Vrije Universiteit Brussel – Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes – France
and
University of Twente – The Netherlands
2001

Adding Systemic Crosscutting and Super-Imposition to Composition Filters

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Patricio Salinas Caro

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoters: Lodewijk Bergmans & Mehmet Aksit (University of Twente)

Dedicated to my family.

Table of Contents

CHAPTER 1 : INTRODUCTION 1

1.1 About the contents of this thesis 1
1.1.1 Motivation 1
1.1.2 The contents 1

1.2 An Example: An Administrative System for Social Security Services 2
1.2.1 An Administrative System for Social Security Services 2
1.2.2 The Software System 2

1.3 System Evolution 4
1.3.1 Evolution 1: Protecting Documents 4
1.3.2 Evolution 2: Adding Workflow 5
1.3.3 Successive evolutions 5
1.3.4 Evaluation 5
1.3.5 The proposed model 6
1.3.6 Subsequent Evolutions 7

1.4 Problem Statement 7
1.4.1 Objectives 7

1.5 Thesis Outline 8

CHAPTER 2 : BACKGROUND 9

2.1 Aspect-Oriented Programming 9
2.1.1 Separation of concerns 9
2.1.2 Not only related to OOP 9

2.2 The Composition-Filters Model 10

CHAPTER 3 : STATE OF THE ART 13

3.1 Super-imposition and crosscutting 13
3.1.1 Super-imposition 13
3.1.2 Crosscutting 14

3.2 AOP Languages based on Java 16
3.2.1 AspectJ 16
3.2.2 HyperJ 18

3.3 Composition Filters 19

3.4 ComposeJ 20
3.4.1 Characteristics 20
3.4.2 Definitions 20
3.4.3 Defining a CF-interface 20

3.5 Super-Imposition and the example problem 21
3.5.1 The previous model 21
3.5.2 Applying Super-imposition 21
3.5.3 Progressive super-imposition 24

3.6 Summary 24

CHAPTER 4 : DESIGNING THE LANGUAGE 25

4.1 The language 25

4.2 Presenting the language 25
4.2.1 The Concern declarations 25
4.2.2 Super-imposition of modules 26
4.2.3 How do we specify the super-imposition? 27

4.3 The Concern grammar 29
4.3.1 Existing Implementations of CF 29
4.3.2 UML-like grammar 29

4.4 The sections of a concern 29
4.4.1 The filterinterfaces section 30
4.4.2 The implementation section 30
4.4.3 The Super-Imposition section 31

4.5 Working out crossing problems 35

4.6 ConcernJ 36

4.7 Parts of a concern 37
4.7.1 The concern specification 37
4.7.2 Filter-Interfaces 37
4.7.3 Implementation 38
4.7.4 Super-Imposition 38

4.8 Summary 38

4.9 Summary Error! Bookmark not defined.

CHAPTER 5 : THE DESIGN OF THE CONCERNJ TOOL 40

5.1 Producing code 40
5.1.1 ComposeJ Code 40
5.1.2 AspectJ Code 41
5.1.3 HyperJ Code 41
5.1.4 Java Code/Class files 42
5.1.5 Making a Decision 43

5.2 How do we produce the code? Error! Bookmark not defined.
5.2.1 Parsers & Visitors 44

5.3 The ConcernJ Architecture 44
5.3.1 The packages distribution 44
5.3.2 The System structure 45
5.3.3 Collecting the information 46
5.3.4 Java Visitors 47
5.3.5 Applying Concerns 47
5.3.6 Concern Visitors 48
5.3.7 Selectors Visitors 49
5.3.8 Structures after a super-imposition and crossing problems **Error! Bookmark not defined.**

5.4 Structures in detail 51
5.4.1 The System 51
5.4.2 The Core Structure 52
5.4.3 The Selectors Visitor 52
5.4.4 Java Visitors 53
5.4.5 Concern Visitors 53

5.5 Conclusions Error! Bookmark not defined.

CHAPTER 6 : CONCLUSIONS 55

6.1 Results versus problem statement 56

6.2 Further Work 56

BIBLIOGRAPHY & REFERENCES 57

APPENDIX A: CONCERNJ GRAMMAR 58

APPENDIX B: SELECTORS GRAMMAR 63

Chapter 1 : Introduction

1.1 About the contents of this thesis

1.1.1 Motivation

Programming languages are implemented and based on others following the traditional *parse-eval-display* structure, that is, source code is read, evaluated and finally produce some result. High-level languages have become necessary to model and apply new paradigms or to simplify tasks than in low-level languages are not trivial or too cumbersome to implement.

In that context, taking for instance the interpretation that a computer chip does from a stream of bits coming from some assembler code, let us suppose for a moment that this stream has come, after several *parse-eval-display* steps, from a sophisticated high-level language. Why do not we still program and produce stream of bits instead of encapsulate them into programs coming from a high-level language?

The response is clear, we need abstraction to focus our efforts in solving another problems instead of worrying us about bits and bytes.

From this point of view, most languages define new ways for encapsulating knowledge and making abstractions easy to model and apply. This is the case of current Aspect-Oriented languages; they define ways for encapsulating concerns though the incorporation of new specifications and translating them finally into a low-level code.

For instance, AspectJ is an AOP implementation for the Java language. AspectJ comes with a new set of specifications that after applying them over AspectJ code produces Java code. The advantage of using AspectJ code instead of directly Java code is the fact that the abstraction allow us to model a system in such a way that is possible to reach modularity, decrease complexity and increase reuse. Therefore, we can conclude that abstraction is necessary.

1.1.2 The contents

Composition-Filters is a technique that allows changing the behavior of objects just by adding some specification over them in a modular way. However, currently there is no a systematic way for super-imposing them over an application.

This thesis work is intended to study and design a language, that based on the Composition-Filters model, allows us the super-imposition of CF-specifications over a system. In this way modularity is achieved by using the Composition-Filters technique and abstraction through using a super-imposition technique.

The study of that new/improved language needs an implementation for proving concepts and without loosing generality, for this thesis work a tool will be created for the super-imposition of Composition-Filters for the Java language. Nevertheless, there is the intention to make the design as much as possible language-independent due to the nature of Composition-Filters, which is a language-neutral model.

1.2 An Example: An Administrative System for Social Security Services

The example we are about to show is also presented in [BeAk-00], and it is intended to illustrate the issue of composing and reusing multiple concerns in object-oriented applications when the requirements evolve.

Also, this example serves as a motivation for showing the necessity of counting with a technique that enables the superimposition of Composition-Filters definitions over a system, and thus this thesis work.

1.2.1 An Administrative System for Social Security Services

Assume that a government-funded agency is responsible for the implementation of disablement insurance laws. As shown in Figure 1-1, the agency implements five tasks:

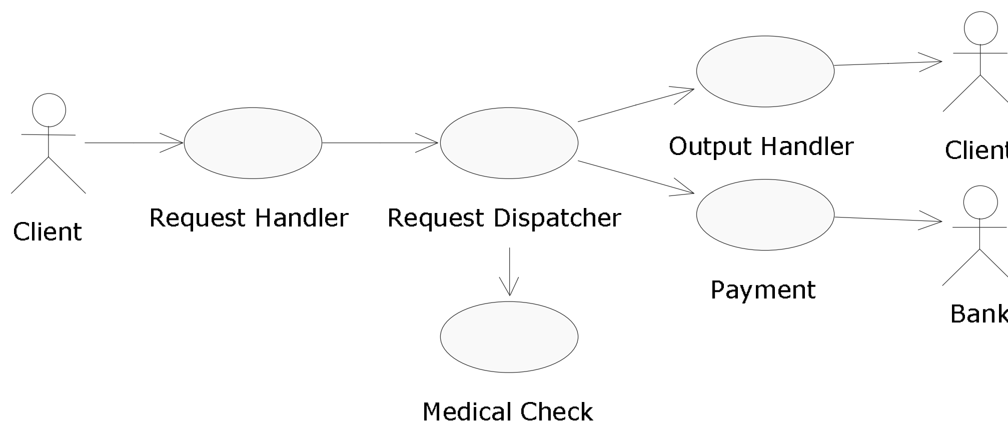

Figure 1-1 . Tasks in the example system

Where each task in the System represents:

1. **Request Handler**: creates an entry for clients (entries are represented as documents)
2. **Request Dispatcher**: implements the evaluation and distribution of the requests to the necessary tasks.
3. **Medical Check**: is responsible for evaluating client's disablement.
4. **Payment**: is responsible for issuing bank orders.
5. **Output Handler**: is responsible for communicating with the clients,

A typical claim is subsequently processed by: *RequestHandler*, *RequestDispatcher*, *MedicalCheck*, *Payment* and *OutputHandler*. Various other interaction scenarios are also possible.

1.2.2 The Software System

Modeling Client's Requests

Now let us assume that the system has been implemented as a set of tasks, for each client's request a ***document*** is created. Depending on the document type and client's data, the document is edited and sent to the appropriate tasks using a standard email system. Each relevant task processes the document accordingly.

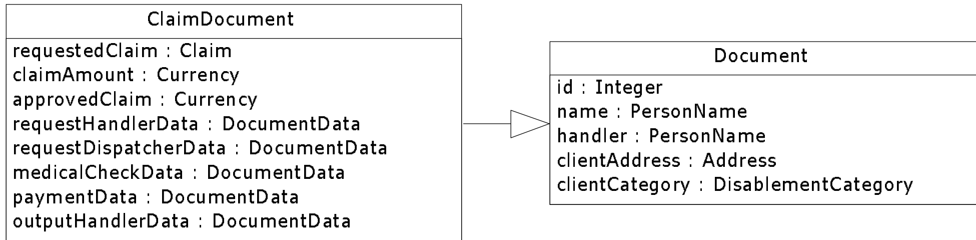

Figure 1-2 . Part of the document class hierarchy, which is used for representing client requests.

As shown in Figure 1-2, class *Document* is the root class for all document types. Every document has 5 attributes:

- ***id***, ***name*** and ***clientAddress*** are used for storing client's data;
- ***handler*** represents the clerk who is in charge of processing the request;
- ***clientCategory*** specifies the classification of the client with respect to the disablement laws.

Class *Document* also implements 10 operations, which are not shown here. These are used for reading and writing the class attributes ('setters' & 'getters').

Class Document has several subclasses. For example, ClaimDocument is used to represent the claims of clients. This class declares 8 attributes:

- ***requestedClaim*** represents the type of client's claim, such as medicine, hospital costs, etc.;
- ***claimAmount*** is the claimed amount of money;
- ***approvedClaim*** is the amount approved by the agency.

The remaining attributes are filled in by various tasks while the document is being processed.

Modeling the Tasks

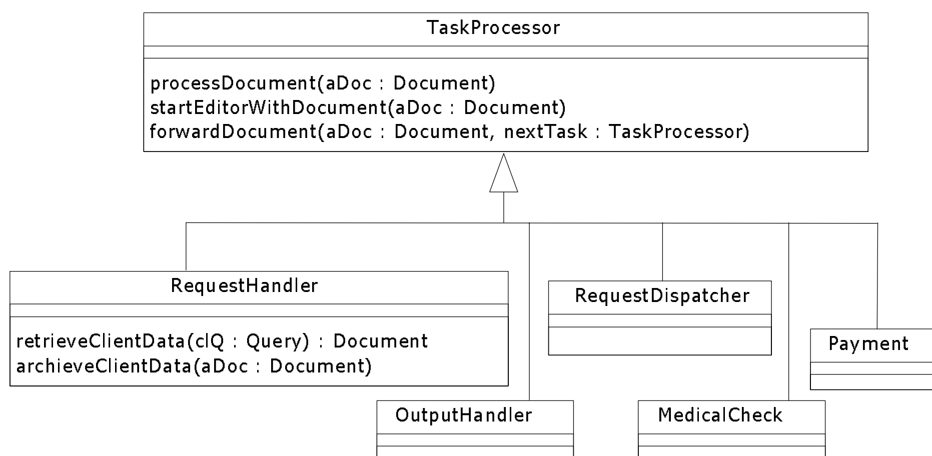

Figure 1-3 . Class hierarchy for tasks

As shown in Figure 1-3, *TaskProcessor* declares the basic operations for all tasks. The operation *processDocument* accepts a document as an argument and opens an editor for the document by calling on *startEditorWithDocument*. When the document is edited by a

particular task, the operation *forwardDocument* is called. Both of those operations are overridden by the subclasses.

Each sub-class of *TaskProcessor* redefines the operations *processDocument* and *forwardDocument*. They are not shown though, to keep the model simple.

RequestHandler implements the front-end of the office. For example, if a client wants to issue a claim, this task creates an object of *ClaimDocument*, retrieves the necessary client data and opens an editor for the document object.

The responsible clerk should enter the data on the field defined for *RequestHandler*. When the task is completed, the clerk selects the next task and calls *forwardDocument*.

The operation *forwardDocument* prepares the document and passes it as an argument to the operation *processDocument* on the next task. Subsequently, each clerk in the process enters data into the appropriate data field and forwards the document according to the office procedure.

In this system, creating a new workflow process can be realized by creating a new structural document subclass.

1.3 System Evolution

The following sections are intended to show possible scenarios of evolution and the possibilities to deal with this evolution that users have with current modeling techniques.

The complete set of evolutions and their fundaments are clearly explained in [BeAk-00]. Therefore, in this case they are going to be briefly explained, just showing only the most interesting aspects for each evolution.

1.3.1 Evolution 1: Protecting Documents

In the initial system, a clerk could edit any field in a document. A request dispatcher clerk could, for instance, accidentally edit the medical data field. Therefore, it was found necessary to protect the documents. We consider two alternatives for enhancing *ClaimDocument* for protection: to modify and recompile *ClaimDocument* or to introduce a new class and reuse class *ClaimDocument* through inheritance or aggregation.

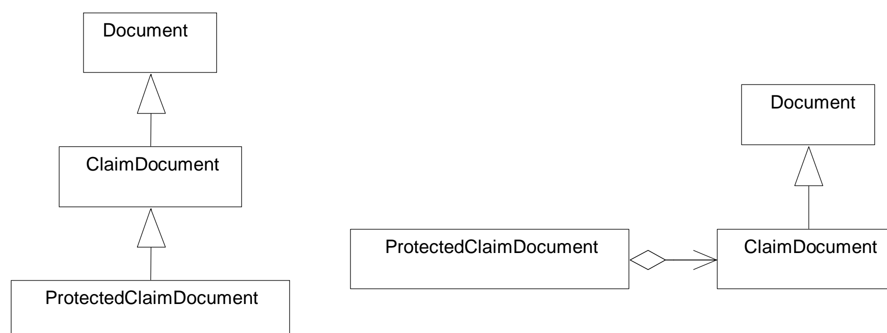

Figure 1-4. Two possibilities for adding ProtectedClaimDocument to the current system, as inheritance (left) and as an aggregation (right).

From the point of view of Software Engineering the option of modifying and recompiling *ClaimDocument* is not acceptable, because in a future evolution the implementation or the requirements for *ClaimDocument* could change, new kind of documents could be added to the system, etc. making reuse and adaptability difficult to apply.

Figure 1-4 shows the other two possibilities for adding *ProtectedClaimDocument* to the system. The inheritance model is shown on the left side and the aggregation model is shown on the right side.

Only authorized clients should be able to view or modify the document's content. A possible solution is adding a new attribute *activeTask* to *ProtectedClaimDocument*, in this way it is possible to know the current task. Therefore, a view-enforcement over the document can be applied depending on the invoked method.

So, the *activeTask* field has to be checked before invoking a real method inside *ClaimDocument*. Otherwise, if the restrictions over *ClaimDocument* are not satisfied, an exception is raised.

When using aggregation, forwarding methods have to be added to the *ProtectedClaimDocument* interface, for invoking those methods into *ClaimDocument* that are not necessary to be checked.

When using inheritance, forwarding methods are not necessary because they are provided by the inheritance mechanism.

The advantage of using an aggregation approach is that the *ProtectedClaimDocument* have a run-time adaptability, because it could change its implementation at run-time just by changing the aggregated object.

1.3.2 Evolution 2: Adding Workflow

In the previous implementation, the clerks had to decide which task is to be executed next. To enforce a particular process, a workflow class is introduced called *WorkFlowEngine*.

Adding a workflow to the system requires redefinition of *forwardDocument* for all task classes. The *forwardDocument* first calls on *selectTask* of *WorkFlowEngine*, which returns the next task. The operation *forwardDocument* cannot be implemented at the *superclass* level, since every task implements this operation in a specific manner.

This implies that methods of every task class have to be redefined, by applying inheritance or aggregation.

1.3.3 Successive evolutions

The application could suffer new evolutions, such as:

1. Adding Document Queues
2. Adding Logging
3. Adding Locking
4. Adding Persistence

Each evolution could imply the introduction of new classes and/or the redefinition of existing ones. When using inheritance or aggregation, methods have to be redefined or forwarded to the original implementations for reusing existing code.

1.3.4 Evaluation

As is presented in [BeAk-00], the Composition Filters model is one of the most appropriated approaches for modeling this system, comparing it with classic object-oriented techniques such as edit & compile, aggregation and inheritance.

1.3.5 The proposed model

The Composition-Filters model will be presented in the following chapter. Nevertheless, Figure 1-5 shows a symbolic representation when filters are added to objects. In this case, filters are represented as rectangles and objects as circles.

Filters are applied to objects in a modular way; the object itself does not suffer any changes, and the interface that contains the filters is in charge of manipulating incoming and outgoing messages.

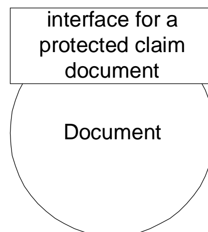

Figure 1-5. ProtectedClaimDocument presented as a Composition-Filters interface*.*

Figure 1-5 shows the proposed version corresponding to the first evolution for a *ProtectedClaimDocument*. A Composition-Filters interface, containing filters, is applied to the original *Document* class (*ClaimDocument*). Incoming messages are filtered, depending on the concext, some methods are forwarded to the original implementation and exceptions are triggered when corresponds.

For the second evolution, the introduction of a *WorkflowEngine*, Composition-Filters interfaces are added over those objects that are part of the workflow. This case is shown in Figure 1-6.

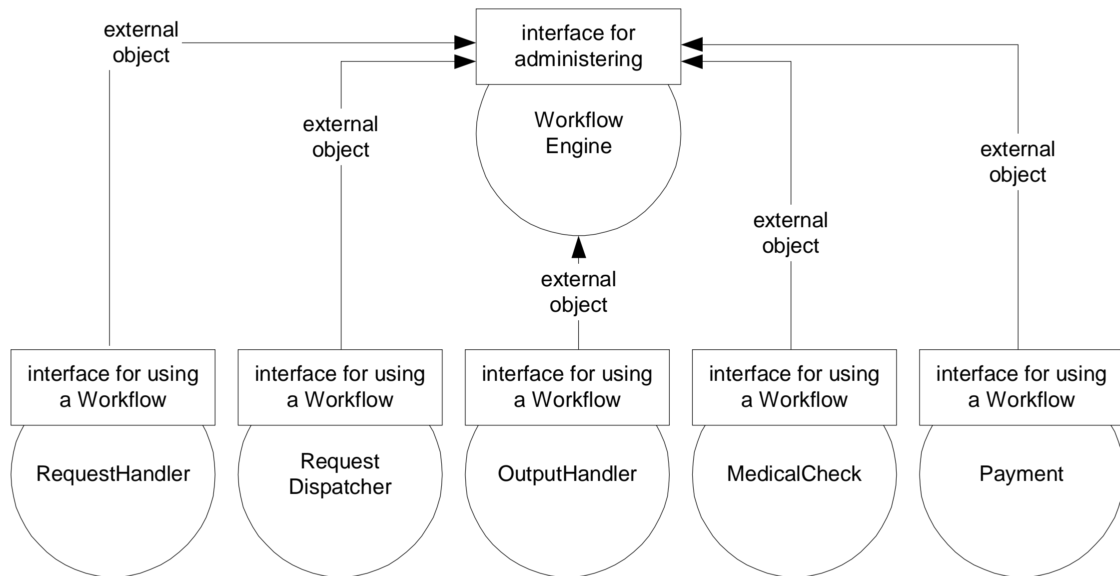

Figure 1-6. The proposed Composition-Filters model for the introduction of WorkflowEngine to the original system.

The objects affected by the introduction of *WorkflowEngine* are *Requesthandler*, *RequestDispatcher*, *OutputHandler*, *MedicalCheck* and *Payment*. Each CF-interface defines an external object for applying the workflow mechanism (*WorkflowEngine*), and each interface has to be added manually over those involved classes.

Currently, there is no a systematic way to apply interfaces over a large set of classes.

1.3.6 Subsequent Evolutions

The mechanism for avoiding subsequent evolutions such as Logging, Locking, Persistence, etc. is the application of new CF-interfaces over the objects involved with each evolution. In this way, filters are added on top of the current object interface.

Figure 1-7 shows the case when adding the Logging interface (black colored) over the existing *ProtectedClaimDocument* interface (gray colored). Filters can be added in this way thanks to the characteristics of the Composition-Filters technique.

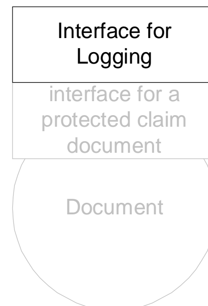

Figure 1-7. Adding a Logging mechanism to ProtectedClaimDocument.

In the example, the Logging interface was only added over *Document*. Nevertheless, it can also be added over other objects into the system.

1.4 Problem Statement

The Composition Filters model can help to solve the presented problems, by applying filter-interfaces over the objects that were affected by the evolution of the original requirements.

However, taking into account the number of interfaces applied to the affected objects, possible new software evolutions and common tasks performed by applied interfaces, increases inevitably the complexity of the given solution, making it harder to reuse existing implementations.

The main key because this technique fails, is the fact that there is currently not a systematic, modular, explicit and automated way to apply interfaces to a large set of objects. Although the current Composition-Filters model allows the reuse of already defined filters, there is no other choice than make and apply interfaces *"manually"*.

1.4.1 Objectives

The ***general objective*** of this thesis work is the study, design and creation of a tool that will give the user the ability to define and apply CF-interfaces to a set of objects, using a systematic and modular approach. The work must be based on currently well proven techniques and tools.

The ***specific objectives*** are to investigate, design and create a specification for super-imposition of Composition-Filters. The grammar for this language will be studied, designed and presented. Finally an implementation for this specification will be built on the Java language.

1.5 Thesis Outline

In this chapter a brief introduction to this thesis was presented. Also an example that shows the necessity of a better approach than using modification, inheritance and aggregation was presented.

Chapter 2 contains background knowledge consisting of current crosscutting techniques such as Aspect-Oriented Programming and the Composition Filters model.

Chapter 3 describes the State of the Art of current crosscutting implementation tools, such as AspectJ, HyperJ and ComposeJ. Also, the super-imposition technique is presented and compared with the other techniques.

Chapter 4 presents the design of a super-imposition language based on the Composition Filters model. This includes the study and design of the grammar, and the most interesting aspects of the super-imposition concepts applied over the new language.

Chapter 5 describes the software design for the super-imposition tool. This tool is very particular and represents the proof of concept of previous chapters. It includes the possibilities for generating code taking into account existing crosscutting tools.

Finally, chapter 6 contains the conclusions for this thesis, showing the advantages and drawbacks of the presented tool, and discussing some further work that can be done based on this thesis work.

Chapter 2 : Background

2.1 Aspect-Oriented Programming

2.1.1 Separation of concerns

The following quote was taken from [HYPJ-01] and explains the concept of "Separation of Concerns":

"Separation of concerns is a concept that is at the core of software engineering. It refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concern (concept, goal, purpose, etc.).

Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts. Many kinds of concerns may be relevant to different developers in different roles, or at different stages of the software lifecycle. For example, the prevalent concern in object-oriented programming is the class, which encapsulates data concerns.

Feature concerns like printing, persistence, and display capabilities, are also common, as are concerns like aspects, roles, variants, and configurations. Appropriate separation of concerns has been hypothesized to reduce software complexity and improve comprehensibility; promote traceability; facilitate reuse, non-invasive adaptation, customization, and evolution; and simplify component integration."

2.1.2 Not only related to OOP

Aspect Oriented Programming [KIC-97], AOP for short, is intended to solve common "separation of concerns" problems. The following quote was taken from [ACM-01] and explains the AOP terminology:

"AOP is a new evolution in the line of technology for separation of concerns – technology that allows design and code to be structured to reflect the way developers want to think about the system. AOP builds on existing technologies and provides additional mechanisms that make it possible to affect the implementation of systems in a crosscutting way.

In AOP, a single aspect can contribute to the implementation of a number of procedures, modules or objects. The contribution can be homogeneous, for example by providing a logging behavior that all the procedures in a certain interface should follow; or it can be heterogeneous, for example by implementing the two-sides of a protocol between two different classes.

As with all other separation of concerns technology, the goal of AOP is to make designs and code more modular, meaning that the concerns are localized rather than scattered and have well-defined interfaces with the rest of the system. This provides us with the usual benefits of modularity, including making it possible to reason about different concerns in relative isolation, making them (un)pluggable, amenable to separate development etc."

Technology	Key concepts	Constructs
Structured programming	Explicit control constructs	Do, while and other loops, blocks etc.
Modular programming	Information hiding	Modules with well-defined enforced interfaces
Data abstraction	Hide the representation of data	Types…
Object-oriented programming	Objects, with classification and specialization	Classes, objects, polymorphism.

As shown, AOP is independent of the paradigm. Nevertheless, Object Oriented programming is today one of the most used paradigms, intended to model the world with

objects, where each object is intended to model different concerns [ACM-01], allowing reuse of code, decreasing software-complexity and increasing modularity. So, most existing AOP implementations are intended for object-oriented languages.

2.2 The Composition-Filters Model

The Composition-Filters model [AKS-92] is an extension to the classical Object-Oriented model. It is related to Separation of Concerns and Aspect Oriented Programming. Filters extend the classic object interface, by interfering -not invasively- the object message-sending mechanism. So, the extension is not at the level of the object itself (implementation), it is at the level of its interface.

Messages among objects, both incoming and outgoing, are caught by filters. Messages are inspected and manipulated at runtime. Filters are added to objects in a modular way. When filters are applied, objects start behaving differently depending on the applied filters and how they were defined.

This enables the modeling of concerns, increases reuse and maintainability. This is because classic OO-approaches, such as inheritance and aggregation, are still possible to use, and with Composition-Filters is possible to apply modularly new features to objects. Finally, we can say that the Composition-Filters model is a dynamic approach because of its runtime features.

More extensive explanations about the Composition-Filters model can be found in [AKS-92], [BERG-94], [SINA-95], [GLAN-95] and [WICH-99].

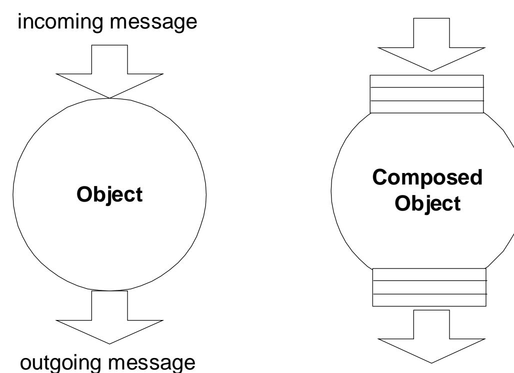

Figure 2-1. A Normal object and a Composed object.

Figure 2-1 shows the two approaches, on the left side the classical Object-Oriented model where messages are received and sent by object itself, and on the right side, the Object-Oriented model empowered with Composition-Filters, where filters manipulate messages before they are really received by or sent from the object.

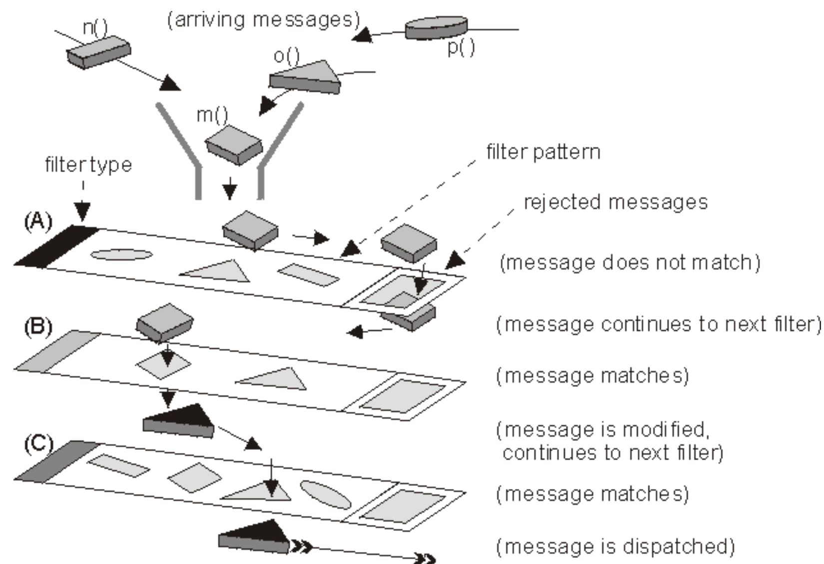

Figure 2-2. An intuitive schema of message filtering.

Figure 2-2 shows the schema for message filtering, filters can be sequentially applied to an object acting as a new *layers*. Each message has to pass the filters one-by-one, until it is discarded or can be dispatched. So, when adding and defining filters, the order in which they are applied does matter.

The following filter-definitions are taken from [BeAk-00]:

- **Dispatch filter**: When a message is accepted, this filter decides where to dispatch the incoming/outgoing message, otherwise the message continues with the next filter. If there are no more filters, an exception is raised. Messages only can be dispatched to a specified target, the target can be the *self* object, internals or externals defined in the applied CF-interface.
- **Error Filter**: When a message is accepted, it continues to the next filter, otherwise an exception is raised.
- **Wait Filter**: When a message is accepted, it continues to the next filter. The message is queued as long as the evaluation of the filter expression results in a rejection.
- **Meta Filter**: When a message is accepted, the reified message is sent as a parameter of another *meta message* to a named object, otherwise the message continues to the next filter. The object that receives the meta message can observe and manipulate the message, then re-activate its execution.

A Composition-Filters specification contains two different modules, the first one defines an interface, which is language independent, and the second one an implementation that realizes the interface.

Both the interface and the implementation specifications are shown in Figure 2-3.

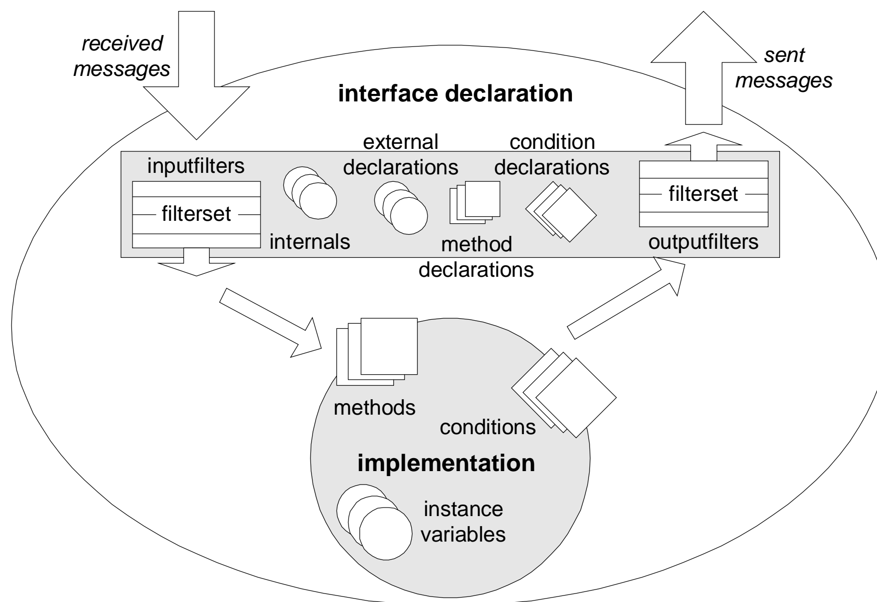

Figure 2-3. A schema for representing the incorporation of Composition-Filters over an object.

Figure 2-3 shows in gray color the interface declaration and the implementation. Respectively one consisting of the following:

1. A Composition-Filters interface that contains:
   - ***internal definitions***. Internals are objects that are part of each CF-interface. So, when an interface is applied over an object, operations can use internals as normal object attributes. Internals can be instantiated when defining an interface.
   - ***external declarations***. Externals are also objects that are part of each CF-interface. The difference with internals is that externals are passed as references to interfaces when they are applied. Externals cannot be instantiated when defining an interface.
   - ***method declarations***. Methods defined inside the CF-interface are intended to complement and extending the current object interface. They can be added/removed systematically when CF-interfaces are applied.
   - ***condition declarations***. Conditions are very specific methods that do not receive any parameter, return always a boolean value and do not change the object state. Conditions are used for inspecting the current state of an object.
   - ***inputfilters definitions***. Filters are defined in this block for inspecting and manipulating incoming messages.
   - ***outputfilters definitions***. Filters are defined in this block for inspecting and manipulating outgoing messages.

2. An implementation-block, that realizes the defined interface:
   - ***Method implementations***. Corresponds to the implementation of all those methods that are not implemented by the current object signature.
   - ***Condition implementations***. Idem as methods, but intended for conditions.
   - ***Instance variables***. Corresponds to variables that can be used by the defined methods and conditions, but cannot be used by filters.

When a message arrives, it is inspected and manipulated by input filters. It can be dispatched to the object itself, internals or externals. When the object sent a message, it is inspected and manipulated by output filters. It can be dispatched to the real destination or redirected to self, internals or externals.

Both the interface and the implementation specifications are defined separately; thus the language-independent part is specified separately from the language-dependent part.

Chapter 3 : State of the art

3.1 Super-imposition and crosscutting

Super-imposition and crosscutting are different concepts:

- ***Super-imposition*** is the technique for adding code, systematically, by specifying the places into the application where the new code will be introduced.

- ***Crosscutting*** is a characteristic of the super-imposed code. When a *crossing* concern is feasible to be modeled separately from main concerns[1], and after be applied over the system by using super-imposition.

3.1.1 Super-imposition

In super-imposition there are four important concepts:

- ***Who***: the modules into a system we want to super-impose. A system contains several modules, when selecting "who" only those selected modules will be part of a super-imposition.
- ***What***: the code we want to super-impose. This code is super-imposed over those modules defined by "who".
- ***Where***: the places where we want to super-impose the new code. When defining "who" we just decide which modules are going to be super-imposed, when defining "where" we decide the place into those modules we want to put the new code.
- ***When***: the instant of time when the new code has to be executed (with respect the original code).

A schematic model for super-imposition is presented in the following figure.

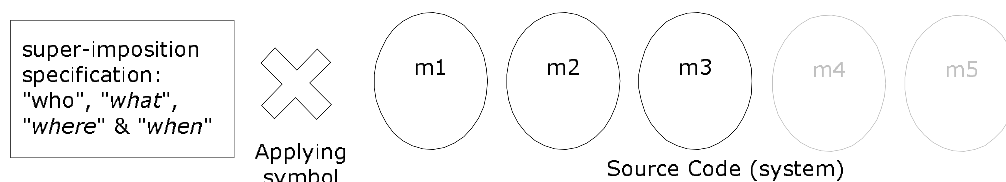

Figure 3-1. Identifying Super-imposition concepts.

Figure 3-1 shows a graphical representation for super-imposition, where each shape represents the following:

1. The *rectangle* represents the super-imposition specification, which contains the information related with ***who, what***, ***where*** and ***when***.
2. The *cross* symbol represents the application of super-imposition.
3. Oval figures represent a system. In this example they are called modules (m1, m2 and m3, m4 and m5).
4. Only selected modules into the system will be super-imposed (in black color). The rest of the modules are not taken into account (in gray color).

In the presented figure, the term "module" was introduced to represent source code. This is mainly because, depending on the used language, modules are named differently and have different meanings. For instance, in Java a module is called "Class" and its source code is defined into a normal file. On the other hand, in Smalltalk a module is also a "Class" but the concept of file does not mean anything. Another example is when defining

[1] Main concerns are particular modules in each paradigm, in OOP a main concern is a class, in functional programming it is a function [ACM-01].

constructors, in languages such as Java and C++ a *constructor* has a meaning, on the contrary to Smalltalk where a *constructor* is just another class operation.

A possible result of that super-imposition is shown in Figure 3-2. Each shape in the figure represents:

1. The arrow symbol represents the result of weaving the super-imposition specification with the original source code.
2. Oval figures represent the same source code presented in Figure 3-1.
3. The dash-rectangles represent super-imposed code.

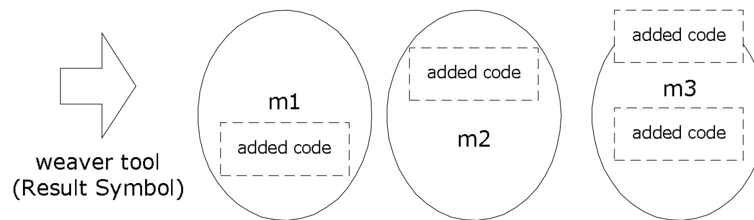

Figure 3-2. Result of a Super-imposition.

3.1.2 Crosscutting

When crossing concerns can be modeled into a separate module, and systematically it can be applied over a system by using super-imposition. In the following figure different modules are crosscutted by different concerns.

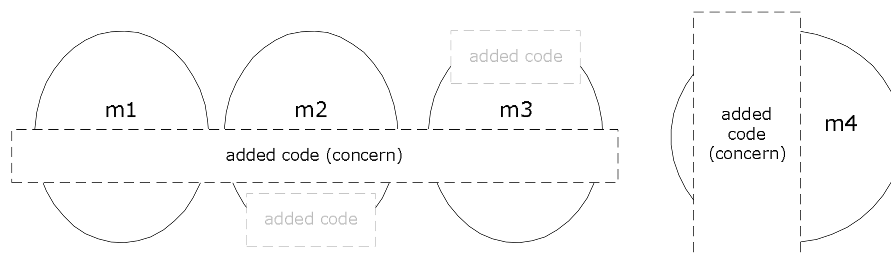

Figure 3-3. Showing the orthogonal characteristic of crosscutting.

Figure 3-3 shows on the left side a concern (a dashed rectangle drawn in black color) that *crosscuts* multiple modules (oval shapes), and on the right side a concern that crosscuts only one module. The dashed rectangles drew in gray color only represent super-imposed code.

Concerns are very specific modules that represent tasks such as security, persistence, logging, tracking, etc. The main characteristic of crosscutting is that specifications can be plugged-in systematically and defined modularly thanks to super-imposition.

When super-imposing several concerns it is also possible to obtain *crossing*[2] code. Figure 3-4 shows this case:

[2] When a concern cross-crosscuts another concern or code.

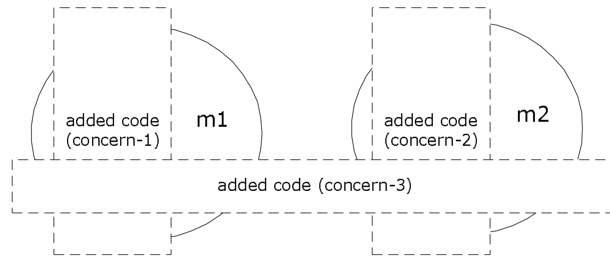

Figure 3-4. Multiple concerns crosscutting modules and each other.

In Figure 3-4 concerns, represented by dashed-rectangles, crosscut source code and each other. How does an applied concern have to be added? The following cases are interesting to take into account:

1. A new concern overrides old code, and thus previous applied concerns. So, it is still possible to use old code (the code existing before super-imposition).
2. A new concern replaces old code, and thus previous applied concerns. This is a drastic option, because always the newest super-imposed code is placed instead of the old existing code.
3. *Crossing* code has to be explicitly modeled as another concern.

These questions are interesting to be analyzed, even when applying only one concern (the case when modeling behavior of the applied concern with respect to the original source code). Concerns are applied by super-imposition, thus these cases have to be specified implicitly or explicitly depending on the used language.

Figure 3-5 shows the crossing-concerns problem:

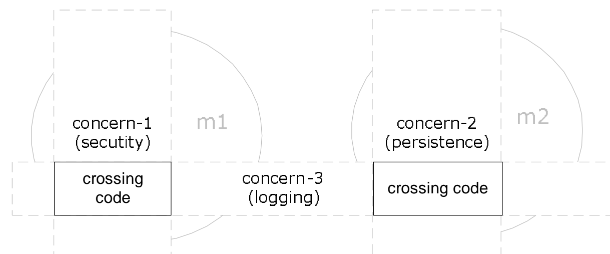

Figure 3-5. Concerns can cross each other.

Concerns are drawn in gray color and *crossing* modules are drawn in black-color. Let us transform these questions in real situations, such as, for instance:

What happens when the logging concern crosses the security concern? Does the logging concern violate security? Does the logging concern have to be applied after security-rules are checked out?

These kinds of questions are interesting to analyze for a correct modeling, and have to be worked out at the super-imposition level. This means that when defining the super-imposition specification, crossing problems have to be explicitly defined with the facilities that the used modeling tool provides.

When introducing some super-imposition languages later in the next section, these questions will be pointed out to show how those languages allow users the specification of this kind of problems.

3.2 AOP Languages based on Java

3.2.1 AspectJ

AspectJ is an AOP implementation for the Java language [ASPJ-01]. It allows the definitions of concerns that are called ***aspects***. A weaver applies aspect definitions over source code (Java Classes) and creates the woven version, also codified in Java.

Aspects are defined one by one. Several aspects can be applied over a system. Each aspect contains the following information:

1. ***Aspect Instantiation***: aspects cannot be explicitly instantiated (a ***new*** message to a class). Therefore, when aspects are defined, the instantiation mechanism has to be specified.

2. ***Event definitions***: in [SAL-00] was presented and proposed that applications have events that can be specified by ***pointcut*** definitions. Those events are basically:

    a) Object Instantiations,
    b) Method invocations, executions,
    c) Field setters and getters,
    d) Exception handlers.

3. ***Actions***: also in [SAL-00] was proposed that the specified events can be caught, and *actions* can be applied to take care of them. Those actions are called ***advices***.

4. ***Introductions***: desired code can be added to existing applications at the level of class definitions, such as, methods, class and instance variables and inheritance.

In AspectJ *aspects* are modeled as objects. Therefore, they can:

- Be defined abstract,
- Be inherited from another aspect (simple inheritance mechanism),
- Be instantiated[3],
- Have fields and operations.

Some Features in brief

Some interesting features of AspectJ are:

- Source files are woven with aspects at compile time.
- Aspects can be defined to follow the flow of an execution.
- Instantiation of aspects are specified explicitly when specifying the aspect.
- Aspects cannot be defined over another ones (it means that only one dimension can be defined).
- Aspects cannot be changed at runtime.
- AspectJ runtime-classes must be available for executing a woven system.

Solving crossing problems

When code is super-imposed by an aspect, there are three possibilities for applying it over previous existing code:

1. The action (advice) is executed ***before*** the code that generated the event. For instance when catching a method execution, the applied advice is executed ***before*** that method is really executed.

---

[3] They cannot be explicitly instantiated by sending a ***new*** message to the Class where they belong.

2. The action is executed ***after*** the code that generated the event. For instance when catching a method execution, the applied advice is executed ***after*** that method was executed.
3. The action is executed ***around*** the code that generated the event. For instance when catching a method execution, the applied advice is executed **instead** that method.

Those three concepts, *before*, *after* and *instead* are specified when an advice is defined.

When more than one aspect is applied over the system, there is the possibility to have crossing problems among aspects. In such a case, the user must specify, when defining an aspect, which aspect dominates the other ones.

Defining an aspect

AspectJ is an AOP implementation for the Java language. Its grammar is very similar to Java, an aspect definition looks like a class definition. An aspect specification looks as follows:

Table 3-1 . Definition of an aspect.

```
 (1) aspect aspectName [of aspectInstantiation]
 (2) {
 (3)   pointcut pointcutName: pointcutDefinition;     } pointcut definitions
 (4)   pointcut pointcutName2:  pointcutDefinition;
 (5)
 (6)   before/after/instead : pointcutExpression
 (7)   {
 (8)      // code for advice (to be added where corresponds)   } an advice
 (9)   }
(10) }
```

Table 3-1 shows the content of an aspect. The most interesting parts of an aspect are:

- Pointcut definitions can be defined as separate expressions (lines 3 and 4).
- There are operations over *pointcuts*, specified in the example by *pointcutExpression* (line 6). It means that pointcuts can be mixed using some operators.
- When defining an advice (line 6), the *before*, *after* or *instead* have to be specified.
- The instantiation mechanism is defined explicitly by *aspectInstantiation* (line 1).

Super-imposition concepts in AspectJ

Making a parallel between the super-imposition concepts (shown in a previous sub-section) and the AspectJ ones, we can notice the following:

Table 3-2. A Parallel between Super-imposition concepts and AspectJ.

SI Concept	Corresponds to
name of Super-imposition specification	aspect
who	aspect and pointcut definitions
what	advice implementations and introductions
where	aspect & pointcut definitions
when	advice definitions (*before*, *after*, *around*)

Table 3-2 shows the comparison between super-imposition concepts with AspectJ definitions.

3.2.2 HyperJ

HyperJ is a Java implementation for *Multi-Dimensional Separation of Concerns* [HYPJ-01]. Multiple concerns can be defined and applied over the original system and each other at the same super-imposition level.

Super-imposition and concerns are modeled and specified by defining *hyperspaces*, *modules*, *hyperslyces* and *hypermodules*.

A ***hyperspace*** contains specifications for dimensions and concerns of importance. A Concern is a set of ***modules***. In Java modules represent classes, methods, instance variables, etc. Concerns are grouped into a single definition module so called Concern-Matrix (a ***hyperspace***).

A Concern-Matrix specifies relations among concerns, identifying integration, encapsulation and identification of each concern. Each axis represents a dimension of concern and each point on an axis a concern in that dimension.

The Concern-Matrix is only a declarative specification. There may be modules that are not specified or not completely specified. A ***hyperslice*** is a set of concerns that is declaretively complete, it declares everything to which it refers. This is important because internally an *hyperslice* defines the coupling among concerns and the way to solve overlapping problems (crossing problems).

An ***Hypermodule*** is a set of hyperslices, that indicates the integration mechanism among each hyperslice, and how they are interrelated.

Some Features in brief

In HyperJ concerns are explicitly specified, *hyperslices* contains declarations about concerns and the modules that they affect, also including codification for those concerns. Crossing problems are specified into *hypermodules* and *hyperslices*.

Concerns are woven with system code and are statically applied at compile-time. They are declaretively applied instead of systematically applied, because they are explicitly specified indicating the affected modules.

How are crossing problems worked out?

The specification and definition of *hypermodules* and *hyperslices* includes also the way for working out crossing problems (overlapping of concerns).

HyperJ provides a set of specifications for modeling and specifying overlapping problems. And each specification depends on the level of the overlapping. Levels can be Classes, Interfaces, Methods, and variables. Some of these specifications are:

1. When overlap appears, it is possible to specify if the involved modules will be mixed, overridden or not allowed.
2. Equate operations, it means that modules can be modeled even if they are not related each other.
3. When methods are merged, an order can be given to specify the moment, with respect to the original method, in which the super-imposed method will be executed.

The set of possibilities is larger than the list presented above. Nevertheless, the presented set was intended to show that HyperJ provides the facilities for specifying them.

Super-imposition concepts in HyperJ

Table 3-3 shows a parallel between super-imposition concepts and HyperJ.

Table 3-3. A Parallel between Super-imposition concepts and HyperJ.

SI Concept	Corresponds to
name of Super-imposition specification	hyperspace
who	hyperspace – concern-matrix
what	hypermodule
where	concern-matrix
when	hyperslice – hypermodule

3.3 Composition Filters

The Composition Filters model can also be considered as an AOP model. Because concerns can also be modeled by defining and applying CF-interfaces to classes, allowing in this way the modeling of concerns such as security, logging, persistence, etc.

The drawback of the current model is the fact that a CF-interface can only be applied to one module (class) each time, this case is shown on the left side of Figure 3-3. Currently there is no possibility for modeling super-imposition of a CF-interface for several modules.

Figure 3-6 shows a schema for identifying Composition Filters concepts and super-imposition.

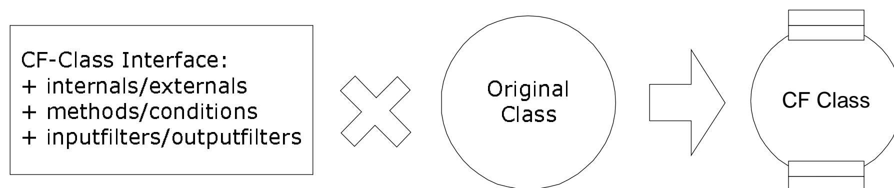

Figure 3-6. Identifying concepts in Composition Filters.

As shown in Figure 3-6, the specification for super-imposition is defined by the CF-interface and is attached to the implementation (represented by the original class). The result is the composed version of the original class.

Solving crossing problems

How crossing problems are worked out in Composition-Filters is a question of a well definition of filters and interfaces and how filters defines the correct semantic of the modeled problem.

Super-imposition concepts in Composition-Filters

Table 3-4 shows the relation between super-imposition and CF concepts.

Table 3-4. A Parallel between Super-imposition concepts and Composition-Filters.

SI Concept	Corresponds to
name of specification	CF class-interface
who	class interface (only one source module)
what	methods, conditions & filters
where	filters definitions (type of filters)
when	inputfilters and outputfilters

3.4 ComposeJ

ComposeJ is a Composition-Filters implementation for the Java language [WICH-99]. Its design and implementation are still evolving as part of the Composition-Filters project.

It is a compiled approach for CF in the Java language, mainly because of performance reasons. There were different versions of implementations for Composition-Filters on reflective languages such as Smalltalk and Sina. When applying reflection over messages, some performance penalties appear when creating the corresponding objects. Those penalties can be avoided when using a compiled approach.

3.4.1 Characteristics

ComposeJ is more than a compiler, it also provides:

- A Graphical User Interface
- An integrated editor for creating and modifying CF projects.

As part of its development, there is the intention for adding:

- Support for other IDE tools.
- A debugging mechanism.

3.4.2 Definitions

To define a Composition-Filters class in ComposeJ, there must be two specifications:

- A class-interface where the user defines: internals, externals, methods, conditions, inputfilters and outputfilters (a ".cf" file).
- An implementation that realizes the class-interface, a ".jwf" file which is a ".java" file without filters.

Current implemented Filters:

In the current ComposeJ implementation, the following filters are implemented:

1. Dispatch Filter
2. Error Filter

Other new filter types, such as Meta Filter and Wait Filter, are still under development.

3.4.3 Defining a CF-interface

Table 3-5 shows a sample code for a Composition-Filters interface.

Table 3-5 . A sample of a CF file specification

```
(1) class USVMail interface
(2) internals
(3)   objectdeclaration
(4)      MailMessage mail = new MailMessage ();
(5) conditions
(6)   private boolean postmanView ();
(7)   private boolean personalView ();
(8) inputfilters
(9)   err : Error = {personalView() =>mail.getContent, true
```

```
                                                                       ~>mail.getContent};
(10)   dis: Dispatch = { mail.* };
(11) end;
```

As shown in Table 3-5, the *internal*, *external*, *method* and *condition* sections (lines 3, 4, 6 and 7) have a Java like-grammar. Mainly, because ComposeJ is an implementation for the Java language, and these specifications are language dependent (object and method definitions). The rest is a specific syntax that is language independent.

3.5 Super-Imposition and the example problem

3.5.1 The previous model

The following figure was already presented in Chapter 1. It corresponds to the example problem and represents the application of several CF-interfaces over some classes in the system.

Figure 3-7 shows, in a gray color, the affected objects (*WorkflowEngine*, *RequestHandler*, *RequestDispatcher*, *OutputHandler*, *MedicalCheck* and *Payment*), and black-colored rectangles represent applied CF-interfaces (for using and administering a workflow).

The problem is that each one of these interfaces was "manually" defined and applied to each affected class.

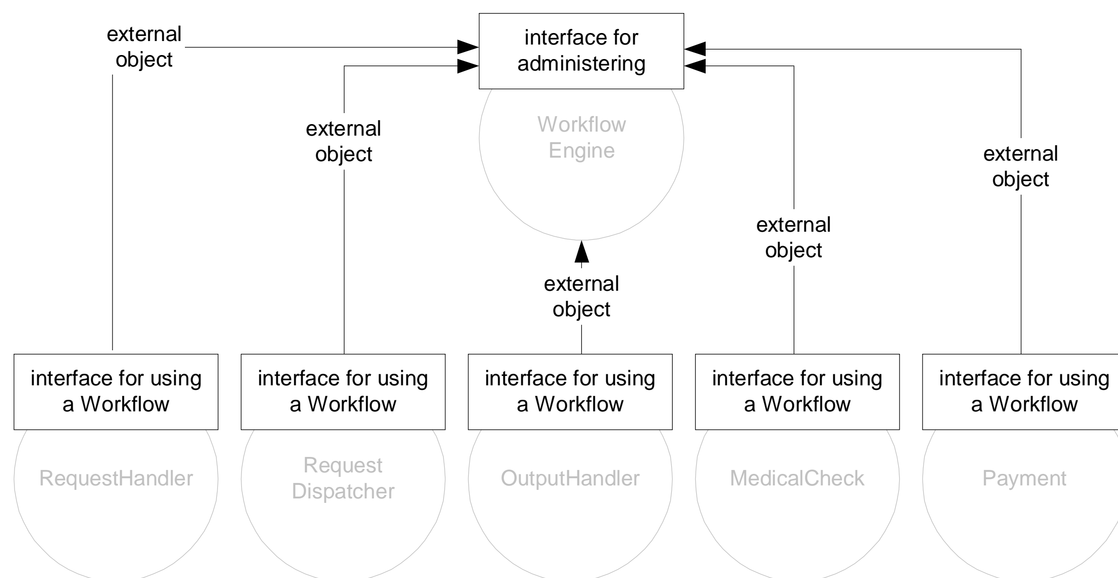

Figure 3-7. The application of CF-interfaces over several objects.

From Figure 3-7 we can observe that the *interface for using a Workflow* is the same for *RequestHandler*, *RequestDispatcher*, *OutputHandler*, *MedicalCheck* and Payment. So, there should be a systematic way to apply that interface over all those objects, without having to specify and apply it "manually" for each class into the system.

3.5.2 Applying Super-imposition

Super-imposition

Taking into account Figure 3-7, we can observe the following super-imposition concepts:

For the *WorkflowEngine* class:

- **who**: *WorkflowEngine*.
- **what**: CF-interface for administering the Workflow.
- **where**: Over the class definition (the class interface changes).
- **when**: When a document is received from some task, it has to be administered depending on the sender and the receiver of the document.

For the rest classes:

- **who**: *RequestHandler*, *RequestDispatcher*, *OutputHandler*, *MedicalCheck* and *Payment*.
- **what**: CF-interface for using the WorkflowManager
- **where**: Over the class definitions (the class interfaces change).
- **when**: When sending a document, a workflow has to be applied over it.

Figure 3-8 shows a schema for the applied super-imposition on the modeled system.

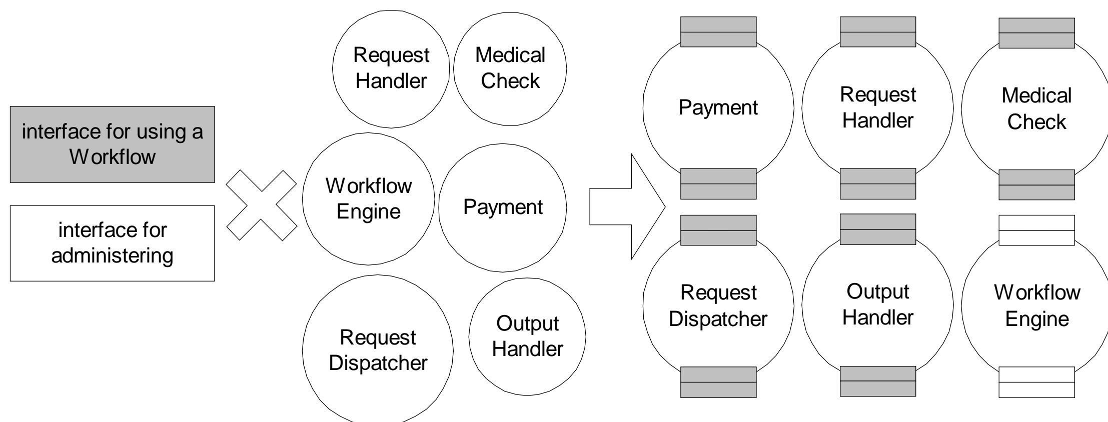

Figure 3-8. Identifying super-imposition concepts.

As shown in Figure 3-8, the CF-interface for using a workflow, drawn as a gray rectangle, is applied over *RequestHandler*, *RequestDispatcher*, *OutputHandler*, *MedicalCheck* and *Payment* (represented by circles). The CF-interface for administering the workflow, drawn as a white rectangle, is applied over WorkflowEngine. The result of those super-impositions is shown on the right side by CF-figures.

Modeling concerns

The main ***concern*** when applying the *workflow interface* to all those objects is the Workflow itself. Internally the interface refers to, as an external object, a *WorkflowManager* instance that allows and empowers the workflow modeling.

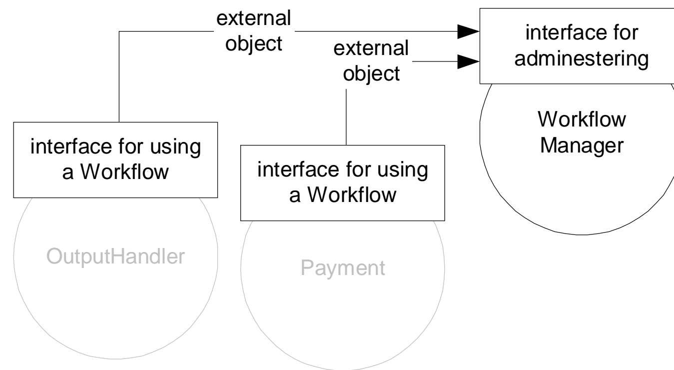

Figure 3-9. The Workflow Manager as an external object.

Figure 3-9 shows the Workflow Manager (on the right side) as an external object of the *interface for using a Workflow* (on the left side, represented by rectangles). In this case, for keeping the model simple, only *OutputHandler* and *Payment* are shown, in gray color, as affected classes.

As shown, the ***concern*** will depend exclusively on how interfaces are applied and how they are codified.

Super-imposing code

A CF-interface is only a specification; it needs an implementation that realizes method and condition definitions. Therefore, it is also necessary to super-impose code to complement methods and conditions.

Figure 3-10 shows the super-imposition of methods and conditions, represented as dashed rectangles, for realizing already super-imposed interfaces.

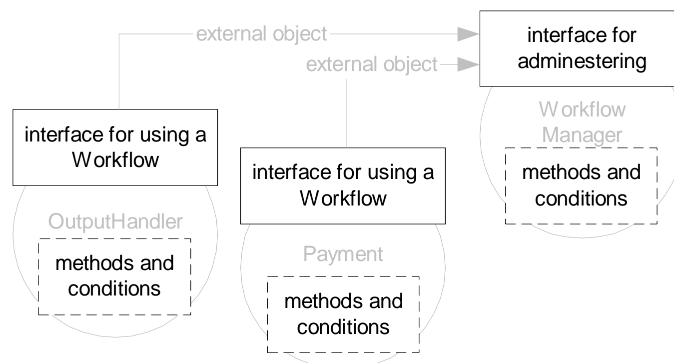

Figure 3-10. Super-imposition of methods and contitions.

In this way, as shown in Figure 3-10, the system is super-imposed systematically by interfaces and implementations.

3.5.3 Progressive super-imposition

To support the incorporation of new concerns, such as logging and persistence, super-imposition of new CF-interfaces can be applied progressively. Finally, the resulting class has a combination of all applied interfaces. That is possible thanks to the modular nature of filters.

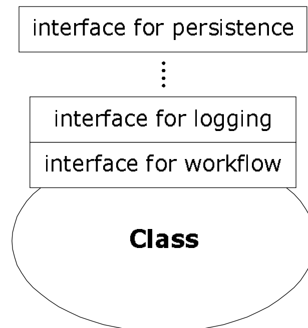

Figure 3-11. Super-imposing several interfaces over the same class.

Figure 3-11 shows the progressive super-imposition of interfaces over an object. Persistence is applied over Logging, Logging over Workflow and Workflow over the original class implementation.

Internally interfaces have defined filters, when super-imposing new interfaces over old applied ones, filters are added sequentially on top of old ones. Figure 3-12 shows this feature:

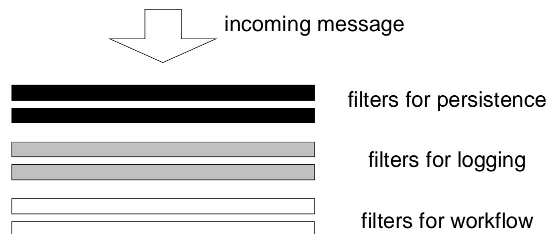

Figure 3-12. Filters acting as sequential modules.

When a message arrives to an object, it is inspected and modified by filters sequentially. Therefore, when combining interfaces it is possible to make only one version of the filters specification. When super-imposing already defined modules, such as methods, conditions, internals, externals, and so on, is a question of solving *crossing* problems and the solution will be explained in a following chapter.

3.6 Summary

There are some aspects that are interesting to highlight from this chapter:

- The super-imposition and crosscutting concepts. Super-imposition is used for applying modeled crosscutting concerns over a defined system.
- Super-imposition concepts *what*, *where*, *when* and *who* are important and interesting to identify when studying a system that needs super-imposition modeling.
- Problems with crossing code. When several concerns are applied, overlapping is possible and there must be ways to avoid these cases.
- How AspectJ, HyperJ and Composition Filters are used for modeling concerns and how they use super-imposition technique.
- How those languages provide ways for solving problems with crossing code. This will be revisited later on in Chapter 4 : section 4.4 .

Chapter 4 : Designing the language

4.1 The language

This chapter contemplates the design of a super-imposition language for the Composition-Filters model. It is intended to allow the modeling, specification and application of interfaces and implementation code. Also, the design includes interesting aspects such as the language grammar and how to solve overlapping/crossing common problems.

Since the language will be based on the Composition-Filters model, the design is intended to be language independent. Because, in this way, it can be applied to several programming languages, enables possible heterogeneity among them and it can be seen as an extension for those languages. UML and OCL grammars have been borrowed and added to the language specification for being used when defining language independent code.

Therefore, for this chapter, a neutral approach has been taken, and just for naming purposes, the name for the super-imposition language will be "**ConcernX**".

4.2 Presenting the language

4.2.1 The Concern declarations

A schematic scenario for super-imposition in the ConcernX language is shown in Figure 4-1. It contains the super-imposition specification, the system and the expected result after super-imposition is applied.

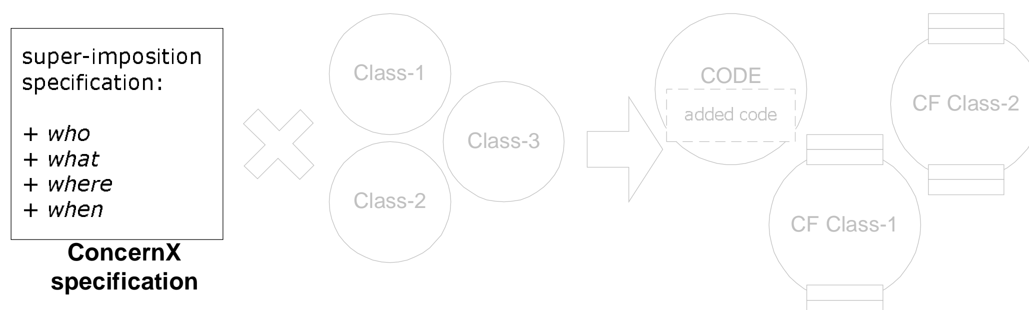

Figure 4-1. A possible scenario for super-imposition of Composition Filters declarations.

As shown in Figure 4-1, on the left side, represented as a rectangle, there is the super-imposition specification, containing *who*, *what*, *where* and *when*. In the middle, in gray color, are shown source files (classes) and on the right side the result of the super-imposition that may result in several CF-classes.

The Concern-specification file will contain:

- Composition-Filters code (interfaces + implementation code).
- Super-imposition itself, which specifies "what", "who", "when" and "where".

4.2.2 Super-imposition of modules

As said before, for obtaining Composition-Filters classes, interfaces and implementation code have to be super-imposed. Figure 4-2 shows the case when CF-specifications are defined to be super-imposed as a whole (interface + implementation).

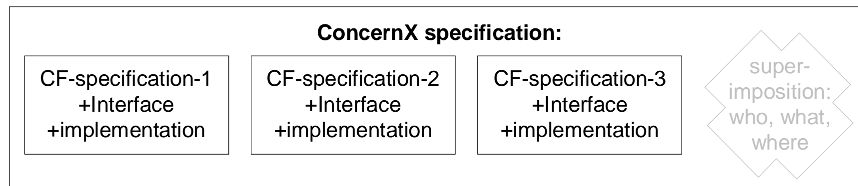

Figure 4-2. Superimposing only Composition Filters specifications.

In Figure 4-2 CF-specifications are represented by rectangles. Each specification contains both the interface and the implementation code. The implementation code is written in some specific language for realizing interfaces. All the implementation blocks are written in the same language. The super-imposition specification is represented by a cross symbol (in gray color).

The problem when super-imposing CF-specifications as a whole, is that interfaces and implementation are bundled (there are only three possibilities to super-impose CF-specifications). So, it is not possible to reuse separately either interface or implementation. Therefore, for reusing some of them, it would be necessary to duplicate the whole CF-specification and make the corresponding changes over the "reused" code.

A possible solution to work out this problem is shown in Figure 4-3. Interfaces and implementation are defined separately each other.

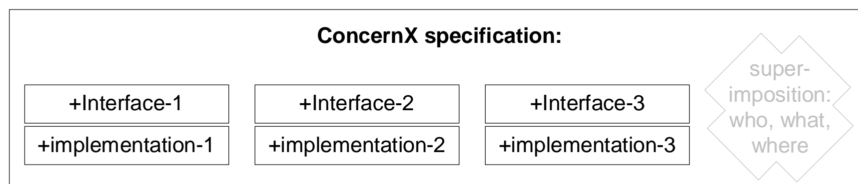

Figure 4-3. Separating CF-interfaces definitions from their implementation.

The problem with this approach is that implementations always realize interfaces, so there can be duplication of methods and conditions inside each implementation block (think about a method "m" duplicated into implementation1 and implementation2).

A third approach is shown in Figure 4-4. Implementation blocks are mixed into only one whole block. In this way, duplication of methods and conditions is avoided, and also there is the possibility to reuse code by choosing the appropriated implementation in each super-imposition phase.

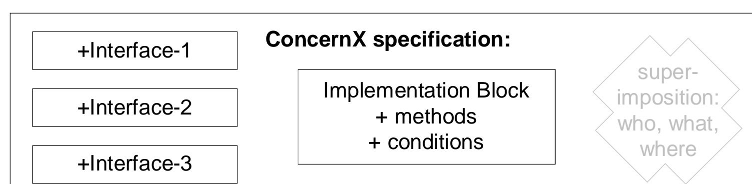

Figure 4-4. Specifying the whole set of implementations that can be used for a super-imposition.

The implementation block represents language-specific code. Therefore, it can be defined as a separate module. In this way language-dependent code is defined separately from the language-independent part.

As an example, for a Java approach, the implementation block will be represented as a Java-Class containing conditions and methods. Also, there is no need for this class to be executable, because it will be only used as a repository for taking method and condition implementations. Nevertheless, it also could be super-imposed and used as every class into the application.

CF-interfaces are not combined into one specification, as we did with the implementation block. We want to keep them in that way, because they well defined entities. The completeness and correctness of each interface can be checked before they are super-imposed (all declared entities are also available in the implementation).

Nevertheless, another approach can be taken. It will be attractive to define multiple implementation blocks in the same ConcernX specification, in such a way, that the same module can be used for defining implementation blocks for different programming languages, for instance:

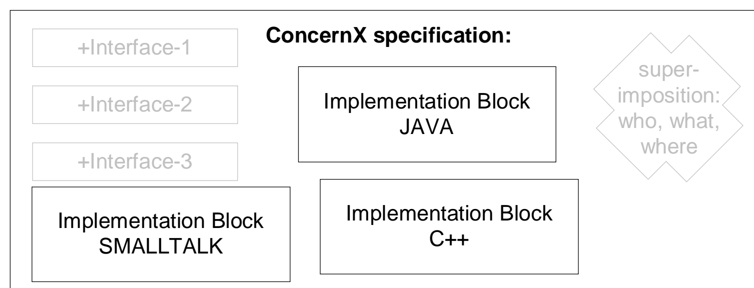

Figure 4-5. Multiple implementation blocks for implementing methods and conditions in different languages.

Figure 4-5 shows the case when defining implementation blocks in different languages, so the ConcernX specification contains everything related with the super-imposition specification.

Nevertheless, this design was not taken into account when implementing and writing implementing this thesis work, because this feature was included after finishing the design and implementation of the ConcernX language.

4.2.3 How do we specify the super-imposition?

For specifying the super-imposition mechanism, the following information must be provided:

- "who" / "where"[1]: which classes are going to be super-imposed.
- "what": ***which*** CF-interfaces and ***which*** implementation will be super-imposed over the desired classes.

[1] Since we are working with Composition Filters, the only way to specify "where" is by applying the super-imposition to the classes specified by "who".

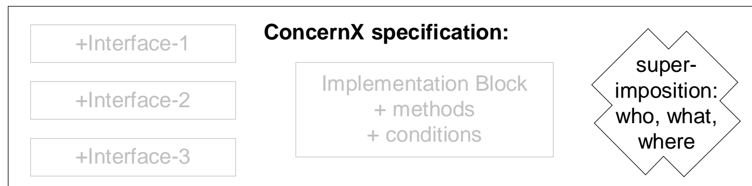

Figure 4-6. Super-imposition specifications.

Figure 4-6 shows, represented as a cross symbol, the specification for super-imposition. In this way, this specification is included as another block into the ConcernX-specification. Rectangles in gray color represent the code that will be used for the super-imposition phase.

Specifying "who" & "where"

Composition-Filters interfaces are applied to objects. In that way, those objects start behaving differently depending on the applied interface. By selecting determined objects into the system, we can apply different concerns over objects of the same type. In this way the approach has more flexibility than just super-imposing over a type in such a way that all objects pertaining to that type are necessarily super-imposed.

So, in this context, the concept "***who***" is represented by objects. A consequence of this, is the fact that the Concern-language will be ***an object-based approach*** for super-imposition. On the contrary to AspectJ, for instance, which is a ***class-based***[2] ***approach***.

A Class-based approach, like AspectJ, is based on modifying completely the class definition at compile time. Aspects are switched-on when compiling the resulting code. So, every instance of modified classes has the same static behavior.

In order to specify those objects, a grammar has to be designed, that allows the definition of sets-of-objects.

Nevertheless, in the implementation of the language, class definitions (the types of those objects) have to be modified in order to incorporate the super-imposition of CF-specifications over them.

Specifying "which" code has to be super-imposed

Interfaces and the implementation-block offer a large set of specifications[3] to choose from. The user must be able to choose ***which*** interfaces and ***which*** code, in particular, from the whole set, are going to be super-imposed over the selected objects. In this context, "**which**" specifies the selection of implementation and interfaces.

Figure 4-7 shows the case when selecting interfaces and methods/conditions from the Concern-specification.

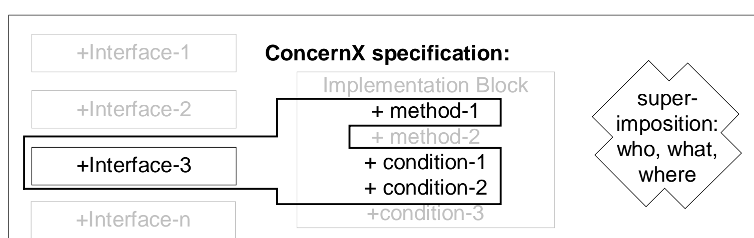

Figure 4-7. Selecting specific implementations from a concern.

[2] In AspectJ, the concept "***who***" is specified by defining the *aspect* or its *pointcuts* over classes.
[3] Interfaces and the implementation block represent the concept "**what**" in super-imposition.

As shown in Figure 4-7, the desired blocks, interfaces and methods & conditions, are chosen separately from the Concern-specification and will be applied over a set of objects.

In this way, for applying each block separately, the language must provide three different sub-sections for each super-imposition.

4.3 The Concern grammar

4.3.1 Existing Implementations of CF

Today, there are already existing implementations for the Composition-Filters model, some of them are:

- **Sina/st**: A bottom-up implementation for CF [SINA-95].
- **C++/CF**: Oriented to work with the C++ language [GLAN-95].
- **ComposeJ**: Oriented to work with the Java language [WICH-99].

Each one of those implementations has adapted the CF-grammar, based on previous implementations, to the language where the CF-model is implemented.

This is mainly because the Composition-Filters model, which is language-independent, has to be applied over a concrete implementation that *basically* is language-dependent. Therefore, objects, methods and conditions specifications/implementations are defined depending on the chosen language.

Moreover, some extra concepts of each language are not available in others. For instance, concepts like *constructors*, *virtual methods*, *destructors, package declarations* and so on depend on each language specification.

This problem appears when a user wants to reuse or migrate specifications among CF-implementations.

Nevertheless, for the ConcernX language grammar, previous language specifications are going to be revisited and reused. The ***CF-specification*** and ***Filters*** grammars are the most interesting to be analyzed, because they represent language independent code.

4.3.2 UML-like grammar

UML stands for Unified Modeling Language [UML-01]. It was introduced a few years ago and it has been becoming a popular language for designing, specifying and modeling object-oriented applications. As a modeling language, UML is intended to be language independent.

One of the benefits of using UML, is the fact that the used OOP language does not matter, an UML CASE-tool can transform the specification model into it.

The UML grammar has been borrowed for the definition of conflictive language-dependent specifications, such as object, method & condition declarations. Another interesting aspect for using UML as a base for specifying those sections is the fact that most users know or have heard about UML. Therefore, it is preferable to make a tool that can be understood by many users.

4.4 The sections of a concern

As shown in previous sections, a Concern-specification contains three main blocks for specifying the following:

1. Interfaces.

2. Implementation code (methods & conditions).
3. Super-imposition specification.

Interfaces and implementation code represent the code to be super-imposed and the super-imposition specification represents the definition of how that code has to be super-imposed and to which objects have to be super-imposed.

4.4.1 The filterinterfaces section

Each *filterinterface* represents a Composition-Filters interface. It contains definitions for internals, externals, methods, conditions, inputfilters and outputfilters. More than one *filterinterface* can be applied to a class in the same phase. The order in which those interfaces are super-imposed represent the behavior of the resulting composed class.

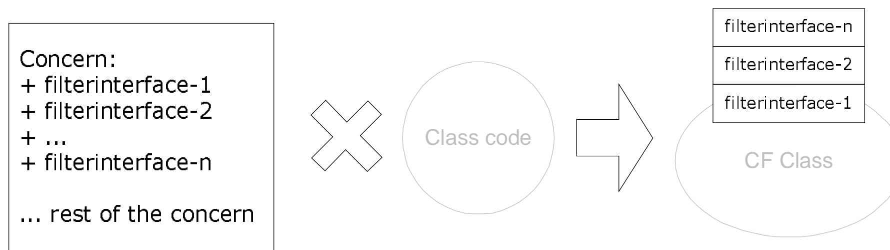

Figure 4-8. Super-imposing several filterinterfaces over a class definition.

Figure 4-9 shows the super-imposition of multiple filterinterfaces over a class, the result is the composed version of the original class.

The following aspects are important when super-imposing multiple filterinterfaces:

1. Internals definitions, externals declarations, methods declarations and conditions declarations can be combined into a single specification without taking care of the order in which the super-imposition took place.
2. Filter-sets may have different behavior depending on the order in which they were super-imposed.

Both points, one and two, correspond to solve *crossing* problems. This will be explained in section 4.5 .

4.4.2 The implementation section

The implementation block contains method and condition implementations that realize a *filterinterface*. Figure 4-9 shows the case when super-imposing methods and conditions over a class.

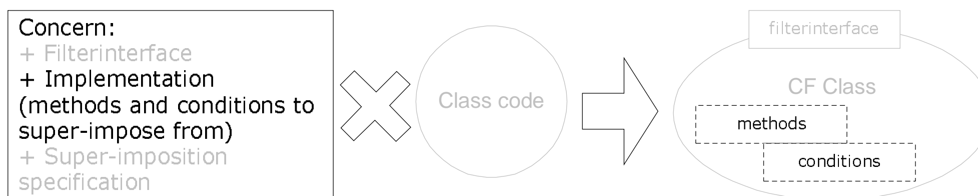

Figure 4-9. Super-imposition of methods and conditions.

Figure 4-9 shows, in dashed-rectangles, the super-imposition of methods and conditions code, taken from the implementation block, over a class definition.

When super-imposing methods and conditions code, there can be *crossing* problems. Mainly because there can be already defined methods with the same signature inside the original class code.

Again, the way for solving these problems will be presented in section 4.5 .

4.4.3 The Super-Imposition section

In this section, the following specifications are defined:

1. Which objects into the system will be part of a modification. Definining in this way sets of objects.
2. Which filterinterfaces will be super-imposed to a set of objects.
3. Which methods and conditions will be super-imposed to a set of objects.

So, in this section, there will be three main sub-sections inside the super-imposition block:

1. *Selector* definitions: for selecting sets of objects.
2. *Superimposition* of Filter-Interfaces: for selecting filterinterfaces to be applied over a set of objects.
3. *Superimposition* of methods and conditions: for selecting methods and conditions to be applied over a set of objects.

selectors

Selector definitions are intended to specify sets of objects. The following information about objects can be obtained from a system:

1. Object name and Type,
2. The file/module names where the object is defined, and
3. The file/module name corresponding to the Type definition.

With all this information, a user can specify which objects into the system he/she wants to take care of.

OCL-like grammar

OCL stands for Object Constraint Language [OCL-01] and is part of the UML specification. It is intended to define constraints when modeling Object-Oriented applications. These constraints are basically invariants, pre-conditions and post-conditions.

OCL, as part of UML, is also intended to be language independent. OCL-expressions mainly return boolean-values and allow definitions of sets and objects. Operations over OCL-expressions are also allowed.

It is possible to use those definitions and make the OCL specification able to return sets of objects. So, with this modification, OCL can be used for defining the ***selectors*** grammar.

The application of a selector of concern over a system is done at compile time. Selectors are based on the selection of sets of objects. So, there must be an initial set that contains all the current objects in the system. In this way, new sets of objects can be defined by applying sets-operations over the system set.

Let us define this set:

system: A set containing all objects inside the system.

The following set of examples is intended to show the capabilities of OCL for defining set of objects. However, before present the examples let us explain some interesting OCL operations:

- **select**: is a set operation that allows selecting objects with some criteria and returns another set of objects.
- **exists**: is a set operation that returns *True* or *False* depending on if the given parameter (an OCL Expression) evaluates to *True* for at least one element in the applied set.
- **includes**: is a set operation that returns True or False depending on the given parameter (an object) is part of the applied set.
- **oclIsKindOf**: is an object operation that returns *True* or *False* depending on if the object is an instance or subtype of the given parameter (a Type).
- **oclIsTypeOf:** is an object operation that returns *True* or *False* depending on if the object is an instance of the given parameter (a Type).

A more extensive explanation about the OCL operations can be found in Appendix B.

Examples:

1. Eventually, all objects into the system.

```
z := system;
```

2. Only those objects that are not an instance of classes {A, B, C, D, E} or their subtypes.

```
a := Set{A, B, C, D, E};
b := system->select( v | not a->exists( w | v.oclIsKindOf(w) ) );
```

3. Only those objects with the type included in {A, B, C, D, E}

```
a := Set{A, B, C, D, E}
b := system->select( v | a->exists( w | v.oclIsTypeOf(w) ) )
```

4. Only those objects with type A or B, which are not an instance of C (C can be a subtype of A or B)

```
a := Set{A, B}
c := system->select( v | a->exists( w | v.oclIsKindOf(w) )
                                        and not(v.oclIsTypeOf(C)) );
```

5. Only those objects with name is not contained in {o1, o2, o3, o4, o5}

```
a := Set{o1, o2, o3, o4, o5}
d := system->select( v | not a->includes(v) );
```

6. Only those objects that their type is A, and their names are not included in {o1, o2}

```
e := system->select( v: A | not Set{o1, o2}->includes(v) );
```

7. Only those objects with type B, and which are not included in "e"

```
f := system->select( v | v.oclIsKindOf(B) and not e->includes(v) );
```

As shown in the examples, the modified version of OCL can be used for selecting objects. Because both objects and sets can be specified and also operations can be applied over them.

filterinterfaces

For super-imposing *filterinterfaces* the user must specify:

1. The selector, a set of objects to super-impose.
2. A set of *filterinterfaces* to be super-imposed over the objects defined by the given selector. Those *filterinterfaces* can be taken from the current concern definitions or from other ones.

Filterinterfaces definitions can also be taken from another concern-specification, allowing in this way the reuse of code.

methods/conditions

For super-imposing methods & conditions, the user must specify:

1. The selector, a set of objects to be super-imposed.
2. A set of methods & conditions to be super-imposed over the objects defined by the given selector. Those methods & conditions can be taken from the current concern definitions or from other ones.

Methods & conditions implementations can also be taken from another concern-specification.

Superimposing filterinterfaces, methods & conditions

Table 5-5 shows an example of a super-imposition of several *filterinterfaces* (the example is also valid when super-imposing *methods* and *conditions*).

Table 4-1. Super-imposing definitions over a selector.

```
selector1 <- { filterinterface1, filterinterface2, filterinterface3, ... }
selector2 <- { method1, method2, ... }
selector3 <- { condition1, condition2, condition3, ... }
```

As shown in Table 4-1, on the left side comes the selector's name, on the right side comes the set of definitions that will be super-imposed at the objects defined by the specified selector, such as *filterinterfaces*, *methods* or *conditions*.

Definitions, such as *filterinterfaces*, *methods* and *conditions*, can be taken from other concerns. That is also valid for a selector, so in this way a concern may super-impose definitions over selectors defined in other concerns.

The symbol "<-" represents the application of a super-imposition. It is one of three different operators; each one intended to solve common *crossing* problems (explained before in sections 4.4.1 and 4.4.2). We will explain these using an example.

Suppose that **selector1** and **selector2** are *selector* definitions. They share some objects, in particular, an object of type **Type**. When super-imposing different *filterinterfaces*, *methods* and *conditions* over **selector1** and **selector2**, the result of the super-imposition may be different depending on the specified application-symbol.

Table 4-2 shows the different symbols for the application of super-imposition over **selector1** and **selector2**.

Table 4-2. Application-symbols for specifying the result of the super-imposition.

<-	The result of the super-imposition over **Type** is the result of combining the applied super-impositions over **selector1** and **selector2**. Newer super-impositions (changes) are inserted before older ones.
<x	The result of the super-imposition over **Type** generates different versions for each applied concern over **selector1** and **selector2** (not yet implemented). When there are more than one applied concern, **concern1** and **concern2**, there will be two different versions for **Type**, one for **concern1** and another for **concern2**. When there is only one applied concern, the result is the same than applying "<-" over **selector1** and **selector2**. In brief this operator generates different **Type** versions as concerns are applied over the same object.
<*	The result of the super-imposition over **Type** generates different versions for each applied concern and each applied selector (not yet implemented). When there are more than one applied concern, **concern1** & **concern2**, and more than one selector, **selector1** & **selector2**, there will be different versions for **Type**. There are some particular cases, that result similar to than applying "<X" or "<-". In brief this operator generates different **Type** versions as concerns are applied over **selectors** that contains the same object.

The way for modeling and designing each application symbol will be presented in Chapter 6 when showing the implementation of ConcernJ.

Applying this grammar to an example

Taking into account the example problem presented in Chapter 1 :section 1.2 , and the modeling presented in 3.5.1 , the following table shows some code for the application of "interface for a workflow" over *Requesthandler*, *RequestDispatcher*, *OutputHandler*, *MedicalCheck* and *Payment*, and "interface for administering" over *WorkflowEngine*.

Table 4-3. An example of a concern specification.

```
concern WorkFlowEngine begin   // introduces centralized workflow control

   filterinterface useWorkFlowEngine begin // this part declares the
   externals                               // crosscutting code
      wfEngine : WorkFlowEngine; // *declares* a shared instance of this
                                 // concern
   inputfilters
      redirect : Meta = { wfEngine.selectTask(target=forwardDocument) };
   end filterinterface useWorkFlowEngine;

   filterinterface engine begin    //defines the interface of the workflow
   methods                         // engine object
      selectTask(Message);
      setWorkFlow(WorkFlow);
   inputfilters
      disp : Dispatch = { inner.* }; // accept all methods implemented by
                                     // myself
   end filterinterface engine;

   superimposition begin
   selectors
```

```
      classes = Set{ RequestHandler, RequestDispatcher, OutputHandler,
                                                  MedicalCheck, Payment};
      allTasks = system->select(v| classes->exists(w| v.oclIsTypeOf(w) ) );
   filterinterfaces
      self <- self::engine;
      allTasks <- self::useWorkFlowEngine;
   end superimposition;

   implementation in "WorkFlowEngine.java";

end concern WorkFlowEngine;
```

The ***useWorkFlowEngine*** filterinterface represents the *interface for using a workflow* and defines a filter of type Meta, which intercepts *forwardDocument* messages and sends them in reified form, as the argument of message *selectTask*, to wfEngine.

The ***engine*** filterinterface, represents the *interface for administering a workflow*, and the implementation part together implement the workflow engine. Next to some methods for accessing and manipulating the workflow representation (in this case none of the implementation methods were shown), it defines the method *selectTask*. This method determines the next task that should handle the document, modifies the corresponding argument of the message object, and then fires the message so that it continues its original execution–but with an updated argument.

The superimposition clause specifies how the concerns crosscut each other. The superimposition clause starts with a selectors part that specifies a sets of objects. Concern WorkFlowEngine defines a single selector named **allTasks**. This selector is defined by using an OCL grammar and specifies all objects that are instances of the various classes that represent tasks.

The selectors part is followed by a number of sections that can specify which objects, conditions, methods respectively filterinterfaces are superimposed upon locations as designated by one or more selectors. In this example the filterinterface ***engine***, is superimposed upon self: this means that instances of *WorkFlowEngine* will include an instance of the ***engine*** filterinterface. In addition, the *useWorkFlowEngine* filterinterface (which can be found in the same concern, as designated by "self:: ") is superimposed upon all the instances defined by the *allTasks* selector.

4.5 Working out crossing problems

When applying super-imposition, *crossing* problems may appear when:

1. Super-imposing code:

   a) Super-imposing *filterinterfaces* with duplicated definitions (filters, internal, externals, methods and conditions).
   b) Super-imposing methods and conditions over already existing ones.

The difference between (a) and (b) is the fact when applying (a) old declarations are wipes out, therefore they connot be used after a new super-imposition has been applied. When applying (b) newer super-impositions are placed on top of old ones, and there should be a way for invoking and executing old code.

Some possible solutions are:

i. Newer definitions replace old ones, keeping the order of super-impositions. This is the easier approach from the tool's point of view.
ii. Newer definitions override old ones. This is more complicated to understand and implement, and it depends on the dynamic capabilities of Composition-Filters, for changing and invoking, at runtime, overridden definitions such as filters, internals, externals, and so on.

The second point is not part of this thesis work. So, the taken approach for solving problem (1) will be the first one (i).

2. An object is part of different selectors at the same time.

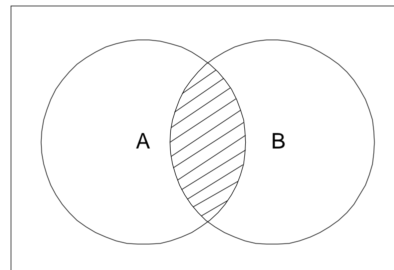

Figure 4-10. Selectors are sets.

Figure 4-10 shows the intersection between selectors A and B. Both A and B could be defined in different concerns.

An object can be part of A and B at the same time. Selectors can be super-imposed differently, by the application of interfaces, methods and conditions.

What should result from the super-imposition of A and B, when different modules were super-imposed at the same phase over that object?

a) The object will contain a combination of all the super-impositions applied over it (The same problem as point 1).
b) There will be different versions of the object, depending on the concern that applied the super-imposition. By using this approach, a problem comes out, which version to use from all possibilities?
c) There will be different versions of the object, depending on the selector that it belongs to. By using this approach, the same problem comes out, which version to use from all possibilities?

For solving problems (a) and (b), there is the possibility for switching dynamically at runtime *filterinterfaces* by invoking certain methods into the super-imposed code. In this way old existing code can be used when necessary. This depends on the dynamic capabilities of filters for applying *filterinterfaces* at runtime depending on the incoming message.

Nevertheless, the user should be able to choose the version he/she wants to use, and the compiler should provide the necessary information for solving those problems when they appear (shown in section 4.4).

Finally, for simplicity reasons, the taken approach for solving problem (2) is (a).

4.6 ConcernJ

ConcernJ represents the implementation for super-imposition of Composition-Filters for the Java language. Thus ConcernJ, as a language, represents just another Composition-Filters implementation. It is intended only as a proof of concept for implementing the language that we have being designing through this Chapter.

The following sections are intended to study step by step the most interesting aspects of the ConcernJ design, taking into account facilities and drawbacks that Java provides.

4.7 Parts of a concern

The following sub-sections are intended to show the most interesting aspects of the ConcernJ grammar. A more extensive specification of it is included in Appendix A.

4.7.1 The concern specification

Table 4-4 shows the grammar of a *concern*. A *concern* is the specification for applying super-imposition in ConcernJ.

Table 4-4 . The grammar of a concern.

```
concern ConcernName begin
   filterinterface (Interface-definition)
   filterinterface (Interface-definition)
   filterinterface (Interface-definition)

   implementation (Implementation-code)

   superimposition (SuperImposition-specification)
end concern ConcernName;
```

As shown in Table 4-4, a *concern* contains *filterinterfaces*, an *implementation block* and a *super-imposition block*.

4.7.2 Filter-Interfaces

Table 4-5 shows the grammar of a *filterinterface*. A *filterinterface* is a Composition-Filters interface specification.

Table 4-5 . The grammar of a filterinterface.

```
filterinterface FilterInterfaceName begin
   internals
      (UML-like grammar for object definitions)
   externals
      (UML-like grammar for object declarations)
   methods
      (UML-like grammar for method declarations)
   conditions
      (UML-like grammar for condition declarations)
   inputfilters
      (Sina/st-like grammar for fitlers specification)
   outputfilters
      (Sina/st-like grammar for filters specification)
end filterinterface FilterInterfaceName;
```

As shown in Table 4-5, a *filterinterface* contain almost the same structure presented in previous CF-implementations. Each block into the *filterinterface* is named depending on the interface specification context (*internals*, *externals*, *methods*, *conditions*, *inputfilters* and *outputfilters*).

The most interesting aspect of a *filterinterface* is the incorporation of the UML grammar for specifying language *conflictive*[4] declarations, such as *internals*, *externals*, *methods* and *conditions*. This makes a *filterinterface* a general language independent specification.

By using UML in these blocks, definitions can be easily translated into specific language code, such as Java, C++, and Smalltalk. In this case, for ConcernJ, UML code will be translated into Java definitions.

[4] Conflictive from the point of view of language-dependent declarations, such as objects, methods & conditions.

Both the *inputfilters* and *outputfilters* sections maintain their grammar, because they are based on Composition-Filters specifications and do not depend on the language.

4.7.3 Implementation

Table 4-6 shows the grammar for the implementation block. This block contains language-dependent code, so it is preferable to define it outside the Concern specification.

Table 4-6. The grammar of the implementation block.

```
implementation in "module-name";
```

As shown in Table 4-6, the implementation block is implemented in another module. In a Java context, the module is represented as a File that contains a class definition.

4.7.4 Super-Imposition

Table 4-7 shows the grammar for the *super-imposition* specification. The *superimposition* block contains definitions for ***selectors*** and superimposition of ***filterinterfaces***, ***methods*** and ***conditions***.

Table 4-7. The grammar of the super-imposition block.

```
superimposition begin
   selectors
      (OCL-like grammar of selector definitions)
   filterinterfaces
      (ConcernJ grammar of super-imposition declarations)
   methods
      (ConcernJ grammar of super-imposition declarations)
   conditions
      (ConcernJ grammar of super-imposition declarations)
end superimposition;
```

As shown in Table 4-7, each sub-block uses a particular grammar. For *selectors*, there is an adapted version of OCL and for *filterinterfaces*, *methods* and *conditions* blocks there is a new grammar, created specially for the ConcernX language.

4.8 Summary

There are some aspects that are interesting to highlight from this chapter::

- The specification for a super-imposition language using Composition Filters can be defined as neutral, because of the nature of the CF model.
- The object-based approach of the Concern-language.
- When designing/creating a super-imposition language is important to know exactly "what", "where", "when" and "who".
- Defining a grammar for super-imposition is an important aspect to take care of.
- UML and OCL grammars are used for defining *conflicting* language-dependent aspects. The UML grammar is used for defining object declarations, methods and conditions definitions. In this way, by using UML as a base, these sections are now defined using a language-independent grammar. The grammar and specification of OCL are used for defining sets of objects.
- The identification of *crossing* problems for the ConcernX language. When they appear, and the facilities that the language provides for taking care of them.

- How the grammar helps to take care of crossing problems. When super-imposing different concerns (or only one) there can be crossing problems. The presented grammar contains specifications for solving these kinds of problems.

Chapter 5 : The Design of the ConcernJ tool

The current chapter is intended to show the most interesting aspects of the ConcernJ implementation. It is based mostly in the design; thus some implementation details have been left out because they are not interesting to be shown in this thesis work.

5.1 Producing code

Computer languages always produce code or information that is interpreted by another module, for instance, the Java compiler produces byte-code that is interpreted by the Java Virtual Machine (JVM), the JVM produces bits and bytes that are interpreted by the computer chip, and so on. In this context, for ConcernJ, there are several possibilities for producing code.

Since ConcernJ is a language intended for applying super-imposition on the Java language, there are already existing tools that do the same for Java. This is the case of AspectJ, HyperJ and ComposeJ. Nevertheless, a bottom-up implementation is also feasible.

The following sections are intended for presenting advantages and drawbacks of each option.

- Producing ComposeJ code
- Producing AspectJ code
- Producing HyperJ code
- A Bottom-up implementation.

5.1.1 ComposeJ Code

Figure 5-1 shows the schema for producing ComposeJ code. So, when the code is produced, the ComposeJ tool will apply the super-imposition over the system generating Java code.

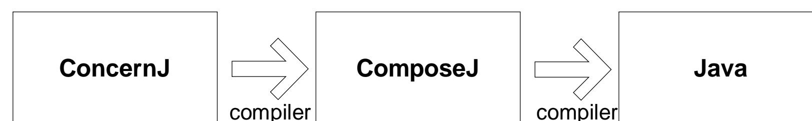

Figure 5-1. Producing ComposeJ code.

As shown in Figure 5-1, rectangles represent language specifications, and arrows represent different compilation phases. In this case, ConcernJ code is translated into ComposeJ specifications, and finally into Java code.

Advantages & Drawbacks

The advantages of using this approach are:

- ✓ It provides a modular approach. So, there can be improvements in each compilation phase, and the system will be still functioning.
- ✓ It is possible to use incremental compilation. This is directly related with the first point.
- ✓ The effort for making this tool could be small. Producing Composition-Filters specifications from ConcernJ and translate them into ComposeJ ones is straightforward. Besides, ComposeJ will produce the final Java code.

The drawbacks of using this approach are:

- × The modularity makes debugging difficult. There will be three different translation phases to debug.
- × Currently, ComposeJ only works with a Graphical User Interface, so it will be necessary to adapt its implementation to support the interaction with ConcernJ.

5.1.2 AspectJ Code

Figure 5-2 shows the schema for producing AspectJ code. So, when the code is produced, the AspectJ tool will apply the super-imposition over the system generating Java code.

Figure 5-2. Producing AspectJ code.

As shown in Figure 5-2, rectangles represent language specifications, and arrows represent different compilation phases. In this case, ConcernJ code is translated into AspectJ specifications, and finally into Java code.

Advantages & Drawbacks

The advantages of using this approach are:

- ✓ AspectJ is currently a good approach to solve some crosscutting problems. There is a big effort in its project team, and a lot of people using it and giving feedback for its improvements and evolutions.
- ✓ It is a modular approach. This is the same case as with ComposeJ.
- ✓ It is possible to use the debugging facilities of AspectJ. AspectJ already provides debugging capabilities.

The drawbacks of using this approach are:

- × Because AspectJ is constantly evolving, it is difficult to make a design based on the current implementation, because the specifications may change from one version to another one.
- × For debugging, the user should know both the Composition Filters and AspectJ ideas.
- × There is another grammar to take care of. There will be several grammars in the tool, such as Java, Composition-Filters, ConcernJ and AspectJ.
- × Currently there is no a public and official AspectJ grammar.

5.1.3 HyperJ Code

Figure 5-3 shows the schema for producing HyperJ code. So, when the code is produced, the HyperJ tool will apply the super-imposition over the system generating Java code.

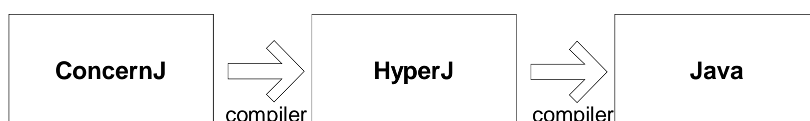

Figure 5-3. Producing HyperJ code.

As shown in Figure 5-3, rectangles represent language specifications, and arrows represent different compilation phases. In this case, ConcernJ code is translated into HyperJ specifications, and finally into Java code.

Advantages & Drawbacks

The advantages of using this approach are:

- ✓ HyperJ is currently a good approach for applying multiple concerns. Different concerns can be defined and applied at the same compilation phase.
- ✓ It is a modular approach. This is the same case as with ComposeJ.

The drawbacks of using this approach are:

- × For debugging, the user should know both the Composition-Filters and the HyperJ models.
- × There is no an official release of HyperJ. HyperJ is still a beta product.
- × There is another grammar to take care of. There will be several grammars into the tool, such as Java, Composition-Filters, ConcernJ and HyperJ.

5.1.4 Java Code/Class files

Figure 5-3 shows the schema for producing directly Java code. ConcernJ will apply both, super-imposition of Composition-Filters and the translations of interfaces onto the Java language.

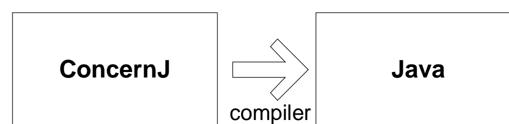

Figure 5-4. Producing directly Java code.

As shown in Figure 5-4, rectangles represent language specifications, and the arrow represents a compilation phase. In this case, ConcernJ code is translated directly into Java code.

Advantages & Drawbacks

The advantages of using this approach are:

- ✓ The integration of ideas, between Super-Imposition and Composition Filters. There is no dependence on third-party software.
- ✓ It is possible to reuse the design from current Composition-Filters implementations. In particular the design and implementation of ComposeJ.
- ✓ For debugging, the tool will only depend on itself.
- ✓ Code optimization and improvements on the same tool. There is no need for depending on third-party software. It can be considered as a drawback as well.

The drawbacks of using this approach are:

- × Code optimization and improvements on the same tool. Because the whole tool has to be redesigned when evolving.
- × It can be considered as an advantage as well.
- × It will need a complete implementation for the Composition-Filters model.

5.1.5 Making a Decision

Considering advantages and drawbacks for each presented option, the most suitable is the first and one, because:

- Translating specifications from ConcernJ to ComposeJ is almost straightforward. In this case, ConcernJ will not implement the application of filters over classes, and thus will use the current facilities of ComposeJ.
- There are not too many grammars to take care of. Only grammars for ConcernJ and Java will be needed.
- ComposeJ may evolve separately from ConcernJ. So, when leaving both the Composition-Filters and the super-imposition modules as separated modules it is possible for each tool to evolve separately.
- Third-party software. There is no direct dependence on other software, such as AspectJ or HyperJ, for applying the super-imposition part.
- As a proof of concept for this thesis work, this option is the easiest one.

5.2 Producing the code

The current ConcernJ design is intended to produce code as follows:

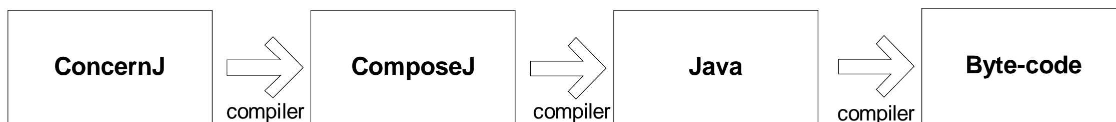

Figure 5-5. ConcernJ will produce ComposeJ code.

Figure 6-1 shows an schema where:

- ConcernJ specifications (concerns + implementation) are translated into ComposeJ code.
- ComposeJ code (CF-interfaces + implementation) finally is translated into Java code.
- Java code is compiled and classes are generated, by using a traditional Java-compiler.

There are several ways to translate ConcernJ specifications into ComposeJ ones:

1. By producing code that can be parsed by ComposeJ.
2. By producing ASTs[1] that can be interpreted by ComposeJ.
3. By modifying ComposeJ to enable it to deal with super-imposition.

Solutions (1) and (2) are very similar from the point of view of compilers, because the only difference between them is that by using solution (2) it is not necessary to parse the code generated from ConcernJ, because it uses the already parsed and generated ASTs (solution (1)).

Solution (3) is not suitable for this thesis work, because it will require modifying a large set of modules into the ComposeJ system. That makes this option not affordable.

Therefore, for this thesis work, as a proof of concept, solution (1) or (2) are acceptable for generating code.

[1] AST stands for Abstract Syntax Tree

5.2.1 Parsers & Visitors

The parsers

Parsers are used for translating source code onto structures that programs can later analyze. They are defined by grammars that contain internally tokens and productions. They represent very specific and particular software, because when something changes into the grammar specification, the whole parser has to change some degree.

Generally, the structures generated by Parsers are Abstract Syntax Trees (AST for short). An AST is a tree composed by nodes, each internal node represents a production into the grammar, and each leaf represents a terminal. Typically, terminals are tokens (this depends directly on the used parser).

Today, it is not necessary and not very common to build a parser from scratch. Compiler-Compilers are software that build parsers by using only a grammar specification. They transform those specifications, tokens and productions, into a parser written in some programming language. That program is able to parse source files and translate them into structures that can be lately analyzed by a program. Commonly, these structures are Abstract Syntax Trees (AST for short).

Because ConcernJ has to work with two kinds of source code (Java files and ConcernJ specifications), there is the need to generate two parsers. It is possible to borrow the Java compiler from ComposeJ and use it for parsing Java files.

In this way, it is only necessary to define and create the ConcernJ parser. The tool used for creating it was JavaCC [JCC-01]. JavaCC only creates parsers, it is necessary to use another tool for creating ASTs, that tool was JJTree [JJT-01] (that comes with the JavaCC distribution).

Visitors

The Visitor design-pattern [GAMM-95] can be used to help to analyze ASTs. Among its features, the most interesting to highlight from is that it enables modularity between algorithms and data. In this way, it is possible to define different tasks (algorithms) such as *pretty printing*, *type checking*, *evaluation*, etc. and apply them, in different phases, to the main structure (the AST)

From the point of view of parsers, in particular the ConcernJ one, the data is represented by ASTs, and the algorithms (also called visitors) represent the logic of the language (they are used for checking correctness, completeness, evaluation of expressions, etc.).

5.3 The ConcernJ Architecture

5.3.1 The packages distribution

Figure 6-2 shows the ConcernJ's package structure.

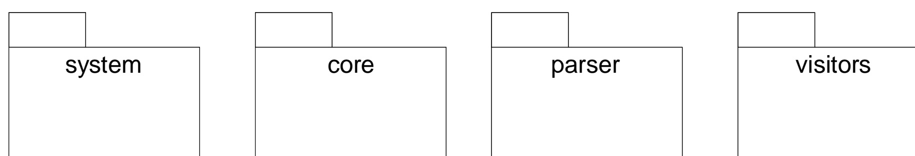

Figure 5-6. The ConcernJ package structure.

Each "folder" represents a particular package, where:

- **system**: contains the Classes for specifying and reading all the files that belongs to a particular system.
- **core**: contains the Classes for modeling systems, using the predefined hierarchical architecture of Java, that is Files, Classes and Objects.
- **parser**: contains the Classes for reading ConcernJ-specification files and transform them onto AST structures.
- **visitors**: contains the Classes for visiting the ASTs of Java and ConcernJ.

5.3.2 The System structure

The *system-package* is intended for modeling systems and for taking information about them at compile time. ConcernJ is a language intended for Java, so in this case a system is represented by a set of files. Those files must be explicitly specified because since Java is a *compiled* language, where classes have to be inspected before applying any super-imposition, i.e, it contains a Meta-level[2] representation of the programs.

The inspection of classes is also needed for selecting objects. The objects have to be known before concerns are parsed, because when parsing *selector-definitions* it is necessary to evaluate which objects are inside the system, their types and the files they are defined from.

Several systems can be defined, and several concerns can be applied over them. However, each time that a system is defined all the information has to be collected at compile-time. If some source file changes, the system has to be regenerated for actualizing all the Meta-information information.

The most interesting information from a system is:

- Which Files are included into the system,
- For each file, *package name*, *imports* and *classes* defined into it. Package name and Imports information is necessary for specifying types and for computing sub-typing.
- For each class, class name, superclass name, interface names and objects it necessary to collect information about typing.
- For each object, its associated types (the same object name can be associated with several types in the same class).

With all that information, it is possible to compute selectors, apply CF-interfaces to classes, apply super-imposition of methods & conditions over classes, apply changes over clients that use super-imposed types, compute super-typing, etc.

There are some interesting aspects to take into account when collecting all this information:

1. Source files have to be parsed and visited.
2. When applying super-imposition, some files have to be modified. So, there is some trade-off to take into account:
   a) Do those files have to be re-parsed for applying the changes over them?
   b) If the ASTs corresponding to those files are kept into memory when parsing the System, do we know before hand how many files and size of each one, and therefore the amount and size of ASTs, a System will contain when parsing it?

There is no oracle for telling us which files must be really needed for being kept into memory. Mainly, because not only direct classes[3] are changed when super-imposing interfaces and code, other files have to be changed as well. Those files are known as *clients*, and they also are part of a super-imposition, because when super-imposing a Type, certain objects have to be modified, thus their type, at compile-time in all the places where instances of that object are created.

[2] When using Meta-level we meant an abstract representation of the program, and we are not talking about reflection at all.
[3] The files corresponding to the types that have been superimposed.

The trade-off is:

- To save memory space, or
- To sacrifice speed & performance.

Computers have been becoming faster year after year, and memory also has been becoming higher and cheaper year after year. The true is that does not matter which option to choose, because from each point of view each option is acceptable.

For the current ConcernJ implementation, Abstract Syntax Trees are kept into memory just for simplicity. Although source files can be re-parsed as long as they are really needed. In this way, modifications over source files are sequentially made and saved once the whole super-imposition process has finished.

5.3.3 Collecting the information

The Core Structure

System files are specified in such a way that it is possible to know beforehand all the information related with a system. Each file is parsed and visited for collecting the needed information. That is done before any concern is applied.

Figure 5-7 shows the created structure when all the information is collected.

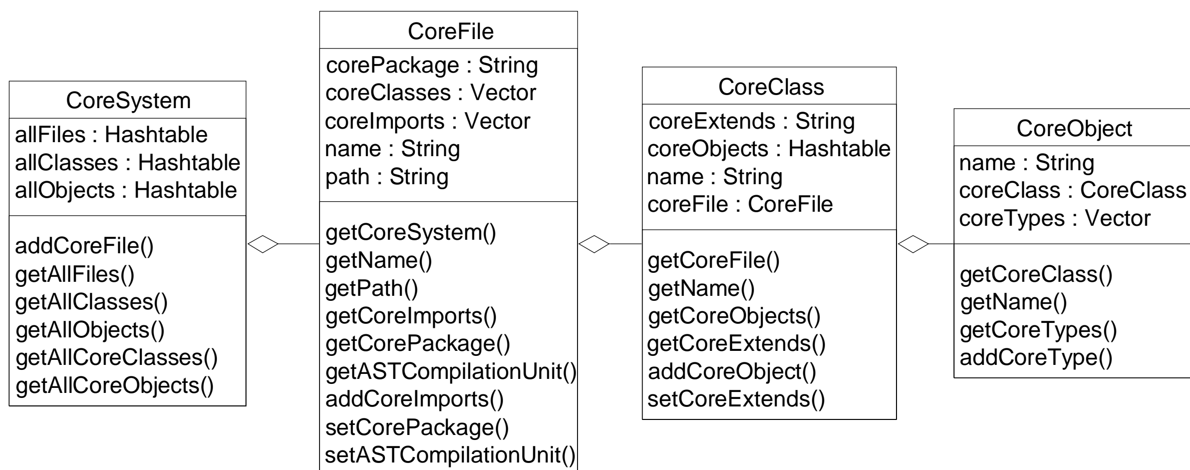

Figure 5-7. The Core structure, used for collecting all the information related with a system.

Figure 5-7 shows the following classes:

- **CoreSystem** contains all the information related with Files, Classes and Objects.
- **CoreFile** contains all the information related with a File, that is package name, imports and classes defined into the file.
- **CoreClass** contains all the information related with a Class, that is class name, super-type, interfaces that it implements and objects it has internally defined.
- **CoreObject** contains all the information related with objects/variables, that is, variable name and the types it has defined into the current class.

Each class contains a reference to the corresponding hierarchical Core-type. In this way, it is possible to know immediately for a class the file where it belongs to or the system where it is defined.

5.3.4 Java Visitors

Java source files are parsed into memory by using the Java-Parser borrowed from ComposeJ. The parsing of each file results into an AST, each AST can be visited by using specific visitors such as pretty printing, type checking, evaluation, etc.

To create the Core structure is necessary to apply to each created AST a visitor that walks through it and collects the information by creating the appropriated Core structures.

Figure 5-8 shows the Java-visitors class model:

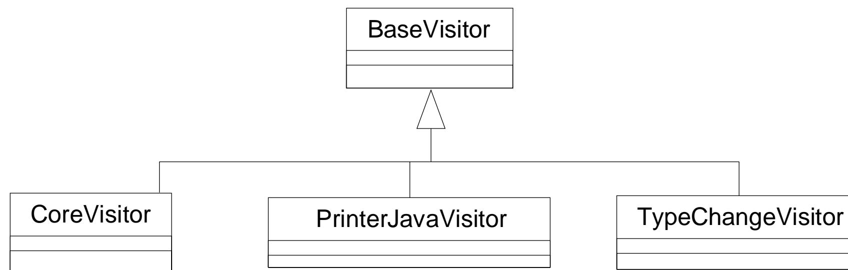

Figure 5-8. Java-Visitors class model.

Figure 5-8 shows the following classes:

- **BaseVisitor**: is an abstract class (a visitor) that implements default actions for each production into the Java AST.
- **CoreVisitor**: is intended to create the Core structure shown in Figure 6-3.
- **PrinterJavaVisitor**: is intended for applying pretty printing over parsed files.
- **TypeChangeVisitor**: is intended for changing type of objects over clients affected by a super-imposition (this visitor will be explained later on).

5.3.5 Applying Concerns

Once the System structure has been created, it is possible to start applying concerns over the system. Concerns are applied in order, one after one. When a concern needs information from another one, such as selectors, interfaces, methods & conditions, these concerns are parsed ***but not*** applied.

In this way, infinite recursion when applying concerns are avoided, because only definitions are parsed and the rest is not evaluated.

Figure 5-9 shows the structure created for storing information related to concerns:

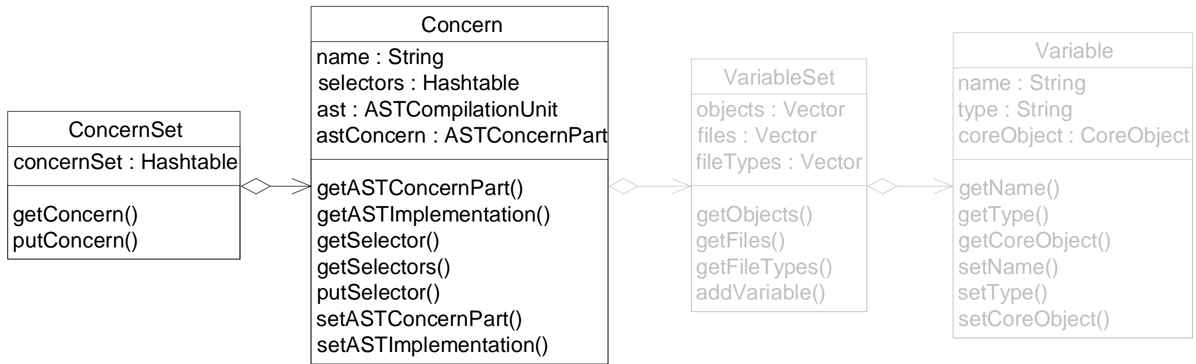

Figure 5-9. The Concern Structure for storing information related with concerns.

Figure 5-9 shows the following classes:

- **ConcernSet** contains all the parsed concerns, some of those could already have been applied and the rest not.
- **Concern** contains all the relevant information related to a Concern, such as concern name, the *selectors* environment (it will be explained later on), and both the implementation and the concern ASTs.
- **VariableSet** contains all objects related to a certain selector. This class is drawn in gray because it will be explained later on.
- **Variable** contains all the relevant information about a particular object. This class is also drawn in gray.

Again, when parsing concerns, there is no an optimal solution for solving this problem. That means, if concerns ASTs have to be kept in memory or re-parsed whenever they are needed again. Nevertheless, for this thesis work, the AST were kept in memory.

5.3.6 Concern Visitors

Concerns specifications are parsed in memory using the parser defined by the ConcernJ grammar (see Appendix A for a more detailed version of this grammar). After concerns are parsed and transformed onto AST structures, they can be visited and operations can be applied over them.

Figure 5-10 shows the class model for concern visitors.

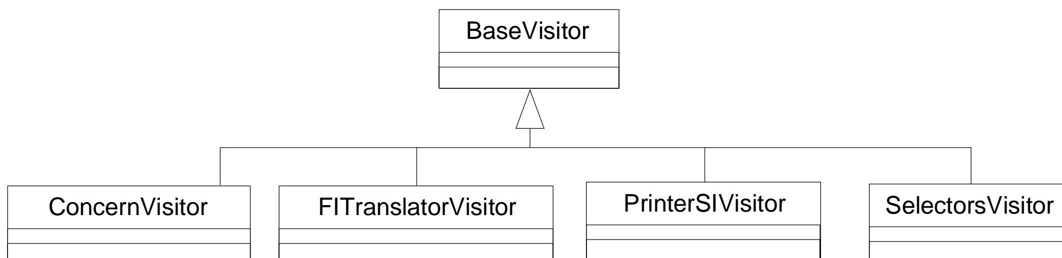

Figure 5-10. The Concern Visitors class model.

As shown in Figure 5-10, classes are modeled in UML, and respectively:

- **BaseVisitor**: is an abstract class (a visitor) that implements default actions for each production into the concern-AST.
- **ConcernVisitor**: is intended for creating the structure shown in Figure 6-5. Concern files are visited and the Concern-structure is created.

- **FITranslatorVisitor**: is intended for translating ConcernJ's CF-interfaces into ComposeJ's CF-interfaces. This is because the specifications for Composition-Filters interfaces are different in each language.
- **PrinterSIVisitor**: is intended for applying pretty printing over parsed concern files, for storing purposes.
- **SelectorsVisitor**: is intended for interpreting selectors, and for creating the VariableSet structure (shown in gray color in Figure 5-9).

5.3.7 Selectors Visitors

When concern ASTs are visited with *ConcernVisitor*, the selectors sub-section (corresponding to the *superimposition* block) is visited and interpreted separately for keeping modularity.

When selector-visitors walk through the selectors-subsection, they create the following structure:

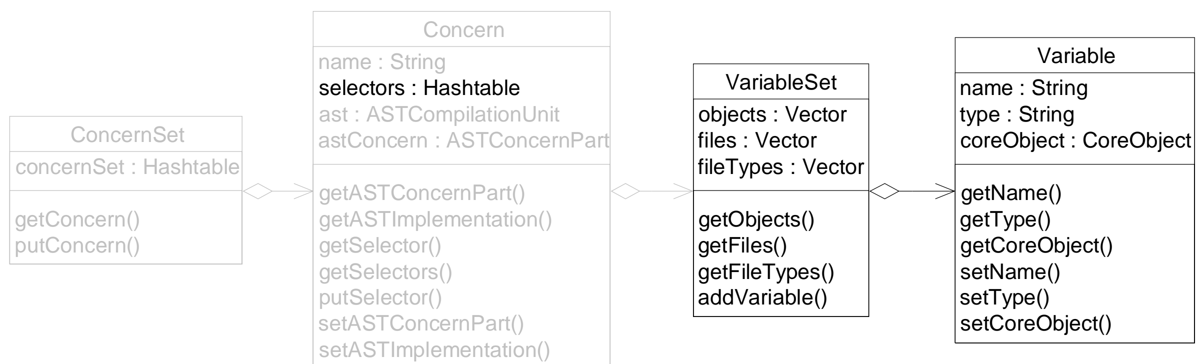

Figure 5-11. The selectors structure.

Figure 5-11 shows the same UML class model drawn in Figure 6-5. The relevant model was this time drawn in black.

Each concern file defines a selectors-block, selectors are interpreted and stored into a kind of "Environment" that contains objects and sets of objects. Lately, this *environment* is used for taking selector definitions and applying super-imposition over the objects they contain.

5.3.8 The effects of processing super-imposition

When super-imposition is applied the types of the objects referred to by the selector, have to be modified. Different super-impositions may affect the same object, and thus its type, at the same super-imposition phase. Let us call that type **Type**, so since not all objects into the system with type **Type** have to be super-imposed (because of selectors), there must exist a way for keeping their original behavior (their type), for the not affected objects, and enabling at the same time, for the affected objects, the super-imposed CF-specifications.

For solving this problem, there are two possibilities:

- **Static approach**: Based on checking and changing its type or instantiation method[4] at compile-time, for every object part of a super-imposition. So, at runtime those objects behave as composed objects.

[4] A *new* message to the Class.

- **Dynamic approach**: Interfaces, and thus their filters, are applied to the object at run-time. The type of a super-imposed object is changed in such a way that when a message comes to the object, it can be inspected and depending on the super-imposition specifications the message is dispatched to the corresponding super-imposed filters.

Static approach

Composed and not composed objects share their type. So, an explicit type relation among them must exist. Figure 5-12 shows a proposed class model for solving the type-relation problem.

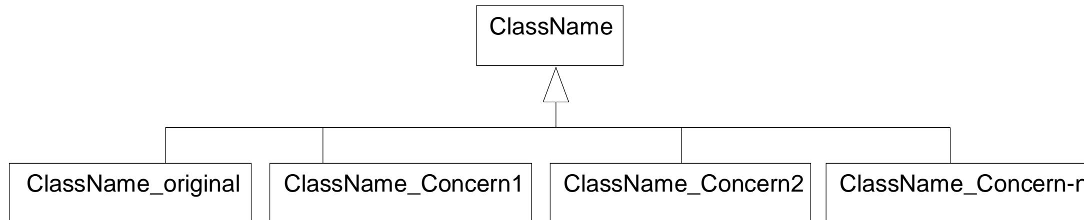

Figure 5-12. Class model for super-imposed classes.

As shown in Figure 5-12, classes are modeled into UML, and each one represents:

- **ClassName** is an abstract class (the original type name), which contains shared definitions for all its sub-types.
- **ClassName_original** is the original implementation for ClassName.
- **ClassName_Concern1** is the original implementation plus the modifications applied by Concern1.
- **ClassName_Concern2** is the same than ClassName_Concern1, but applied to Concern2.
- **ClassName_Concern-n** is the same than Classname_Concern1, but applied to Concern-n.

The different versions of **ClassName** are related with the approach for solving crossing problems shown in the last chapter. How different versions of super-imposition are explicitly specified will depend on the used "application" symbol. In any case, only one application symbol, when all the concerns are mixed, has been implemented. Nevertheless, for the rest symbols, the procedure and problems for applying those changes into the source code are almost similar.

For applying the changes to all those objects part of a super-imposition, the instantiation methods have to be replaced to the corresponding super-imposed classes. In this case, the object can change its implementation, and thus its composed class-version, just by invoking the corresponding instantiation class method.

When doing that a problem arises, to know at compile time whether a instantiation methods corresponds to certain composed object or not. This is because clients outside the system cannot be checked for changing all the needed instantiations, or because there can be really complex expressions that make type checking difficult to apply.

Dynamic approach

This second approach needs features that are not currently available on the traditional reflection capabilities of Java[5]. For applying CF-interfaces there must be at least a way for knowing the name of an object at runtime, and only the JVM knows that information.

[5] Reflection in Java is more similar to introspection.

Nevertheless, this solution is based on applying a general CF-interface over a **Type**. When every object is instantiated gets the same interface (**Type**). This implies that there is no modification of client code, which is a great feature. When messages are sent to objects, the CF-interface decides depending on the object name and the super-imposition specification which version to apply (some of the super-imposed versions or the original one).

Figure 5-13 shows an schema for representing the Dynamic approach.

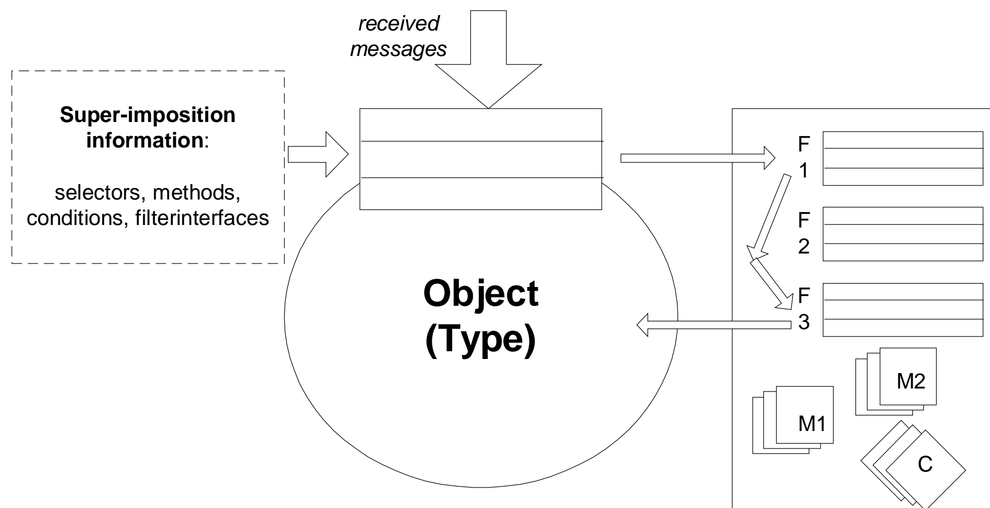

Figure 5-13. Dynamic approach using Compositon-Filters.

As shown in Figure 5-13, when a message arrives to the interface of **Object** (of type **Type**), it is inspected and depending on its name and the super-imposition information (represented as a dashed rectangle) the corresponding *filterinterfaces*, and thus their filters, are applied to the incoming message (in the example filterinterfaces are represented by F1, F2 and F3). In the example also methods (M1 and M2) and conditions (C) are kept into a modular way inside the general CF-interface.

Mainly because of the lack of better current reflection capabilities of Java, for instance it is not possible to know the name of an object at runtime, the taken option is the static approach.

5.4 Structures in detail

5.4.1 The System

For defining systems there are the following possibilities:

1. To define a project file specifying each file that pertains to the system.
2. Do not define a project file, but read all the needed files taking into account their package names and import statements.

Option (1) seems to be the most suitable approach, because systems are statically defined. Therefore, a user can specify which files are part of the system and which are not, decreasing the complexity of the super-imposition and the modification of clients programs. In this way, the user has more fine-grained control for defining the scope of super-imposition[6].

[6] AspectJ and HyperJ also define project files, where users specify a system.

5.4.2 The Core Structure

When visiting Java ASTs with a *CoreVisitor*, the following information is relevant and collected from each file:

- File Name
- Package name
- Import names
- Class names
- For each class: extends name and implements names
- For each class: objects and their types (an object name can have associated several types).

Objects are identified within the following parts:

- Field declarations,
- Method Parameters, and
- Local declarations

With all this information it is possible to build the Core-Structure. On the other hand, only classes with source files are allowed for being visited. This is because the lack of reflection capabilities of the Java language (only fields and methods are possible to know from a compiled class).

5.4.3 The Selectors Visitor

When a *SelectorsVisitor* walks through a selectors-declaration, it builds an Environment structure containing the interpretation for each OCL expression (a selector). The Environment is a *hashtable* containing as a key the selectors name and as a value the evaluation of the OCL-expression. Figure 5-14 shows the Environment structure.

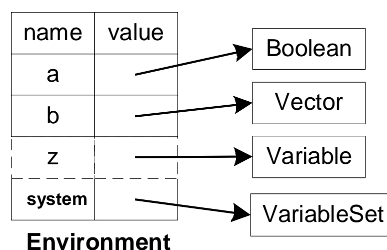

Figure 5-14. The Environment Structure.

Possible values that are accepted into the Environment hashtable are:

- **Boolean** values: for OCL-operations that return *true* or *false*.
- **String** values: for literal values, like class names.
- **Variable** values: for representing an object into the system.
- **VariableSet** values: a set of Variable objects.
- **Vector** values: a set of literal values. It is intended to represent an OCL *Set*.

With all those values is possible to evaluate OCL expressions. Only the following subset of operations, from the shown operators in Appendix B, is currently correctly implemented.

- Operations over Sets: ***select***, ***exists***.
- Operations over Objects: ***oclIsTypeOf***.

Only those operations where implemented because with that small set it *is possible* to specify selectors. Which proves the concept that OCL-expressions are suitable for defining sets of objects.

Nevertheless, there are operations that are difficult to implement like ***oclIsKindOf***; it needs more effort because it is necessary to compute super-types using extends, implements, imports and packages information from classes definitions (all this information is possible to obtain from the *Core-Structure*).

Another interesting aspect of selectors is the fact that when a selectors-block is visited there is already a *VariableSet* in the Environment. This set represents the system itself with all the objects defined into it and is called "**system**".

5.4.4 Java Visitors

The most important visitors included in this package are the *CoreVisitor* (already explained) and the *TypeChangeVisitor*. *TypeChangeVisitor* applies changes to clients that use objects affected by super-imposition.

The approaches for modifying clients were already explained in section 4.4 . So, *TypeChangeVisitor* is intended basically for changing object instantiations[7]. This is done by making a **Table** for super-imposed objects and their types. When instantiations are performed, the table is checked to see if corresponds a change on the instantiation method or not.

5.4.5 Concern Visitors

The most interesting visitor from this package is *ConcernVisitor*. Several features and characteristics were already shown in previous sections but some relevant details have not been explained yet.

When a *ConcernVisitor* walks through a Concern-AST, some checking can be done over each block inside the concern, such as:

- The correctness and completeness of each Composition-Filters interface.
- The correctness of the implementation block.
- The current concern completeness.

Currently, none of the activities take place, but it is interesting to take them into account for future improvements of ConcernJ.

Since ConcernJ produces ComposeJ code, it creates three kinds of files:

- CF-specifications: When concerns are applied over source files; they create modified versions of class declarations, by applying over them CF-interfaces
- JWF-files: When concerns are applied over source files; they create modified versions of class declarations, by applying over them methods and conditions.
- JAVA-files: When program clients need to be modified because they contain super-imposed objects (this would not be necessary in the case of creating a dynamic approach for super-imposing interfaces and code over types).

All those files are kept in memory in the structure *ModifiedFiles*. Which contains the corresponding ASTs for each modified type and client. When the super-imposition process finishes, all those files are saved to disk or passed directly as ASTs to ComposeJ for being translated into Java code.

---

[7] Currently this visitor is not completely built, because when checking object instantiations, there may be in between of the object and its instantiation really complex Java expressions that make their evaluations not a trivial work for a compiler (this is not a main goal for this thesis work).

5.5 Summary

The most important aspects to highlight from this chapter are:

- The possibilities for producing code. Several possibilities were studied for producing code, such as AspectJ, HyperJ and ComposeJ. These tools currently apply super-imposition on the Java language.

- Making a decision for the most suitable option for producing code. The most interesting aspects of each super-imposition languages were studied and a decision was made, based on presented advantages & drawbacks.

- The concept of System. Files into the system are explicitly specified, so a user can specify whether a file is interesting for a modeling or not.

- The Core Structure for collecting all the information related with a system. It is necessary to know before applying concerns all the objects inside the specified system.

- How information is collected and what is interesting for collecting from. Object information is collected from Fields, Local declarations and methods signatures.

- The Concern structure for collecting all the information related with concerns. When a concern is evaluated a concern-structure is created for keeping information about already applied concerns.

- Concerns are applied explicitly. When a concern is parsed, it will not necessarily be applied. Concerns are also parsed when collecting information about selectors and application of filterinterfaces, methods and conditions.

- The Environment structure for storing selectors. When the selectors sub-section is parsed, OCL expressions are evaluated and stored into an Environment structure. That structure is lately inspected when applying super-imposition.

- Approaches for modifying and applying the super-imposition to objects. When super-imposing objects, their types have to be modified. For applying those modified types there are two approaches, a static one based on a compile-time approach and a dynamic one based on reflection.

Chapter 6 : Conclusions

The present thesis work was intended to study the main aspects of the design and implementation of a tool that could apply super-imposition of Composition-Filters to a system, in a modular and systematic way.

Super-imposition is a technique that several tools and languages, such as AspectJ, HyperJ and ComposeJ adopt and implement for generating code.

Also, those tools were presented and studied as examples for showing the super-imposition concepts. Moreover, they were presented as possible solutions for applying super-imposition of Composition-Filters in the Java language.

Relevant aspects of super-imposition were presented (Chapter 2):

- The identification of "**who**", "**what**", "**when**" and "**where**" when doing super-imposition.
- The identification of crossing problems (overlapped super-imposition).

How they are related with current tools using super-imposition and how common problems in those tools are worked out.

Also, when designing a language that uses super-imposition, is important:

- The grammar,
- How it is related with super-imposition (identifying correctly ***who***, ***what***, ***where*** and ***when***), and
- How crossing problems are avoided by using the grammar.

Later, when implementing the tool:

- How super-impositions concepts are included,
- The design and implementation of structures to enable super-imposition concepts.
- How crossing problems are worked out, taking into account the objective programming language, its features and capabilities.

All those concepts and aspects that involves super-imposition are important to analyze when defining a language.

On the other hand, in this thesis work we tried to make a language-independent specification to make a language that uses super-imposition for applying Composition-Filters specifications. Thanks to the nature of the Composition-Filters model that was possible. The ConcernJ implementation was intended to apply all collected concepts on the Java language.

Based on the expressed above and the contents of this thesis work, the following aspects are important to conclude:

- The majority of the chosen approaches were static, mainly because they were simply to implement or there were not the facilities to implement those approaches dynamically.

- UML and OCL were useful for super-imposition of Composition-Filters. UML for defining interfaces and OCL for specifying selectors. The main reason for using them is they are well proven and defined approaches, with a lot of people improving it design and with a heterogeneous grammar.

- The main reason for using only a subset of OCL is its expressiveness for selecting objects. Nevertheless, we can extend it later for adding new existing OCL features.

- Only was implemented a small subset of the OCL operators, they can be extended later

- Specifications of super-imposition for Composition-Filters can be ported to a new implementation on another language.

6.1 Results and problem statement

We intended to design and implement a language for supporting the super-imposition of Composition-Filters specifications. We base our design on base an example problem. Nevertheless we tried to make a very general specification. We focused out efforts mostly in the design part, however an implementation of that design was made for the Java language.

A consequence of this design is the specification of a Language independent grammar for the language. It is based on previous implementations of Composition-Filters and the incorporation of UML for defining language dependent sections. Also, another contribution of this design was the incorporation of OCL for the definition of selectors (sets of objects).

Several crosscutting tools were considered for inspecting its implementation and also for generating super-imposition. Nevertheless, none of them was considered for generating code.

Another contribution of this thesis work, although it is only a proof of concept, is the implementation of ConcernJ, the super-imposition language for Java. The design of the structures is based mostly on static approaches. Nevertheless, we hope that in future improvements dynamic approaches can be included.

6.2 Further Work

Some concepts, such as abstract classes and class interfaces were not taken into account when we defined and designed the structure of the super-imposition language.

There is the need to change the naming of some sections in the current grammar specifications. For instance, when defining a *filterinterface*, it may result confusing relate concepts as interface and filters, because the concept interface, in Object-Oriented languages, is oriented to express the signature of a class. On the other hand, filters are intended for declaring Composition-Filters definition.

Another section to take care of is the implementation-section, because the grammar is not clear enough to express that the implementation is defined in a separated module.

New implementation-blocks can be added to the concern specification. Each one can be written in a distinct programming language.

When super-imposing different specifications over an object, the dynamic approach could be the most appropriated for solving this kind of problems. Because it allows a non invasive approach for applying Composition-Filters interfaces over classes (only “servers” are changed and not “clients”).

Bibliography & References

[ACM-01] Aksit, Mehmet. Kiczales, Gregor. Lieberherr, Karl. Ossher, Harold. *"A Discussion on Aspect-Oriented Programming. Frequently-Asked Questions"*, Publications of the ACM – *not yet published*. October 2001.

[AKS-92] Aksit, Mehmet. Bergmans, Lowewijk. Vural, S. *"An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach"*, Proceedings of ECOOP '92, LNCS 615, Springer-Verlag, 1992, pp. 372-395.

[ASPJ-01] *"AspectJ: Aspect-Oriented Programming for Java"*, AspectJ Web Site, http://www.aspectj.org, 2001.

[BeAk-00] Bergmans, Lodewijk. Aksit, Mehmet. *"Composing Multiple Concerns Using Composition Filters"*, TRESE Group, Computer Science Department, University of Twente, The Netherlands, 2000.

[BERG-94] Bergmans, Lodewijk. "Composing Concurrent Objects", Ph.D. Thesis, Computer Science Department, University of Twente, The Netherlands, 1994.

[GAMM-95] Gamma, Erich. Helm, Richard. Johnson, Ralph. Vlissides, John. *"Design patterns, Elements of Reusable Object Oriented Software"*. Addison-Wesley, 1995.

[GLAN-95] Glandrup, M.H.J. "Extending C++ using the concepts of Composition Filters", Master of Science thesis, Computer Science Department, University of Twente, The Netherlands. November 1995.

[HYPJ-01] *"HyperJ: Multi-Dimensional Separation of Concerns for Java"*, IBM Research Web Site, http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm, 2001.

[JCC-01] "JavaCC - The Java Parser Generator", *http://www.webgain.com/products/metamata/java_doc.html*. Metamata Inc., 2000.

[KIC-97] Kiczales, Gregor. Lamping, John. Mendhekar, Anurag. Maeda, Chris. Videira-Lopes, Cristina. Loingtier, Jean-Marc. Irvin, John. *"Aspect-Oriented Programming"*, http://www.parc.xerox.com/csl/projects/aop/selectedPapers.shtml. June 1997.

[OCL-01] "Object Constraint Language Specification", Klasse Objecten Web Site, http://www.klasse.nl/ocl/index.html, 2001.

[SAL-00] Salinas, Patricio. *"How to use AspectJ"*, Capita Selecta report, EMOOSE exchange program, Ecole des mines de Nantes, France. December 2000.

[SINA-95] Koopmans, Piet. "On the Definition and Implementation of the Sina/st Language", Master of Science thesis, Computer Science Department, University of Twente, 1995.

[UML-01] "The Unified Modeling Language"

[WICH-99] Wichman, J.C. "ComposeJ, The development of a preprocessor to facilitate Composition Filters in the Java Language", Master of Science thesis, Computer Science Department, University of Twente, The Netherlands. December 1999.

Appendix A: ConcernJ grammar

The objective of the present appendix is explaining each part of the ConcernJ grammar, used to define concerns for the Composition-Filters model.

Because we are trying to explain a grammar, it could be useful to remember some interesting BNF rules:

[]	:	It means that everything inside brackets is optional (zero or one time).
()*	:	It means that everything inside parentheses can appear zero or more times.
()+	:	It means that everything inside parentheses can appear one or more times.
()LIST	:	It is a shortcut for (element1, element2, element3, ...), a set of elements separated by comma. In this document a LIST will contain at least one element.

A Concern:

A concern will define the behavior of one or more objects inside the system or application, it contains 3 interesting blocks:

```
CONCERN =
<concern> CONCERN_NAME <begin>
   ( FILTERINTERFACE )*
   [ IMPLEMENTATION  ]
   [ SUPERIMPOSITION ]
<end> [<concern> CONCERN_NAME];
```

Where:

CONCERN_NAME: Represents the name of the current concern.
FILTERINTERFACE: Represents a CF-interface specification.
IMPLEMENTATION: Represents an implementation-block (a class containing methods and conditions).
SUPERIMPOSITION: Represents the super-imposition specification.

A concern can contains several **filterinterface**-declarations, at most one **superimposition** declaration and one **implementation** block.

The Filter-Interface Block:

A *filterinterface* block contains the same definitions of a CF-interface (internals, externals, methods, conditions, inputfilters and outputfilters).

```
FILTERINTERFACE =
<filterinterface> FILTERINTERFACE_NAME <begin>
   [INTERNALS]
   [EXTERNALS]
   [CONDITIONS]
   [METHODS]
   [INPUTFILTERS]
   [OUTPUTFILTERS]
<end> [<filterinterface> FILTERINTERFACE_NAME];
```

Where:

FILTERINTERFACE_NAME: Represents the name given to this CF-interface definition. It will be used for being identified and super-imposed over source code.

The Internals sub-block:

```
INTERNALS=
<internals>
   (VARIABLE_SET <:> TYPE <;>)*

VARIABLE_SET=
   VARIABLE_NAME |
   VARIABLE_NAME <,> VARIABLE_SET
```

As shown, variables are defined in UML-style. A Type may contain the package name, separated by double-colons (currently not supported).

Another interesting aspect is the fact that variables cannot be instantiated directly in this block. They have to be instantiated directly into the source code, or by calling a method.

The Externals sub-block

```
EXTERNALS=
<externals>
   (VARIABLE_SET <:> TYPE <;>)*

VARIABLE_SET=
   VARIABLE_NAME |
   VARIABLE_NAME <,> VARIABLE_SET
```

As shown, variables are defined in UML-style. A Type may contain the package name, separated by double-colons (currently not supported).

The Conditions sub-block

```
CONDITIONS=
<conditions>
   (CONDITION_NAME <;> )*
```

Conditions are defined without specifying their type. This is because conditions can only return as a value a boolean.

The Methods sub-block

```
METHODS=
<methods>
   (METHOD_NAME <(> [ ARGUMENT_SET ] <)> [ <:> TYPE <;> ])*

ARGUMENT_SET=
   ARGUMENT_DEF ( <;> ARGUMENT_DEF )*

ARGUMENT_DEF=
   [ VARIABLE_NAME (<,> VARIABLE_NAME)* <:> ] TYPE
```

Method signatures are defined by using an UML-like grammar. Also, as explained for internals, a Type can be specified with a package name, but it is not implemented.

The Input-Filters sub-block

```
INPUTFILTERS=
<inputfilters>
   (FILTER_NAME <:> FILTER_TYPE <=> <{> FILTER_SET <}> <;>)*

FILTER_SET=
   [CONDITION FILTER_OP] OBJECT_SET |
   [CONDITION FILTER_OP] OBJECT_SET COMPOSITION_OP FILTER_SET

FILTER_OP=
   <=>> | <~>>

COMPOSITION_OP=
   <,>

OBJECT_SET=
   <{> OBJECT ( OBJECT_OP OBJECT )* <}> |
   OBJECT

OBJECT=
   [ TARGET <.>] SELECTOR [ <(> [ <TARGET> <=> TARGET1 <,> <SELECTOR> <=>
                                                           SELECTOR1 ]

OBJECT_OP=
   <,>

SELECTOR=
   METHOD_NAME [ <(> TYPE ( <;> TYPE )* <)> ]

TARGET=
   <inner> | OBJECT_NAME
```

This block represents the classic CF-interface. A better explanation about this block can be found at [WICH-99].

The Output-Filters sub-block

The outputfilters grammar is almost the same than the inputfilters one, only changes the token for specifying this block.

The implementation block:

```
IMPLEMENTATION=
   <implementation> <in> <”> MODULE_NAME <”> <;>
```

This block specifies the location of the implementation block.

The Super-Imposition Block:

This block is the most interesting of this work, because it will define the behavior of our applications, how filter-interfaces will interact each other, how classes in our applications will be affected directly or indirectly, etc.

```
SUPERIMPOSITION =
<superimposition> <begin>
   [SELECTORS]
   [CONDITIONS]
   [METHODS]
   [FILTERINTERFACES]
<end> <superimposition>;
```

Where:

SELECTORS: Represents the block for defining set of objects.
CONDITIONS: Represents the block for defining super-imposition of conditions.
METHODS: Represents the block for defining super-imposition of methods.
FILTERINTERFACES: Represents the block for defining super-imposition of Composition-Filters interfaces.

The selectors sub-block:

```
SELECTORS =
<selectors>
  ( SELECTOR_NAME <:=> OCL_EXPRESSION <;> )*
```

Where:

SELECTOR_NAME: The name of the current selector definition.
OCL_EXPRESSION: An OCL expression that generates a set of objects (more information in Appendix B)

The methods sub-block:

```
METHODS =
<methods>
   ( [CONCERN_NAME1 <::>] SELECTOR_NAME <<-> METHOD_NAME_SET <;> )*

METHOD_NAME_SET =
   METHOD_NAME | <{> (METHOD_NAME)LIST <}>

METHOD_NAME =
   ([CONCERN_NAME2 <::>]METHOD_NAME_ID) |
   ([CONCERN_NAME2 <::>]FILTERINTERFACE_NAME <::> <*>

CONCERN_NAME=
   <self> | IDENTIFIER
```

Where:

CONCERN_NAME1::SELECTOR_NAME : Represents a specific selector with name SELECTOR_NAME defined into **CONCERN_NAME1**.

METHOD_NAME_SET: Represents a set of possible methods that will be superimposed over the specified selector.

METHOD_NAME: Represents specific methods taken from the implementation block defined into CONCERN_NAME2. If CONCERN_NAME2 is not given, the default concern will be the current one (self).

If **FILTERINTERFACE_NAME** is specified, all the methods defined into that interface will be taken from **CONCERN_NAME2** and super-imposed over the selector.

Several method super-impositions can be defined in this block.

The Conditions block:

```
CONDITIONS =

<conditions>
  ( [CONCERN_NAME1 <::>] SELECTOR_NAME <- CONDITION_NAME_SET ; )*

CONDITION_NAME_SET =
   CONDITION_NAME | <{> (CONDITION_NAME)LIST <}>

CONDITION_NAME =
   ([CONCERN_NAME2 <::>]CONDITION_NAME_ID)LIST      |
   ([CONCERN_NAME2 <::>]FILTERINTERFACE_NAME <::> <*>

CONCERN_NAME =
   <self> | IDENTIFIER
```

The CONDITIONS block is almost the same than the METHODS one, but it is intended for super-imposing conditions over selectors.

The filter-interfaces sub-block:

```
FILTERINTERFACES=

<filterinterfaces>
  ([CONCERN_NAME1 <::>]SELECTOR_NAME <<-> FILTERINTERFACE_SET <;> )*

FILTERINTERFACE_SET=
  [CONCERN_NAME2 <::>]FILTERINTERFACE_NAME |
  <{> ([CONCERN_NAME2 <::>] FILTERINTERFACE_NAME)LIST <}>
```

The FILTERINTERFACES block is almost the same than the METHODS one, but it is intended for super-imposing Composition-Filters interfaces.

Appendix B: Selectors grammar

Mainly a selector represents a set of objects related with certain criteria; OCL has operations over collections, specifically Sets. Some of its interesting operations are:

SETS

Set

A Set is a mathematical-set. It contains elements without duplicates.

Declaration

```
set = Set{ elements }   // a comma list of elements.
```

Operations over a Set

Union: The union of set and set2:

```
set->union(set2 : Set(T)) : Set(T)
```

Equal: Evaluates to true if *set* and *set2* contain the same elements.

```
set = (set2 : Set(T)) : Boolean
```

Intersection: The intersection of *set* and *set2* (i.e. the set of all elements that are in both *set* and *set*2).

```
set->intersection(set2 : Set(T)) : Set(T)
```

Minus: The elements of *se*t, which are not in *set*2.

```
set – (set2 : Set(T)) : Set(T)
```

Including: The set containing all elements of *set* plus *objec*t.

```
set->including(object : T) : Set(T)
```

Excluding: The set containing all elements of *set* without *objec*t.

```
set->excluding(object : T) : Set(T)
```

Symmetric Difference: The sets containing all the elements that are in *set* or *set*2, but not in both.

```
set->symmetricDifference(set2 : Set(T)) : Set(T)
```

Select: The subset of *set* for which *expr* is true.

```
set->select(expr : OclExpression) : Set(T)
```

Reject: The subset of *set* for which *expr* is false.

```
set->reject(expr : OclExpression) : Set(T)
```

Operations over collections (a Set is a sub-type of Collection):

Let's call *collection* an instance of Collection.

Size: The number of elements in the collection *collection.*

```
collection->size : Integer
```

Includes: True if *object* is an element of *collectio*n, false otherwise.

```
collection->includes(object : OclAny) : Boolean
```

Excludes: True if *object* is not an element of *collectio*n, false otherwise.

```
collection->excludes(object : OclAny) : Boolean
```

Count: The number of times that *object* occurs in the collection *collection* (not interesting).

```
collection->count(object : OclAny) : Integer
```

Includes all: Does *collection* contain all the elements of *c2* ?

```
collection->includesAll(c2 : Collection(T)) : Boolean
```

Excludes all: Does *collection* contain none of the elements of *c2* ?

```
collection->excludesAll(c2 : Collection(T)) : Boolean
```

Is empty: Is *collection* the empty collection?

```
collection->isEmpty : Boolean
```

Is not empty: Is *collection* not the empty collection?

```
collection->notEmpty : Boolean
```

Exists: Results in true if *expr* evaluates to true for at least one element in *collection.*

```
collection->exists(expr : OclExpression) : Boolean
```

For all: Results in true if *expr* evaluates to true for each element in *collectio*n; otherwise, result is false.

```
collection->forAll(expr : OclExpression) : Boolean
```

Results in true if *expr* evaluates to a different value for each element in *collectio*n; otherwise, result is false.

```
collection->isUnique(expr : OclExpression) : Boolean
```

Iterate: Iterates over the collection. See "Iterate Operation" on page 7-25 for a complete description. This is the basic collection operation with which the other collection operations can be described.

```
collection->iterate(expr : OclExpression) : expr.evaluationType
```

OBJECTS:

Equal: True if *object* is the same object as *object2*.

```
object = (object2 : OclAny) : Boolean
```

Not equal: True if *object* is a different object from *object2*.

```
object <> (object2 : OclAny) : Boolean
```

Is kind of: True if *type* is one of the types of *object*, or one of the supertypes (transitive) of the types of *object*.

```
object.oclIsKindOf(type : OclType) : Boolean
```

Is type of: True if *type* is equal to one of the types of *object.*

```
object.oclIsTypeOf(type : OclType) : Boolean
```

Operations over types:

Name: The name of *type*.

```
type.name : String
```

Super-type: The set of all direct supertypes of *type*.

```
type.supertypes : Set(OclType)
```

All Super-types: The transitive closure of the set of all supertypes of *type*.

```
type.allSupertypes : Set(OclType)
```

All instances: The set of all instances of *type* and all its subtypes in existence at the snapshot at the time that the expression is evaluated.

```
type.allInstances : Set(type)
```