving downgrades the use of Java interfaces. By weaving on Java interfaces, we can eleminate the problems mentioned above.

```
1 List l = getList();
2
3 l.first(); //results in a compile error
4 (ArrayList)l.first(); // casting is needed
```
Listing 6.5: In practice we cannot use references to interface types

The next section proposes a solution for modularizing interface-based weaving in the composition filters model.

6.3 Solution Proposal

Currently, programmers can only superimpose composition filters on classes. However, the selector language of Compose★ possesses the tools to select interfaces as well. Listing 6.6 shows a list of predicates that can be used in the selector definition to select interfaces. For example, we can select all private interfaces by executing the query ”isInterfaceWithAttribute(Interface, ’private’)”.

```
isInterface(Interface).
isInterfaceWithName(Class).
isInterfaceWithAttribute(Class).
```
Listing 6.6: Interface predicates.

To solve the problems mentioned in Section 6.2 we need to weave on Java interfaces. Now, a possible way to tell the Compose★/J weaver to weave explicitly on Java interfaces is to select an interface in the selector definition, as is shown in Listing 6.7. A problem with this approach is that it may seem that we are trying to superimpose filters on instances of Java interfaces, which sounds awkward. Messages are not sent to Java interfaces. Messages are sent to objects, which are always instances of classes. Java interfaces are *abstract*, they cannot be directly instantiated. In other words, putting a filter on a Java interface makes no sense.

To avoid this, we say that selecting an interface in the selector definition is purely syntactic sugar for selecting all classes that implement the interface, but it additionally has some semantic meaning, namely that it tells the weaver to weave on the Java interface as well.

```
concern HumanizeListConcern {
    filtermodule HumanizeList {
        externals
            h : aPackage.HumanizedList = aPackage.HumanizedList.instance();
        inputfilters
            disp : Dispatch = { [*.first] h.first, [*.last] h.last }
    }

    superimposition {
        selectors
            classes = { I | isInterfaceWithName(I,"java.util.List")}
        filtermodules
            classes <- HumanizeList;
    }

    //implementation
}
```

Listing 6.7: Weaving on Java interfaces by selecting an interface in the selector definition.

6.4 Summary and Conclusion

An interface in Java is a group of related methods with empty bodies. They form a contract between classes and the outside world. Java interfaces speed up the development process, improve maintainability and improve readability.

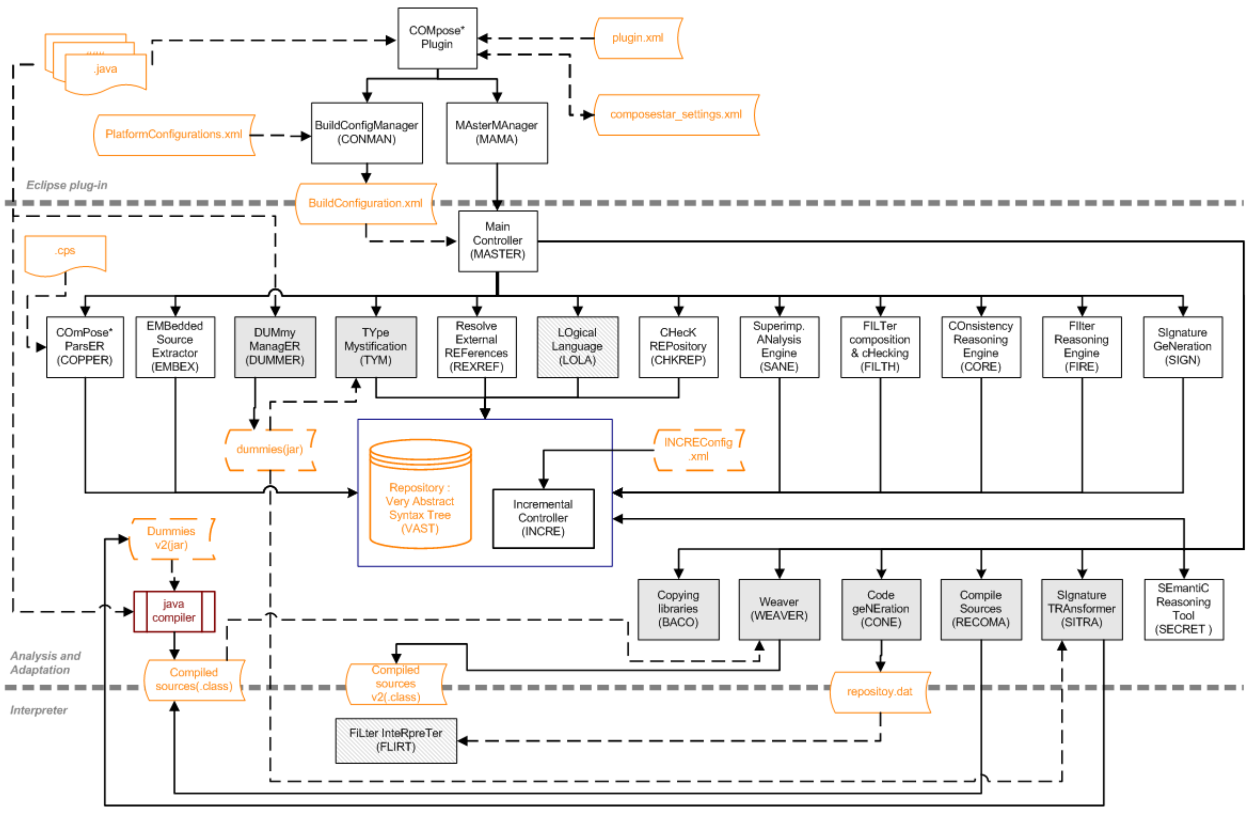
Currently, Compose⋆ only supports class-based weaving, which means that the Compose⋆/J weaver does not weave on Java interfaces. This restriction can lead to unwanted results when we try to superimpose filters on classes that implement an interface. In particular, problems arise when the filters create a signature mismatch in the form of a growing interface. These problems include redundant castings and runtime errors. To avoid downgrading the use of Java interfaces, we concluded that Compose⋆/J should support interface-based weaving.

In this chapter we proposed a solution for modularizing interface-based weaving in the form of selecting interfaces in the selector definition. We defined that selecting an interface is purely syntactic sugar for selecting all classes that implement the interface. It additionally tells the Compose⋆/J weaver to weave on the selected interface.

Chapter 7

Design of Compose⋆/J

As explained in Chapter 2, the architecture of Compose*/J is divided in four layers: IDE, compile-time, adaptation and runtime. The compile-time layer reasons about the composition filters model, and thus is considered language independent. All implementations of Compose⋆ share the analysis tools that exist in this layer. The other three layers are partially language dependent. Figure 7.1 shows a more detailed figure of the Compose⋆/J architecture. In the figure, the compile-time layer and adaptation layer are integrated as one layer, called the *Analysis and Adaptation*-layer. This layer consists of modules. The modules that are java specific are greyed out, the language independent modules are filled with white color.



.java
COMpose* Plugin
plugin.xml
composestar_settings.xml
PlatformConfigurations.xml
BuildConfigManager (CONMAN)
MAsterMAnager (MAMA)
Eclipse plug-in
BuildConfiguration.xml
Main Controller (MASTER)
.cps
COmPose* ParsER (COPPER)
EMBedded Source Extractor (EMBEX)
DUMmy ManagER (DUMMER)
TYpe Mystification (TYM)
Resolve External REFerences (REXREF)
LOgical Language (LOLA)
CHecK REPository (CHKREP)
Superimp. ANalysis Engine (SANE)
FILTer composition & cHecking (FILTH)
COnsistency Reasoning Engine (CORE)
FIlter Reasoning Engine (FIRE)
SIgnature GeNeration (SIGN)
dummies(jar)
Repository : Very Abstract Syntax Tree (VAST)
Incremental Controller (INCRE)
INCREConfig .xml
Dummies v2(jar)
java compiler
Copying libraries (BACO)
Weaver (WEAVER)
Code geNEration (CONE)
Compile Sources (RECOMA)
SIgnature TRAnsformer (SITRA)
SEmantiC Reasoning Tool (SECRET)
Analysis and Adaptation
Compiled sources(.class)
Compiled sources v2(.class)
repositoy.dat
Interpreter
FiLter InteRpreTer (FLIRT)

Figure 7.1: A detailed overview of the Compose⋆/J architecture.

This chapter discusses the design choices made in the layers and presents the design of the language specific parts of Compose⋆/J. We discuss each layer, except the compile-time layer, in a separate section.

7.1 Integrated Development Environment

The Compose⋆/J compiler uses a build configuration file for building a Compose⋆ application. This build configuration file contains information about the source files, concerns and settings of the Compose⋆/J compiler. For automatic creation of such a build configuration, Compose⋆/J interfaces with a native Integrated Development Environment (IDE).

There are many IDE's on the Java platform. The most popular are NetBeans [29] and Eclipse [13]. We use Eclipse as our IDE platform for Compose⋆/J, because of its following characteristics:

Platform for multiple IDE's. Eclipse is an extensible platform for building IDE's. Besides the built-in Java IDE, there are language IDE's for most of the popular programming languages, such as C/C++ Development Tooling (CDT). This saves us some effort in writing plug-ins for other Compose⋆ implementations. An example of such an implementation is Compose⋆/C [38], which uses an Eclipse plug-in as well.

Extensible nature. Eclipse has an open architecture, making it extensible. This extensible nature of the Eclipse Platform enables us to easily add other features (e.g. debugging) or use other applications in the future.

7.1.1 Eclipse Architecture

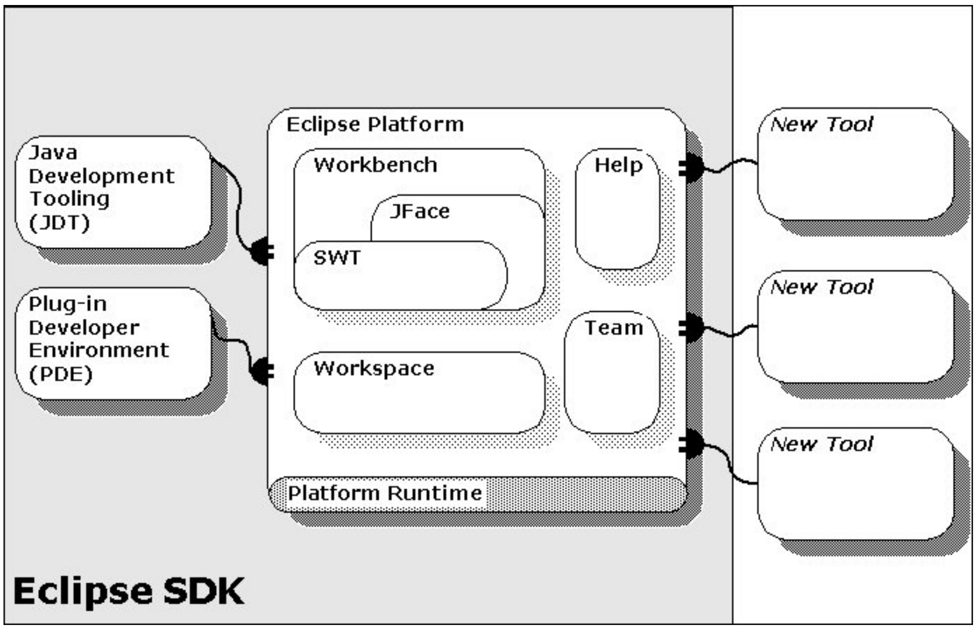


Figure 7.2: Overview of the Eclipse architecture.

Figure 7.2 illustrates an overview of the Eclipse architecture. The Eclipse SDK includes the Eclipse Platform, a Java Development Tool (JDT) and the Plug-in Development Environment (PDE). The JDT and PDE are plug-ins to the Platform.

The Eclipse Platform is built on a mechanism for discovering, integrating, and running plug-ins. A plug-in is the smallest unit of function that can be developed and delivered separately. Plug-ins may rely on services provided by other plug-ins (”extensions”) or provide services on which yet other plug-ins may rely (”extension points”). Usually a small tool consists of a single plug-in. A complex tool has its functionality split over multiple plug-ins.

The Eclipse Platform consists of a couple of components that contain a bunch of built-in plug-ins. These plug-ins provide standard services that can be used by the Compose⋆/J plug-in. Section 7.1.2.2 describes which services the plug-in uses. The major components of the Eclipse Platform are:

Platform Runtime. A small kernel that loads the plug-ins.

Workbench. Component that implements the graphical interface of Eclipse, and its subcomponents JFace (i.e. a toolkit with classes for handling many common UI programming tasks) and the Standard Widget Toolkit (SWT). E.g. it provides services to create dialogs and wizard pages.

Workspace. Component that ”holds” information about the development environment. E.g. it provides services to extract project information.

Team. Component that adds version and configuration management (VCM) capabilities to projects in the Workspace. It handles the issues of checking-in and checking-out code versions when a group of developers are working on the project.

Help. Component that provides online documentation and context-sensitive help to applications.

The Compose⋆/J plug-in is a new tool that can be integrated in the Eclipse Platform. Below, we present the design of this plug-in.

7.1.2 Design of the Compose⋆/J Eclipse plug-in

This section presents the design of the Compose⋆/J Eclipse plug-in. It first presents a use case diagram that shows what actions a user can do with the plug-in. After that, it describes which services provided by the Eclipse SDK the plug-in uses.

7.1.2.1 Use case diagram

Figure 7.3 shows a use case diagram of what actions a user can do with the plug-in. The diagram defines the following actions:

1. **Create a Compose⋆/J project.** The action that creates a Compose⋆/J project. A user can create a Compose⋆/J project by using a wizard. The wizard acts similar as the wizard for creating a standard Java project, found in the JDT plug-in. The difference is that the Compose⋆/J wizard adds the Compose⋆/J libraries to the Java project.

2. **Configure compiler settings.** The action that configures the compiler settings. A user can change the settings of the Compose⋆/J compiler either globally (i.e. each new Compose⋆/J project receives the initial global settings) or per project.

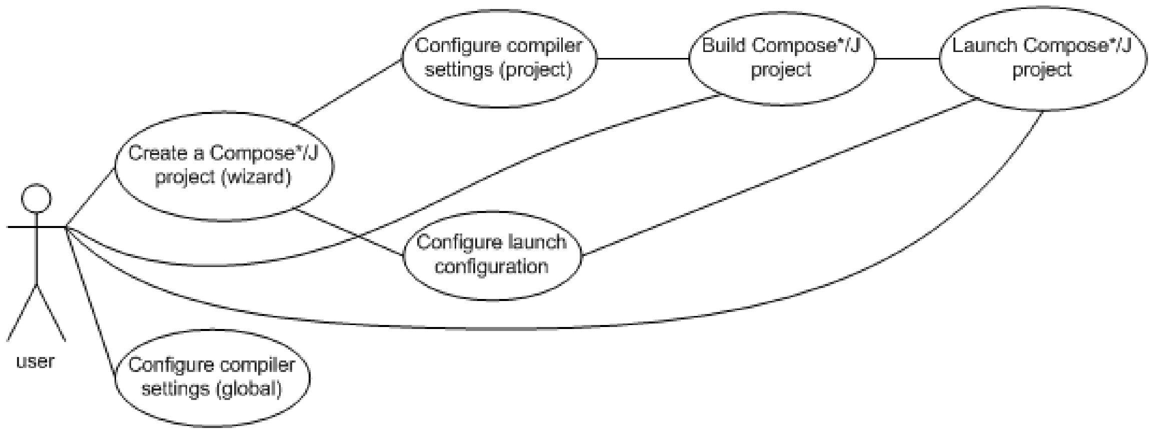

Configure compiler settings (project)
Build Compose*/J project
Launch Compose*/J project
Create a Compose*/J project (wizard)
Configure launch configuration
user
Configure compiler settings (global)

Figure 7.3: Use case diagram: Compose*/J Eclipse plug-in actions.

3. **Build a Compose⋆/J project.** The action that builds a Compose⋆/J project. The build process consists of two steps. First, a build configuration file is created that contains the project information and compiler settings. After that, the Compose⋆/J compiler is triggered with the build configuration file as input.

4. **Configure launch configuration.** The action that configures a Compose⋆/J launch configuration. Similar to creating a launch configuration for Java applications, a user can create a launch configuration for Compose⋆/J projects. The launch configuration automatically adds the Compose⋆/J runtime libraries to the classpath.

5. **Launch a Compose⋆/J project.** The action that launches a Compose⋆/J project.

7.1.2.2 Using services provided by Eclipse

As described in Section 7.1.1, the Eclipse Platform provides standard services that simplify the process of implementing a plug-in. Such a service has an unique identifier. The Compose⋆/J plug-in uses the following standard services:

1. **org.eclipse.ui.newWizards** This extension point provides a way to register resource creation wizard extensions. When using this service the wizard is automatically added to the graphical interface of Eclipse. The Compose⋆/J plug-in uses this service for registering the wizard that creates a new Compose⋆/J project. This service is found in the Workbench component.

2. **org.eclipse.ui.preferencePages** The workbench provides one common dialog box for preferences. This extension point provides a way to add pages to the preference dialog box. The Compose⋆/J plug-in uses this service to add the preference page that enables a user to change the global Compose⋆/J compiler settings. This service is found in the Workbench component.

3. **org.eclipse.ui.propertyPages** This extension point provides a way to add additional property pages to objects of a given type. The Compose⋆/J plug-in uses this service to add a property page to a project that enables a user to change the Compose⋆/J compiler settings. This service is found in the Workbench component.

4. **org.eclipse.ui.popupMenus** This extension point provides a way to add menuitems to a popupmenu to objects of a given type. The Compose⋆/J plug-in uses this service to add a "build" and "launch" menuitem to the popupmenu that popups when right-clicking on a project. This service is found in the Workbench component.

5. **org.eclipse.debug.ui.launchConfigurationTabGroups** This extension point provides a mechanism for contributing a group of tabs to the launch configuration dialog for a type of launch configuration. The Compose⋆/J plug-in uses this service to create a tabgroup to configure the launch of a Compose⋆/J application. This service is found in the Workbench component.

6. **org.eclipse.debug.core.launchConfigurationTypes** This extension point provides a configurable mechanism for launching applications. When using this service the launch configuration is automatically added to the launch configuration dialog. The Compose⋆/J plug-in uses this service to add the launch configuration mechanism for Compose⋆/J applications to the Eclipse workbench.

7. **org.eclipse.jdt.launching.classpathProviders** This extension point provides a way to dynamically compute and resolve classpaths and source lookup paths for Java launch configurations. The Compose⋆/J plug-in uses this service to compute the classpath for the Compose⋆/J launch configuration. This service is found in the JDT plug-in.

7.2 Adaptation

The adaptation layer in Compose⋆/J consists of components that connect the platform independent compile-time layer to the Java platform. The result of the adaptation layer is a target program that interfaces with the Compose⋆/J runtime. This section describes the design choices made for the language specific parts of this layer in Compose⋆/J.

7.2.1 Collecting type information

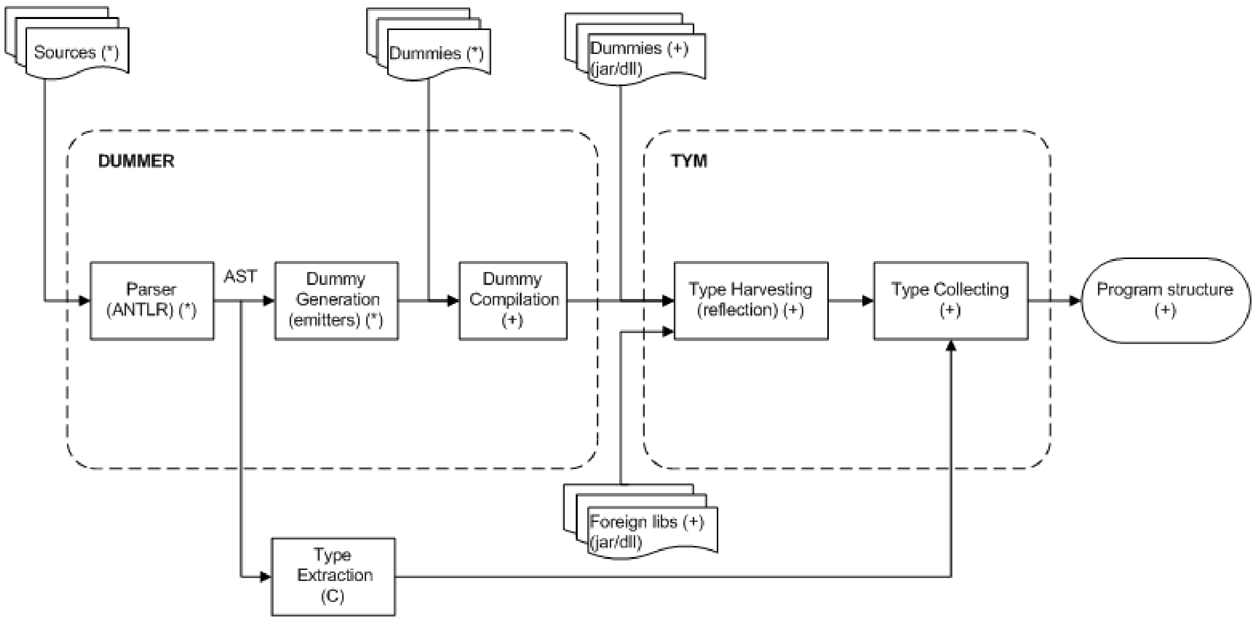


Figure 7.4: Uniform approach for collecting type information.

In order to reason about concerns, the Compose⋆/J compiler needs to collect the structure and annotations within the source program and store it in a knowledgebase. Since Compose⋆ is a language independent solution, it is useful that this is done in a uniform way in each Compose⋆ implementation. Figure 7.4 presents such a uniform approach. An asterix (*) in the model indicates a language specific procedure or output (e.g. Java or C#). A plus sign (+) indicates a platform specific procedure or output (e.g. Java or .NET).

Since most OO-languages support a reflection mechanism, we use this mechanism for collecting type information. This requires compilation of all program sources, but since composition filters can create a mismatch between the class interfaces and real interfaces (signature mismatch), errors can occur while compiling these sources. A solution for this problem was found in the form of using dummy sources [20].

Dummy sources are created from program sources by replacing all method bodies with empty or default return statements. A solution for creating these dummy sources, is to use a parser for each language. Fortunately, we can use ANTLR as the sole parser generator since it supports grammars for a lot of OO-languages. An apprehensive list of supported languages can be found on the ANTLR-site [1].

ANTLR creates an Abstract Syntax Tree (AST) for each program source. This tree is used to create a dummy source. After the dummy sources are created, the dummy sources are compiled. This process is performed in one module, called `DUMMER`.

A second module, called `TYM` (TYpe Mystification), uses the compiled dummies to collect the type information. This process is done in two procedures. The first procedure (Type Harvesting) uses reflection to retrieve the type information from the compiled dummies. The second procedure (Type Collecting) collects the types and stores them into the repository.

Compose⋆/J uses this general approach for collecting type information.

An example of a Compose⋆ implementation that uses an other approach is Compose⋆/C. Compose⋆/C uses source code weaving, thus it does not need to create dummies. The type information is directly retrieved from the Abstract Syntax Tree's.

7.2.2 Weaving

A crucial part of the adaptation layer is the weaving process. In context of Compose⋆, weaving is the process that manipulates a target program with changes introduced by the composition filters. This section describes the design choices made for weaving in Compose⋆/J.

7.2.2.1 Determining the weaving process

There exist many approaches for AOP weaving. A complete domain analyses is found in [41]. To determine the approach in Compose⋆/J, we answer the questions below.

How to weave?

The first question we answer is *how* to weave. There exist three alternatives:

- Source code modification: changing the source code of the program directly. This happens before the program is running (i.e. at compile time). This requires source code to be available.
- Byte code modification: changing the byte code of the program. This happens after the program has been compiled. In Java, this can happen either at compile time or at load time (i.e. when class is loaded by a class loader). This alternative does not require source code to be available.
- Reflection: changing the code with a reflection mechanism. This happens when the program is running (i.e. at runtime). The reflection mechanism is used to determine the exact position in the code that is being executed.
- VM-level modification: adding AOP-knowledge to a VM. This alternative requires a JVM that internally keeps track of the joinpoints (e.g. method calls) and executes advice code at these joinpoints.

Selected: Byte code modification

The reflection mechanism is a costly operation, which results in poor performance at runtime.

At the moment, there does not exist a stable JVM that supports AOP, although they are being developed (e.g. JRockit [24]). Changing a JVM by ourselves takes too much time. Furthermore, using a JVM that supports AOP decreases our audience, since every user must install this JVM to be able to use Compose⋆/J.

This leaves us with either source code or byte code modification. We choose byte code modification, since it has the advantage that it does not require source code to be available. Furthermore, it is simpler, because byte code is normalized, i.e. it does not contain syntactic sugar.

What to weave?

The second question we answer is *what* to weave. There exist two alternatives:

- Actual code: inserting the advice code directly in the code where it is needed. Once the program is executed, the advice code is executed automatically.
- Hooks: inserting calls to a process which decides which advice code to execute. These calls are called hooks.

Selected: Hooks

The first alternative requires a complete model of what code to weave. The second alternative is more easier to implement. This alternative is also used by Compose⋆/.NET. It uses hooks to an interpreter to execute filter advise code [16]. The main part of this interpreter was written in J#, with the intention to support porting it to the Java platform.

When to weave?

The third question we answer is *when* to weave. There exist three alternatives:

- Compile time: changing the byte codes of a program while the program is not running (i.e. byte code of the files stored on disk are changed).
- Load time: changing the byte codes of a program, as the classes, of which the program exists, are loaded into memory by a class loader.
- Runtime: changing the behavior of a program while the program is running by linking aspects and objects at runtime.

Selected: Compile-time

Currently, the Compose⋆ language does not express dynamic weaving at runtime, so weaving at runtime is not useful yet. This leaves us with either compile-time or load-time weaving. Load-time weaving is virtual machine dependent, which requires different implementations to do load-time weaving. We only need one implementation to do compile-time weaving, so we choose compile-time weaving.

7.2.2.2 Selecting a byte code manipulator

As described in Section 7.2.2.1, we use byte code modification as our weaving mechanism in Compose⋆/J. A couple of byte code manipulation tools exist on the Java platform. In this section we shortly describe three of them, compare them and select one for Compose⋆/J.

BCEL

BCEL (Byte Code Engineering Library) [3] is a toolkit for the static analysis and dynamic creation or transformation of Java class files. It enables developers to implement the desired features on a low level of abstraction (i.e. byte code instructions). Therefore, it has a high learning curve.

ASM

ASM [2] is a Java byte code manipulation framework. It can be used to dynamically generate stub classes or other proxy classes, directly in binary form, or to dynamically modify classes at load time. It offers similar functionalities as BCEL, but is much smaller and faster.

Javassist

Javassist (Java Programming Assistant) [23] makes Java byte code manipulation simple. Unlike BCEL and ASM, Javassist provides two levels of API: source level and byte code level. Using the source level API, it is possible to edit a class file without knowledge of the specifications of the Java byte code. On the other hand, the byte code level API allows programmers to directly edit a class file as well, like BCEL and ASM.

Selected: Javassist

We only need simple instructions to implement the weaving functionality in Compose⋆/J (i.e. adding methods or changing a single statement). All three tools support these kind of instructions.

From performance point of view, ASM is the best option. It is significantly smaller and faster than BCEL and Javassist. On the other hand, Javassist is easiest to use. Since performance is not the most important requirement in Compose⋆/J, and due to the short time frame for implementing Compose⋆/J, our choice is Javassist. If performance becomes an issue, then switching to ASM is recommended.

7.2.2.3 Signature transformation dummies

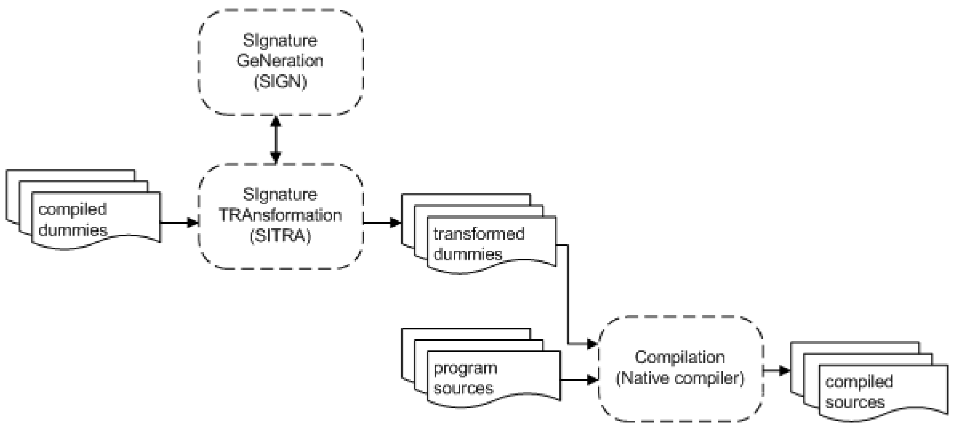

SIgnature GeNeration (SIGN)
compiled dummies
SIgnature TRAnsformation (SITRA)
transformed dummies
program sources
Compilation (Native compiler)
compiled sources

Figure 7.5: Compilation process of program sources in Compose⋆/J

As described in Section 7.2.1, dummy sources are used for collecting type information, but these dummy sources serve another purpose as well, as is shown in Figure 7.5.

As described in Section 7.2.1, composition filters can create a mismatch between the class interfaces and real interfaces (signature mismatch). Thus, errors can occur while compiling the program sources. Now, in order to compile the program sources, the Compose⋆/J compiler transforms the signatures of the compiled dummy sources, using the results produced by the compile time module, called `SIGN` (SIgnature GeNeration). `SIGN` calculates the changes of the signatures introduced by the concerns. The actual transformation is performed by a language dependent module, called `SITRA` (SIgnature TRAnsformation). After the transformation, the program sources are compiled ”against” the changed dummies.

7.3 Runtime

Our main reason for using an interpreter in Compose⋆/J is the fact that already an interpreter exists in Compose⋆/.NET. This interpreter is largely written in J# to simplify the process of porting it to the Java platform.

The interpreter is responsible for executing concern code at the joinpoints. It is triggered by function calls that are woven in by the weaver. It uses a reduced copy of the repository to evaluate and execute the filters.

Porting the interpreter to the Java platform only requires the implementation of a small set of classes. These classes are presented in Chapter 8.

7.4 Summary and Conclusion

The architecture of Compose⋆/J is divided in four layers: IDE, compile-time, adaptation and runtime. The compile-time layer is language independent. Thus, Compose⋆/J reuses it. The other three layers are partially language dependent. This chapter presented the design choices made in these layers. To summarize, we here briefly present a list of the main design choices:

- Compose⋆/J interfaces with the Eclipse IDE.
- Compose⋆/J uses a general approach for collecting the structure and annotations within a source program. This approach is based on ANTLR and the reflection mechanisms of OO-languages.
- Compose⋆/J uses byte code manipulation at compile time as the weaving process. The weaver inserts hooks to an interpreter. Javassist is used as the byte code manipulation tool.
- Compose⋆/J reuses the interpreter of Compose⋆/.NET.

The next chapter presents a detailed implementation of the language specific parts of Compose⋆/J.

Chapter 8

Implementation of Compose⋆/J

The previous chapter presented the design choices of the various language specific parts of Compose⋆/J. This chapter presents a detailed implementation of those language specific parts.

8.1 Eclipse plug-in

First, we present the implementation of the Compose⋆/J Eclipse plug-in. The Eclipse plug-in consists of two components: a core component and a language specific component. All Compose⋆ implementations that interface with the Eclipse IDE share the core component. The language part is language dependent. Currently, the implementations of Compose⋆ that interface with the Eclipse IDE are Compose⋆/J and Compose⋆/C.

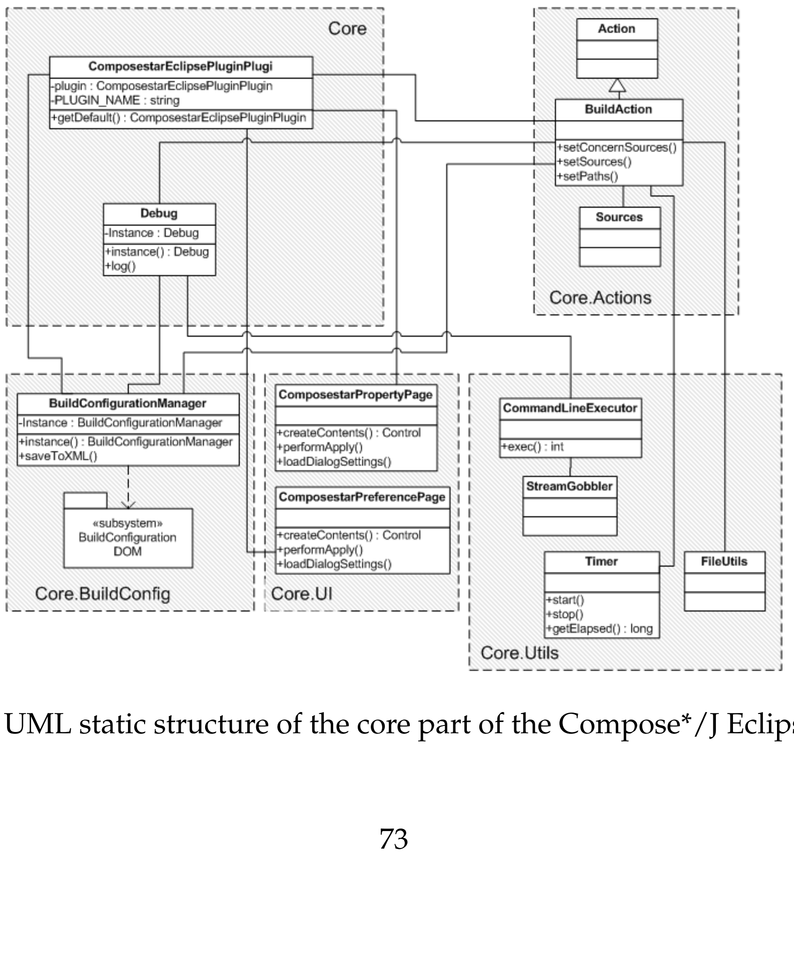


Figure 8.1: UML static structure of the core part of the Compose*/J Eclipse plug-in.

8.1.1 Core part

Figure 8.1 shows the implementation of the core part of the plug-in. The main class is `ComposestarEclipsePluginPlugin`, which represents the heart of the plug-in. The class `Debug` is used for printing out debug information on a console. Furthermore, the core part consists of four packages: *BuildConfig*, *UI*, *Actions* and *Utils*. We describe the components in each package below.

BuildConfig

The package *BuildConfig* contains logic for building a build configuration file. The class `BuildConfigurationManager` creates such a file. It uses a data object model for mapping the data to xml, shown in Figure 8.2. Appendix A shows an example of a build configuration file.

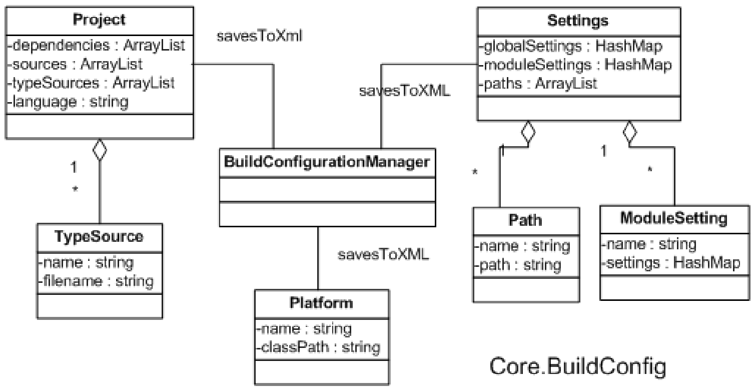


Figure 8.2: build configuration data object model.

UI

The package *UI* contains core classes that extend the graphical interface of the Eclipse Workbench. As described in the previous chapter, the plug-in uses a property page and a preference page.

Actions

This package contains core classes for dealing with actions. The class `BuildAction` represents the action "building a Compose⋆ application". The class `Sources` extracts the sources of a project from the Eclipse workspace.

Utils

This package contains utility classes. The classes `CommandLineExecutor` and `StreamGobbler` provide the ability to run an application (e.g. the Compose⋆/J compiler) from the command line and catch the output. The class `FileUtils` contains methods that deal with file handling (e.g. converting backslashes in filenames to slashes). The class `Timer` represents a stopwatch.

8.1.2 Language dependent part

Figure 8.3 shows the implementation of the language dependent part of the plug-in. As a naming convention, all package names start with the name of the language (i.e. Java). We describe the components in each package below.

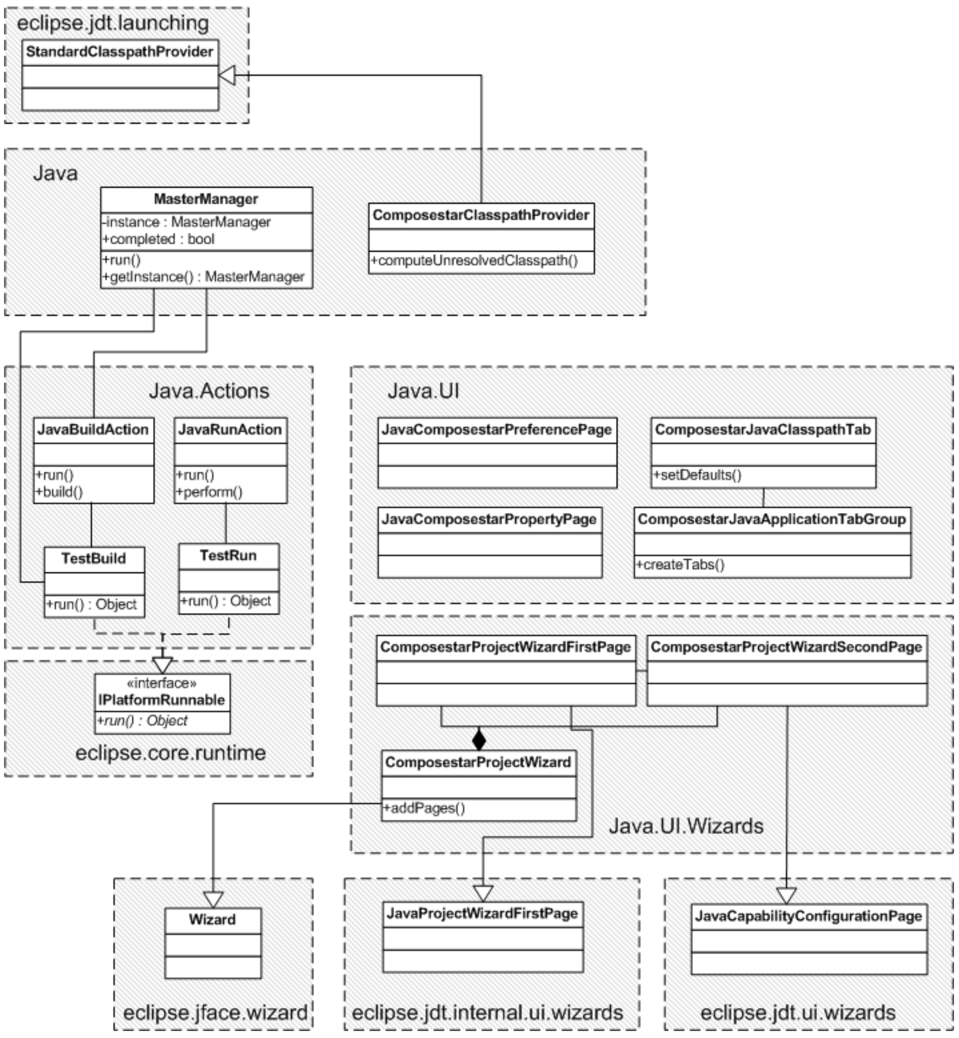

eclipse.jdt.launching
StandardClasspathProvider
Java
MasterManager
-instance : MasterManager
+completed : bool
+run()
+getInstance() : MasterManager
ComposestarClasspathProvider
+computeUnresolvedClasspath()
Java.Actions
JavaBuildAction
+run()
+build()
JavaRunAction
+run()
+perform()
TestBuild
+run() : Object
TestRun
+run() : Object
Java.UI
JavaComposestarPreferencePage
ComposestarJavaClasspathTab
+setDefaults()
JavaComposestarPropertyPage
ComposestarJavaApplicationTabGroup
+createTabs()
«interface»
IPlatformRunnable
+run() : Object
eclipse.core.runtime
ComposestarProjectWizardFirstPage
ComposestarProjectWizardSecondPage
ComposestarProjectWizard
+addPages()
Java.UI.Wizards
Wizard
eclipse.jface.wizard
JavaProjectWizardFirstPage
eclipse.jdt.internal.ui.wizards
JavaCapabilityConfigurationPage
eclipse.jdt.ui.wizards

Figure 8.3: UML static structure of the language part of the Compose*/J Eclipse plug-in.

Java

This package contains components that do not belong specifically to any subpackage.
The class `MasterManager` triggers the Compose⋆/J compiler. To accomplish this, it uses the `CommandLineExecutor` of the core part. `ComposestarClasspathProvider` provides a way to load the classpath used by the Compose⋆/J compiler in the Eclipse workbench. It extends the standard classpath provider service of the Eclipse Platform, located in the *eclipse.jdt.launching* package.

Java.Actions

This package contains classes that perform specific Compose⋆/J actions.
The class `JavaBuildAction` represents the action ”building a Compose⋆/J application”. Section 8.1.3 presents a detailed control flow of this process. `JavaRunAction` represents the action ”running a Compose⋆/J application”. Furthermore, the classes `TestBuild` and `TestRun` provide services to test both actions in headless Eclipse (i.e. without IDE).

Java.UI

This package contains classes that extend the graphical interface of the Eclipse Workbench. The classes `JavaComposestarPreferencePage` and `JavaComposestarPropertyPage` provide respectively a preference page and a property page. `ComposestarJavaApplicationTabGroup` creates a tab group containing several pages to configure the launch of a Compose⋆/J application. Many of these pages are reused from the JDT tool that is part of the Eclipse SDK for launching a Java application. The only tab page that is different is the page that configures the Compose⋆/J compiler classpath. The class `ComposestarJavaClasspathTab` implements this page.

Java.UI.Wizards

This package is a subpackage of *Java.UI*. It contains classes that extend the Eclipse Workbench with wizards for creating a Compose⋆/J project. Again, the classes use several classes from the JDT tool. Appendix B shows screenshots of the graphical interface of the plug-in.

8.1.3 Building a Compose⋆/J application

This section describes the control flow of building a Compose⋆/J application in the Eclipse plug-in. Figure 8.4 shows an UML sequence diagram of the control flow. The diagram defines the following sequence of actions:

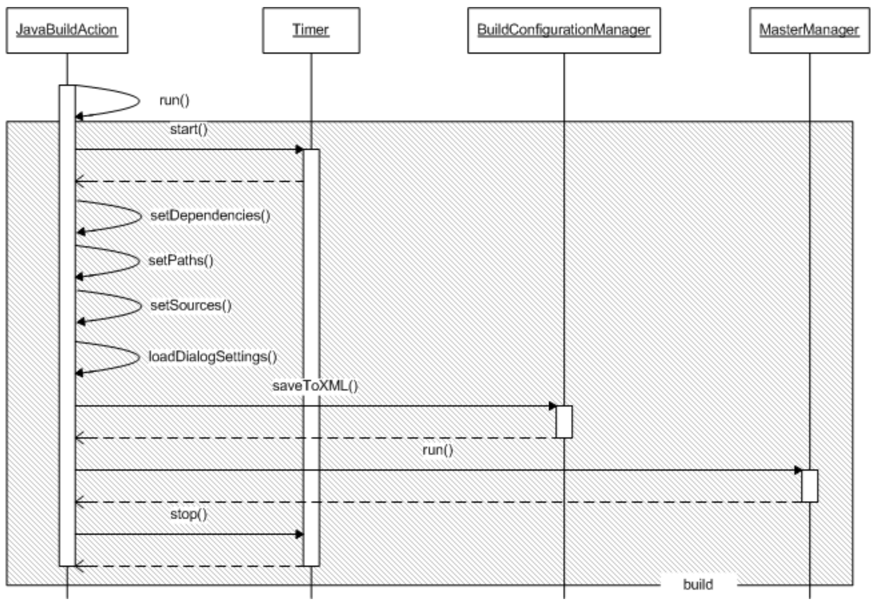


Figure 8.4: UML sequence diagram - control flow of building a Compose⋆/J application.

1. **Start the build process.** The build process is started by calling the `run()` method of an instance of the class `JavaBuildAction`. This method starts a new thread for refreshing the Eclipse UI during execution of the build process. This thread performs the build process by calling the `build()` method of `JavaBuildAction`.
2. **Start a timer.** The build process starts with running a stopwatch to time the process. This is done by calling the `start()` method of an instance of the class `Timer`.
3. **Collect necessary information to build a configuration file.** This process collects the necessary information for building a configuration file. This information is retrieved by calling the methods `setDependencies()`, `setPaths()`, `setSources()` and `loadDialogSettings()`.
4. **Build a configuration file.** The build configuration file is created by calling the `saveToXML()` method of an instance of the class `BuildConfigurationManager`.
5. **Compile the Compose*/J application.** A Compose⋆/J application is compiled by the Compose⋆/J compiler. This compiler is triggered by calling the `run()` method of an instance of the class `MasterManager`.
6. **Stop the timer.** After compilation, the stopwatch is stopped by calling the `stop()` method of the class `Timer`.

8.2 Adaptation

This section presents the implementation of the various language specific parts of the adaptation layer in Compose⋆/J.

8.2.1 Dummy generation

The process of creating dummies is performed by the module `DUMMER` (DUMmy ManagER). Figure 8.5 shows the UML structure of this module.

The class `DummyManager` is the starting point of the module. The `DummyManager` uses an instance of the class `DummyEmitter` to create the dummy files. Each Compose⋆ implementation uses a specific language emitter. In this case, Compose⋆/J uses an instance of `JavaDummyEmitter` to create dummies from sources written in the Java language.

Furthermore, the classes `JavaRecognizer` and `JavaLexer` represent respectively a parser and a lexer for the Java language. ANTLR creates these classes from a specified grammar file.

Finally, the class `JavaCompiler` defines logic for compiling dummies and program sources with a native Java compiler.

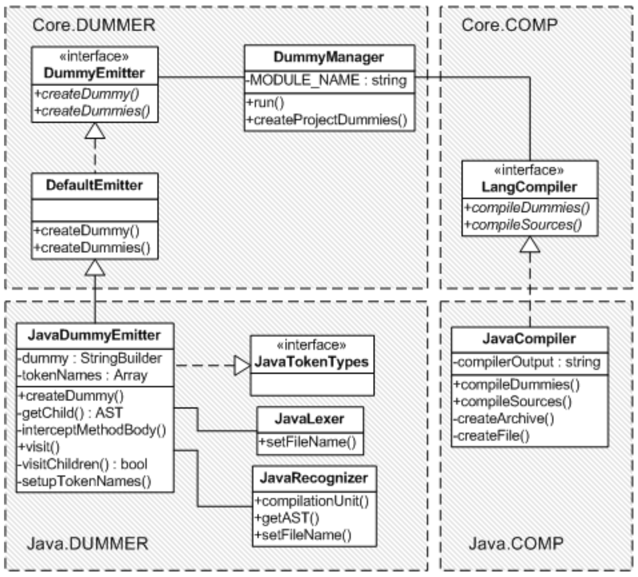


Figure 8.5: UML static structure of DUMMER.

8.2.1.1 Control flow DUMMER

Figure 8.6 shows an UML sequence diagram of the control flow of the module `DUMMER`. The diagram defines the following sequence of actions:

1. **Start the module.** The module `DUMMER` is triggered by the class `JavaMaster` by calling the method `run()` of an instance of the class `DummyManager`. `JavaMaster` controls the execution of all modules.

2. **Create dummies.** The process of creating dummies is triggered by calling the method `createDummies()` of an instance of the class `JavaDummyEmitter`.
   `JavaDummyEmitter` creates a dummy source of every program source in a Compose⋆/J project. This process is performed in a sequence of actions: (a) the method `compilationUnit()` of an instance of the class `JavaRecognizer` creates an Abstract Syntax Tree (AST) of the program source, (b) the method `visit()` of `JavaDummyEmitter` is recursively called to construct a dummy from the AST and (c) the method `emit()` emits the dummy source to a file.

3. **Compile dummies.** After the dummies are created, the dummies are compiled by calling the method `compileDummies()` of an instance of the class `JavaCompiler`.

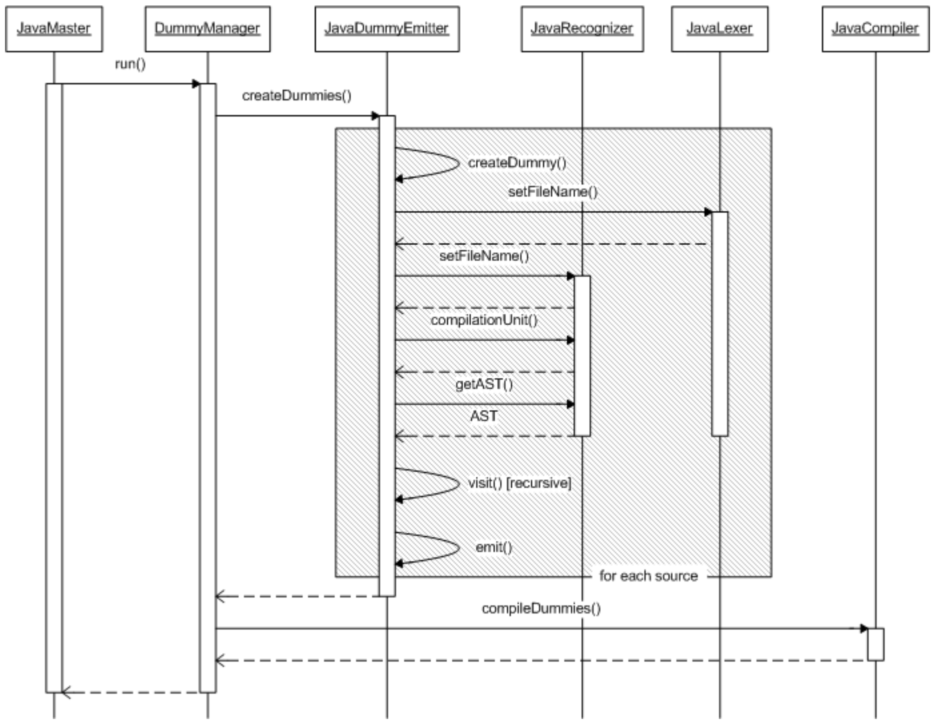


Figure 8.6: UML sequence diagram - control flow of DUMMER.

8.2.2 Collecting type information

The process of collecting type information in Compose⋆/J is performed by three modules:

1. **HARVESTER** This module extracts classes from the compiled dummies.
2. **COLLECTOR** This module collects the type information (other than annotations) from the extracted classes and stores them in the repository.
3. **AnnotationCollector** This module collects the annotations from the extracted classes and stores them in the repository.

Figure 8.7 shows the UML structure of these three modules. All three modules are located in the package TYM (TYpe Mystification). The starting point of the module `HARVESTER` is the class `JavaHarvestRunner`. This class uses an instance of the class `JarLoader` to load classes from a jar file (i.e. the dummies are stored in a jar file). Exceptions that occur during this process are represented as objects of the type `JarLoaderException`. In order to locate the jar file, `JavaHarvestRunner` uses an instance of the helper class `ClassPathModifier`. This class can change the classpath at runtime.

Furthermore, `HARVESTER` stores the classes in an instance of the singleton class `ClassMap`. The classes `JavaCollectorRunner` and `AnnotationCollector` use the information stored in this singleton class to map it to a representation in the repository.

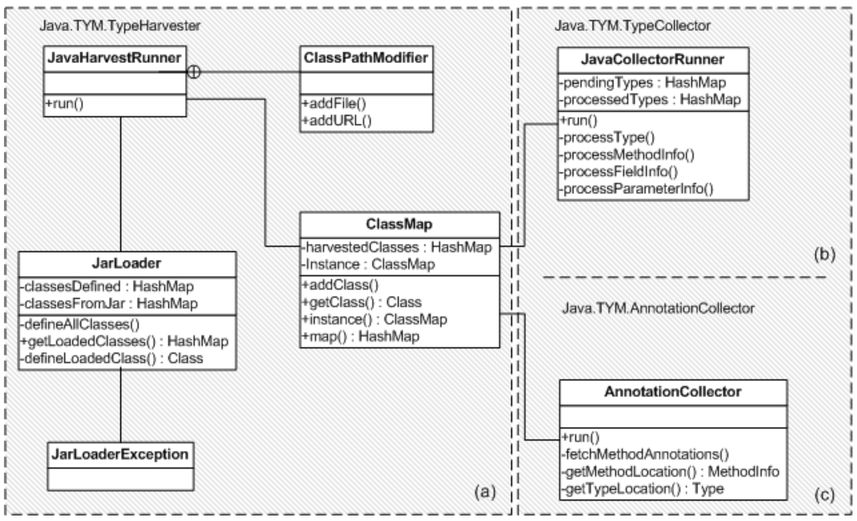


Figure 8.7: UML static structure of collecting type information - (a) HARVESTER (b) COLLECTOR (c) AnnotationCollector.

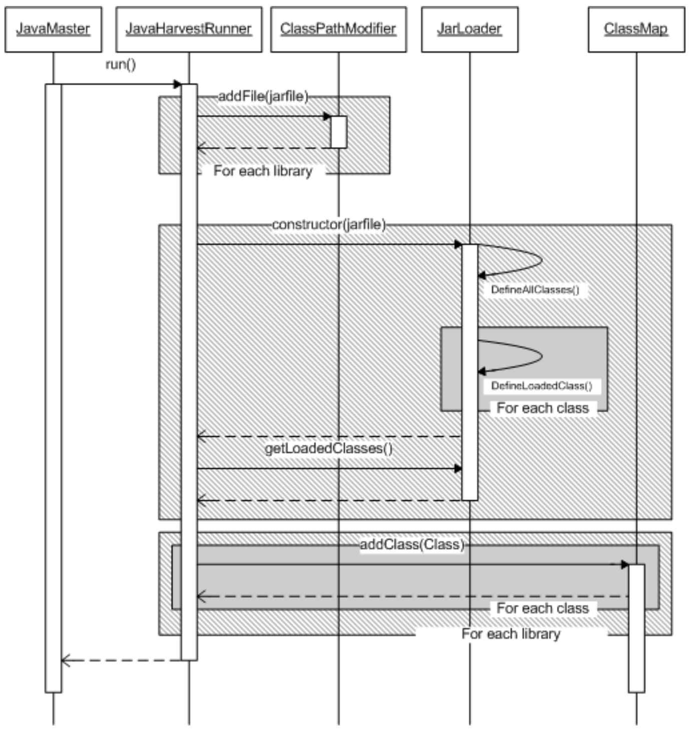


Figure 8.8: UML sequence diagram - control flow of HARVESTER.

8.2.2.1 Control flow harvesting

Figure 8.8 shows an UML sequence diagram of the control flow of the process of harvesting the type information (i.e. the module `HARVESTER`). The diagram defines the following sequence of actions:

1. **Start the module `HARVESTER`.** The module `HARVESTER` is triggered by the class `JavaMaster` by calling the method `run()` of an instance of the class `JavaHarvestRunner`.
2. **Add libraries to classpath.** In order to extract the classes from the dummy library and dependent libraries of a Compose⋆/J project, the classpath must contain the paths to the libraries. The paths are added to the classpath by calling the method `addFile()` of the static class `ClassPathModifier` for each library.
3. **Extract the classes from libraries.** The next step is to extract all classes from the libraries. This action is performed by calling the constructor of an instance of the class `JarLoader`, which has a jarfile as argument. The classes are retrieved by calling the method `getLoadedClasses()` of the `JarLoader`.
4. **Store the classes in memory.** The module `HARVESTER` ends with storing all the extracted classes in memory. This is done by calling the method `addClass()` of the singleton class `ClassMap` for each extracted class.

8.2.2.2 Control flow collecting

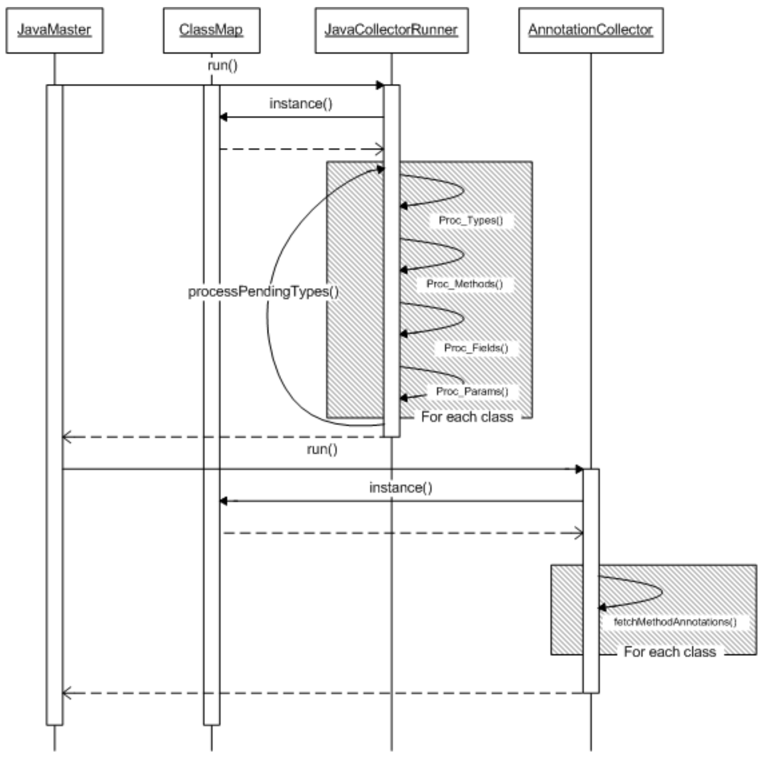


Figure 8.9: UML sequence diagram - control flow of `COLLECTOR` and `AnnotationCollector`.

Figure 8.9 shows an UML sequence diagram of the control flow of the process of collecting the type information (i.e. the modules `COLLECTOR` and `AnnotationCollector`). The diagram defines the following sequence of actions:

1. **Start the module `COLLECTOR`.** The module `COLLECTOR` is triggered by the class `JavaMaster` by calling the method `run()` of an instance of `JavaCollectorRunner`.

2. **Retrieve classes.** The module `COLLECTOR` starts with retrieving the classes from memory. This is done by calling the method `instance()` of the class `ClassMap`. `instance()` returns an instance of `ClassMap`, which contains the classes extracted by the module `HARVESTER`.

3. **Extract the type information from the classes.** This action extracts all type information (except annotations) from the classes. The type information consists of classes, methods, fields and parameters found in the classes. These objects are mapped to a uniform language model. Section 8.2.2.3 presents this language model.

4. **Start the module `AnnotationCollector`.** The next phase is to extract the annotations. This is done by the module `AnnotationCollector`. The module is triggered by the class `JavaMaster` by calling the method `run()` of an instance of the class `AnnotationCollector`.

5. **Retrieve classes.** The first step of the module `AnnotationCollector` is the same as the module `COLLECTOR`. It reads the classes from memory by retrieving an instance of the class `ClassMap`.

6. **Fetch the annotations from the classes.** This action extracts the annotations from the methods of the classes. This is done by calling the method `fetchMethodAnnotations()` of the class `AnnotationCollector`.

8.2.2.3 Language model

Figure 8.10 shows an abstraction of the Java language model.

The Java language model is specific for the Java language, but the various tools in the compile time layer (e.g. SIGN, FIRE and LOLA) are platform and language independent. In order to reason about concerns in a uniform way in each Compose⋆ implementation, these tools need a platform and language independent representation of the static structure of a Compose⋆ application. This representation is shown in Figure 8.11.

As we can see, the platform and language independent model (LAMA) closely resembles to the model shown in Figure 8.10. It contains the most important components that exist in a OO-language: types, methods, fields, parameters, annotations and namespaces. Each class other than the class `Annotation` extends from the abstract class `ProgramElement`. Every `ProgramElement` can contain zero or more annotations. Furthermore, every platform specific program element extends from a platform independent counterpart. (e.g. `JavaMethodInfo` represents a method in Java and `DotNETMethodInfo` represents a method in .NET, they both extend from the class `MethodInfo`). This mapping allows us to access the program elements in a uniform way in the platform and independent compile-time layer.

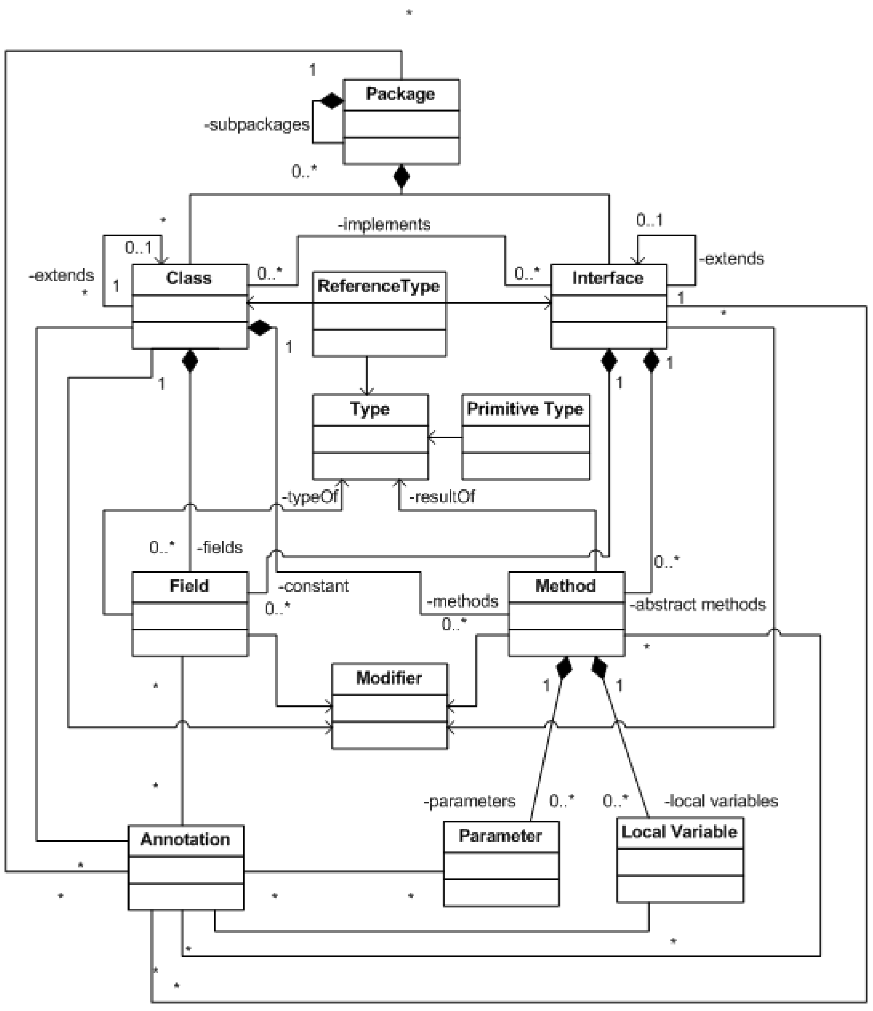


Figure 8.10: abstraction of the Java language model.

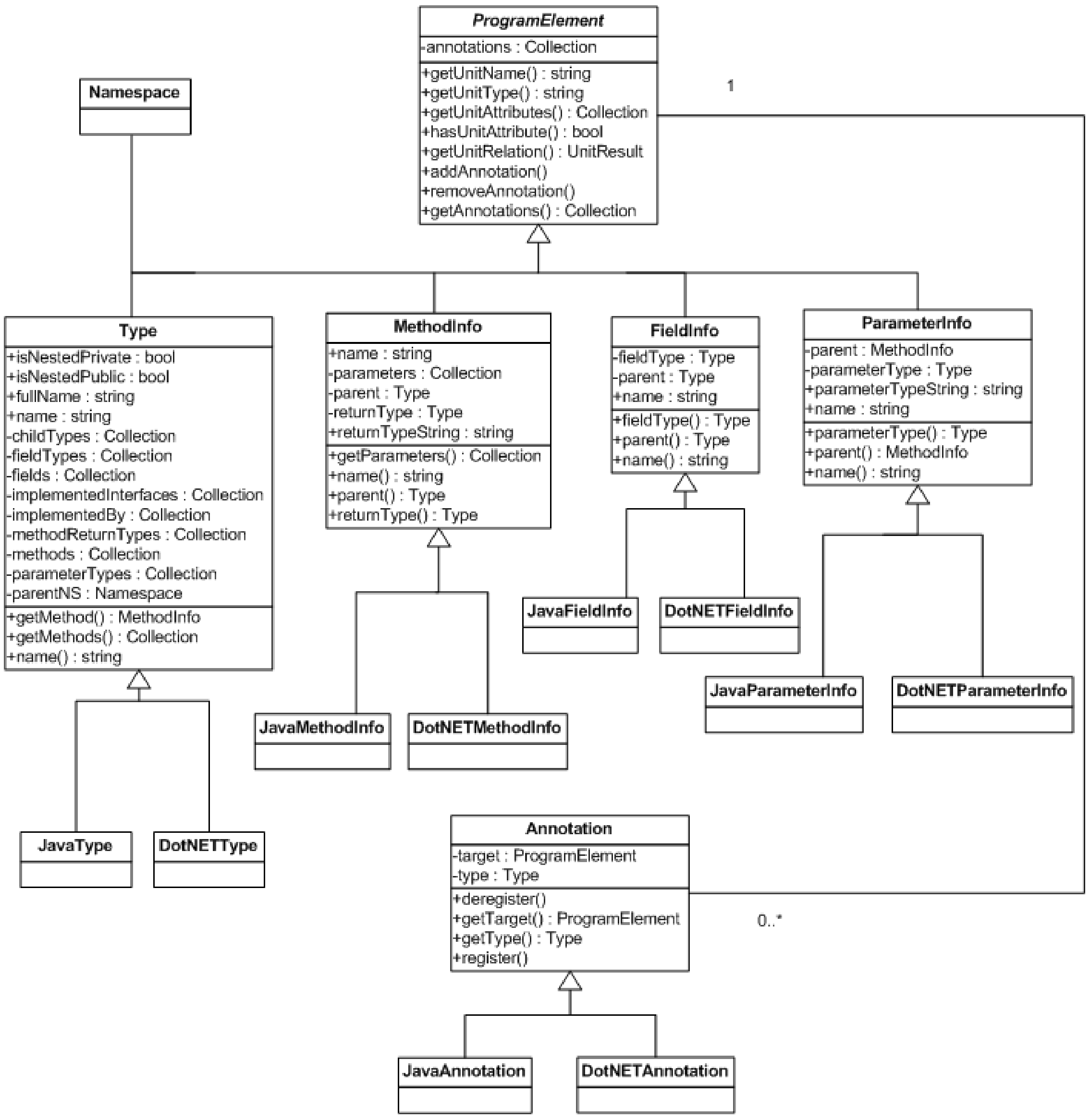


Figure 8.11: UML static structure of LAMA.

8.2.3 Signature Transformation

The process of signature transformation of the dummy sources is performed by the module `SITRA` (SIgnature TRAnsformation). Figure 8.12 shows the UML structure of this module.

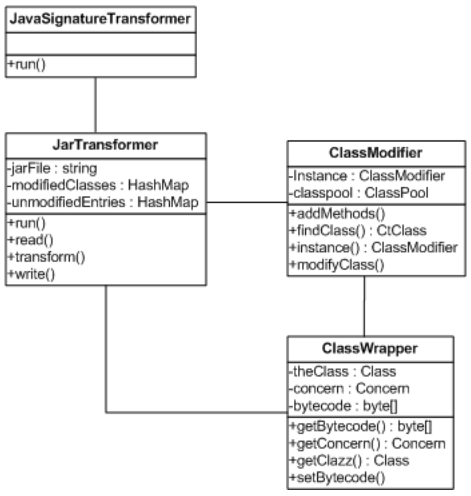


Figure 8.12: UML static structure of SITRA.

The class `JavaSignatureTransformer` is the starting point of the module. It uses an instance of the class `JarTransformer` to transform the compiled dummies inside a jarfile. `JarTransformer` uses an instance of the class `ClassModifier` to perform the transformations on a class. This is done by using Javassist as the bytecode manipulation tool. The class `ClassWrapper` acts as a wrapper for manipulated classes. It contains the original `Class` object, the transformed bytecode of the class and a `Concern` object that holds information about what needs to be transformed.

8.2.3.1 Control flow SITRA

Figure 8.13 shows an UML sequence diagram of the control flow of the module `SITRA`. The diagram defines the following sequence of actions:

1. **Start the module.** The module `SITRA` is triggered by the class `JavaMaster` by calling the method `run()` of an instance of the class `JavaSignatureTransformer`.

2. **Read classes from jar.** `JavaSignatureTransformer` calls the method `run()` of an instance of the class `JarLoader`. `Jarloader` initiate the transformation process by reading all classes from the specified jarfile. This is done by the method `read()`.

3. **Transforming the classes.** The actual transformation is performed by calling the method `modifyClass()` of an instance of the class `ClassModifier` for each class. The method `addMethods()` of `ClassModifier` adds the signatures of the new methods, introduced by the concerns, to a class. The new bytecode is set by calling the `setBytecode()` method of the `ClassWrapper`.

4. **Write classes back to jar.** The module `SITRA` ends with writing the changes back to the original jarfile. This is done by calling the method `write()`.

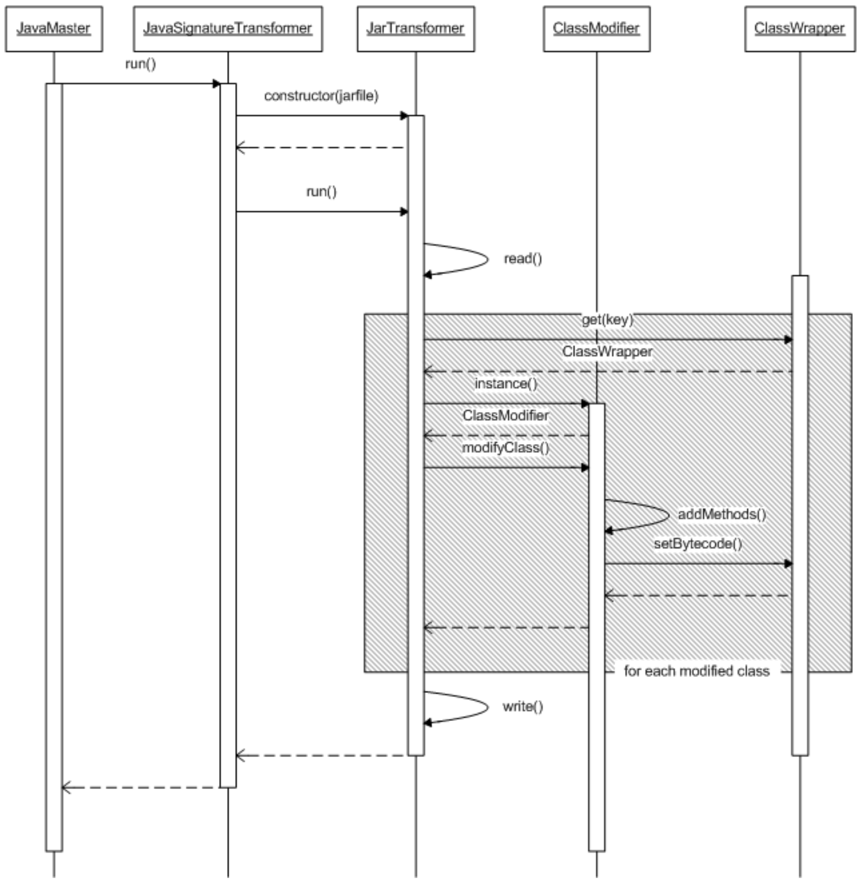


Figure 8.13: UML sequence diagram - control flow of SITRA.

8.2.4 Weaver

As described in the previous chapter, we use Javassist as our weaving tool. In this section, we first describe how Javassist deals with editing expressions. After that, we present the implementation of the Compose*/J weaver.

8.2.4.1 Editing expressions in Javassist

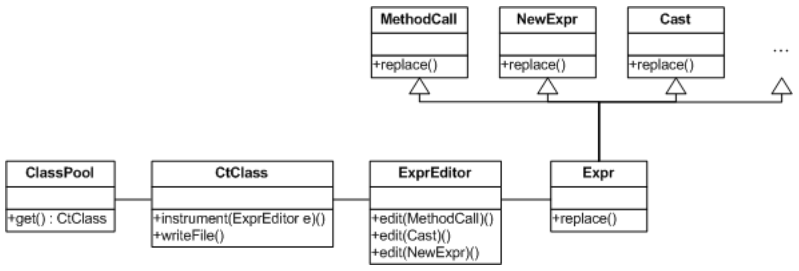


Figure 8.14: UML static structure - editing expressions in javassist.

Figure 8.14 shows the classes in Javassist that play a role in editing expressions. We describe the classes below:

CtClass The `CtClass` object is an object that represents a class obtained from a given class file. It provides almost the same functionality as the `java.lang.Class` class of the standard java reflection API. Similar to `CtClass`, Javassist also has classes that represent methods, constructors, fields, etc (i.e. `CtMethod`, `CtConstructor` and `CtField`). Unlike the standard reflection API, Javassist allows programmers to change the definition of a class through the `CtClass` object. One method that can be used is `instrument(ExprEditor)`. This method scans all the method bodies of the methods declared in the class, and edits its expressions found in the methods through an instance of the class `ExprEditor`. The method `writeFile()` of `CtClass` writes the changes back to a class file.

ClassPool The `ClassPool` object is an container of `CtClass` objects. A `CtClass` object is obtained through the `get()` method. This method finds a class file through the classpath and creates a `CtClass` object from it.

ExprEditor As described above, the `ExprEditor` object represents a translator for method bodies. The `edit()` method inspects and modifies an given expression.

Expr The `Expr` object is an object that represents an expression. Each type of expression extends from this class, e.g. `MethodCall`, `Cast` and `NewExpr` (object creation). The method `replace()` replaces the expression with the bytecode derived from the given source text.

Now, as described above, Javassist edits expressions by calling the `replace()` method of expression objects. This method takes source text as an argument. In the source text, we can use special meta variables, listed in Table 8.1, to perform reify and reflect operations on demand. A complete explanation of these meta variables is found in [8].

$0, $1, $2, ...	parameter values
$_	result value
$$	a comma-separated sequence of the parameters
$args	an array of the parameter values
$r	formal type of the result value
$w	the wrapper type
$proceed(..)	execute the original computation
$class	a java.lang.Class object representing the target class
$sig	an array of `java.lang.Class` representing the formal parameter types
$type	a java.lang.Class object representing the formal result type
$cflow(..)	a mechanism similar to cflow of AspectJ

Table 8.1: Meta variables

Listing 8.1 shows an example of the usage of these meta variables. The code shows a translator that edits expressions of the type `MethodCall`. More specific, it replaces method calls made to methods with the name `methodX`. If such a method call is found, the expression is replaced with two expressions. The first expression prints out the name of the method. The second expression ”`$_ = $proceed($$)`” executes the original expression. Note that this example is an example of an implementation of a before logging concern.

```
new ExprEditor(){
    public void edit(MethodCall mc){
        if (mc.getMethodName().equals("methodX")){
            mc.replace("{ System.out.println("methodX");"
            + "$_ = $proceed($$); }");
        }
    }
}
```

Listing 8.1: Example usage of meta variables.

8.2.4.2 Static structure of the weaver

Figure 8.15 shows the implementation of the Compose⋆/J weaver. In the figure, above on the right, we see the part of Javassist that we explained above. The core part of the weaver consists of only two interfaces. They tell us that the Compose⋆/J weaver is a `WEAVER` and a Compose⋆ module. The most interesting part is the language dependent part. This part connects with Javassist and it contains the following classes:

JavaWeaver The `JavaWeaver` object is the entry point of the weaver.

HookDictionary The `HookDictionary` object is a storage place for hooks. A hook is a location in the program where a call to the interpreter should be inserted by the weaver. This information is also available in the repository, but computing and accessing it directly takes more time. Thus, we use a mapping for it. The `MethodBodyTransformer` object uses the `HookDictionary` to check whether or not to place hooks in particular expressions.

ClassWeaver The `ClassWeaver` object is an object that performs the weaving on the classes of a given project. It contains a `ClassPool` object that stores the `CtClass` objects that need to be transformed.

MethodBodyTransformer The `MethodBodyTransformer` object is a translator for method bodies. This object replaces expressions with hooks to the interpreter. It extends the class `ExprEditor` that is part of Javassist.

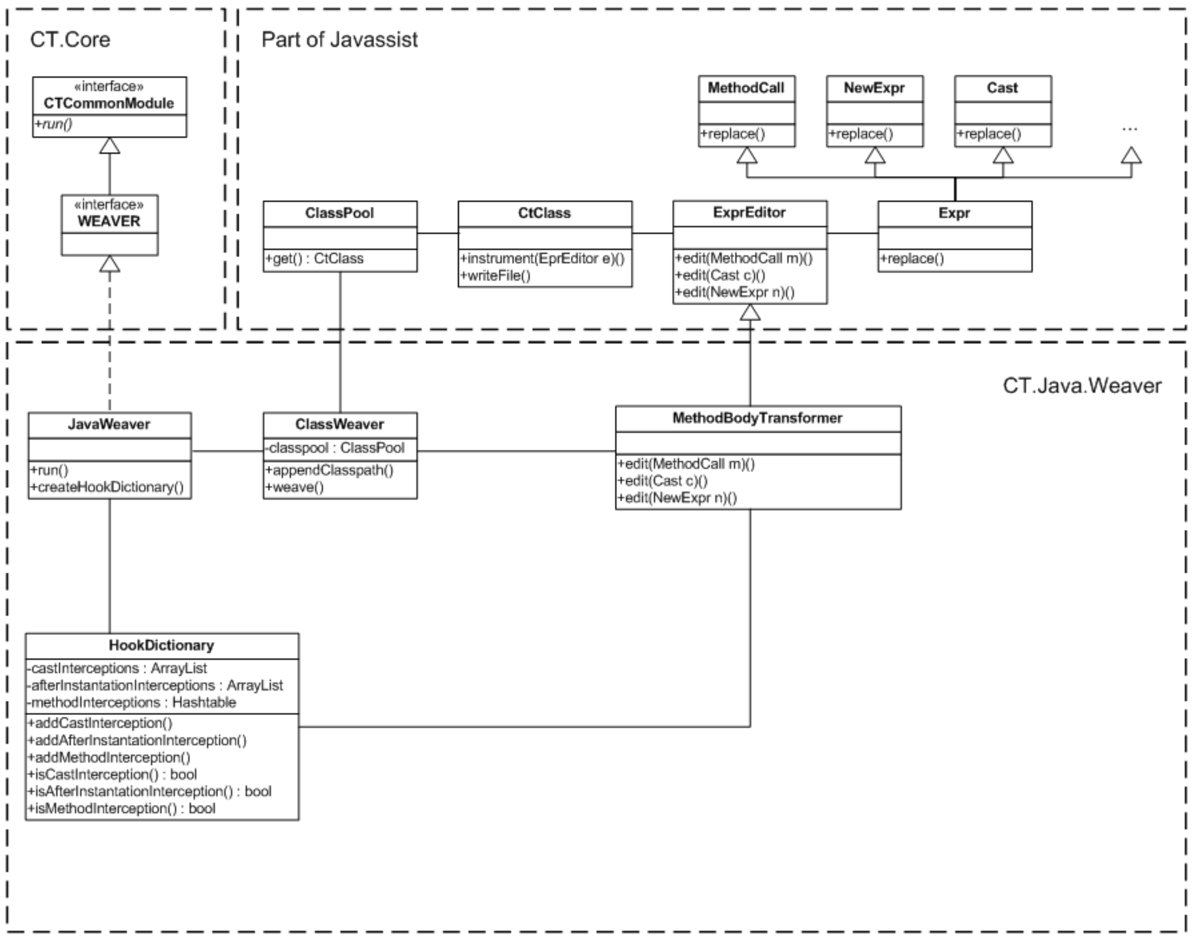


Figure 8.15: UML static structure of the Compose*/J weaver.

8.2.4.3 Control flow of the weaver

Figure 8.16 shows an UML sequence diagram of the control flow of the weaver. The diagram defines the following sequence of actions:

1. **Start the weaver.** The weaver is triggered by the class `JavaMaster` by calling the method `run()` of an instance of the class `JavaWeaver`.

2. **Create the hook dictionary.** The `HookDictionary` is created by calling the method `createHookDictionary()` of the `JavaWeaver` object. This method inserts three types of hooks to the dictionary: a) method calls b) object creations and c) castings.

3. **Add libraries to classpath.** This action adds the libraries of a Compose*/J project to the classpath, in order to find the classes that need to be transformed. This is accomplished by calling the `addClassPath()` method of the `ClassWeaver` object for each library.

4. **Transform the classes.** This action is triggered by calling the `weave()` method of the `ClassWeaver` object. The `weave()` method starts with adding the application start info to the main class of the given project. This is done by calling the `writeApplicationStart()` method. The application start info contains logic for initializing the interpreter. Furthermore, the `weave()` method instruments each class in the project with a `MethodBodyTransformer` object, which performs the transformation of interesting expressions. After the transformations are performed, the classes are written back to disk by calling the `writeFile()` method of the `CtClass` objects.

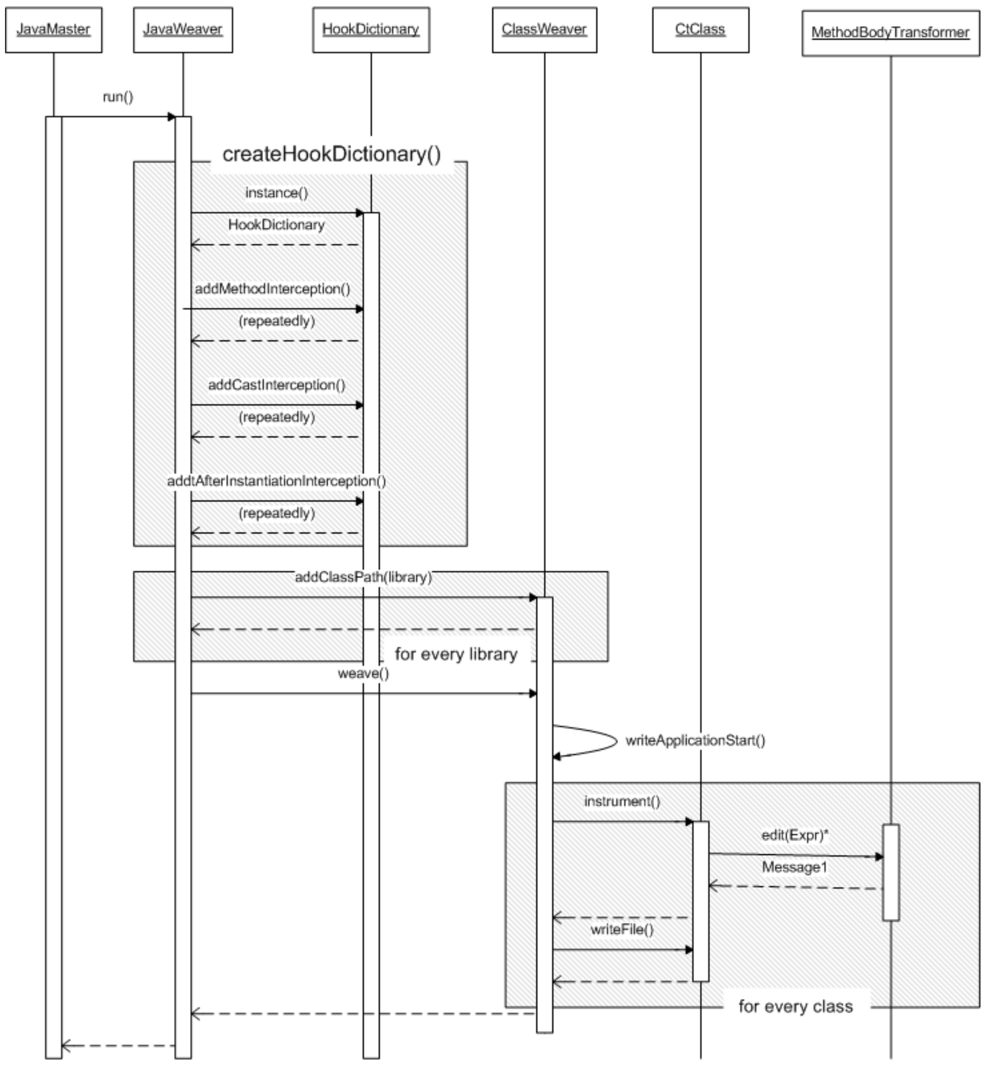


Figure 8.16: UML sequence diagram - control flow of the Compose⋆/J weaver.

8.3 Runtime

As described in Section 7.3, Compose⋆/J uses the interpreter of Compose⋆/.NET. A detailed description of the interpreter is found in [16]. Porting the interpreter to the Java platform only requires the implementation of a small set of classes. These classes are shown in Figure 8.17. We describe the classes below:

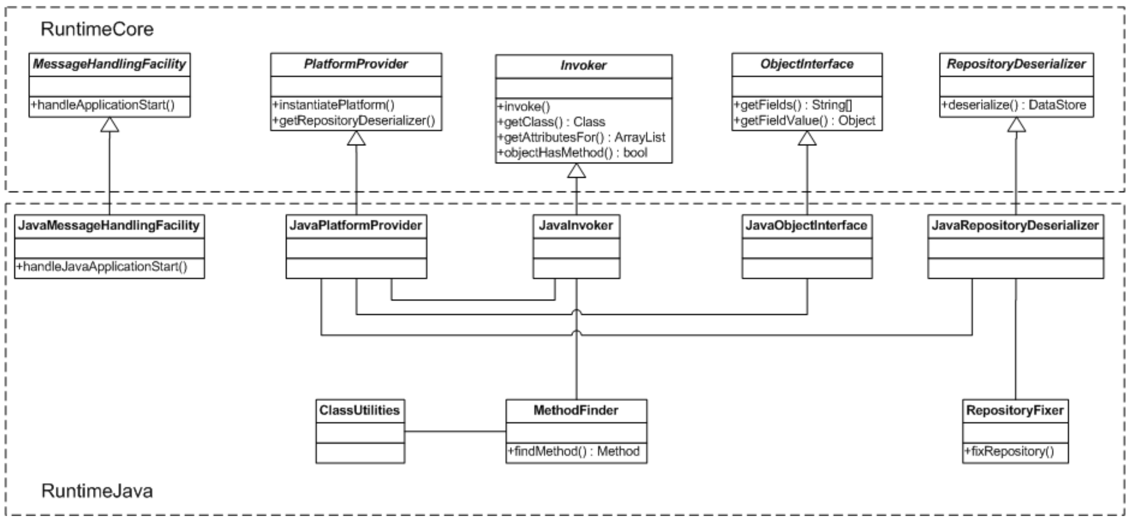


Figure 8.17: UML static structure of Compose*/J interpreter.

JavaMessageHandlingFacility The class `JavaMessageHandlingFacility` is the entry point of the interpreter. Calls to this class are woven in by the weaver in a Compose*/J application. `JavaMessageHandlingFacility` inherits the functionality from the core class `MessageHandlingFacility`.

JavaPlatformProvider The class `JavaPlatformProvider` initializes the interpreter and provides a repository deserializer for the Java Platform. It inherits the functionality from the core class `PlatformProvider`.

JavaInvoker The class `JavaInvoker` provides the ability to invoke intercepted messages using the Java reflection API. It inherits the functionality from the core class `Invoker`.

JavaObjectInterface The class `JavaObjectInterface` provides access to the interface of an object using the Java reflection API. It inherits the functionality from the core class `ObjectInterface`.

JavaRepositoryDeserializer The class `JavaRepositoryDeserializer` provides the ability to deserialize the repository created at compile time. It inherits the functionality from the core class `RepositoryDeserializer`.

RepositoryFixer The class `RepositoryFixer` fixes the repository after it has been deserialized.

MethodFinder The class `MethodFinder` is a utility class for finding methods. It uses the standard Java reflection API.

ClassUtilities The class `ClassUtilities` is a utility class for classes. It uses the standard Java reflection API.

Chapter 9

Conclusion, Future Work and Related Work

In this thesis, we presented the design and implementation of Compose⋆/J, the Compose⋆ implementation for the Java Platform. Furthermore, we investigated the possibilities of supporting specific Java features in Compose⋆/J.

In this final chapter, we conclude this thesis. First, we summarize the main conclusions drawn in this thesis. Then we describe some possible future work on Compose⋆/J and at the end of this chapter we present some related work.

9.1 Conclusions

This section is divided in two parts. In the first part, we summarize the main design choices of Compose⋆/J. In the second part, we summarize the conclusions drawn from the investigation of supporting specific Java features in Compose⋆/J.

Main design choices of Compose⋆/J

- Compose⋆/J interfaces with the Eclipse IDE. The main reason for this is that Eclipse is an extensible platform for building IDE's. Besides the built-in Java IDE, there are language IDE's for most of the popular programming languages, such as C/C++ Development Tooling (CDT). This saves us some effort in writing plug-ins for other Compose⋆ implementations for other base languages.

- Compose⋆/J uses a general approach for collecting the program structure and annotations within a source program. This approach is based on source code parsing and the reflection mechanisms of OO-languages. The program structure is mapped to an uniform language model for OO-languages. This mapping is needed to create and reuse platform independent analysis tools that reason about the composition filters model.

- Compose⋆/J uses byte code manipulation at compile time as the weaving process. The weaver inserts hooks to an interpreter. Javassist is used as the byte code manipulation tool. Our main reason for using Javassist is the fact that it provides a source-level API, unlike other byte code manipulation tools. This means, that we can edit a class file without knowledge of the specifications of the Java bytecode, which speeded up the implementation of the Compose⋆/J weaver.

- Compose⋆/J reuses the interpreter of Compose⋆/.NET. The interpreter of Compose⋆/.NET was developed with the intention of porting it to the Java Platform. The biggest part was written in J#. Porting the interpreter to the Java Platform only required the implementation of a couple of classes.

Supporting specific Java features

- In the world of AOP, exception handling is often mentioned as an example of a crosscutting concern, so we investigated the possibility of modularizing exception handling with composition filters. We presented three composition filters models, each supporting exception handling in a different way, and we compared them based on a qualitative study. We concluded that a model based on returnfilters is the best choice, because it exposes the most intuitive semantics, it offers real-time condition evaluation and it provides the highest amount of possible filter orderings.
- We also investigated the possibility of expressing crosscutting concerns on inner classes. We argued that crosscutting concerns that apply to top-level classes, may also apply to inner classes. Furthermore, we described that the current selector language of Compose⋆/J does not possess the tools to select inner classes properly, so we proposed new predicates to the selector language.
- Finally, we discussed the possibility and benefits of weaving on Java interfaces. We concluded that supporting only class-based weaving, makes working with Java interfaces cumbersome. Thus, we proposed a way to modularize interface-based weaving in Compose⋆/J.

9.2 Future Work

In this section we describe some future work on Compose⋆/J.

Extend functionality of plug-in

The current version of the Compose⋆/J Eclipse plug-in provides a wizard to create a Compose⋆/J application, windows for changing the Compose⋆/J compiler settings and various ways to build and run a Compose⋆/J application. In the future, it can be extended with new functionality, e.g. debug capabilities, visualization tools and a help section.

Change weaving strategy

As described above, Compose⋆/J uses Javassist as the byte code manipulation tool. From a performance point of view, ASM [2] is a better choice. If performance becomes an issue, then switching to ASM is recommended.

Furthermore, we concluded in this thesis that the current interpreter-based weaving strategy exposes a drawback when dealing with call-backs. The inlining weaving strategy solve these problems partially.

Further investigation on specific Java features

We mainly focused our investigation of supporting specific Java features in Compose⋆/J on a language level. Further research should be performed to implement the various changes introduced to the Compose⋆/J language.

Speed up Compose⋆/J compiler

Currently, the Compose⋆/J compiler is non-incremental. However, [34] describes a way to speed up the compiler.

9.3 Related Work

To conclude this chapter, we here briefly present some work related to Compose⋆/J.

Compose⋆/.NET and Compose⋆/C

The composition filters concept of Compose⋆ can be applied to any programming language, given that certain assumptions are met. Beside Compose⋆/J, Compose* currently supports two other platforms: .NET 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⋆/C contains support for the C programming language. The implementation is different from the .NET counterpart, 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. Similar to Compose⋆/J, Compose⋆/C provides a plug-in for Eclipse.

ComposeJ and ConcernJ

ComposeJ and ConcernJ are two previous implementations of the composition filters model on the Java Platform.

ComposeJ [43] is the oldest of the two. It is constructed as a preprocessor to the Java compiler. ComposeJ bases its implementation of the composition filters model on successive source code transformations directed by the composition filter specification.

ConcernJ [33] acts as a preprocessor for ComposeJ. It introduced the notion of superimposition to the composition filters model. This allows for reuse of the filter modules and facilitation of crosscutting concerns.

Bibliography

[1] Antlr. grammar list. URL http://www.antlr.org/grammar/list.

[2] ASM. Java byte code manipulation framework. URL http://asm.objectweb.org/.

[3] BCEL. Byte code engineering library. URL http://jakarta.apache.org/bcel/.

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

[5] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, Oct. 2001.

[6] S. R. Boschman. Performing transformations on .NET intermediate language code. Master's thesis, University of Twente, The Netherlands, Aug. 2006.

[7] R. Bosman. Automated reasoning about Composition Filters. Master's thesis, University of Twente, The Netherlands, Nov. 2004.

[8] S. Chiba and M. Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. 2003. URL http://www.csg.is.titech.ac.jp/paper/chiba-gpce03.pdf.

[9] O. Conradi. Fine-grained join point model in Compose*. Master's thesis, University of Twente, The Netherlands, Aug. 2006.

[10] A. J. de Roo. Towards more robust advice: Message flow analysis for composition filters and its application. Master's thesis, University of Twente, The Netherlands, Mar. 2007.

[11] D. Doornenbal. Analysis and redesign of the Compose* language. Master's thesis, University of Twente, The Netherlands, Oct. 2006.

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

[13] Eclipse. An open development platform. URL http://www.eclipse.org.

[14] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Comm. ACM*, 44(10): 29–32, Oct. 2001.

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

[16] C. F. N. García. Compose* - a runtime for the .NET platform. Master's thesis, Vrije Universiteit Brussel, Belgium, Aug. 2003.

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

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

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

[20] F. J. B. Holljen. Compilation and type-safety in the Compose* .NET environment. Master's thesis, University of Twente, The Netherlands, May 2004.

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

[22] S. H. G. Huttenhuis. Patterns within aspect orientation. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[23] Javassist. Java programming assistant. URL http://www.csg.is.titech.ac.jp/~chiba/javassist/.

[24] JRockit. Bea. URL http://dev2dev.bea.com/jrockit/.

[25] 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.

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

[27] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd ICSE2000*, page 7. Xerox Corporation, 1999.

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

[29] Netbeans. URL http://www.netbeans.org.

[30] 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. ISBN 0-7923-7576-9.

[31] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In G. Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 141–147. ACM Press, Apr. 2002.

[32] A. Popovici, G. Alonso, and T. Gross. Just in time aspects. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 100–109. ACM Press, Mar. 2003.

[33] P. Salinas. Adding systemic crosscutting and super-imposition to Composition Filters. Master's thesis, Vrije Universiteit Brussel, Aug. 2001.

[34] D. R. Spenkelink. Incremental compilation in Compose*. Master's thesis, University of Twente, The Netherlands, Oct. 2006.

[35] T. Staijen. Towards safe advice: Semantic analysis of advice types in Compose*. Master's thesis, University of Twente, Apr. 2005.

[36] Sun. Sun microsystems inc. URL `http://www.sun.com`.

[37] P. Tarr, H. Ossher, S. M. Sutton, Jr., and W. Harrison. N degrees of separation: Multi-dimensional separation of concerns. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 37–61. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.

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

[39] TIOBE. Tiobe programming community index. URL `http://www.tiobe.com/tpci.htm`.

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

[41] C. Vinkes. Superimposition in the Composition Filters model. Master's thesis, University of Twente, The Netherlands, Oct. 2004.

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

[43] J. C. Wichman. The development of a preprocessor to facilitate composition filters in the Java language. Master's thesis, University of Twente, 1999. URL `http://trese.cs.utwente.nl/oldhtml/publications/msc_theses/wichman.thesis.pdf`.

Appendix A

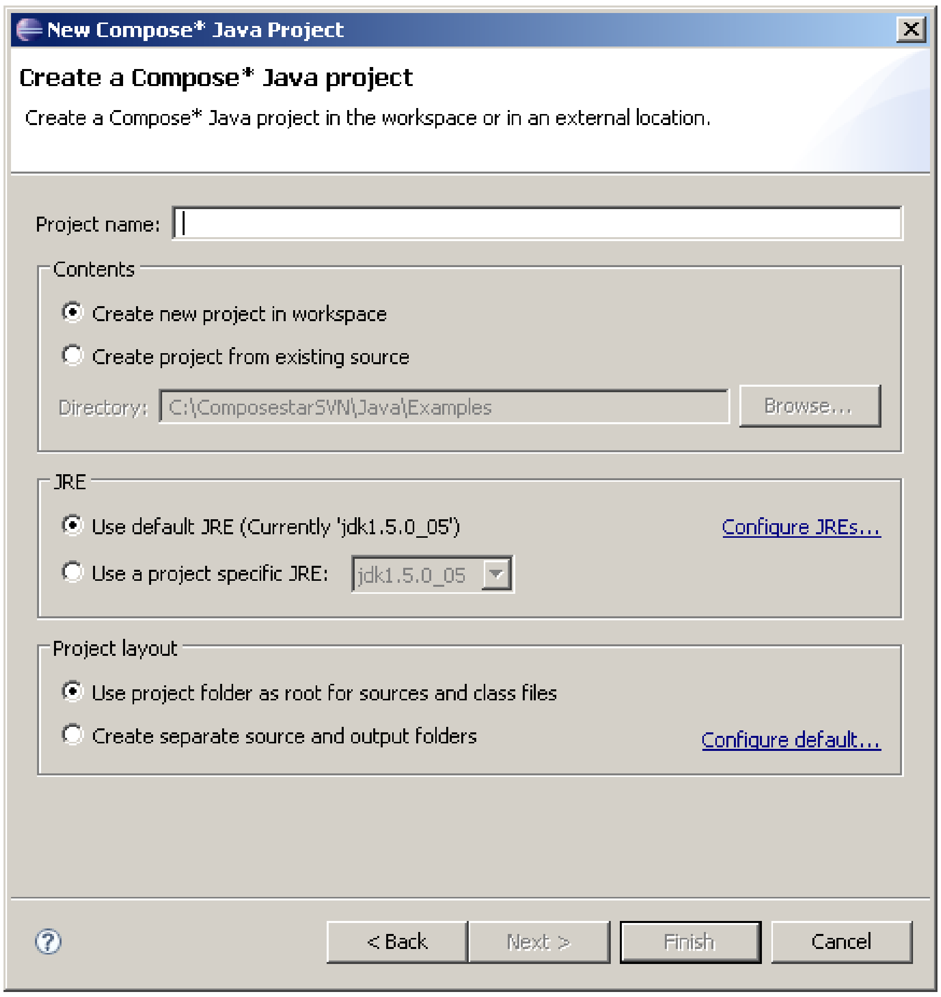
Example BuildConfiguration-file

```xml
<?xml version="1.0" encoding="us-ascii"?>
<!--This BuildConfiguration file is automatically generated by the Composestar
    Eclipse Plugin.-->
<BuildConfiguration version="1.00">
  <Projects buildDebugLevel="4" applicationStart="PlatypusExample.Main"
      runDebugLevel="1" outputPath="C:/ComposestarSVN/Java/Examples/Platypus/bin/">
    <Project name="Platypus" language="Java" basePath="C:/ComposestarSVN/Java/
        Examples/Platypus/" >
      <Sources>
        <Source fileName="C:/Platypus/PlatypusExample/Bird.java" />
        <Source fileName="C:/Platypus/PlatypusExample/Animal.java" />
        <Source fileName="C:/Platypus/PlatypusExample/Main.java" />
        <Source fileName="C:/Platypus/PlatypusExample/Food.java" />
        <Source fileName="C:/Platypus/PlatypusExample/Egg.java" />
        <Source fileName="C:/Platypus/PlatypusExample/Caretaker.java" />
        <Source fileName="C:/Platypus/PlatypusExample/Mammal.java" />
        <Source fileName="C:/Platypus/PlatypusExample/Platypus.java" />
      </Sources>
      <Dependencies>
        <Dependency fileName="C:/Program Files/eclipse32/eclipse/plugins/
            ComposestarCore/Binaries/ComposestarCORE.jar" />
        <Dependency fileName="C:/Program Files/eclipse32/eclipse/plugins/
            ComposestarCore/Binaries/ComposestarJava.jar" />
        <Dependency fileName="C:/Program Files/eclipse32/eclipse/plugins/
            ComposestarCore/Binaries/ComposestarRuntimeInterpreter.jar" />
      </Dependencies>
      <TypeSources>
      </TypeSources>
    </Project>
    <ConcernSources>
      <ConcernSource fileName="C:/Platypus/PlatypusExample/Platypus.cps" />
    </ConcernSources>
  </Projects>
  <Settings>
    <Modules>
      <Module name="INCRE" enabled="False" config="C:/Program Files/eclipse32/
          eclipse/plugins/ComposestarJava/INCREconfig.xml" />
      <Module name="FILTH" input="" />
      <Module name="SECRET" mode="-1" />
    </Modules>
    <Paths>
```

```
      <Path name="Base" pathName="C:/Platypus/" />
      <Path name="Composestar" pathName="C:/Program Files/eclipse32/eclipse/plugins/
          ComposestarCore/" />
      <Path name="EmbeddedSources" pathName="embedded/" />
      <Path name="Dummy" pathName="dummies/" />
    </Paths>
  </Settings>
  <Platforms>
  <Platform name="Java" mainClass="Composestar.Java.MASTER.JavaMaster" classPath="%
      composestar%/binaries/ComposestarCORE.jar;%composestar%/binaries/
      ComposestarJava.jar;%composestar%/binaries/antlr.jar;%composestar%/binaries/
      prolog/prolog.jar;%composestar%/binaries/groove/castor-0_9_5_2-xml.jar;%
      composestar%/binaries/groove/groove-1_2_0.jar;%composestar%/binaries/groove/
      jgraph.jar;%composestar%/binaries/groove/xerces-2_6_0-xercesImpl.jar;%
      composestar%/binaries/groove/xerces-2_6_0-xml-apis.jar;%composestar%/binaries/
      javassist.jar" options="">
    <Languages defaultLanguage="Java">
      <Language name="Java">
        <Compiler name="JavaCompiler" executable="javac.exe" options=""
            implementedBy="Composestar.Java.COMP.JavaCompiler">
          <Actions>
            <Action name="Compile" argument="{OPTIONS} {SOURCES}" />
            <Action name="CreateJar" argument="jar {OPTIONS} {NAME} {CLASSES}" />
          </Actions>
          <Converters />
        </Compiler>
        <DummyGeneration emitter="Composestar.Java.DUMMER.JavaDummyEmitter" />
        <FileExtensions>
          <FileExtension extension=".java" />
        </FileExtensions>
    </Language>
    </Languages>
    <RequiredFiles>
        <RequiredFile fileName="ComposestarCore.jar" />
        <RequiredFile fileName="ComposestarJava.jar" />
        <RequiredFile fileName="prolog/prolog.jar" />
    </RequiredFiles>
  </Platform>
</Platforms>
</BuildConfiguration>
```

Appendix B

Screenshots of the Eclipse Plug-in


New Compose* Java Project

Create a Compose* Java project

Create a Compose* Java project in the workspace or in an external location.

Project name:

Contents

- [✓ Create new project in workspace]
- Create project from existing source

Directory: C:\ComposestarSVN\Java\Examples

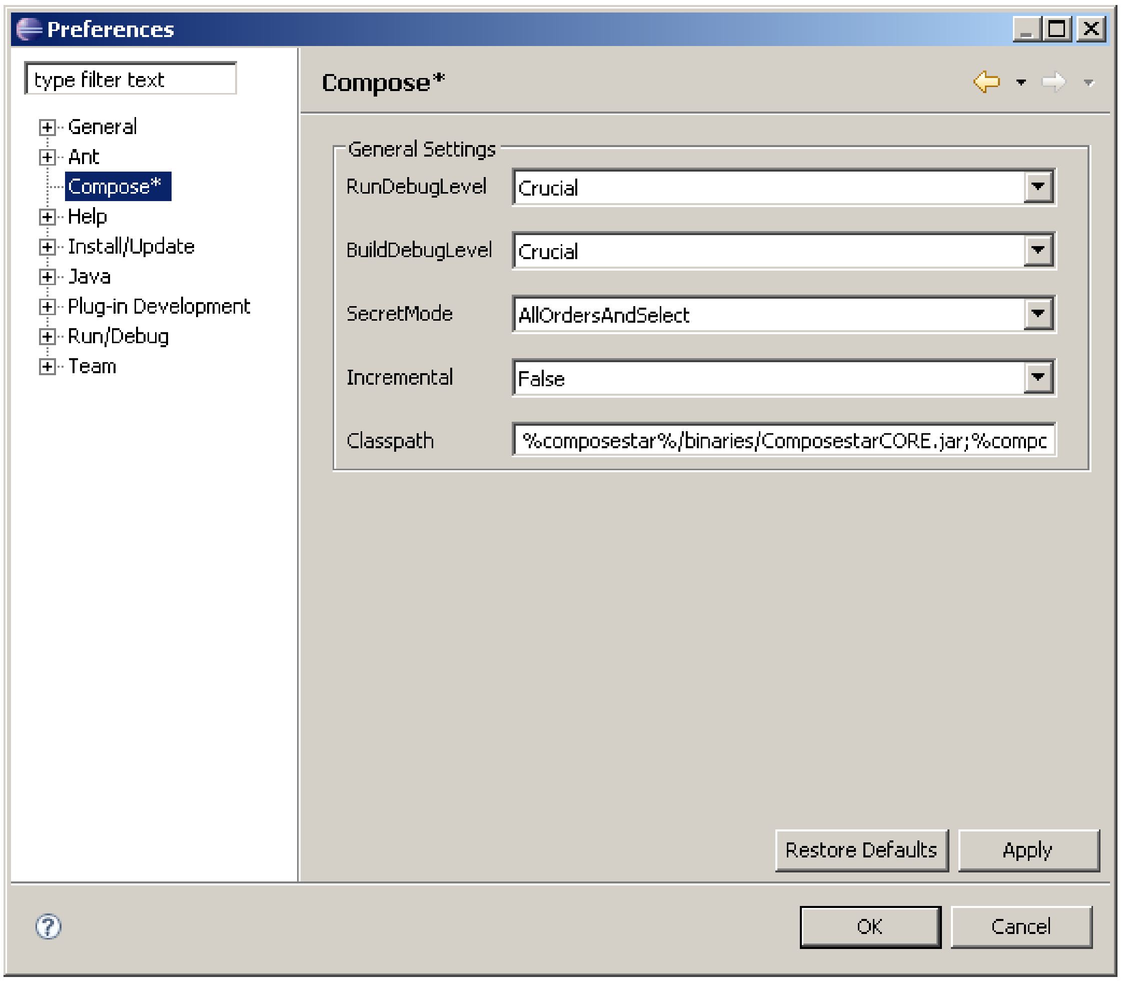
JRE

- [✓ Use default JRE (Currently 'jdk1.5.0_05')]
- Use a project specific JRE: jdk1.5.0_05

Project layout

- [✓ Use project folder as root for sources and class files]
- Create separate source and output folders

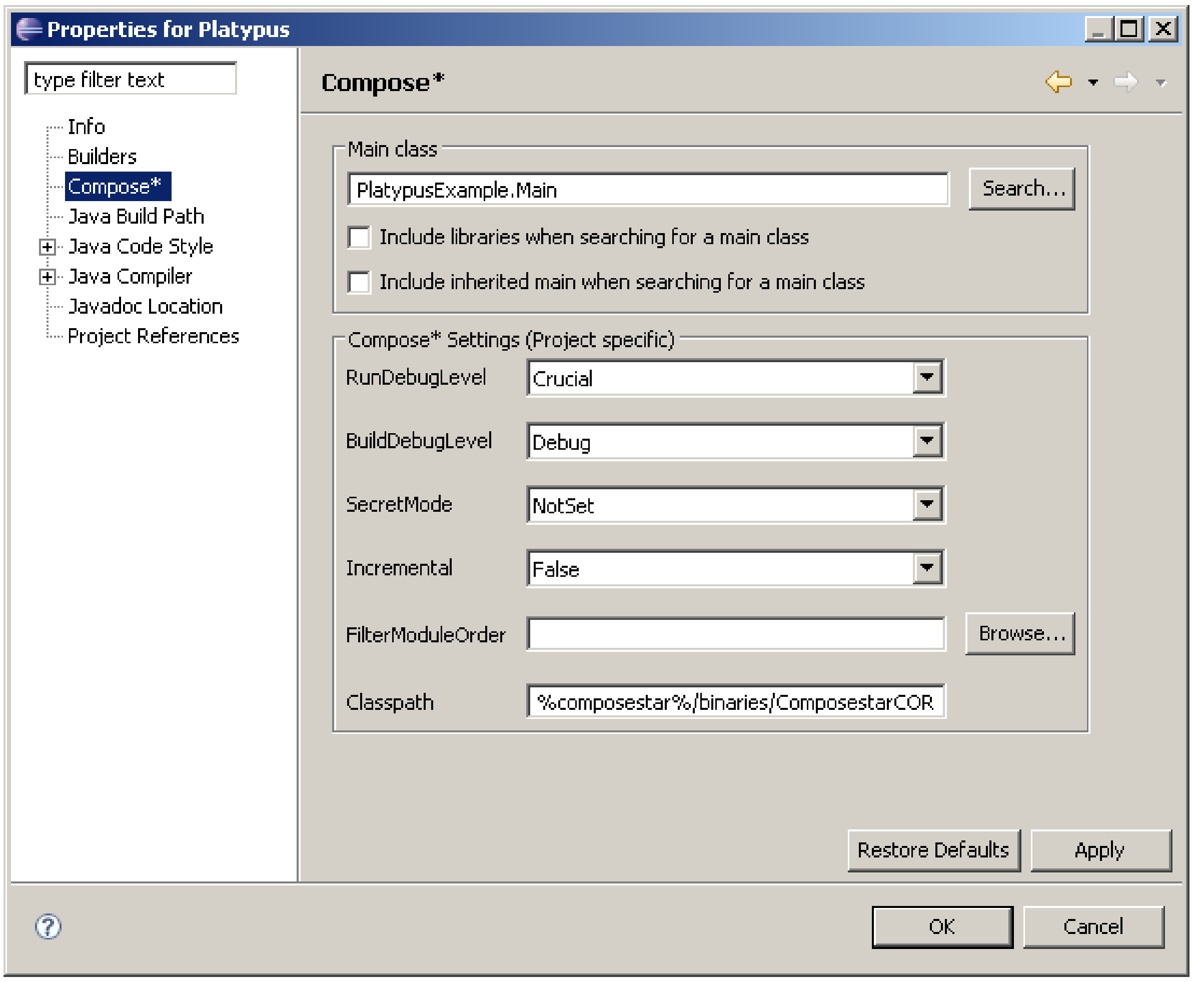
Figure B.1: Wizard for creating a Compose⋆/J project.



General Settings

RunDebugLevel: Crucial
BuildDebugLevel: Crucial
SecretMode: AllOrdersAndSelect
Incremental: False
Classpath: %composestar%/binaries/ComposestarCORE.jar;%compc

Figure B.2: Setting the global Compose⋆/J compiler settings.


Properties for Platypus

type filter text

- Info
- Builders
- Compose*
- Java Build Path
- Java Code Style
- Java Compiler
- Javadoc Location
- Project References

Compose*

Main class

PlatypusExample.Main

Search...

☐ Include libraries when searching for a main class

☐ Include inherited main when searching for a main class

Compose* Settings (Project specific)

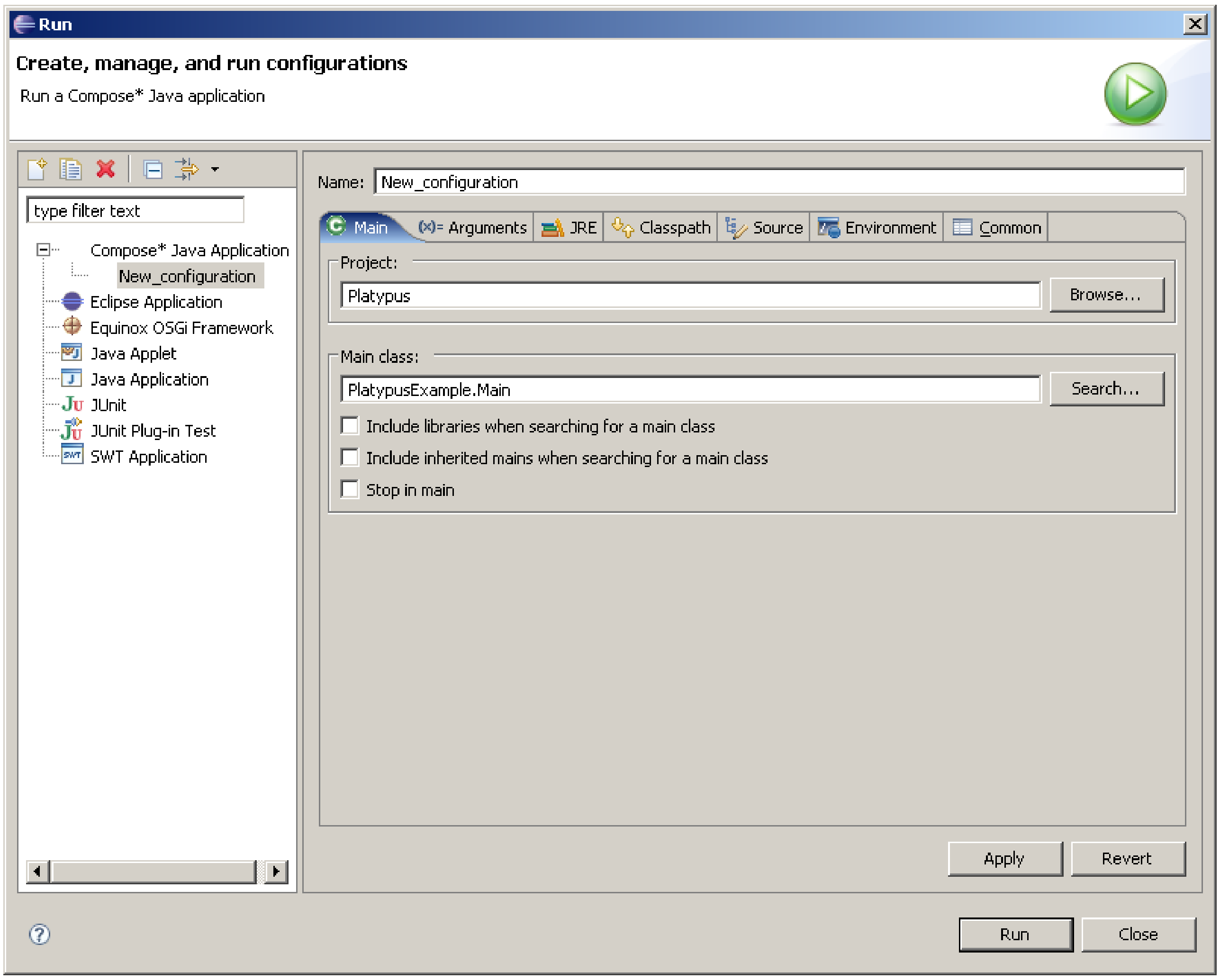
RunDebugLevel	Crucial
BuildDebugLevel	Debug
SecretMode	NotSet
Incremental	False
FilterModuleOrder	
Classpath	%composestar%/binaries/ComposestarCOR

Browse...

Restore Defaults   Apply

OK   Cancel

Figure B.3: Setting the project Compose⋆/J compiler settings.



Name: New_configuration

Project: Platypus

Main class: PlatypusExample.Main

- [ ] Include libraries when searching for a main class
- [ ] Include inherited mains when searching for a main class
- [ ] Stop in main

Figure B.4: Launch configuration for launching a Compose⋆/J project.