Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
and
Universidad National de La Plata - Argentina
2001-2002

Dynamic Aspect Composition using Logic Metaprogramming

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: Jessie Dedecker

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Maximo Prieto (Universidad National de La Plata)

Abstract

Separation of concerns has always been one of the most important goals in computer science. However, not all concerns can be easily modularized in the current programming languages. This resulted in the fact that some concerns are implemented ad-hoc, that is by placing code that is spread across one or more modular units. Such code is often called tangled or crosscutting code. Crosscutting code results in software that is less reusable, less readable and harder to maintain. Several programming tools have been proposed to modularize these crosscutting concerns in aspects. Many of these programming tools for advanced separation of concerns are built by creating or modifying the compiler of the existent languages that weave the crosscutting concern in the code. However, modifying a compiler makes it difficult to introduce aspects at run-time. Introducing aspects at run-time is particularly useful in distributed applications, where the crosscutting concerns can depend on the machine where the object is located, thus requiring aspects to be woven each time an object is migrating over the network. In this dissertation we propose a composite aspect object to support dynamic weaving of aspects. Furthermore, we propose to compose these composite aspect objects using a logic meta programming language. A logic meta programming language allows us to reason about the source code and the structure of the base level in a declarative style. Reasoning about the source code is useful in a dynamic context, because it allows to adapt our composition strategy depending on the changes that are made at run-time. Our model for dynamically typed object oriented languages defines a meta-architecture that configures the meta-object protocol using logic meta programming rules for supporting dynamic aspect compositions.

Acknowledgements

This dissertation would not have been what is it is today, without the immense help and discussions of other people.

I would like to thank my promotor Theo D'Hondt, who gave me the opportunity to do the EMOOSE master that has been a wonderful and unforgettable experience.

Also a big thank you goes to Maximo Prieto for our many interesting discussions about this dissertation even though he had a busy work schedule.

Many thanks go to Johan Brichau and Kris Gybels for discussing and questioning my ideas and for reading the dissertation in all its stages.

Tom Mens and Gustavo Rossi for proof-reading my dissertation and for providing me with many useful suggestions for improving it.

Thanks also go to my parents for supporting me even though I was abroad.

Contents

Chapter 1

The Thesis

1.1 Introduction

In this dissertation, we show how the composition problem of dynamically woven aspects can be addressed flexibly using composite aspect objects. We use logic meta programs to compose these composite aspect objects. Composition issues are more difficult to resolve when we can dynamically add and remove aspects in an application.

Modularization to manage complexity has always been one of the most important goals in computer science[Dij76, Par72]. This has been expressed with the development of new programming languages, i.e., crude assembly languages have been replaced with object-oriented programming languages over the years. However, it has come to the attention that not all concerns can be easily modularized in the current programming languages. This resulted in the fact that some concerns are implemented ad-hoc, that is by placing code that is spread across one or more modular units. Such code is often called tangled or crosscutting code. Crosscutting code results in software that is less reusable, less readable and harder to maintain. The research community is responding to this problem by proposing multiple programming tools [KLM+97, BA01, OT01, LOO01] to extend existing programming languages to support modularization of such crosscutting concerns and hence, achieve a more advanced level of separation of concerns. The language extensions often provide some syntactical construct for modularizing these crosscutting concerns. Many of these programming tools for advanced separation of concerns are built by creating or modifying the compiler of the existent languages that weave the crosscutting concern in the code [KLM+97, BA01, OT01, LOO01]. Another approach to achieve advanced separation of concerns is through the use of reflection [Sul01]. The advantage of reflection over a modified compiler is that it allows one to dynamically weave and unweave the crosscutting concern into or out of the application. This dynamism of adding and removing crosscutting concerns

is especially important in the context of distributed systems and component-based systems. For example when we have a program that allows you to dynamically plug and unplug components so that the availability of the system is optimized while you can still upgrade its components. This all works well until we come to realize that the components we want to plug into the system can crosscut over the system.

In a distributed environment objects are passed over the network, but often the available services of the distributed environments are different (such as security policies, available memory, processing power, ...). Hence, each time an object migrates to another environment it needs to consider other concerns. Therefore, each time an object migrates we might want to adapt the crosscutting concerns that are woven in the object to take care of the new constraints and possibilities that pose itself in the new distributed environment.

Composition of crosscutting concerns is not straightforward, for example when two or more crosscutting concerns need to be woven in the same place in the code they can create conflicts. Problems like these are harder to solve when the weaving is done dynamically, because the conflicts have to be detected and resolved at run-time.

A logic meta programming language allows us to reason about the source code and the structure of the base level in a declarative style. Reasoning about the source code is useful in a dynamic context, because it allows to adapt our composition strategy depending on the changes that are made at run-time.

1.2 Thesis Statement

In this dissertation, we show how the composition problem of dynamically woven aspects can be addressed flexibly using composite aspect objects. Secondly, we also show that logic meta programs provide an expressive means to compose these composite aspect objects.

1.3 Outline of the Dissertation

This section describes the road map for reading this dissertation.

- Chapter two explains the composition problem of modularizing some concerns in an application and finally describes some tools for advanced separation of concerns that were proposed by the research community.
- Chapter three describes common composition issues that we think must be solved in any good tool for doing advanced separation of concerns.

- Chapter four introduces the composite aspect object model for doing advanced separation of concerns for aspects that can be dynamically woven into the application.
- Chapter five describes how we can employ logic meta programming for composing the composite aspect objects and how it can help us resolve the composition issues described in chapter three.
- Chapter six describes the implementation of two small cases to show the usefulness of the composite aspect object in an application.
- Chapter seven concludes this dissertation by giving a summary and a discussion on the contributions we made. It also discusses future work.

Chapter 2

Techniques for Separation of Concerns

2.1 Introduction

Programming has evolved from assembly programming into more advanced programming paradigms, such as procedural programming, structured programming, functional programming, logic programming and object-oriented programming. These programming paradigms strive for better decomposition and modularization of programs. A better decomposition attributes to a better separation of concerns [Dij76, Par72]. Separation of concerns in a program has many benefits and is claimed to enhance the quality attributes of the source code [FBLL02, OT99]:

- **Adaptability**: the lifespan of software increases and there is a need to let software evolve and adapt to new requirements. A better separation of concerns shapes software, so that it is easier to make changes to it, without experiencing problems such as ripple effects.
- **Reusability**: a good modularized design with a strong cohesion and weak coupling attributes to the reusability of the software. One can reduce the cost and development time by reusing existing pieces of software.
- **Comprehensibility**: the code becomes more understandable when we have a clear separation of concerns, because we only have to read code that is about the concern. In other words, the reader of the code does not have to perform a mental switch between the different concerns.

Currently object-oriented programming is the most popular programming paradigm. With object-oriented programming the problem is decomposed into objects. One of the merits of object-oriented programming is

that it allows to modularize most of the concepts in the problem and solution domain allowing for better evolution of the software. The idea is to encapsulate all the relevant entities, that are likely to be subject to change, into objects. This way we can extend these points of variability using the common techniques, such as inheritance and aggregation.

The design process of an application often results in a single decomposition of the problem domain. This decomposition is often called the *dominant* decomposition. Bass et al. [BCK98] gives two remarks on this:

1. There is no single correct decomposition of a software system.
2. No decomposition can ever achieve full separation of *all* the concerns.

The idea of having multiple views on software is also present in the UML design language [FS97, Kru95], where you have different types of diagrams on the same piece of software. Some of these diagrams express different concerns. For example, the collaboration diagram expresses how a set of objects interact together, while a class diagram expresses the static structure between the different classes.

2.1.1 Concerns

Every good software engineer intuitively strives for separation of concerns in all phases of the development. As with most terms used in computer science, there is no well established definition for a concern. We use the definition from [Boa00]:

Definition 1 (Concern) *Those interests which pertain to the system development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. Concerns can be logical or physical concepts, but they may also include system considerations such as performance, reliability, security, distribution, and evolvability.*

Examples of functional concerns are business rules and software features. Examples of non-functional concerns are security policies, synchronization, distribution, ... Some of these concerns are hard to modularize using the standard techniques of modularization. Implementing such concerns often leads to tangled code and crosscutting code.

2.1.2 Tangled Code and Crosscutting Code

Concerns that are spread across one or more modular unit often result in tangled code and crosscutting code, because they are not confined to one place in the code. This makes it hard to implement them, because the developer needs to be aware of them when he implements other concerns. Another problem is evolution. As the concerns are not modularized it is

difficult to make changes to them. Parts of the changes are easily forgotten as the code is dispersed over multiple parts.

Definition 2 (Tangled Code) *We say that source code in one modular unit is tangled when it contains pieces of code dealing with different concerns.*

Definition 3 (Crosscutting Code) *Source code implementing a given concern is crosscutting when it is contained in more than one modular unit.*

Definition 4 (Modularized Code) *Modularized code is when the code of a concern is grouped together in a unit and is loosely coupled with the code of another unit. (i.e., a unit could be a class in a class-based object-oriented language)*

The code fragment below is an example of tangled and crosscutting code:

```
ShoppingCart>>buy

   "synchronization concern"
   monitor lock: self.

   "authentication concern"
   [ auth isAuthenticated ] whileFalse: [
      auth authenticate
   ].

   items do: [:item | item buy ].

   "synchronization concern"
   monitor unlock: self.
```

2.2 Achieving Separation of Concerns

If we have a simple figure editor application, then we can represent the figures as objects. As the program needs to evolve we can add new kinds of figures and/or adapt existing ones. However, when we consider the problem of updating the screen each time a figure has been manipulated, we have to put code spread across the different methods to update the screen each time the object is changed. The code for updating the screen is crosscutting the different classes. The problem is that the *concern* of updating the screen is not modularized. This crosscutting causes that the quality attributes of the source code, as described above, are reduced.

A UML class diagram of part of the application is shown in figure 2.1. A figure can be manipulated by sending a message to the object that represents the figure. Figures can be grouped using the composite pattern [GHJV94].

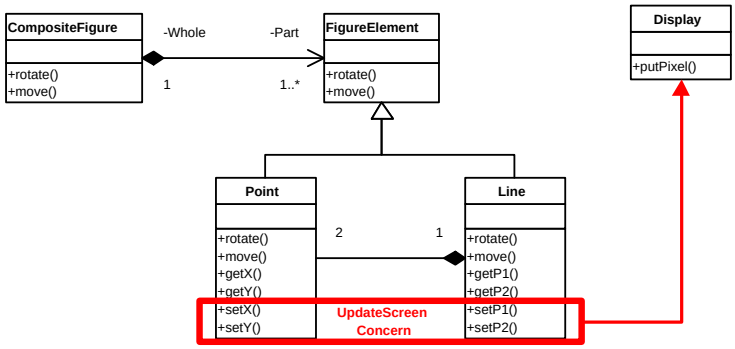

Figure 2.1: UML Diagram for Simple Figure Editor

We will now try to separate the concern for keeping the screen up-to-date with the internal data model using techniques available in some object-oriented programming languages.

2.2.1 Inheritance

We can separate the code for updating the screen by creating a subclass for each figure and override the methods that change the state of that figure. In the overriding method we call the overridden method and after that we can update the screen. This is shown in figure 2.2. There are 2 problems with this solution:

1. The code for updating the screen is not tangled with the other code anymore, but it is still spread across the different subclasses.

2. When we would like to make changes to the figure classes, for example to introduce a specialization of the existing Line class, we can do this by creating a subclass of *Line*. The problem is that the inheritance relationship in most class-based languages is static and that we cannot always reuse existing screen updating code for the different classes in the hierarchy. Hence encapsulating crosscutting concerns in different subclasses is not sufficient for separation of concerns.

The notion of inheritance is not made for separation of concerns, but it is rather based on specialization for elements from the problem and solution domain. Many variations of the inheritance mechanism exist that allow a better separation of concerns, but many have the problem that the concerns are still tangled across the code and do not allow reusability of the concern to its full extend.

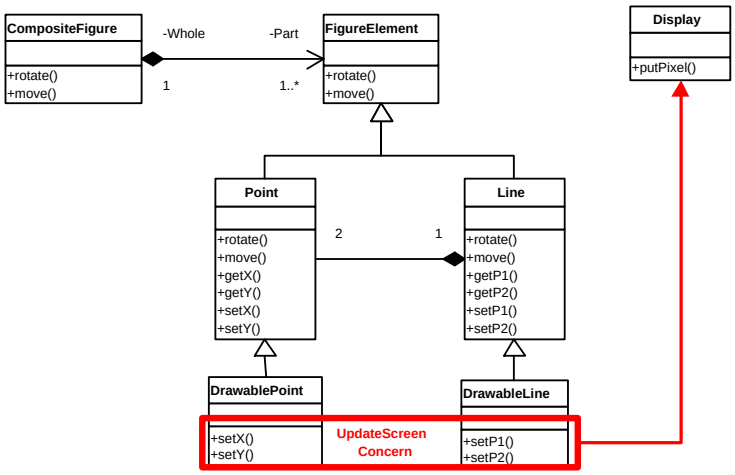

Figure 2.2: Simple Figure Editor: Separation of Concerns Using Inheritance

2.2.2 Design Patterns

Some of the design patterns [GHJV94], tackle the problem of separation of concerns. For example, the visitor pattern allows to separate a concern that is spread across a class hierarchy into one class, namely the visitor. In the example of the figure editor, we could use the observer pattern. In the observer pattern we have a subject, the object that is interesting to be observed and the observer, the object that is interested in the changes of the subject. At each place where the state of the subject is updated we notify the observers that have subscribed themselves at the subject. The design is shown in figure 2.3. Using design patterns allows better separation of concerns, but we still have some problems:

- Design patterns are implicitly present in the code, this makes it hard to identify them and to read them.

- The hooks for the design patterns need to be present in the code. In other words, they need to be foreseen, while we also want to implement unanticipated changes in the requirements of the code. Changing the code for installing the design pattern can require major changes in design of the application.

- The code for the hooks is also crosscutting across the other code. For example, the calls to the notify method in the observer design pattern are implicitly placed across the other code.

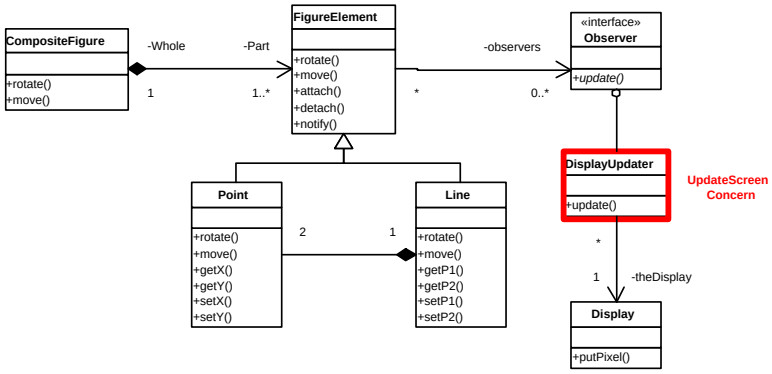

Figure 2.3: Simple Figure Editor: Separation of Concerns Using Design Patterns

2.2.3 Reflection

Computational reflection [Mae87] is another means for achieving separation of concerns [Sul01]. With computational reflection it is possible to add code at the meta-level of the program. For example, with computational reflection it is possible to intercept messages that are sent. These messages are then reified and can be manipulated, before they are eventually received by the object. We can achieve separation of concerns, because the programming happens at the meta-level rather than in the code of that method in the base-level. For the figure editor example we could reify the messages that change state and put code for updating the screen there. Using computational reflection allows for a great deal of flexibility, but also has some disadvantages:

- Computational reflection is not always supported by the programming language. This means that if we want to use it, we are restricted in the choice of our programming language.
- Programming at the meta-level is difficult and can introduce some subtleties.
- Meta-level programming also has some performance issues that need to be taken into account.

2.3 Aspect-Oriented Software Development

Another way for having separation of concerns is using aspect-oriented programming (AOP). AOP provides extensions to already existing programming languages, so that it becomes possible to modularize crosscutting concerns. The word *aspect* is often used to denote a modularized

crosscutting concern. Currently there are many AOP tools in research [KLM+97, BA01, OT01, LOO01]. Most of them are based on the idea of having a language that specifies where the concern is spread across existing code and what the should be done at these points. After the concerns have been specified a *weaver* is employed to weave the aspects into the existing source code, that is the code without the tangled concerns. AOP has also some disadvantages:

- While the understandability of the separate modules is improved, the understandability of the whole program becomes harder. This is because the weaver weaves code at some places where there is no explicit call to that code.
- Debugging of aspect oriented code is harder too, because debugging of the code is done over the weaved code and not in the separate modules.

In the next section we provide an overview of some AOP tools and try to identify their problems and merits.

2.3.1 Issues in Aspect-Oriented Programming Languages

An AOP language is not a stand-alone programming language. It merely complements an already existing language. Most of the time this underlying language is using the object-oriented paradigm, however this is not a requirement. In object-oriented programming the idea is to search for commonalities, and push them up in the hierarchy. With AOP the idea is to specify tangled concerns and represent them as first-class entities.

Definition 5 (Base Program) *The base program is the code where the aspects are applied to.*

When designing an AOP language we must take the following issues into account [EFB01]:

Specification of Aspects

One important aspect of an AOP language is how it defines an aspect. The modular construct it provides to encapsulate aspects. The syntactical constructs that are provided to define how the aspect crosscuts over the base program. The extension mechanisms the AOP language provides to let the aspect evolve. The constructs it provides to improve reusability of the aspect.

Composition Mechanisms

Once the aspects are specified, they need to be composed together in the program. Some issues involved in the composition mechanisms are:

Generality Is the underlying aspect-language domain specific or is it a general-purpose programming language?

Visibility Are aspects visible to each other or is only the underlying code base visible for the aspects?

Conflicts How are conflicts between different aspects resolved?

Behavior Can aspects only add behavior or can they also remove behavior from the underlying code base?

Implementation Mechanisms

The implementation techniques boil down to the difference between static and dynamic composition mechanisms of the aspects:

Static Static composition mechanisms are compilers or pre-processors that weave the code of the aspects at the different pointcuts. They have the advantage that they are fast because the introduction of the code is done at compile-time. The disadvantage is that it is impossible to adapt them at run-time.

Dynamic Dynamic composition mechanisms could be implemented by using reflection and meta-object protocols. Their advantages and disadvantages are the opposite of the static one. The design of a reflective architecture usually involves a tradeoff between the the level of flexibility it provides and the performance.

Software Process

Implementation tools for realizing advanced separation of concerns are one thing, but a methodology for creating aspects is also needed. Another important issue of aspect-oriented programming techniques is the software process they propose. Also the language constructs that enhance the reusability of aspects are a part of the software process. Another issue is the verification of correctness of the aspects and the ability to debug them.

2.3.2 AspectJ

AspectJ [KLM+97] is a language extension for doing aspect-oriented programming in Java. The tool is a consolidation of previous work on reflection and meta-object protocols.

In AspectJ a concern is expressed in terms of an aspect. An aspect is a module that allows the developer to express crosscutting implementations. AspectJ supports the expression of both static and dynamic crosscutting.

Static crosscutting changes the static type signature of a program.

Dynamic crosscutting gives the possibility to add and/or remove behavior from a program at certain points in the flow of the program.

Note that static and dynamic crosscutting differs from static and dynamic composition. In the next subsections we discuss the dynamic crosscutting features of AspectJ.

Dynamic Joinpoint Model

AspectJ allows to change the behavior of software at certain well-defined points in the flow of the execution. These points in the execution flow can be specified using the dynamic *joinpoint* model. *Pointcuts* are a means to specify sets of joinpoints and to refer to values at these joinpoints. Pointcuts can be defined with a name or anonymously. An example of an anonymous pointcut:

```
receptions(void Point.setX(int)) ||
receptions(void Point.setY(int))
```

This defines the point in the execution flow where an object of type Point (i.e., the class Point or a subclass of it) receives a message setX() or setY(). It is also possible to give a name to the pointcuts:

```
pointcut moves(Point p):
    receptions(void p.setX(int)) ||
    receptions(void p.setY(int))
```

Also notice that the pointcut definition moves() has a parameter p, which is bound to the value of the receiver of the message.

Advice

After having defined the places (joinpoints) in the execution flow and the necessary data, it is possible to define the behavior that should be executed at these points. The behavior is defined using advice. There are three different kinds of advice:

Advice Before Executes at the instruction just before the joinpoint is reached.

Advice After Executes at the instruction just after the joinpoint is finished.

Advice Around can be distinguished from the two other types of advice in that they are strictly additive, that is to say that they do not remove any computation from the base program. Around advice on the other hand, has the capability of selectively preempting computation from the base program.

```
abstract aspect Trace
{
  abstract pointcut changes(Object o);

  after(Object o): changes(o)
  {
    System.out.println(o + " has changed!");
  }
}

aspect PointTrace extends Trace
{
  pointcut changes(Object p):
    receptions(void p.setX(int)) ||
    receptions(void p.setY(int))
}
```

Figure 2.4: Example of modular aspects

An example advice logging the movements of the points onto the screen would translate in the following advice:

```
after(Point p): moves(p)
{
  System.out.println(p + " has moved!");
}
```

Reusability

AspectJ introduces aspect modules for encapsulating the definition of the aspects. To promote the reusability of aspects pointcut definitions can be postponed by declaring them as abstract. Using inheritance we can make the aspects concrete by defining the abstract declaration. Figure 2.4 shows an example the reusable abstract aspect Trace. However, the reusability of aspects is restricted, because it is impossible to extend pointcuts that are not abstract, so the PointTrace aspect cannot be extended anymore. We can therefore say that AspectJ only supports anticipated reusability of the aspects.

Composition Mechanisms

Composition is done statically using a modified compiler. If multiple aspects are crosscutting in the same point then it is possible to define a static order

on them by using the dominates keyword. If aspectA dominates aspectB then the advices of aspectA are executed before the advices of aspectB.

2.3.3 Aspectual Components

Aspectual components [LLM99] are proposed as an extension to AspectJ and Adaptive Plug and Play (AP&P) Components. AP&P components [ML98] are used for modelling behavioral composition, but they can also be used for expressing aspects. The problem of AspectJ is that the modular units provided by AspectJ do not provide a generic data model and therefore it is too tightly coupled with the base program.

Figure 2.5 shows the example given in [LLM99]. It shows the implementation of an observer aspect for a game. The problem is that the aspect is bound with the base, namely the class *TicTacToe* to which the aspect is applied[1]. Hard coding the structure of the base program in the aspect harms their reusability. We can say that the AspectJ modules are tightly coupled with the class structure of the base and do not have their own independent structure.

[LLM99] gives following definition for the aspectual components:

Definition 6 (Aspectual Component) *:*

1. *a set of participants forming a graph called the PG (represented by, e.g., a UML class diagram.) A participant is a formal argument which consists of:*
   - *expected features (keyword **expect**)*
   - *re-implementations (keyword **replace**)*
   - *local features (data and operations.)*
2. *aspectual component-level definitions*
   - *local classes, visible only within the aspectual component*
   - *features (data and operations: there is a single copy of each global data member for each deployment)*

We further explain this definition using an example borrowed from [LLM99]. Figure 2.6 shows the implementation of the observer protocol aspect using aspectual components. The components form a modular reusable entity and consists of multiple participants. Each participant consists of a number of *expected* operations, such as the *changeOp* operation. 'Expected' operations can be compared with the abstract methods notion of the usual

[1] In current versions of AspectJ this issue is solved by a mechanism based on interfaces that can have code attached to them.

```
aspect TTTDisplayProtocol {
  static new Vector TicTacToe.observers = new Vector () ;
  static new TicTacToe TTTObserver.game ;
  static new void TicTacToe.attach (TTTObserver obs) {
    observers.addElement( obs ) ;
  }

  static new void TicTacToe.detach (TTTObserver obs) {
    observers.removeElement ( obs ) ;
  }

  static new void TTTObserver.update() {
    board.update(game);
    status.update (game);
  }

  // all methods that change state
  static after TicTacToe.startGame, TicTacToe.newPlayer,
               TicTacToe.putMark,TicTacToe.endGame {
    for (int i = 0 ; i != observers.size(); i++)
      ((Observer)observers.elementAt(i)).update();
  }
}
```

Figure 2.5: Example: AspectJ code harming the reusability, because it is dependent on the base program.

object-oriented languages: they are not implemented in the component itself. 'Expected' operations can also be *replaced* with a new implementation, as a method from a superclass can be redefined in a subclass in the context of inheritance. The previous definition of the replaced operation can be called with the expected() keyword, which is similar to the super-mechanism for accessing the previous definition of the method in the class/superclass relationship.

After defining the components we can deploy them in onto an existing class hierarchy. Deployment is done using a *connector* mechanism. Figure 2.7 shows the code for using the observer protocol aspect to connect the TicTacToe game with two observers. The connector mechanism binds one or more classes to the participants defined in the aspectual component. In the example the class TicTacToe is bound to the Subject participant and both the classes BoardDisplay and StatusDisplay are bound to the Observer participant. In the connection with TicTacToe the 'expected' change operator is bound to the operations startGame, newPlayer, putMark, endGame defined in the TicTacToe class. The 'expected' operation subUpdate from the Observer participant is given in the connector.

Reusability

The reusability of aspectual components improves that of AspectJ, because both the base program and the aspectual components have their own independent hierarchy. The connectors can also be incrementally extended using a technique similar to inheritance.

Composition Mechanisms

The composition mechanism is static and is similar to that of AspectJ. The execution order of the aspects can be manipulated with composite connectors.

2.3.4 Composition Filters

Composition Filters [AT98, OT01] are used to express crosscutting concerns in an application. The idea is that filters are wrapped around a class. Before a message is received by an object it first passes through the input message filters that are wrapped around its class. An incoming message filter either accepts or rejects messages depending on a message pattern and the result of evaluating a boolean expression. The semantics of accepting or rejecting a message depends on the type of the filter. Examples of filter types semantics are given in table 2.1. Analogous to the input message filters are the output message filters for messages that were sent from within a certain object. Figure 2.8 shows the conceptual model of the input and output filters.

```
component ObserverSubjectProtocol
{
  participant Subject
  {
    expect void changeOp(Object[] args);
    protected Vector observers = new Vector();
    public void attach(Observer o)
    { observers.addElement(o); }
    public void detach(Observer o)
    { observers.remove(o); }
    replace void changeOp()
    {
      expected();
      for (int i=0; i<observers.size(); i++)
      {
        ((Observer)observers.elementAt(i)).update(this);
      }
    }
  }
  participant Observer
  {
    expect void subUpdate(Subject s);
    protected Subject s;
    public void update(Subject aSubject)
    {
      s = aSubject;
      expected.subUpdate(aSubject);
    }
  }
}
```

Figure 2.6: Example: Observer/Subject protocol aspect using Aspectual Components

Filter Type	Accept Action	Reject Action
Dispatch	Dispatch message to the target object	Proceed to the next filter
Error	Continue to the next filter	Raise an exception
Wait	Continue to the next filter	The message is kept in a queue until the evaluation expression evaluates to true.

Table 2.1: Example Filter Types

```
connector ObserverSubjectConnectorToTicTacToe
{
  TicTacToe is Subject with
  {
    changeOp = { startGame, newPlayer, putMark, endGame }
  };

  { BoardDisplay, StatusDisplay } is Observer with
  {
    void subUpdate(Subject aSubject)
    {
      setGame((Game)aSubject);
      repaint();
    }
  };
}
```

Figure 2.7: Example: Connecting Observer/Subject protocol aspect to existing classes.

Reusability

In the composition filters model the aspects are represented by the filters. Each filter represents an orthogonal extension, that is to say that each filter extension to a class is considered independent from other filters. The orthogonality eases the composition of the aspects, but it also makes the system harder if not impossible to use for aspect interactions.

Composition Mechanisms

The composition mechanism is based on a modified compiler. The order of the aspects is determined by the order in which the filters are put in the system. The order of the filters is statically determined.

2.3.5 Multidimensional Separation of Concerns

Multidimensional separation of concerns (MDSOC) [OT99] is an extension on previous work done on subject-oriented programming [HO93]. The idea is that it is impossible to encapsulate all concerns in a single decomposition of the program. Each decomposition has different properties and accommodates for different kinds of changes in the software. Decomposing the software in a single dimension causes the code to become tangled and crosscutting as explained in section 2.1. With hyperspaces the idea is to achieve

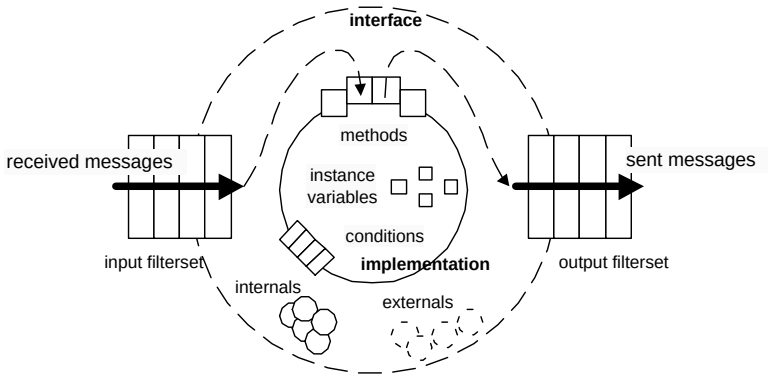

Figure 2.8: Conceptual Model of Composition Filters

three goals with respect to the concerns:

1. Identification:
   the first step in identifying concerns is by selecting them in the source code, naming them and populate the concerns with the units that contribute to them.

2. Encapsulation:
   Encapsulating concerns allows them to be handled as first-class entities and keeps the code of a concern localized.

3. Integration:
   Once the concerns have been identified and encapsulated we must integrate them in order to create the software that contains the multiple concerns.

The items are further explained in the three following subsections.

Identification

Concerns are put in a multidimensional space. Each axis represents a dimension of a concern. Each point on the axis represents a concern in the dimension. By making the dimensions of the concerns explicit and identifying the code units in these dimensions it is possible to find which concerns are affecting code. We can also find the interaction points of the different concerns.

Encapsulation

The matrix of dimensions explained above allows to identify the sets of concerns and overlapping areas between them, but it does not provide an

encapsulation mechanism. The sets of units cannot be seen as encapsulated entities, because the units call upon each other and are too tightly coupled. The units are decoupled with hyperslices. Hyperslices are declared to be *declaratively complete*. This means that a hyperslice must at least provide a declaration for each function and variable that is used by its members. The definition however, does not need to be provided, the items can be declared abstract. By making a hyperslice declaratively complete we state what the hyperslice needs in order to be functional. It also makes the hyperslice loosely coupled, making the unit encapsulated.

Integration

Several hyperslices can be composed and form pieces of working software or new hyperslices, depending on whether there are abstract declarations left. Hyperslices can also give conflicts, for example, when two hyperslices provide the same method. There are several solutions for resolving this conflict:

- one hyperslice overrides the method of the other
- both methods are executed:
  we need to define in what order and how the return value is computed

Hypermodules are used to specify the relationships between the hyperslices and also specify the *composition rules* on how the conflicts are to be resolved. Composition rules for two hyperslices PayRoll and Personnel:

```
hypermodule PayrollPlusPersonnel
  hyperslices: Payroll, Personnel;
  relationships:
    mergeByName;
end hypermodule
```

These composition rules will join the behavior of the methods that have the same name in the two hyperslices together.

Reusability

Hyperslices are composed in a component-oriented fashion. The tool which implements MDSOC for Java, named Hyper/J, includes a tool for composing hyperslices. Reusability of aspects is claimed to be better, because aspects do not rely on a dominant model.

Composition Mechanisms

The composition tool is based on a pre-compiler that translates the hypermodules to regular Java code.

2.3.6 Java Aspect Components

Java Aspect Components (JAC) [PSDF01] is a framework that supports aspect-oriented programming in Java. Aspects can be woven and unwoven at run-time using reflective properties of the language. Unlike the other aspect-oriented programming tools JAC does not propose new language extensions, instead it tries to define a generic architecture. JAC consists of the following program parts:

JAC aspect objects are attached to participants (regular base objects). A JAC aspect object can consist of the following methods:

- Wrapping methods are used to attach behavior before or after a base object method.
- Role methods are used to extend the behavior of one or more base objects with some code.
- Exception handlers are automatically activated when an exception is raised within the base object it is attached to.

Weaver The weaver deploys the JAC aspect objects in the correct base objects.

Composition Aspect Object When multiple JAC aspect components are attached to the same base object then conflicts can arise. Composition aspect objects are employed to resolve such conflicts.

Reusability

JAC aspect objects are claimed to be fully reusable and extensible, because they do not define any aspect pointcuts in the code. In our opinion this is not completely true, because the way the aspect object is activated is not externalized and must be handled in the aspect code. When an aspect object is activated it is passed the method and argument list of the method it is wrapped around. The code of the aspect object has to use this information to get information from the base program. The problem is that the way an aspect should be activated is not externalized and this is tangled in the code of the aspect. This harms the reusability of the aspects.

Composition Mechanism

JAC distinguishes itself from the other aspect-oriented programming tools in that it has a weaver that can install aspects at run-time, rather than at compile-time.

	AspectJ	Aspectual Components	Composition Filters	MDSOC	JAC
Generality	General Purpose	General Purpose	General Purpose	General Purpose	General Purpose
Visibility	Yes	No	No	Yes	Yes
Conflicts	Order	Order	No	Order	Composition Aspect
Behavior	A/R	A/R	A/R	A/R	A/R
Impl. Mechanics	Compile	Compile	Compile	Compile	Run-Time
Reusability	Dependent	Good	Good	Good	Dependent

Table 2.2: Comparison of Aspect Tools (A=Add/R=Remove)

2.4 Comparison

Table 2.2 compares the different approaches to the issues that were described in section 2.3.1. From the comparison we learn that most AOP tools cannot be used for distributed applications or component based applications that require run-time weaving of the aspects.

2.5 Summary

In this chapter we have briefly discussed the problems of concerns that are difficult to modularize. Such concerns decrease the quality attributes of the source code and often result in so-called spaghetti code. Existing language constructs are often insufficient to modularize all concerns. Concerns are often added implicitly by editing the code of the base program instead of adding them in a modularized way. Many tools have been developed to complement existing programming languages with the ability to modularize crosscutting concerns. We have discussed and compared some of these tools. Most of the tools are not able to express dynamic crosscutting concerns.

Chapter 3

Composition Issues

3.1 Introduction

When we need to weave multiple aspects to the same base program then the composition mechanism should allow someone to regulate the interactions and possible conflicts among the aspects. First, we give an overview of the different composition issues that occur when trying to weave aspects in an application. We then identify how these issues are addressed in some of the existing aspect-oriented composition tools.

3.2 Composition Issues

In [PSDF01] an overview of some composition issues is given. Most of them are open issues, that is to say that they are still under discussion in the research community. We have reduced these composition issues to five problems associated with aspect-oriented programming. The composition issues and the illustrative examples used in this section are from [PSDF01]. The AspectJ code from the examples serves to illustrate how the composition issues tend to get tangled in the advice code. This harms the reusability of the aspects in other contexts.

3.2.1 When Activated

The decision to activate an aspect sometimes depends on run-time properties of the application or even on the context of the caller. We give two examples of each of these activation problems:

Property-Dependent Activations

Sometimes an aspect should only be activated when a certain condition is fulfilled. For example, an authentication aspect only needs to get activated if the user has not already been authenticated:

```
aspect authentication {
 abstract pointcut authentications();

 before(): authentications() {
   while (!isAuthenticated()) do
     // authenticate
 }
}
```

Another example is that a display only needs to be updated if the changes to the state of a model affect the view.

Context-Dependent Activations

Some aspects need to perform dynamic context-dependent tests to remain semantically consistent. Consider the example of a counting aspect. The aspect is responsible for counting the number of times it has been activated. The code for a counting aspect that increments a counter each time before the method m1 in class A has been activated.

```
aspect CountingAspect {
  private int counter;
  pointcut pc1(): target(A) && call(void m1());
  before(): pc1() {
    counter++;
  }
}
```

Suppose we know that method m2 in the same class calls m1 ten times. We could optimize the counting aspect by incrementing the counter with ten instead of incrementing it ten times separately. The code for this is shown below:

```
aspect WrongOptimizedCountingAspect {
  private int counter;
  pointcut pc1(): target(A) && call(void m1());
  pointcut pc2(): target(A) && call(void m2());
  before(): pc1 {
    counter++;
  }
  before(): pc2 {
    counter += 10;
  }
}
```

This aspect will not function correctly, because when the message m2 is sent it will increment the counter with ten and continue to increment the counter with ten times one. In order to function correctly we need to specify that the advice attached to the pointcut pc1 should not be activated when the advice attached to pointcut pc2 has already been activated. Code for solving this problem is shown below:

```
aspect OptimizedCountingAspect {
  private int counter;
  private boolean applied = false;
  pointcut pc1(): target(A) && call(void m1());
  pointcut pc2(): target(A) && call(void m2());

  before(): pc1 {
    if ( applied ) { applied = false; skip }
    counter++;
    applied = false;
  }
  before(): pc2 {
    counter += 10;
    applied = true;
  }
}
```

Context-dependent aspects are called *jumping aspects* in [BMD00], because the pointcuts depend on the context in which a component is used.

3.2.2 How Activated

If we consider the optimized counter aspect from above again, then we can see that the aspect has become more dependent on the base program. This makes the aspect less reusable and harder to read. Ideally we would have a simple counter aspect and depending on the run-time or static properties we decide to increment the counter with another value. How the aspect is activated should be fully externalized from the aspect code, so that is it decoupled from the base program.

3.2.3 Choosing Aspects

Sometimes it is useful to have two aspects that handle the same crosscutting concern, but in a different way. For example, depending on the network-load we could decide to alter the compression algorithm. When there is a fast connection we use a compression algorithm that provides better quality, but needs more bandwidth. When the connection is saturated we could use a

compression algorithm that requires less bandwidth, but that delivers worse quality:

```
aspect MediaCompression {
 abstract pointcut pc1();
 before(): pc1 {
   if (Network.getLoad() > TRESHOLD)
     // low-quality compression
   else
     // high-quality compression
 }
}
```

The code above shows how the code for choosing between the two aspects gets tangled in the advice code.

3.2.4 Compatibility

When an aspect is to be woven into an existing application a compatibility conflict can occur. For example, when an logging aspect is woven into the base program, but the base program already contains some logging facilities, then the weaver should be able to detect this and refuse to weave the logging aspect.

3.2.5 Order

Sometimes, two or more aspects need to be woven at the same place in the base program. For example a semaphore aspect and a logging aspect that both need to be wrapped around a single method. In that case it might be necessary to specify which aspect should be handled before the other aspect. We can distinguish between a static order and a dynamic order of the aspects.

Static Order

Static order between two aspects defines an *unconditional* precedence between two or more aspects. The semaphore and logging aspect are an example of a static order, the semaphore aspect always needs to be activated before the logging aspect.

Dynamic Order

Dynamic order between two aspects defines a *conditional* precedence between two or more aspects, that is to say that the order can depend on the run-time properties of an application. For example, if we have an authentication and a logging aspect, then depending on the status of the application

we want to activate the logging before or after the authentication. If a security breach is suspected we decide to log before authenticating, in the other case we decide to log after authentication.

3.2.6 Dependencies

Sometimes one aspect depends on another aspect to perform a certain feature. For example, a billing aspect might need a timing aspect to let the invoice depend on the time the service was utilized. In the case we want to weave the billing aspect we might want to automatically weave the timing aspect with it or report an error when the aspect cannot be found. Also, when the billing aspect is skipped, then the timing aspect can also be skipped (unless there are other aspects depending on it).

3.3 Evaluation of Existing Composition Tools

This section compares how the previously mentioned issues are resolved in some of the aspect-oriented programming tools that we discussed in the previous chapter. They are summarized in table 3.1.

3.3.1 AspectJ

When Activated

Property-dependent activations . When the pointcut that is attached to an advice does not match, then the advice is not executed. Pointcuts can depend on the run-time properties using an if-pointcut. The advantage is that the condition is moved outside of the advice so that the reusability of the aspect is not harmed.

Context-dependent activations are achieved using the control flow based pointcuts. The problem is that the consistency rules are mixed in the pointcuts. Moreover, the consistency rules are more permanent than the pointcuts. In the example from section 3.2.1 we would have to repeat the consistency rule (using the control flow pointcut) each time we reuse the aspect in another base program. So each time the aspect is reused in a different context, the developer that reuses the aspect has to keep the consistency rule in mind.

How Activated

How the aspect is activated can be partially externalized by making use of the pointcut parameters. However, when we would need to convert some of the collected parameters we have to do this in the aspect code, because

Composition Technology	When Activated	How Activated	Choosing Aspects	Compatibility	Order
AspectJ	P/C	Partially Tangled	Implicit	No	Static
CF	P	Tangled	Implicit	No	Static
AC	No	Tangled	No	No	Static
Hyper/J	C	Partially Tangled	No	No	Static
JAC	P/C	Tangled	Yes	Yes	Static Dynamic

Composition Technology	Dependencies	Composition Coding Style
AspectJ	No	Declarative
CF	No	Declarative
AC	Yes	Composition Language
Hyper/J	No	Composition Language
JAC	Yes	Imperative

Table 3.1: Summary of Composition Technologies vs. Composition Issues (P=Property C=Context)

the pointcuts do not allow to put code to convert the parameters in the pointcuts. This harms the reusability of the aspect code.

Choosing an aspect

Choosing an aspect can be simulated in AspectJ by splitting the choices between the different aspects in different advices and specifying which advice should become active in the pointcut that is attached to the advice. This solution however, is not completely satisfying. One could say that the code to choose between several aspects is tangled in the pointcuts of the different aspects.

Compatibility

Compatibility is not supported.

Order

Static Order is supported using the dominates relationship. The dominates relationship is introduced so that if two or more advices need to be woven at the same joinpoint, then the weaver knows in which order the advices should be placed. When an aspect A *dominates* aspect B, then all the advices from aspect A are executed before the advices of aspect B. There are several problems with this solution:

1. The granularity of the dominates relationship is limited to that of a whole aspect. If you consider the problem that an aspect `A` contains advices `a` and `b` and an aspect `B` contains advices `c` and `d`. Then we are unable to express that advice `a` should be woven before advice `c`, and that advice `b` should be woven after advice `d`.

2. The dominates relationship is tangled in the aspect declaration, which makes it harder to reuse the aspect. For example in the case that the aspect is to be reused in an application where the other aspect is not needed.

Dynamic Order Advices are woven at compile-time and cannot be altered at run-time.

Dependencies

Aspect dependencies are not supported.

3.3.2 Composition Filters

When Activated

Property-dependent activations Filters either accept or reject a message. The acceptance or rejection of a message is determined by a set of patterns over the message name together with a condition. The message can be rejected if either the pattern does not match or the condition evaluates to false. Depending on the semantics of the filter the message is then handled by the next filter.

Context-dependent activations are not supported.

How activated

It is not possible to specify how a filter should be activated.

Choosing an aspect

Choosing an aspect can be done by weaving multiple filters and use opposite boolean conditions to choose between one filter or another.

Compatibility

Checking the compatibility is not supported.

Order

Static Order is done by ordering the incoming and outgoing filters. The order of the filters is determined by their order in the composition rules.

Dynamic Order Filters are statically woven into the program text at weave-time and the order cannot be changed at run-time.

Dependencies

Specification of dependencies is not supported.

3.3.3 Aspectual Components

When Activated

Property-dependent activations is not supported.

Context-dependent activations are not supported.

How Activated

It is not possible to specify how an aspectual component should be activated.

Choosing an aspect

Choosing an aspect is not supported.

Compatibility

Checking the compatibility is not supported.

Order

Static Order is done using composite connectors. Connectors can be used to determine a static order for the different aspects.

Dynamic Order is not supported.

Dependencies

Dependencies can be specified by creating a new component that is composed of all the necessary components. Instead of weaving each aspectual component separate, the composed component needs to be woven using a connector requiring all the weaving directions.

3.3.4 Multi-Dimensional Separation of Concerns

When Activated

Property-dependent activations are not supported.

Context-dependent activations are achieved using an extended version of the **bracket** construct in the composition of several hyperslices. The bracket construct can be used to add a call before or after a certain method in a hyperslice. The construct has been extended with a **from** directive to determine when the bracket is applicable using the origin of the call site. The call sites are partitioned in hyperslices.

How Activated

Aspects that are bracketed around a method can either be given the original parameters or the name of the class and the name of the method. It is not specified how arguments can actually be converted, so this would need to be done in the aspect code, which harms the reusability.

Choosing an aspect

Choosing an aspect is not supported.

Compatibility

Checking the compatibility is not supported.

Order

Static Order is done using the order relationship. The order is specified on the level of methods and helps in defining a partial order when several methods are merged together.

Dynamic Order is not supported.

Dependencies

Specifying dependencies is not supported.

3.3.5 Java Aspect Components

JAC supports all the issues using composition aspects, except for the issue of how to activate the aspect. However, JAC does not provide any syntactical constructs to support these issues and the issues are implemented in an imperative coding style. This means that the behavior of the composite aspect objects becomes hard to read, and therefore it is more difficult to predict the behavior of the overall system. In our opinion the composition issues should be written in a language that declaratively specifies how the composition issues should be solved, rather than in an imperative coding style. Such a declarative coding style would enhance the readability and understandability of the system.

The activation of the JAC components is not externalized and is tangled in the aspect code as we already explained in section 2.3.6. This harms the reusability of the aspects.

3.4 Proposed Solution

In this section we describe what features the composition technologies should provide in order to successfully handle the issues discussed in section 2.3.1 and section 3.2 in a dynamic context:

- Static vs. Dynamic Weaving
  In [RV97, Bol99, Sul01] several examples are given why dynamic weaving of aspects is preferred over a static weaving method for some

applications. Consider for example a component system that allows someone to plug and unplug components at run-time. Crosscutting concerns, such as logging can also be seen as components. A second application where dynamic aspects are useful is in the context of distributed systems. For example, when an object is transported to another device the object needs to consider different resources and services that are provided by its new environment. The adaptations to its new environment can be seen as aspects, but the aspects depend on the environment where the object is living. This means that new aspects sometimes have to be woven when an object moves from one device to another. Considering such applications we choose for a composition mechanism that allows dynamic weaving of aspects.

- Composability of Aspects
  The composition mechanism needs to be able to compose different aspects together [ATB00]. This means effectively that if you take two aspects and you combine them together that you have a new aspect that is again reusable.

- Aspect Conflicts
  When different aspects are composed together they can cause conflicts [PTC00, Pul00, PSDF01, BMD02]. In section 3.2 we discussed some of these issues in more detail. The composition mechanism needs to provide mechanisms to resolve or avoid conflicts in a declarative coding style and untangled from the aspect code.

- Visibility of Aspects
  To resolve conflicts, an aspect sometimes needs to be able to access and therefore be aware of the presence of the other aspects in the application.

- Reusability
  To improve the reusability of the different aspects, both the aspect code, the composition issues and the aspect activation code should be fully separated.

- Composition Language
  To improve the readability and the understandability of the full program, the composition of aspects should be declaratively specified rather than in an imperative way.

3.5 Conclusion

In this chapter we discussed some composition issues that rise when we want to weave one or more aspects into a base program. We tried to identify how

these issues are solved in some existing composition tools for aspect-oriented programming. A summary is shown in table 3.1. Most of the issues are not supported. The issues that are supported by the programming tool are often crosscutting the aspects or are solved implicitly. These problems affect the reusability of the aspects. In section 3.4 we described what features the composition mechanism should provide to solve these composition issues in a dynamic context.

Chapter 4

Composite Aspect Objects

4.1 Introduction

In this chapter we introduce a new model for achieving advanced separation of concerns based on run-time weaving of the aspects. The design of our model is based on the solution we proposed in section 3.4.

4.2 The Model

In this section we introduce the notion of a *composite aspect object*. In composite aspect objects the code of a crosscutting concern is encapsulated in an object. The specification of how the concern is crosscutting the base program is put in a separate module, called an activation scheme.

4.2.1 Composite Aspect Objects

When an object receives a message in a language with a reflective meta-architecture, then the meta-object searches for the code for handling the message that was sent. It does this by locating the source code and executing that code. In the composite aspect object model we extend this by searching the code of multiple objects instead of one object. Each of these objects represents a software concern that should be executed at that point. Multiple composite aspect objects can point to the same code of a single object. This means that the code of a single object can encapsulate the code of a concern that is crosscutting over multiple other objects. The composite aspect object is living at the meta-level of the language, but has an object identity associated with it, so that base level objects can send messages to it. Figure 4.1 shows what happens at the meta-level when a message is sent to the composite aspect object. The composite aspect object provides several services:

- locating and executing the code of the objects.

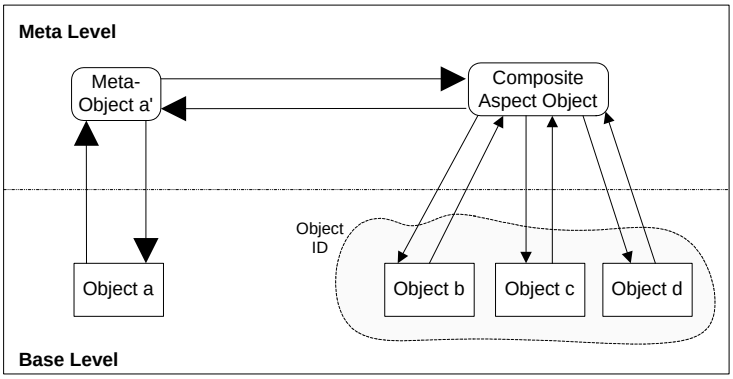

Figure 4.1: Example Model of a Composition Aspect Object

- providing the necessary parameters so that the code can execute, hence the composite aspect object acts as an adapter to activate objects that have a different protocol.
- determining the order in which the objects are activated by sending the objects a message. The order of the aspects can be determined dynamically so that the order can depend run-time values.
- exposing details from the execution environment to the objects if needed. For example, the sender of the message.

These services are provided at run-time and are therefore all adaptable at run-time.

Running Example To make the model a bit more concrete we use a running example for the next subsections. Consider an object representing a queue. The queue-object has four operations: push:, pop, top and isEmpty. The queue can be accessed concurrently by different processes and we would like to log the operations on the queue. The synchronization aspect and the logging aspect are crosscutting the functionality provided by the queue. The structure of the composite aspect object is shown in figure 4.2.

4.2.2 Aspects

In our model an aspect is represented by two entities:

1. The concerns that are crosscutting the program are encapsulated in regular objects that live at the base level. Objects that are part of a composite aspect object are called *part-objects*. Each part-object has a unique *identifier* within the composite aspect object and implements a software concern from the problem or solution domain of

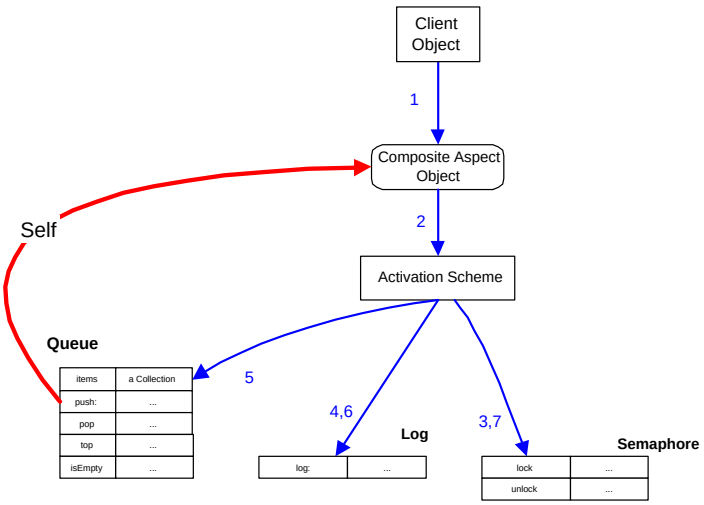

Figure 4.2: Queue Example using a Composite Aspect Object

the application. One part-object can be put in multiple composite aspect objects, this makes it possible to modularize concerns that are crosscutting multiple objects. The part-objects have a protocol that is specific to the software concerns they implement. Representing the crosscutting concerns as objects has the advantage that we can change their implementation at run-time by plugging, unplugging or replacing them in the composite aspect object. Furthermore, we can extend the functionality of the crosscutting concerns using regular object-oriented techniques such as inheritance and aggregation.

2. The way the services of the composite aspect object are configured determine how the functionality implemented in the part-objects are crosscutting the program. The configuration of these services is determined by what we call an *activation scheme*. An activation scheme is a first class entity, represented as an object so that it can be adapted to changes in the environment.

It is important to note in our model that both the part-objects and the activation scheme are represented as first class entities. Since both entities define an aspect we can say that our aspects are adaptable at run-time. Also note that our aspects are based on objects rather than classes. We can take one object and adapt its behavior rather than having to change its class for changing the behavior.

Running Example The activation scheme describes how the queue-, semaphore- and logging object are activated when the composite aspect object receives a message. An example activation scheme for the method

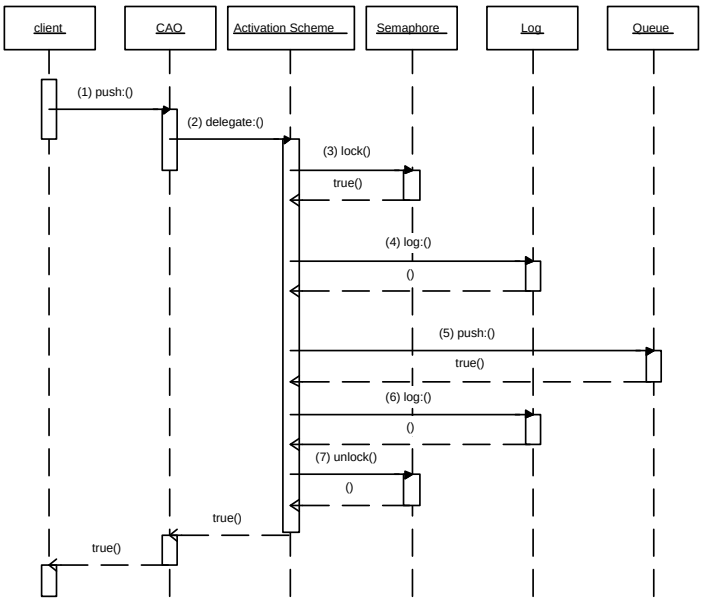

Figure 4.3: Activation Scheme for Figure 4.2

push: for the above example is shown in figure 4.3. The activation scheme can be determined dynamically. We could for example add a condition to each activation sequence, so that when the condition evaluates to true we execute the activation and when it evaluates to false we skip to the next activation sequence.

4.2.3 Weaving

There are several possibilities to apply the composite aspect object depending on the time the crosscutting concerns are identified:

Compile-Time Installation

When the crosscutting concerns are identified at compile-time we can change the meta object of the base object, so that it constructs a composite aspect object that includes the base and the objects that implement the different crosscutting concerns, instead of creating the base object immediately.

Running Example When a new queue is constructed we return a composite aspect object that contains the queue, a semaphore object and a logging object as part-objects. Figure 4.4 shows an example of how the constructor could be implemented in this case.

```
Queue class>>new
  | compositeAspect scheme queue lock logging |
  compositeAspect := CompositeAspect new.
  scheme := QueueScheme new.
  "set object that fulfills the Queue role"
  scheme queue: super new.
  "set object that fulfills the Semaphore role"
  scheme lock: Semaphore new.
  "set object that fulfills the Logging role"
  scheme logging: Log new.
  "set the activation scheme"
  compositeAspect strategy: scheme.
  ^compositeAspect.
```

Figure 4.4: Example Implementation the Constructor of the Queue in Smalltalk

Run-Time Installation

When the crosscutting concerns need to be applied at run-time we can create a composite aspect object at run-time and replace all existing objects with the composite aspect object. In Smalltalk this can be achieved using the become: operation. The become: operation is a primitive method understood by all objects. When it is sent to an object it changes all the references to the receiver of the message to the object that was passed as a parameter. We realize that this is a strong requirement on the language. An alternative that is less restrictive is to change the constructors of all classes so that they always return a composite aspect object that contains an instance of the class. Once we have a composite aspect object we can change its structure at run-time and thus adapt the composite aspect object (i.e., add the objects that implement the crosscutting concerns and change the activation scheme so that it reflects how the part-objects are crosscutting the behavior).

4.2.4 The Notion of Self

An important aspect of the part-objects is their binding of "Self" in a composite aspect object. There are two ways to bind the "Self"-value:

1. rebind "Self" to the composite aspect object.
2. keep "Self" bound to the part-object.

Below we show an example that shows that both possibilities to bind "Self" are useful in one composite aspect object. Consider an object from which we want to log the different messages that were sent to the object. We compose

a composite aspect object that consists of two part-objects, an object that implements the logging aspect and the object that we want to log. If we want to log the messages that were sent from within the object itself then we need to have its "Self" bound to the composite aspect object. However, binding "Self" of the logging object to the composite aspect object is not desired, because we would have to pollute the protocol of the composite aspect object with the the protocol of the logging object. We do not want to do this, because in this example the composite aspect object does not compose the concept of logging, but rather the concept of the object that we are logging. Since the two possibilities are common we allow both in our model.

Running Example The composite aspect object is composed of three objects, but the concept that the composite aspect object composes is determined by one single object, the queue object. Furthermore, we might want to log the operations done that are activated by self-sends. For this reason we decide to bind the "Self" of the queue object to that of the composite aspect object.

4.2.5 Adapting the Composite Aspect Object at Run-Time

Imagine that the queue object is passed over the network to a device that has limited memory resources. Therefore, we decide to restrict the size of the queue. When an element is pushed onto the queue when it is at its full capacity we decide to throw an OverflowException. The aspect for throwing exceptions at the correct time depends on the resources available in the machine where the object will live. Hence, the aspect needs to be woven each time the object migrates to another device. We must also throw an exception when a pop-operation is performed while there are no elements in the queue. We encapsulate the code of the aspect for throwing and checking the exceptions in a class QueueError that has three methods:

1. willOverflow
   returns true or false depending on the memory available in the machine where the object lives of the queue.

2. willUnderflow
   returns true when there are no more elements in the queue and false when there is at least one element in the queue.

3. throw:
   takes an object representing the exception as parameter that needs to be thrown.

When the composite aspect object is deserialized we can weave the instance of the QueueError class to handle the change in memory resources that is

Activation Sequence	What	Activation Condition
1	queueError throw: OverflowException	error willOverflow
2	semaphore lock	true
3	logging log: 'push: accessed'	true
4	queue push: (args at: 0)	true
5	logging log: 'push: terminated'	true
6	semaphore unlock	true

Table 4.1: Adapted Activation Scheme for push: message

specific to the device by adding it to the composite aspect object and changing the activation scheme so that the exceptions are checked and activated at the correct moment:

```
DynamicWeaver>>adaptForErrorHandling: compositeAspect
    | oldStrategy newStrategy |
    compositeAspect add: QueueError new.
    "retrieve the old activation scheme"
    oldStrategy := compositeAspect strategy.
    "convert the old activation scheme to throw exceptions"
    newStrategy := oldStrategy withErrorHandling.
    "Set the new activation scheme"
    compositeAspect strategy: newStrategy.
    ^compositeAspect.
```

4.3 Conclusion

In this chapter we introduced the notion of a composite aspect object to achieve advanced separation of concerns. This model is dynamic and aspects can be added, removed and replaced at run-time. The model has several advantages:

- The activation scheme can be replaced so that different schemes can be chosen and adapted to the changes in the composite aspect object.
- The code of the crosscutting concerns is encapsulated in regular classes. It is the activation scheme that defines how the crosscutting concerns should be activated in the specific context. Hence, the code of the crosscutting concerns can be reused in different contexts, but with an activation scheme that is adapted to the context of its use.

A disadvantage of the model is that because the activation scheme is determined dynamically each time a message is received that this may have

a negative impact on the performance. We believe this negative impact on the performance is minimal, because we can use more static versions for activation schemes that do not have dynamic properties or optimize them using caching.

This model forms the basis for our weaving technology. The next step is to define an activation scheme that expresses the pointcuts of the part-objects in a single object and how multiple aspects relate together to help us resolve the composition issues explained in chapter 3.

Chapter 5

Composing the Composite Aspect Object using Logic Metaprogramming

5.1 Introduction

In the previous chapter we introduced composite aspect objects as a meta-object that is composed of part-objects and an activation scheme. An activation scheme defines how messages received by a composite aspect object should be processed by the part-objects of which it is composed. We described the activation scheme as a black-box entity that determines how and when certain part-objects had to be activated. In section 3.4 we pointed out that we could benefit from a declarative language for expressing the composition of different aspects. In this chapter we discuss a possible implementation of an activation scheme using logic meta-programming rules to determine the activation sequence. We choose for logic metaprogramming, because of the inherent declarative nature. Another advantage of logic metaprogramming is the ability to inspect the structure of our program, which can be useful to express dynamic aspect compositions depending on the dynamic program structure. The next three sections serve as an introduction to logic metaprogramming. They are followed by a discussion of an activation scheme expressed using logic metaprogramming.

5.2 What is Declarative Metaprogramming

When we write programs that act on other programs then we are doing *meta-programming*. Hence, programs that are acting on other programs are called *meta-programs* and the programs that are acted upon are called base programs. We can distinguish between at least two types of metaprogramming:

1. Compile-time Metaprogramming:
   The meta-program is employed at compile-time to change other programs. They usually come in the flavor of a preprocessor or are integrated with the compiler of the language. An example of such a system is the template-system in C++ [Str97].

2. Run-time Metaprogramming:
   The meta-program is employed at run-time and are usually integrated in the programming language, such as in Smalltalk [GR83]

Declarative metaprogramming combines a declarative language used at the meta-level together with a certain base language (e.g. an object-oriented language).

5.3 Logic Metaprogramming

Logic metaprogramming is a particular instance of declarative metaprogramming. The declarative language that is used at the meta level is a logic programming language (e.g. Prolog).

The base level is the program where the meta-program is reasoning about. In logic metaprogramming this level is expressed as facts, rules and terms at the meta-level. Meta level programs are used to *reason* and *manipulate* the base-level.

In this section we discuss two different implementations of logic metaprogramming languages, but first we give a short introduction to logic programming.

5.3.1 Logic Programming

Logic programming is about implementing *relations*. A logic program is a collection of horn clauses. A horn clause is of the form:

$H \leftarrow B_1, ..., B_n$

H is called the head of the rule, while the $B_1, ..., B_n$ is usually called the body of the rule. H is said to be proven if $B_{1..n}$ is proven to be true. When there is no body and thus $n = 0$, then H is unconditionally true. Such horn clauses are usually called *facts*, while horn clauses with $n > 0$ are called *rules*. Results are computed by querying the system. A query is usually of the form $\leftarrow Q$. When querying the system we are in fact trying to construct a proof by contradiction. When a solution can be found for the query we say that the query succeeds; when no solution is found we say that the query failed. A query is computed by an algorithm that is called *resolution*. Resolution is based on a special variable binding mechanism that is called unification.

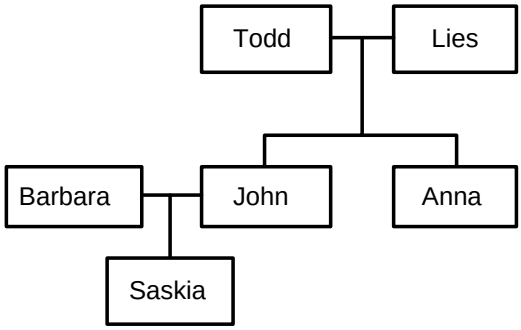

Figure 5.1: Family Tree

Example The syntax is similar to the rules describes above. Differences are the ← is represented as "if" and the logic variables that start with a "?". Consider the family tree shown in figure 5.1. The tree is expressed as following facts:

```
married(todd, lies).
parent(todd, john).
parent(lies, john).
parent(todd, anna).
parent(lies, anna).

married(john, barbara).
parent(john, saskia).
parent(barbara, saskia).

female(barbara).
female(anna).
female(lies).
male(todd).
male(john).
```

We now define some rules that allow us to identify the father and mother relationships in the family tree.

```
father(?Parent, ?Child) if
  parent(?Parent, ?Child), male(?Parent).
mother(?Parent, ?Child) if
  parent(?Parent, ?Child), female(?Parent).
```

The father-rule should be read as ?Parent is the father of ?Child if there exists a parent-relationship between ?Parent and ?Child and there exists a male-relationship for the ?Parent. The mother-rule is constructed similarly. Rules can be used to construct other rules, which is shown in the grandfather-rule shown below:

```
grandfather(?Grandfather, ?Grandchild) if
  parent(?Parent, ?Grandchild),
  father(?Grandfather, ?Parent).
```

When multiple rules with the same head are in the rule repository, then the inference engine tries to construct a proof by trying out all the rules in the order they are put in the repository. Rules with the same head are expressing the logical OR-operator. The example shown below is a recursive rule for defining the ancestor-relationship between different persons:

```
ancestor(?Person, ?Ancestor) if parent(?Ancestor, ?Person).
ancestor(?Person, ?Ancestor) if parent(?Parent, ?Person),
                                ancestor(?Parent, ?Ancestor).
```

Note that the non-recursive rules must be listed first in the repository. This is necessary in order to stop the recursion.

Launching the query:

```
if ancestor(saskia, ?Ancestor)
```

produces the following answers using backtracking:

- $?Ancestor \leftarrow john$
- $?Ancestor \leftarrow barbara$
- $?Ancestor \leftarrow todd$
- $?Ancestor \leftarrow lies$

For more information on logic programming we refer to [Fla94].

5.3.2 SOUL

The acronym SOUL stands for Smalltalk Open Unification Language, a logic programming language that has been implemented and integrated with Smalltalk [Wuy98]. SOUL is used for reasoning about the structure of object-oriented programs. An example use of SOUL is the extraction and synchronization between design information and the code of the program. SOUL is actively available at run-time in the Smalltalk image.

SOUL provides a layered set of rules to reason about the base language structure:

logic layer: contains the predicates that add core logic-programming functionality, such as list handling, arithmetic, program control, repository handling,...

representational layer: this layer reifies some of the concepts from the base language (e.g., class, superclass, methods and instance variables).

basic layer: this layer adds more predicates to facilitate reasoning about the base code (e.g., parse tree traversal, typing, code generating, accessing code and other auxiliary rules). This layer is necessary, because the representational layer only provides the most primitive information.

design layer: groups all predicates that express particular design notations, such as programming conventions, design patterns and UML class diagrams.

Examples

As an example we will describe the class-rule, that interacts with the underlying Smalltalk image. The rule can be used to retrieve classes from the base level or verify their existence. The class-rule is the logical representation of a class in the logic meta level.

```
class(?C) if
    atom(?C),
    [Smalltalk includes: ?C].

class(?C) if
    var(?C),
    generate(?C, [Smalltalk allClasses]).
```

The first rule handles the case where the logic variable has been assigned a value to validate if the class is available in the Smalltalk image:

- atom(?C)
  checks to see if there is a value bound to the ?C variable
- `[Smalltalk includes: ?C]`
  Uses the language symbiosis to check if the Smalltalk image includes the class bound to the ?C variable. Note that the value of the block is the value of the last method-call that is executed in the block. When the block is used in the body of the rule then it has to return a boolean value.

The second rule handles the case the the logic variable is unbound and generates the classes that are available in the Smalltalk image.

- var(?C)
  checks to see if there the variable ?C is unbound.
- generate(?C, [Smalltalk allClasses])
  the generate-predicate binds the values from a collection (generated here by the "Smalltalk allClasses"-command) one by one, while backtracking, to the variable ?C.

```
class Stack {
int pos = 0 ;
Stack() {
contents = new Object[SIZE];}
public Object peek ( ) {
return contents[pos]; }
public Object pop ( ) {
return contents[--pos]; }
... }
```

Figure 5.2: Regular Java Code for Stack Implementation.

```
class(Stack).
var(Stack,int,pos,{int pos = 0;}).
constructor(Stack,[],{public Stack()},
{contents = new Object[SIZE]; }).
method(Stack,Object,peek,[],
{public Object peek()},{return...}).
method(Stack,Object,pop,[],
{public Object pop()},{return...}).
...
```

Figure 5.3: Stack Implementation using Logic Propositions.

In current research SOUL is extended with basic predicates to facilitate the construction of aspect-oriented languages for specific domains.

5.4 Aspect-Oriented Logic Metaprogramming

5.4.1 TyRuBa

The acronym TyRuBa stands for Type-Oriented Logic Metaprogramming for Java[1]. The principle of TyRuBa is to use a logic programming to manipulate types at compile-time [DD99]. TyRuBa can be distinguished from SOUL in that TyRuBa is a pre-compiler, while SOUL is actively available in the Smalltalk image at run-time. Base language programs are represented as a set of logic propositions. One of the most important features of TyRuBa is the use of *quoted code blocks*[2] into the logic rules. Quoted code blocks allow pieces of Java code to be used as terms in the logic rules. Quoted code blocks are surrounded with curly braces. Figure 5.3 shows a set of logic propositions that represent the Java program shown in figure 5.2. Eventually the TyRuBa system can use the logic propositions in figure 5.3 to constructs the regular Java code.

TyRuBa can be used to do aspect-oriented programming[DD99]. Consider the aspect of synchronization. The method pop() with the synchronization concern tangled in the code of the method is shown in figure 5.4.

With TyRuBa we can separate the synchronization code from the base code as is shown in figure 5.5. The COOL_allRequired-rule is used to generate a combined condition from all the required-rules. The conditions are then combined using a conjunction.

Depending on the instructions given to the TyRuBa system it will compose the aspect code with the base code using the logic inference process.

[1] Java is a registered trademark of Sun Corporation

[2] Quoted code blocks are now also available in SOUL

```
Object void pop() {
  synchronized (this) {
    while(this.isEmpty()) {
      try {
        wait();
      } catch (InterruptedException e) {  }
    }
    return contents[--pos];
  }
}
```

Figure 5.4: POP method with synchronization tangled

```
method(COOL,?class,?Return,?name,?Args,?head,{
  synchronized (this) {
    while(?condition) {
      try {
        wait();
      } catch (InterruptedException e) {  }
    }
    ?body;
  }
}) if method(JCore,?class,?Return,?name,?Args,?head,?body),
      COOL_allRequired(?class,?name,?condition).

required(Stack, pop, {this.isEmpty()}).
```

Figure 5.5: Separation of the synchronization aspect

The code for the aspects is separated from the base program, because they are encapsulated in different logic facts. Querying the TyRuBa system for JCore method infers a regular non-synchronized version of the stack, while querying for a COOL method infers a synchronized version of the Stack.

5.4.2 Aspect Specific Languages

With aspect-oriented logic metaprogramming we can provide a framework that allows developers to implement their own aspect-specific languages (ASL) [BMD02]. An ASL has the advantage over other approaches to aspect-oriented programming in that they allow to create constructs that are closer to the problem domain of the aspect language. Hence it is more natural to specify the aspects for a particular application in an aspect-specific language. The framework provides several primitive weaving rules that can be used to create the ASL. In the case that the ASLs are not fully orthogonal to each other we can have conflicts. Since all ASLs are using the same primitive rules for instructing the weaver one can specify how the different ASLs should cooperate together.

Imagine that we have the primitive weaving rules shown in table 5.1. We can construct a small aspect specific language for tracing the execution of a program using the rules shown below:

```
adviceBefore(method(?class,?selector),
             { Logger log: 'Enter ?class>>?selector'
                      for: thisObject }) if
  logMethod(?class,?selector).

adviceAfter(method(?class,?selector),
            { Logger log: 'Exit ?class>>?selector'
                     for: thisObject }) if
  logMethod(?class,?selector).
```

The aspect-specific language we have constructed has one language construct, namely logMethod. We can specify the methods to log by specifying logMethod-rules such as for example: `logMethod([Pipe],drain:)` that specifies that we want to trace the method drain: in class Pipe. The logMethod-rule can also contain a body. For example, if we want to trace the method drain: in all the subclasses of the class Pipe, then we could define the logMethod-rule as:

```
logMethod(?subclass, drain:) if
  subclass([Pipe], ?subclass)
```

More complex ASLs can be constructed similarly.

Rule	Explanation
adviceBefore(?m, ?c)	execute the code bound to the variable ?c before the method bound to the variable ?m
adviceAfter(?m, ?c)	execute the code bound to the variable ?c after the method bound to the variable ?m

Table 5.1: Primitive Rules for Instrumenting the Weaver

5.5 Logic Activation Scheme

In most of the current work done on logic metaprogramming the logic rules have been employed at compile-time to reason about and change the structure of the programs. In this section we describe how we can apply the logic meta-rules to make run-time decisions. In chapter 4 we discussed that aspects are defined as part-objects implementing a crosscutting concern and an activation scheme that determines how the services provided by the composite aspect object should be configured. Hence, the activation scheme can benefit from a declarative language, because of the declarative nature of the activation schemes. Also, since we want to express a decision schema a programming language gives us more flexibility and expressivity. As our declarative language we use a logic programming language.

The logic activation scheme consists of different modules of rules:

- Aspect Modules
  - Activation Rules
  - Internal Rules
  - Compatibility Rules
- Aspect Configuration Module
- Order Module
- Aspect Activation Modules

Each of these set of rules are explained in the following subsections. Figure 5.6 shows how the modules are layered together.

5.5.1 Aspect Modules

Aspect modules define **how** and **when** part-objects are activated. A logic activation scheme can contain multiple aspect modules. Aspect modules are associated with one or more part-objects and are implemented with the part-objects. An aspect module consists of three types of rules:

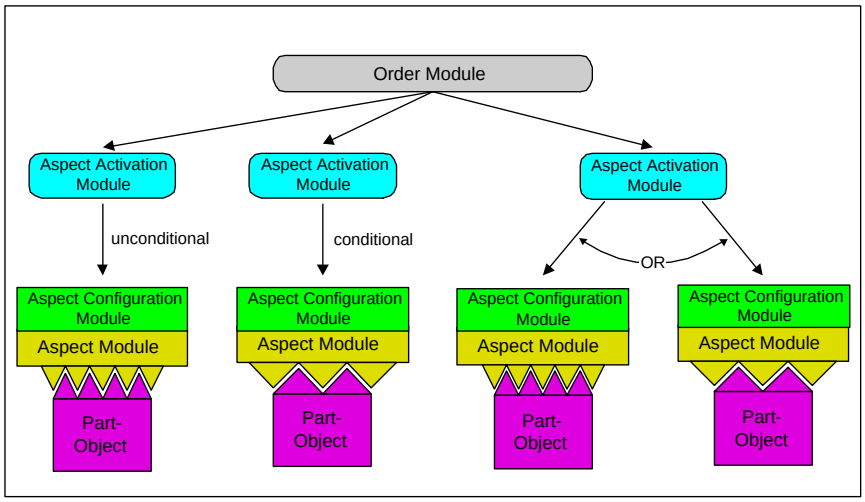

Figure 5.6: Layering of the Modules in a Logic Activation Scheme

Activation Rules

The activation rules are resolved when either a message is sent to a composite aspect object or when an exception occurs while processing a message. The activation rules provide the interface between the composite aspect object and the aspect modules. There are four different types of activation rules that specify how the part-objects are activated:

1. before(+?receivedSel, +?receivedArgs, -?partName, -?sel, -?args)
   the before-rules specify what part-objects should be activated before the base functionality is executed.

2. base(+?receivedSel, +?receivedArgs, -?partName, -?sel, -?args)
   the base-rules specify the core functionality that should be executed. The base-rules give the possibility of spreading the core functionality over multiple part-objects.

3. after(+?receivedSel, +?receivedArgs, -?partName, -?sel, -?args)
   the after-rules specify what part-objects should become active after the base functionality is executed.

4. catch(+?exc, +?receivedSel, +?receivedArgs, -?partName, -?sel, -?args)
   defines how exceptions, that are generated within the composite aspect object, should be handled. When no rule matches the exception that was generated the exception is thrown outside to the caller.

The "+" sign in front of the logic variables means that the rule is inferred with the logic variables bound to a value. The logic variables with a "-" sign are left open and are searched for when inferring the rules. The meaning

of the logic variables is shown in table 5.2. The rules are inferred at runtime after a well-defined event has occurred. Depending on the results of inferring the rules the composite aspect object activates zero, one or more part-objects. When multiple results are computed by inferring the activation rules, then all the results are used to activate the part-objects. Table 5.3 defines when the rules are activated.

Example To make the activation rules more concrete we work out an aspect module for a simple caching aspect. The aspect module covers the activation of two part-objects. One part-object fulfills the role of an object that needs to be cached and has the identifier cached. The other part-object is responsible for caching the results of methods and has the identifier cache.

```
base(?selector, ?args, cache, retrieve:withArgs:, ?partArgs) if
  cachedSelector(?selector),
  inCache(?selector, ?args),
  append(<?selector>, ?args, ?partArgs).

base(?selector, ?args, cached, ?selector, ?args) if
  not(inCache(?selector, ?args)).

after(?selector, ?args, cache, store:selector:args:, ?partArgs) if
  cachedSelector(?selector),
  not(inCache(?selector, ?args)),
  result(cached, ?selector, ?result),
  append(<?result, ?selector>, ?args, ?partArgs).
```

Looking at the rules we can distinguish between two cases:

1. the method that is called is in the cache
   the first rule is applicable and will activate the part-object named cache, to retrieve the value that was stored in the cache, by sending the message retrieve:withArgs:.

2. the method that is called is not in the cache

   (a) the second rule is applicable and activates the part-object named cached, by delegating the message that was received, to compute the value.

   (b) The third rule is activated right after the second rule and will activate the part-object cache to store the value computed by the second rule the cache.

Variable	Semantics
?exc	Exception that occurred while processing a message
?receivedSel	Selector received by the composite aspect object
?receivedArgs	Arguments received by the composite aspect object
?partName	Name of the part-object that has to be activated
?sel	Name of the selector that has to be activated
?args	Argument list that needs to be provided for activating the object

Table 5.2: Semantics of the variables in the activation rules.

Rule	When
before	The composite aspect object received a message
base	The composite aspect object received a message and after the before-rule
after	The composite aspect object received a message and after the base-rule
catch	An exception was thrown while processing a message
check	A part-object has been added or removed from the composite aspect object

Table 5.3: Rule Activation Points.

Internal Rules

The internal rules usually determine conditions on the part-objects and can be used while resolving the activation rules.

Example In the caching example the inCache-rule is an example of an internal rule. An example implementation of the inCache-rule is shown below:

```
inCache(?selector, ?args) if
  partObject(cache, ?cacheObj),
  [?cacheObj isCached: ?selector withArgs: ?args].
```

The rule searches for the part-object that fulfills the role as cache and uses the symbiosis of the logic metaprogramming language to check of the part-object has the result associated with the selector and arguments in cache.

Compatibility Rules

It is possible to change the structure of a composite aspect object at runtime. When we change the part-objects in a composite aspect object we might want to check if the changed object still provides the required services that are required by the aspect modules and check the compatibility of the new aspects. For this reason we propose to add compatibility rules to the aspect modules. The compatibility rules are of the form:

check(+?partName, +?partObject)

The rules are activated each time a part-object is added, removed or replaced from the composite aspect object. When the compatibility rule in the composite aspect object cannot be inferred then the part-object is rejected and an exception is thrown.

Example The rules of the caching aspect module presumes that the part-object that fulfills the role as cache understands the messages send to it. We can check the compatibility with the rule below:

```
check(cache, ?cacheObj) if
  instanceOf(?cacheObj, ?cacheClass),
  understands(?cacheClass, retrieve:withArgs:),
  understands(?cacheClass, store:selector:args:),
  understands(?cacheClass, isCached:withArgs:).
```

The first rule in the body searches for the class from the part-object named cache. The subsequent rules are part of the SOUL reasoning library and are used to check if the class has a method that associated with a certain message.

5.5.2 Aspect Configuration Module

We discussed in section 5.4 that we can create aspect specific languages using logic metaprogramming for static aspects. We can do the same to compose the composite aspect object. An aspect module then defines an aspect specific language. The rules that configure the aspect module cannot be included in the aspect module itself, because they would harm the reusability of the aspect modules. Therefore we propose to separate the configuration rules in a separate aspect configuration module.

Example The caching aspect module needs to be configured by defining which selectors should be cached in the composite aspect object. For example if we want to cache the methods read: and translate: we add the rules:

```
cachedSelector(read:).
cachedSelector(translate:).
```

5.5.3 Order Module

When multiple aspects need to be woven at the same place, then conflicts on the order of the aspects can exist. For this reason we have dominates-rules which are similar to the dominates relationships as used in AspectJ. However, the dominates-rule is inferred at each event, which means that the relationship can depend on dynamic information put in the body of the rule. One problem is that the logic engine can generate multiple solutions for the ordering. To overcome this problem we choose to take the first order that has been generated.

dominates(?aspectBefore, ?aspectAfter)

5.5.4 Aspect Activation Module

Sometimes we want to fully deactivate a module depending on some conditions. For example, when the logging aspect has been deactivated. For this reason we introduce aspect activation modules. An aspect activation module declares when a module should become active. Aspect activation modules contain activation-rules. The head of an activate-rule contains the name of the aspect module that it regulates. The body of an activate-rule defines what conditions should be met before the aspect module can become active. An active-rule without a body unconditionally activates the aspect module.

active(?aspectModule)

5.6 Run-time Reasoning Library

In the previous section we have introduced the logic activation scheme. Logic rules are inferred at run-time to determine the activations of the composite aspect object and check the compatibility of the part-objects. SOUL comes with rules to reason about the static information about the code (such as the class-hierarchy), but does not provide rules for reasoning about the run-time environment. In this section we introduce a small library of rules that makes it possible to reason about the run-time environment. As such, it represents the dynamic information can be used to program the modules from the logic activation schemes.

5.6.1 Typing

We have two typing rules that are for checking the type of objects living in the environment:

1. instanceOf(?obj, ?class)
   expresses the type-relationship between an object and its class.

2. kindOf(?obj, ?class)
   expresses the kind-of-relationship between an object and a class.

Remember that we can use logic rules in different ways so we can search for all objects of a certain class living in the environment.

5.6.2 Collaborators

Often objects need to collaborate with other objects in order to provide the requested services. We have three rules that allow us to inspect the internal collaborators of an object:

1. field(?obj, ?varName, ?refObj)
   The field rule expresses the relationship between an object, its instance variable name and the object the instance variable refers to.

2. accessor(?obj, ?varName, ?sel)
   instance variables are always private in Smalltalk, however often methods are available for accessing the instance variables. The accessor rule expresses the relationship between the object, variable name and the selector for accessing the variable name.

3. referToEachOther(?objX, ?objY)
   the referToEachOther rule expresses the relationship of objects that collaborate together.

5.6.3 Control Flow

Sometimes we want to know in which context a certain message was sent. For this reason we have rules expressing the control flow history of a certain message:

1. cflow(?obj, ?sel, ?args)
   The cflow-rule infers all objects that are currently activated on the call-stack.

2. sender(?c, ?cSel, ?cArgs, ?r, ?rSel, ?rArgs)
   The sender-rule expresses the caller-receiver relationship between two activations on the call-stack.

5.7 Composite Aspect Object Reasoning Rules

Besides reasoning about the run-time of the environment we also want to reason about the structure and the environment of the composite aspect object. The following rules allow to reason about the composite aspect object:

1. whole(?whole)
   The whole-rule is used for retrieving the composite aspect object in which a certain rule was activated.

2. partObject(?partName, ?partObj)
   The partObject-rule expresses the relationship between the objects and their unique identifier that are part of the composite aspect object in which the rule is evaluated.

3. aspectModule(?logicModule)
   The aspectModule-rule matches with the names of the aspect modules that are currently present in the system.

4. aspectSender(?caller, ?callerSel, ?callerArgs)
   This rule is used to match the object that sent a message to the composite aspect object.

5. receivedSelector(?selector, ?arguments)
   The rule matches the selector and the arguments in which the composite aspect object was activated.

6. result(?partName, ?selector, ?result)
   The rule matches a value that was computed by the part-object named ?partName and was activated with the selector ?selector to the variable ?result. The results that were computed by part-objects are only kept during the handling of one message by the composite aspect object.

5.8 Solving Composition Issues

In this section we show how to solve the composition issues that were discussed in chapter 3 using the logic activation scheme in conjunction with the rules introduced above.

5.8.1 Activation

Some crosscutting concerns need to be activated depending on some run-time properties of the system or depending on the context in which they are called.

Property-Dependent Activations

In the bodies of the before-, base-, after- and catch-rules we can use the language symbiosis provided by the logic metaprogramming language to access the run-time properties of the system. For example, if we only want to activate a logging aspect when the logging is activated we can use the following rule:

```
before(?selector, ?receivedArgs, logging, log:, ?selector) if
  logSelector(?selector),
  partObject(logging, ?logObject),
  [?logObject isActive].
```

Explanation of the rule:

- logSelector(?selector)
  an internal rule used to infer if we are interested in logging the selector.
- partObject(logging, ?logObject)
  rule is used to find the part-object in the composite aspect object that is responsible for logging.
- `[?logObject isActive]`
  using the language symbiosis provided by the logic meta programming language we send the message isActive to the part-object that is responsible for logging to find out if it was activated.

Context-Dependent Activations

The activation of an aspect sometimes depends on the context in which the aspect is activated. We work out the example of a view that is installed on a collection to display it contents. The collection has two selectors to add elements: add: for adding a single element and addAll: for adding a collection of elements. We encapsulate the crosscutting concern of updating the screen

in an aspect. Hence the composite aspect object contains two objects, a collection object and an object for notifying other objects of changes. The notify aspect needs to be activated each time the state of the collection is updated, therefore the aspect needs to be activated when add: or addAll: is send. Now, suppose the addAll: method is implemented in terms of the add: method:

```
Collection>>addAll: aCollection
  aCollection do: [:element | self add: element ].
```

In that case the notify aspect will be activated once after the addAll: and n times after the add: (if there are n elements in aCollection), while we only want to activate the notify aspect after the addAll: has been executed. We want to express that the notify aspect should not be activated if it is called in the context of executing addAll: method.

To express this we can make use of the aspectSender-predicate that was introduced above:

```
after(?selector, ?args, notifier, notifyAll, <>) if
  stateChange(?selector),
  partObject(collection, ?caller),
  not(aspectSender(?caller, addAll:, ?callerArgs))
```

Explanation of the rule:

- stateChange(?selector)
  check to see if the selector that was sent to the composite aspect object changed the state.
- partObject(collection, ?caller)
  this rule unifies with any of the object that fulfills the role as collection in the composite aspect object.
- not(aspectSender(?caller, addAll:, ?callerArgs))
  this rule defines that the caller must not be the collection in the context of the addAll: method.

5.8.2 Choosing an Aspect

When two or more part-objects provide a different implementation for the same crosscutting concern and are woven in the same composite aspect object, then we must specify when which part-object should be activated. One option would be to put this decision in the activation rules in the aspect modules. However, we think this decision should not be specified in the activation rules because of three reasons:

1. The decision process is implicitly put in the activation rules.

2. When multiple activation rules are used to define when the aspect should become active, then we need to put the same condition in all the activation rules.

3. It harms the reusability of our activation rules in a context where there is only one implementation of the aspect necessary.

Consider again the example of the two compression aspects that we discussed in section 3.2.1. We have two objects implementing a high compression algorithm (providing a lower quality) and a low compression algorithm (providing good quality). A class diagram of the compression classes is shown in figure 5.7. We add both objects to the composite aspect object, with the identifiers highCompression and lowCompression. The activation rules, that defines when and how the compression objects should be activated, should not use the explicit name of the part-object that needs to be activated (such as the name highCompression or lowCompression). Instead the activation rule uses an abstract name (such as compression) in the activation rules. For example:

```
before(?selector, ?args, compression, compress, <>) if
  ...
```

We can now separately specify the decision process between the two part-object as follows:

```
partObject(compression, ?compressionObject) if
  [Network isSaturated],
  partObject(highCompression, ?compressionObject).

partObject(compression, ?compressionObject) if
  [Network isSaturated not],
  partObject(lowCompression, ?compressionObject).
```

Now, when the composite aspect object infers the before rule it will resolve the abstract name compression using the partObject rules.

Note that these rules are not put in the activation rules of the aspect module. The complete process of choosing the correct part-object is put in these two rules and is not crosscutting multiple aspect modules. The advantage is that it becomes more readable, changeable and the aspect module can be fully reused.

5.8.3 Compatibility

Because of the dynamic model of the composite aspect object we can add, remove or change the part-objects at run-time. This allows for a great deal of flexibility, which is not always needed and can sometimes endanger the

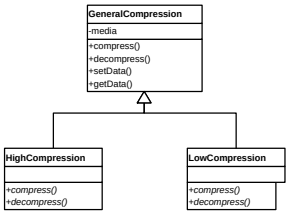

Figure 5.7: UML Diagram of Compression Classes

safeness of the application, for example if we put a part-object that does not implement the protocol required by the logic activation scheme (i.e. enforcement of type-safeness). To resolve these issues we can use the compatibility-rules introduced above in section 5.5.1. The compatibility rules define what properties are required from the part-objects and are included in the logic activation scheme. The rules can check both the structure and the dynamic properties of the part-objects using the run-time reasoning library. If the check-rules cannot be inferred when the part-objects are added or removed, then an InconsistentCAOError exception is thrown.

5.8.4 Order

When two or more aspects need to be activated at the same point in time, then it might be necessary to specify the order in which the aspects need to be activated. As explained above in section 5.5.3, we have introduced a dominates-relationship between two aspect modules in a composite aspect object.

Static Order

When the order between two aspects is static then we can use the rule with an empty body. For example, if we have an object that is accessed by multiple threads then we need a semaphore aspect to ensure the consistency of the data accesses in your object. Now, if we need logging on the same object, we need to lock the object before we log the accesses to the object. Example: dominates(semaphore, logging).

Dynamic Order

Sometimes we might want to let the order depend on dynamic properties. For example [PSDF01], if we have an object that is available over the network with both an authentication and a logging aspect we might want to log operations before the authentication if we suspect someone is trying to intrude in the system and to log after authentication in other cases. We can

specify the dynamic order of aspects by putting a body in the dominates-rule. The example can be specified with the following rules:

```
dominates(logging, authentication) if
  [Network intrusionDetected].

dominates(authentication, logging) if
  [Network intrusionDetected not]
```

5.8.5 Dependencies

As pointed out in section 3.2.6 sometimes an aspect depends on other aspects to provide a certain service. For example, when we have a billing aspect that creates invoices for the time a certain service was provided. The billing aspect depends on a timing aspect to measure the time a certain service was used. We can use the aspect activation modules to specify the dependencies between one or more aspect modules. Suppose we have an aspect module for the billing aspect called billing and an aspect module for timing aspect called timer. We can now define that the billing aspect can only be activated if the timer aspect is present and active with the following rule:

```
active(billing) if
    aspectModule(timer),
    active(timer)
```

We can also add a rule that weaves a timer aspect if no timer aspect is present:

```
active(billing) if
    not(aspectModule(timer)),
    whole(?cao),
    [?cao addModule: TimerAspect. true],
    [?cao add: 'timer' part: Timer new. true]
```

The rule first checks to see if there is no timer aspect module present. Then we bind the composite aspect object to the cao variable. Eventually we add the a timer aspect module and weave a part-object responsible for timing services using the language symbiosis.

5.9 Performance Issues

Logic programming languages have a bad reputation when it comes to performance. Since the outcome of the results of several queries steer the activation of the different part-objects we can wonder if it is appropriate to use a logic language. In [Roy90] it is shown that logic programs can be compiled into code that approaches the efficiency of an imperative programming

language. The system could further be optimized using a caching system, because some of the rules compute the same results are computed for inferring different rules. However, the SOUL logic engine that was used to implement the logic activation scheme is not optimized and this is an area for future work discussed in section 7.2.1.

5.10 Conclusion

In this chapter we have explained how to define the composite aspect object protocol using a logic activation scheme. Such a logic activation scheme consists of logic metaprogramming rules. These rules are separated in several modules related to the issues that they resolve. Aspect modules serve to define the protocol of the composite aspect object and activate the objects implementing a certain crosscutting concern. They include activation rules that define how and when the different part-objects in the composite aspect object should be activated. Furthermore, they provide rules to define the required properties from the part-objects. The order module is introduced to define static and dynamic orders between the aspect modules. The aspect activation modules define rules to specify when to activate and deactivate an aspect module. The modules provide an expressive means to solve the issues that were discussed in chapter 3. The issues are resolved in separate modules, so that they do not affect the reusability of the aspect modules and their part-objects.

Chapter 6

Examples

6.1 Introduction

In the previous chapter we introduced the logic activation scheme and the run-time reasoning library. In this section we work out two small examples to show the practical use of the composite aspect object. The first example shows the capability for dynamically weaving aspects in a distributed system and the second example demonstrates how some composition issues are solved in our composite aspect object model.

6.2 Distributed Library

The first case demonstrates the dynamic weaving capabilities of the composite aspect objects. As a case we work out a digital library system that can be accessed from multiple clients over a network. Users can search for books in the digital library. Users can add books to the digital library. The books can be saved both in the digital library (to prevent that the books are deleted if the digital library program terminates) and on the client side if the user decides that he likes the book and wants to view it when he is not connected to the network. The books are saved in different ways depending on the infrastructure available on the machine where the book is located. The digital library server will make the books persistent in a SQL database to speed up the queries for searching the books by the numerous clients on the system. The clients on the other hand will make the books persistent in files, since the workstations are not equipped with a SQL database. Another issue we want to take into account is that of the available bandwidth of the network. The bandwidth depends on the quality of the connection between the digital library server and the client connection. Corporate clients might have a fast connection that has a good throughput, while mobile clients usually have a small throughput. Other factors that could influence the network connection are the geographical locations and the general saturation of the

network (dip vs. peak hours). We should be able to adapt the strategy of transferring files depending on these conditions.

6.2.1 Core Classes

In this subsection we describe the classes that are used for our case and their behavior:

Library

Represents a collection of books.

- add: aBook
  method for adding a book to the library
- remove: aQuery
  method removes all books that match the query that was passed as a parameter.
- search: aQuery
  method returns a collection of books that match the query given as parameter

Book

Represents a book from the library.

- author: anAuthor
  sets the author of the book
- author
  returns the author of the book
- isbn: anISBN
  sets the isbn number of the book
- isbn
  returns the isbn number of the book
- title: aTitle
  set the title of the book
- title
  return the title of the book
- at: aPageNumber put: aPage
  adds or replaces a page in the book

- at: aPageNumber
  returns the page number in the book

- numberOfPages
  returns the number of pages in a book

Page

Represents a single page within a book.

- text: aText
  sets the text in the page

- text
  returns the text contained in the page

Query

The abstract query class is used to determine whether a book matches the criteria set by the query. There is a subclass for each field in the book (author, title, isbn, ...). Queries can be combined (with the logical and/or-operator) using the composite pattern [GHJV94].

- keep: aBook
  method returns true when the book matches the query and false in the other case.

LibraryServer

Provides the portal for accessing a library through a network connection.

- library: aLibrary
  sets the library the server provides access to

- library
  return the library the server provides access to

- initialize
  initializes the server

- start
  starts the server

- stop
  stops the server

6.2.2 Aspects

In the small case presented above we can identify three aspects, namely distribution, synchronization and persistence. In this case we focus on the distribution and persistence aspects. Most of the current aspect-oriented programming tools cannot handle these aspects sufficiently, because of their dynamic properties such as the available bandwidth of the network and the persistence of the books that differ depending on the machine where the books are located.

Distribution

Most of the object-oriented programming languages have support for two strategies for passing objects over a network connection.

- By copy
  The object and all its collaborators are recursively copied and transmitted over the network. On the other end of the network the object and its collaborators are reconstructed. A new object identity is created on the machine to which the object has been copied.

- By reference
  The object is not copied, instead a reference to the object is sent over the network. On the other end of the network the reference is constructed and all messages passed to this reference are forwarded over the network to the computer where the object lives. Hence, the reference on another machine acts as a proxy that redirects all its method calls to the real object.

One strategy is more appropriate than another, depending on the properties of the application we want to construct. For example: pass-by-reference is mostly appropriate for big objects, but it requires a continuous network connection.

Persistence

The implementation of the persistence aspect depends on the machine at which books need to be made persistent. For example, when books are made persistent in the library server we want to save them into a database system. Database systems provide superior capabilities for searching the books. However, the machines of the users that search for books also want to save the books so that they can view them when they are disconnected from the network. The machines of the regular users do not have a database server installed, so they want to save the book in a regular file. When a user is using a monochrome palm device he might want to remove the illustrations in the book to save memory on the device. The above examples

show that the persistence aspect is dynamical and depends on the machine where the book is located. Each time a book is passed over the network the receiving machine needs to weave the correct persistence aspect into the book. Below we work out the persistence of books in a SQL database and files. To automate the synchronization between the run-time state and the persistent state, we specify that the run-time state should be made persistent when the state of a book object has changed ten times.

6.2.3 Implementation Details

This subsection discusses some implementation details that improve the understandability of the implementations of the different aspects below. More particularly we discuss the serialization protocol that is provided in Smalltalk and the framework that allows us to distribute objects in Smalltalk.

Serialization

In Smalltalk, each object understands a set of four messages that are used to serialize and deserialize objects. The first two messages are for serialization and the last two messages are for the deserialization of objects. We shortly discuss them below, because they are used to implement the distribution aspect:

1. objectForDataStream: aStream
   This method is called before an object is serialized and takes a streaming object as parameter. It gives the object the possibility to provide a surrogate object for serialization instead of itself.

2. storeDataOn: aStream
   This method writes the object onto the stream that was provided as a parameter.

3. readDataFrom: aStream size: sizeOfObject
   reads the data from the stream to reconstruct the serialized object.

4. comeFullyUpOnReload: aStream
   This method gives the opportunity to perform some actions after the object has been deserialized from a stream. The value returned from this method is what is considered to be the deserialized object.

Distribution Framework

For distributing object over several computers we have made use of *Remote Smalltalk* [Dec02], a framework for distributing objects in the Smalltalk programming language. Each object has a method remoteType. The method

returns the symbols `#copy` or `#reference`, specifying how the object should be passed over the network. Objects are by default send by reference, although this can be changed by overriding the method remoteType in a subclass. Objects representing numbers, strings, boolean values, the nil object, ... are passed by copy. The main problem with the distribution framework is that the code that is responsible for the distribution is tangled in the class of the objects that we want to access over the network. Methods have to be added to one or more classes to regulate their distribution properties, hence the distribution concern is still tangled.

6.2.4 Part-Object Classes

In this section we describe the classes of the objects responsible for the persistence and distribution of objects. Note that, like in the previous examples, we do not intend to make the perfect classes for persistence and distribution. The classes are for the persistence and distribution of general objects, but can be further extended using inheritance and aggregation for specific classes of objects.

SQLPersistence

Class responsible for the persistence of general objects to a SQL database server.

- openWithPrimaryKey: aKey from: aTable
  returns the object that was saved using the key aKey from a table named aTable
- saveWithPrimaryKey: aKey to: aTable object: anObject
  saves the object anObject into a table named aTable with the key aKey
- refreshWithPrimaryKey: aKey from: aTable object: anObject
  synchronizes both the object anObject that was made persistent in the table aTable and the living object with the key aKey

FilePersistence

Class responsible for the persistence of general objects to a file.

- saveOnFile: aFileName object: anObject
  saves anObject into a file named aFilename
- openFromFile: aFileName
  returns the object that was saved into a file named aFileName
- refreshFromFile: aFileName object: anObject
  synchronizes both the object anObject that was made persistent and the living object from the file named aFileName

NetworkObject

The class NetworkObject provides methods for specifying the object transmission mode:

- byCopy
  returns the symbol `#copy`
- byReference
  returns the symbol `#reference`
- transferObject: anObject
  sets the object to be transferred

Counter

The class Counter is used to check the number of times the state of an object has been changed:

- increment
  increments the counter with one.
- = aNumber
  returns true if the counter has reached aNumber.
- < aNumber
  returns true if the counter is below aNumber.
- reset
  sets the counter to zero.

6.2.5 Book-Objects

In this subsection we discuss how the different aspects are weaved into the objects that represent the books from our digital library.

Part-Objects

The composite aspect object of book-objects contains the part-objects with the following names:

book the part-object is an instance of the class Book.

persistence the part-object is an instance of the SQLPersistence class when the object resides on the server-side and is an instance of the FilePersistence class on the side of a client.

network the part-object is an instance of the class NetworkObject

counter the part-object that is responsible for counting the number of times an object has changed.

Aspect Modules

The composite aspect objects for book-objects are composed of four aspect modules that are named the same as the part-objects.

book In the book aspect module we have one rule that declares that all selectors that are understood by the part-object named book should be delegated to that object, except for the selectors that are named objectForDataStream: and remoteType. These selectors have to be explicitly mentioned, because they are understood by every object (and thus also the part-object book), but we want to handle them in the other aspect modules.

```
base(?rSel, ?rArgs, book, ?rSel, ?rArgs) if
    not(equals(?rSel, objectForDataStream:)),
    not(equals(?rSel, remoteType)),
    partObject(book, ?baseObject),
    instanceOf(?baseObject, ?baseClass),
    understands(?baseClass, ?rSel).
```

This solution is not completely satisfying, because the rules that check that the received selector is not objectForDataStream: or remoteType are put in the body of the base-rule, because of their conflicting nature with the network aspect module. This restricts the reusability of the book aspect module. We move these rules out of the base-rule and put them in an aspect activation module to improve the reusability of the book aspect module.

```
active(book) if
    not(receivedSelector(objectForDataStream:, ?)),
    not(receivedSelector(remoteType, ?))
```

The rule specifies that the book aspect module should not become active when the selector received by the composite aspect object is either objectForDataStream: or remoteType.

counter We want to synchronize the run-time state with persistent state after each ten times the state of the object has changed. For this we need to employ a counter aspect that counts the number of times the state has changed. We can update the counter with the following rules:

```
after(?rSel, ?rArgs, counter, increment, <>) if
  updateCounter(?rSel),
  partObject(counter, ?counterObj),
  resetAt(?resetNumber),
  [?counterObj < ?resetNumber].

after(?rSel, ?rArgs, counter, reset, <>) if
```

```
  updateCounter(?rSel),
  partObject(counter, ?counterObj),
  resetAt(?resetNumber),
  [?counterObj = ?resetNumber].
```

The rules check if the counter needs to be updated and if the counter needs to be reset or incremented.

Aspect Configuration Module The counter aspect module can be configured with two rules:

- updateCounter(?selector)
  defines when the counter should be updated.
- resetAt(?aNumber)
  defines the number at which the counter should be reset.

In our case the counter if configured with the following rules:

```
updateCounter(author:).
updateCounter(isbn:).
updateCounter(title:).
updateCounter(at:put:).

resetAt(10).
```

The first four facts are the selectors that change the state of a book and the last fact defines the number at which the counter should be reset.

We can make the updateCounter-rule less susceptible to changes in the book class by using the SOUL library to check which selectors are changing the state. The following rule checks if the selector that is received has changed the state of the book:

```
updateCounter(?rSel) if
  partObject(book, ?bookObj),
  instanceOf(?bookObj, ?bookClass),
  selectorParseTree(?bookClass,?rSel,?tree),
  assignmentStatement(?tree, ?var, ?value).
```

The rule checks the parse tree of the source code to see if the method associated with the selector contains an assignment statement. Using such a rule at run-time has the advantage that changes made to the composite aspect object (i.e. if the book object is replaced with another object that has a different implementation) will automatically adapt to the new changes. The disadvantage of such rules is that they have a negative impact on the efficiency. We believe however, that this could be optimized with advanced caching techniques.

persistence As we explained above, the SQLPersistence and the FilePersistence class have a different protocol that is more specialized to the crosscutting concern they are implementing. For this reason we have to define two different persistence aspect modules. One that is used for the SQLPersistence class on the server side and another that is used for the FilePersistence class.

Aspect Module for SQL Database The rules for the SQLPersistence class are shown below:

```
base(load:, ?args, persistence,
     openWithPrimaryKey:from:, ?argsWithTable) if
    table(?table),
    append(?args, <?table>, ?argsWithTable).

base(save, <>, persistence,
     saveWithPrimaryKey:to:object:, <?key, ?table, ?object>) if
    table(?table),
    persistentPart(?object),
    dbkey(?key).

base(refresh, <>, persistence,
     refreshWithPrimaryKey:from:object:, <?key, ?table, ?object>) if
    table(?table),
    persistentPart(?object),
    dbkey(?key).

after(?rSel, ?rArgs, persistence,
      refreshWithPrimaryKey:from:object:, <?key, ?table, ?object>) if
    needsSynchronization(?rSel),
    table(?table),
    persistentPart(?object),
    dbkey(?key).

dbkey(?key) if
    persistentPart(?object),
    keyField(?field),
    equals(?key, [?object ?field asString]).
```

1. The first rule makes the composite aspect object delegate load:-messages to the persistence object. The argument passed with load: contains the isbn number of the book. The rule adapts the parameter list by appending the table name of the database where the objects are stored.

2. The second rule is for handling the save-message. It retrieves the key of the object by the internal rule dbkey, which is used to retrieve the value of the field that determines the primary key (in our case the field that hold the isbn). Furthermore, it uses the persistentPart-rule to find the part that should be made persistent.
3. The third rule is similar to the previous one.
4. The fourth rule is an after rule and defines when the run-time state should be synchronized with the persistent state.
5. The dbkey-rule is used to retrieve the value that should serve as key in the database.

Aspect Configuration Module Our aspect module is reusable for different objects by reconfiguring the aspect module with three rules:

1. the persistentPart rule specifies which part of the composite object will be made persistent.
2. the keyField rule specifies which field of the object will serve as the primary key in the database.
3. the table rule specifies the table where the books will be stored.
4. the needsSynchronization(?rSel) what conditions should be fulfilled to synchronized the run-time state with the persistent state.

The server side is configured with the following rules:

```
persistentPart(?object) if
    partObject(book, ?object),

keyField(isbn).

table(books).

needsSynchronization(?rSel) if
  updateCounter(?rSel),
  partObject(counter, ?counterObj),
  resetAt(?aNumber),
  [?counterObj = ?aNumber].
```

Notice that the interaction between the counter aspect and the persistence aspect are fully encapsulated in the aspect configuration module. In effect the interaction between the two aspect modules is specified in the needsSynchronization-rule. Encapsulating these interactions in the aspect configuration modules allows us not only to fully reuse the counter and persistence

part-objects, but also the counter and persistence aspect modules in different contexts.

Aspect Module for Files Now we define the aspect module that saves the objects in files for the composite aspect object of books on the client-side:

```
base(load:, ?args, persistence, openFromFile:, ?args).

base(save, <>, persistence,
     saveOnFile:object:, <?filename, ?object>) if
    persistentPart(?object),
    filename(?filename).

base(refresh, <>, persistence,
     refreshFromFile:object:, <?filename, ?object>) if
    persistentPart(?object),
    filename(?filename).

after(?rSel, ?rArgs, persistence,
     refreshFromFile:object:, <?filename, ?object>) if
    needsSynchronization(?rSel),
    persistentPart(?object),
    filename(?filename).
```

The rules are similar to the rules above from the persistence aspect module for the databases.

Aspect Configuration Module The persistence aspect for files is reusable by configuring three rules:

1. The persistentPart-rule is identical to the one above.
2. The needsSynchronization-rule is identical to the one above.
3. The filename-rule is used to retrieve the filename that should be used to save and refresh the object. In this example we used the title of the book.

The client is configured with following rules:

```
persistentPart(?object) if
    partObject(book, ?object).

needsSynchronization(?rSel) if
  updateCounter(?rSel),
```

```
  partObject(counter, ?counterObj),
  resetAt(?aNumber),
  [?counterObj = ?aNumber].

filename(?title) if
    persistentPart(?partName),
    partObject(?partName, ?partObject),
    equals(?title, [?partObject title])
```

network The rules of the network aspect module define how and which part-objects from the composite aspect object should be passed over the network:

```
base(objectForDataStream:, ?receivedArgs, network,
     transferObject:, <?object>) if
    transfer(?object).

base(remoteType, ?receivedArgs, network, byCopy, <>) if
    transferMode(copy).

base(remoteType, ?receivedArgs, network, byReference, <>) if
    transferMode(reference).
```

The first base-rule delegates the objectForDataStream: selector to the transferObject: method of the network object with as argument the part that is selected to be transferred. The following two base-rules are for handling a remoteType-message. The first rule delegates the remoteType-message to the byCopy method on the network part-object if the transferMode-rule is set to copy. The second rule delegates the remoteType-message to the byReference method if the transferMode-rule is set to reference.

Aspect Configuration Module There are two rules that need to be configured when reusing the network aspect module:

- transfer(?object)
  specifies the object that should be exported.

- transferMode(?mode)
  specifies if the object should be passed by copy or passed by reference.

We can now configure our persistence aspect, so that the transfer mode of the objects depend on different heuristics. We can for example transfer the book by reference if the book contains more than hundred pages and transfer the book by copy if it has less than hundred pages with the following rules:

```
transfer(?object) if
  partObject(book, ?object).

transferMode(copy) if
    partObject(book, ?book),
    [?book numberOfPages < 100].

transferMode(reference) if
    partObject(book, ?book),
    [?book numberOfPages >= 100].
```

Other heuristics such as the network throughput or a combination of multiple heuristics can also be used. In [LK97] an aspect-oriented language is constructed for handling the aspect of distribution. They allow to change the properties of the way an object is passed over the network depending on the method that is remotely invoked. Our approach is more dynamic as it allows to take any run-time property into account to adapt the way an object is passed over the network.

Order Module

There is a conflict between the counter and the persistence aspect modules. The persistence aspect interacts with the counter to know whether it should synchronize the run-time state with the persistent state. In order to function correctly the counter aspect should be updated before the persistence aspect is checking the value of the counter. We can specify this in an order module with the following rule:

```
dominates(counter, persistence).
```

Dynamic Weaving of the Persistence Aspect

As we already mentioned above, we have to dynamically weave the persistent aspects depending on the computer where a book object arrives. Each time a book object arrives over the network we create a composite aspect object and weave the aspects that are needed on the machine where the object has arrived. In section 6.2.3 we saw that after an object is deserialized the method comeFullyUpOnReload: is called. We can use this method to create a composite aspect object and weave the aspects that are needed for the machine where it arrived. Because we do not want to tangle the dynamic weaving of the book objects in the Book class we invoke the dynamic weaver at the meta-level. The dynamic weaver that weaves book objects on the client-side is shown in figure 6.1. Its counter-part for weaving the book at the server-side is shown in figure 6.2.

```
DynamicWeaver>>weaveBook: aBook
    | compositeAspect strategy persistence network |
    "Create new composite aspect object"
    compositeAspect := CompositeAspect new.
    "Create new activation scheme based on"
    "Logic Metaprogramming"
    strategy := LMPStrategy new.
    "Assign the logic activation scheme to"
    "the composite aspect object"
    compositeAspect strategy: strategy.
    "Create the object that will be responsible for"
    "the persistence using files"
    persistence := FilePersistence new.
    "Create the object that will be responsible for"
    "the object transfers over the network"
    network := NetworkObject new.
    "Add all the part-objects in the composite aspect"
    "object"
    strategy add: 'book' part: aBook.
    strategy add: 'persistence' part: persistence.
    strategy add: 'network' part: network.
    strategy add: 'counter' part: Counter new.
    "Configure the composite aspect object to use the"
    "following logic activation scheme"
    strategy addModule: BookAspect.
    strategy addModule: CounterAspect.
    strategy addModule: FileAspect.
    strategy addModule: NetworkAspect.
    strategy addModule: ClientConfiguration.
    ^ compositeAspect
```

Figure 6.1: Wrapping the book object after it is deserialized at the client-side

```
DynamicWeaver>>weaveBook: aBook
    | compositeAspect strategy persistence network |
    "Create new composite aspect object"
    compositeAspect := CompositeAspect new.
    "Create new activation scheme based on"
    "Logic Metaprogramming"
    strategy := LMPStrategy new.
    "Assign the logic activation scheme to"
    "the composite aspect object"
    compositeAspect strategy: strategy.
    "Create the object that will be responsible for"
    "the persistence using a SQL database"
    persistence := SQLPersistence new.
    "Create the object that will be responsible for"
    "the object transfers over the network"
    network := NetworkObject new.
    "Add all the part-objects in the composite aspect"
    "object"
    strategy add: 'book' part: aBook.
    strategy add: 'persistence' part: persistence.
    strategy add: 'network' part: network.
    strategy add: 'counter' part: Counter new.
    "Configure the composite aspect object to use the"
    "following logic activation scheme"
    strategy addModule: BookAspect.
    strategy addModule: CounterAspect.
    strategy addModule: SQLAspect.
    strategy addModule: NetworkAspect.
    strategy addModule: ServerConfiguration.
    ^ compositeAspect
```

Figure 6.2: Wrapping the book object after it is deserialized at the server-side

6.3 Secured Objects

The second example focusses on some composition issues. In some applications we want to restrict the access to the selectors of one or more objects to a limited number of people in a distributed application. Since the object contains critical data we would like to know who has accessed the object. When we are suspecting someone is hacking the system we want to show all attempts to access data in the secured object, so then we want to log the accesses that have not been authorized. This example has different software concerns that are crosscutting some classes and that would be tangled in an object-oriented programming language without support for advanced separation of concerns. In the subsections below we show how these crosscutting concerns can be separated using the composite aspect object model.

6.3.1 Part-Objects

In the small example we can distinguish between at least three aspects:

1. Secured Object
   The implementation of the object that needs to be secured. In a real-world example, this can be a complex object, but for the simplicity of this case we assume that the object has a selector which prints sensitive information on the screen.

2. Authentication
   We have an object that is responsible for the authentication of the users that want to access the secured object from over the network.

3. Logging
   We have an object that is responsible for logging the information on an output device.

The UML class diagram of the classes that implement these three software concerns is shown in figure 6.3. These three objects are composed together in a composite aspect object. In the composite aspect object we give each of the objects the following unique identifiers:

Object	Identifier
Secured Object	data
Authentication	authenticator
Logging	logger

6.3.2 Aspect Modules

We need three aspect modules, one for each part-object. The modules are named identical to the part-objects authenticator, logger and data. In this section we discuss the rules included in each aspect module.

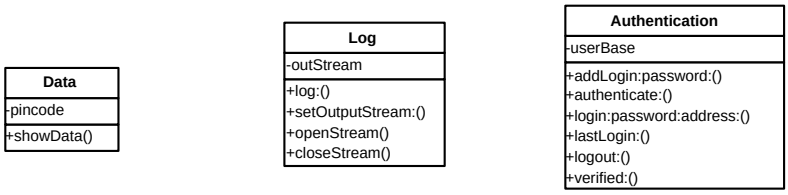

Figure 6.3: UML Class Diagram of Secured-Objects Example

```
base(?rSelector, ?rArgs, data, ?rSelector, ?rArgs) if
    partObject(data, ?baseObject),
    instanceOf(?baseObject, ?baseClass),
    understands(?baseClass, ?rSelector)
```

Figure 6.4: Rule for Exposing the Interface of the Data-Object to the Composite Aspect Object

In a first step we define the interface of the composite aspect object. We want the interface of the composite aspect object to match that of the secured object and we want to add a logout selector to invalidate a session.

Data Aspect Module

The data aspect module contains one rule, which is shown in figure 6.4. This rule defines that all the selectors of the data object are part of the interface of the composite aspect object. The base-rule consists of three rules in its body:

1. The partObject-rule unifies the ?baseObject variable with the object that is put in the composite aspect object as data-object.

2. The instanceOf-rule unifies the object that was found and binds its class to the ?baseClass variable.

3. The understands-rule is used to check if the selector that was received by the composite aspect object is understood by the class.

So, if this base-rule can be proved when the composite aspect object has received a message, then the same selector and the same argument list is used to activate the data-object. Note that this rule will adapt to the interface of the data-object that is currently used in the composition of the composite aspect object. The data-object can be interchanged with any other object without having to change the data aspect module. This improves both the reusability and run-time adaptability of our aspect module.

Authentication Aspect Module

The authentication aspect module is more complicated. The calls to the secured object can originate from different clients accessing the computer through the network. Depending on the machine where the call came from we need to check if the person using that machine has already been authenticated. Remote calls to the secured objects are handled by a thread that is listening on a certain port and when a new connection is made then a socket object is created and passed to the broker as a parameter of the message process: for handling the call. To be able to distinguish the different machines we need to know the network address of the machine where the call came from. The network address of the machine is stored in the socket. We can find the network address of a socket using the control flow of the call with the following internal rule:

```
remoteAddress(?address) if
  cflow(?broker, process:, <?socket>),
  instanceOf(?broker, [RSTBroker]),
  equals(?address, [?socket remoteAddress])
```

The rule searches in the active method calls for the activation of the method from the broker that handled the call. To implement this without the cflow-rule we would end up in changing a lot of methods for passing the remote address as a parameter.

The second internal rule is used to check whether the call comes from a machine that has been authenticated before. The internal rule is shown below:

```
authenticated(?login) if
  remoteAddress(?address),
  partObject(authenticator, ?authObject),
  [?authObject verified: ?address],
  equals(?login, [?authObject lastLogin: ?address])
```

The rule unifies the ?login variable with the person that has been logged in. The rule consists of four parts:

1. remoteAddress(?address)
   the network address of the machine where the call originated from is bound to the logic variable ?address

2. partObject(authenticator, ?authObject)
   unifies the object that fulfills the role as authenticator part-object with the variable ?authObject

3. [?authObject verified: ?address]
   we make use of the SOUL symbiosis with the Smalltalk environment

to send the message verified: with as parameter the network address of the machine where the call originated from. The authenticator object should return true if the machine is properly authenticated and false in the other case.

4. equals(?login, [?authObject lastLogin: ?address])
   the equals-rule is used to bind the person that is authenticated to the ?login variable.

The three subsequent rules are used to define how and when the authentication should be applied in the composite aspect object:

```
before(?securedSelector, ?, authenticator, authenticate:,
       <?address>) if
    not(authenticated(?login)),
    remoteAddress(?address),
    securedSelector(?securedSelector).

catch(?exception, ?securedSelector, ?args, whole,
      ?securedSelector, ?args) if
    instanceOf(?exception, [AuthException]).

base(logout, <>, authenticator, logout:, <?address>) if
    remoteAddress(?address)
```

1. The before-rule defines that the composite aspect object should activate the authentication (by sending the object that fulfills the role as authenticator the authenticate: message with the network address of the remote machine) when we are not already authenticated and when the selector we received is secured.

2. The catch-rule determines the strategy what should happen when the authentication fails. In this case we resend the message that we received to the composite aspect object, which results in a new try to authenticate the user. So, in our example the policy for failed authentication attempts is to retry the authentication. It is possible to have a more complicated policy (for example, when a maximum of three authentication tries are allowed) by adding an object and adapting the catch-rule so that it defines a failure policy and redirect all the exceptions to that object.

3. The base-rule makes the composite aspect object understand the logout: message, by delegating it to the authenticator object to invalidate the session.

Aspect Configuration Module The authentication aspect module is configured with one rule that defines the selectors that need authentication:

```
securedSelector(showData)
```

The rule defines the fact that showData is a secured selector. Note that if we want to secure all the selectors of the base object that we could define a securedSelector that is similar to the base-rule from above, instead of having to define all the selectors separately. Another example of a more advanced configuration rule is that we could check the source code of the data object to determine which selectors have to be secured. For example, if the data that needs to be secured is put in an instance variable named pincode in the data object, then we can use the SOUL library [Wuy98] to search for all the methods that access the pincode instance variable and secure them instead of manually specifying the secured selectors.

Logging Aspect Module

A second requirement of the example was that we wanted to log who had accessed a certain selector in the secured object. Note that this requires an interaction between the logging aspect and the authentication aspect. For the logging functionality we have added an object which fulfills the role as the logger. For activating the logger we use the rule shown below:

```
before(?securedSelector, ?, logger, log:, ?message) if
    logSelector(?securedSelector, ?message).
```

Aspect Configuration Module We configure the logging aspect module with with the following rules:

```
logSelector(?selector, <{?login has accessed ?selector}>) if
    securedSelector(?selector),
    authenticated(?login),
    [Network intrusionDetected not]

logSelector(?selector, <{Someone tried to access ?selector}>) if
    securedSelector(?selector),
    [Network intrusionDetected]
```

Basically the first rule defines that the logging aspect should log when the message is a secured selector and the access to the selector was authenticated when the network is in a normal state. The second rule defines to log every attempt to access a selector when an intrusion in the network is detected.

6.3.3 Compatibility Rules

In the case of the authentication the dynamic weaving capabilities might compromise the security of the system. For example, a malicious host could replace the authentication object with another class that has the same interface, but always authenticates the call. For this reason we might specify that the part-object authenticator should always be of the class Authentication that we provided with the system. Furthermore, we want to check that the call to change the composite aspect object does not come from a remote machine. For this we can use the following compatibility rule:

```
check(authenticator, ?authObject) if
    instanceOf(?authObject, [Authentication]),
    not(remoteAddress(?address)).
```

The first rule in the body of the compatibility rule checks that the new object is of the class Authentication. The second rule checks that the change to composite aspect object does not originate from a remote machine.

6.3.4 Order Module

This example shows a dynamic order conflict between the logging aspect and the authentication aspect. Dynamic orders cannot be expressed with most of the current aspect-oriented programming tools. More particularly, the order of the the logging and authentication aspect depends on the fact that we are suspecting if someone is hacking the system or not. In the former case the logging aspect should be activated before the authentication aspect, to log all the accesses to the system, while in the latter case we activate the logging aspect after the authentication aspect in order to log the user that accessed the secured object. We can express this very simply in the logic activation scheme by adding an order module with the following rules:

```
dominates(logging, authentication) if
  [Network intrusionDetected].

dominates(authentication, logging) if
  [Network intrusionDetected not]
```

6.3.5 Aspect Activation Module

The logging message of the logging aspect includes the user that has accessed the secured object. This means that the logging aspect depends on the activation of the authentication aspect for functioning correctly in that case. When everything is being logged we do not depend on the authentication aspect, because we do not include the user name. This means that the logging aspect depends on the authentication aspect depending on the some condition. We can specify this dependency with the following rule:

```
active(logging) if
  [Network intrusionDetected not],
  active(authentication)

active(logging) if
  [Network intrusionDetected]
```

The first rule denotes that the logging aspect module can only be activated if the authentication aspect is also active when there is no network intrusion detected. The second rule says that the logging aspect does not depend on any other aspects when there is a network intrusion detected.

6.4 Conclusion

In this chapter we worked out two cases to show the applicability of the composite aspect object model in real-world cases. The first case showed how the model can be used in a distributed environment with aspects that depend on the client where the object lives and emphasized the dynamic weaving capabilities of the composite aspect object model. The second example of secured objects showed how we can resolve conflicts. Both cases also demonstrated how aspect interactions can be encapsulated in the aspect configuration modules so that both the part-objects and the aspect modules are reusable in different contexts.

Chapter 7

Conclusion

Separation of concerns has always been a major goal in software engineering. It has come to the attention that some concerns cannot be modularized cleanly in current programming languages. Such concerns lead to tangled and crosscutting code and therefore language extensions are proposed to encapsulate these crosscutting concerns. Most of these language extensions are based on creating a pre-compilers.

Pre-compilers work fine until we come to realize that some aspects need to be introduced dynamically at run-time. Examples of such systems are component systems that support dynamic plugging and unplugging of components or distributed systems that want to weave aspects into migrating objects to support their new environment.

There are some issues when composing a program using aspects. For example, an aspect may depend on one or more other aspects in order to perform its task or the order of the aspects may be depending on several conditions. These composition issues are even more important when aspects can be dynamically added to the application. Solving these composition issues in most of the current composition tools is impossible or harms the reusability of the aspects.

We proposed a composite aspect object model that supports advanced separation of concerns in a dynamic environment. It does so by using the reflective capabilities and the meta-architecture of the underlying language. Aspects are first-class entities that are divided into part-objects (objects that provide the behavior of the software concern) with an activation scheme (that determines when and how the objects should become active). When a message is sent to a composite aspect object it uses the activation schemes to determine which part-objects should be activated in order to process the message. With composite aspect objects the aspects can be added, adapted or removed at run-time.

We have shown that logic metaprogramming provides an expressive means to specify the activation schemes. Indeed, the declarative nature allows us

to create activation schemes that are easily reusable and definable. We have split up the logic activation schemes into aspect modules that define how and when the part-objects should be activated. Aspect modules also include compatibility rules that allow us to declaratively specify what is expected from the part-objects in order to have a correct behavior. Further we have order modules that specify the dynamic sequence in which the aspect modules should be activated. Aspect activation modules are used to specify dependencies between the aspect modules of a composite aspect object. The logic activation scheme allows us to specify relationships between multiple aspects in a declarative way, helping us to solve the composition issues without harming their reusability.

We can view our model as a meta-architecture that configures the meta-object protocol using logic meta programming rules for supporting dynamic aspect compositions.

The cases described in chapter 6 have been implemented to test the implementations of the composite aspect object model and the logic activation schemes. They showed the usefulness of the model and the logic activation schemes.

7.1 Technical Contributions

This dissertation has led to the following implementations:

- Composite aspect object model has been implemented using the advanced meta-architecture of Smalltalk.
- Logic activation schemes have been implemented using SOUL as the declarative metaprogramming language.
- The run-time reasoning library for SOUL has been implemented. It extends and enhances a prototype that was provided by Kris Gijbels.

With these implementations we have shown both the feasibility of the composite aspect object model and the logic activation schemes.

7.2 Future Work

This dissertation provides a model for doing advanced separation of concerns with dynamic aspects. Further it uses logic metaprogramming to define activation schemes. There are however several areas that need to be further looked at:

7.2.1 Efficiency

The use of a logic metaprogramming language for defining the activation rules provides an expressive means, but lowers the efficiency of our model. It is possible to optimize SOUL or logic metaprogramming language by means of advanced caching (so that we can avoid calculating the same results again) and compiling the rules into native code as we explained in section 5.9.

7.2.2 Language Extensions

The composite aspect object and the logic activation scheme provide the implementation mechanisms for dynamic weaving of aspects. A language could be constructed that eases the application of the composite aspect object in practical situations. Such a language should also have support for specifying the points where the composite aspect object should be adapted and how this should be done.

7.2.3 Validation

We have validated our model by implementing some small cases, because of the limited time available. However, using the composite aspect object model in a real world applications is necessary to further assess the ease of use of our model.

7.2.4 Modelling Techniques and Process

A good implementation of an application starts with a good design. There is a need for enhancing the current modelling techniques to take aspects into account. Much work still needs to be done expressing a model using static or dynamic aspects in an application. There is also a need to adapt the software process for developing software using aspect oriented development tools.

7.2.5 Language Dependence

Currently the model has been implemented in Smalltalk. Further study is needed to see if the model is implementable in other dynamically typed object-oriented languages.

The model we have presented is only applicable for dynamically typed systems, because the set of messages understood by the composite aspect object is determined by the logic metaprogramming rules. We feel that porting the system to static typing system would become possible if we pose certain restrictions in the composite aspect object, like fixing the set of understood messages upon creation. However, this has not been looked into and should be studied further.

7.2.6 Selecting Objects for Adaptation

When dealing with dynamic aspects we have to define the points where the aspects need to be woven into objects. In this dissertation we have always done this before or after the execution of certain methods at the meta level. Using the declarative metaprogramming language and the runtime reasoning library we could define logic queries over the objects that are living in a machine to determine the objects that need to be adapted.

7.2.7 Dynamic Unweaving

In this dissertation we have not explicitly discussed the unweaving of aspects. There are two possibilities for the unweaving of aspects. We could disable the aspect by adapting the aspect activation module or we could remove the part-object and the aspect module from the composite aspect object. Both options are feasible in the composite aspect object model with few modifications.

Appendix A

Implementation

A.1 Introduction

In this chapter we describe our design and implementation issues of the composite aspect object model. As our implementation language we have chosen for Smalltalk, because of its meta-architecture that is one of the most advanced. Another reason was the availability of the logic metaprogramming language SOUL for the Smalltalk language that enabled us to implement the logic activation scheme we described in chapter 5.

A.2 Implementation Issues

In this section we describe some of the implementation issues we encountered while implementing the composite aspect object model in Smalltalk.

A.2.1 Reifying Messages at Run-Time in Smalltalk

When we take a look back at the conceptual model of the composite aspect object in figure 4.1 on page 38, then we can see that the composite aspect object decides what object to activate next depending on the message it received. We can do this in two ways:

1. Implement a method for each message that can be send to the composite aspect object and execute the activation scheme for that method.

2. Reify the message that was sent to the composite aspect object and reason.

Because the structure of the composite aspect object can change at run-time (the part-object can be changed and the activation strategy can be adapted) the protocol of the composite aspect object can also change at run-time. Hence the first option of implementing a method for each message that could be send to the composite aspect object is overly static and cannot be

used in this case. Thus each time a message is sent to the composite aspect object then its meta-object needs to send a reified version of that message to the composite aspect object. In Smalltalk, when a message is sent to an object and that message does not appear in the protocol of the class of the object then the message that was sent is reified and passed as an argument to the method doesNotUnderstand:. The composite aspect object overrides this method and uses the reified message to decide what objects to activate next. One problem however, is that messages that are understood by the class Object and its parent classes are not catched since they appear in the protocol. To overcome this problem we had to override certain methods and redirect them manually to the doesNotUnderstand: method.

A.2.2 Method Wrappers

In this section we explain the use and implementation of method wrappers [BFJR98]. Method wrappers are useful to add code before and after a method of a certain class without altering the hierarchy of its class. Smalltalk is a pure object-oriented language in the sense that everything is represented as an object, so a class of an object is also an object. One of the responsibilities of classes is to keep the shared behavior of its instances. Therefore, each class object has an instance variable methodDict, that points to a dictionary mapping selectors onto compiled methods. The compiled methods themselves are also represented as objects of the class CompiledMethod. One of the methods of CompiledMethod class, named valueWithReceiver:arguments:, is for executing the method in the context of an object with a set of parameters. Method wrappers are installed by replacing the CompiledMethod object for a certain selector in the method dictionary with an instance of a method wrapper. The MethodWrapper class extends the CompiledMethod class with an instance variable clientMethod, keeping a reference to the old CompiledMethod instance that it replaced. Furthermore it overrides the behavior of valueWithReceiver:arguments: with the following code:

```
MethodWrapper>>valueWithReceiver: anObject
               arguments: anArrayOfObjects
    "This is the general case where you want both a
     before and after method, but if you want just a
     before method, you might want to override this
     method for optimization."
    self beforeMethod.
    ^ [self clientMethod valueWithReceiver: anObject
                         arguments: anArrayOfObjects]
        ensure: [self afterMethod]
```

The overridden method does three things:

1. first executes some beforeMethod that should be executed before the method it replaced
2. it calls the old method that it replaced
3. eventually it calls some behavior that should be executed after the original method.

Method wrappers are used by extending the MethodWrapper class and override the beforeMethod and afterMethod methods.

A.2.3 Changing the Notion of Self

As discussed in section 4.2.4 we sometimes want to change the notion of self depending on the composite aspect object it is put. Messages that are send to "self" can be identified when m is a message so that $m_{sender} = m_{receiver}$. To detect such messages we can use method wrappers we explained in the previous section. When an object is added to a composite aspect object, where we want to bind its "self" to the composite aspect object. For this we have introduced a *self-redirector*. A self-redirector is a method wrapper with the specific purpose of intercepting messages that were sent to self and redirect them to another objects. All the messages that were not sent from within the object to "self" should be executed normally. The class SelfRedirector inherits from the class MwMethodWrapper and has three instance variables:

1. recipient:
   reference to the object where the messages send to self should be redirected to. In our case this will be the composite aspect object.
2. object:
   method wrappers are wrapped around classes and not around objects. We want to redirect the self of one specific object to another object and not that of a set of classes. The object instance variable determines the object where the redirector should be applied to.

```
SelfRedirector>>valueWithReceiver: anObject
                arguments: anArrayOfObjects
    | client |
    "get the sender of the message"
    client := thisContext sender client.
    "check to see if the sender of the message"
    "equals the receiver of the message and"
    "that the method wrapper is active"
    (client = anObject and: [self object = anObject]
                      and: self active)
```

```
ifTrue: [
    "redirect the call to the composite aspect object"
    ^self recipient perform: self selector
              withArguments: anArrayOfObjects
]
ifFalse: [
    "execute the method this is wrapped around"
    ^self clientMethod valueWithReceiver: anObject
                       arguments: anArrayOfObjects
]
```

A.2.4 Adapting a Composite Aspect Object

Until now, when the composite aspect object had to be adapted at run-time we have always described that this was implemented with a dynamic weaver. This weaver is invoked at certain points in the execution. To avoid that the invocation of the dynamic weaver gets tangled in the code of the base program we can make use of the method wrappers or put the dynamic weaver as a part-object in the composite aspect object that is constructed at compile-time. For example, in section 6.2.5 we had to adapt the composite aspect object when the message comeFullyUpOnReload: was sent to an object of the class Book installed. For this we can install a method wrapper that adapts the composite aspect object.

A.3 Design

This section discusses the design of the composite aspect object model and its implementation in Smalltalk.

A.3.1 Class Diagram

Figure A.1 shows the class diagram of the CompositeAspect class. Each composite aspect object has a reference to a strategy and to a collection of part-objects that can be activated to handle a message. The strategy has to be a subclass of the ActivationScheme class that provides some basic functionality. The strategy is used to determine what part-object and what method that needs to be activated to handle a certain message that was sent to a composite aspect object. When a part is added with its self bound to the composite aspect object, then a reference is kept to the metaclass that is responsible for redirecting the self of that part.

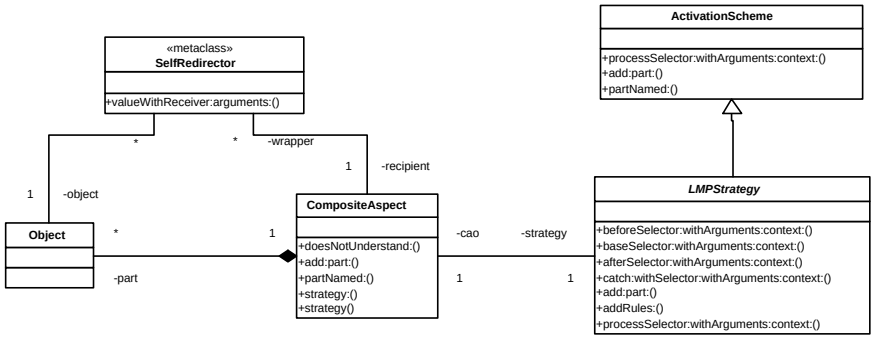

Figure A.1: UML Class Diagram of the Composite Aspect Object Implementation in Smalltalk

A.3.2 Processing a Message

In this section we discuss how the objects collaborate to process a message that is sent to the composite aspect object. For this we take the example of the secured objects that we discussed in section 6.3. We have some data encapsulated in an object that can be accessed by the selector showData. We secure the data object using the composite aspect object model by weaving both a logging and authentication aspect into the object. Figure A.2 shows the sequence diagram what happens if an object client sends the message showData to the composite aspect object[1].

A.4 Conclusion

In this chapter we showed the feasibility of the composite aspect object model by discussing the implementation and its design of the composite aspect object in Smalltalk.

[1]The sequence diagram does not show all the interactions that are necessary to handle the message as it would make the sequence diagram less understandable

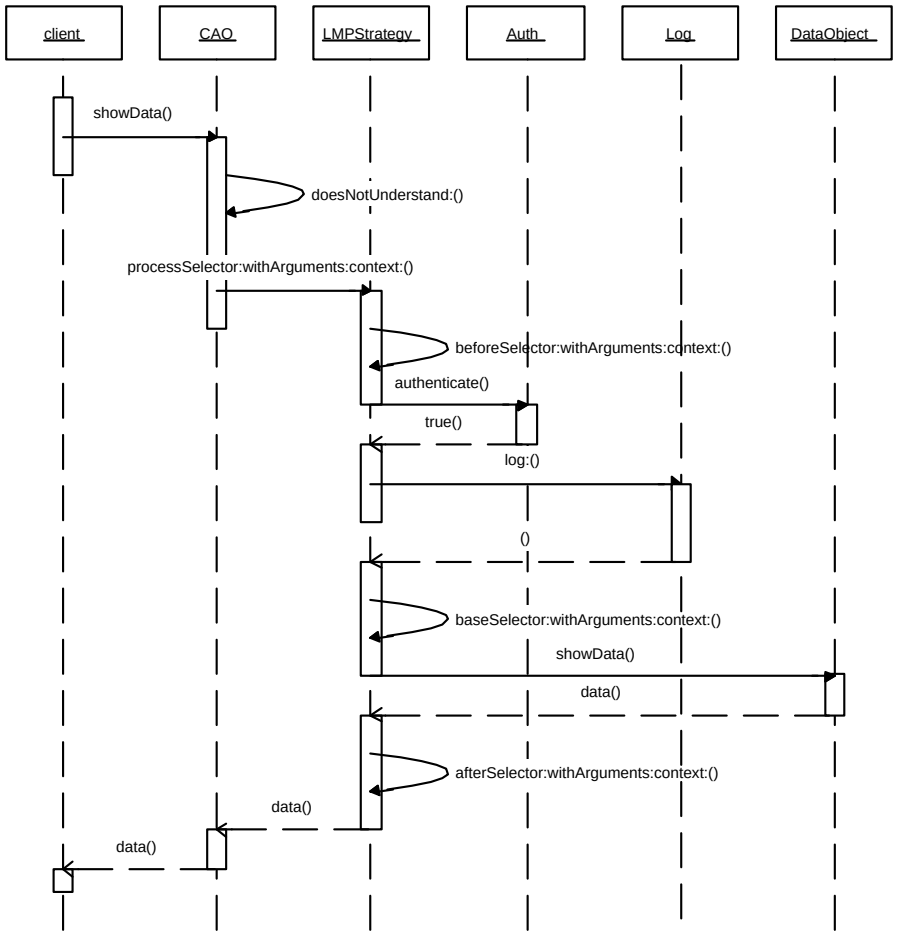

Figure A.2: UML Sequence Diagram for handling the message showData

Bibliography

[AT98] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. In *ECOOP98 Proceedings*, 1998.

[ATB00] M. Aksit, B. Tekinerdogan, and Lodewijk Bergmans. The six concerns for separation of concerns. In *ECOOP2000 Proceedings*, 2000.

[BA01] Lodewijk Bergmans and Mehmet Aksits. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BFJR98] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. Wrappers to the rescue. *Lecture Notes in Computer Science*, 1445:396–??, 1998.

[BMD00] J. Brichau, W. Meuter, and K. De Volder. Jumping aspects. In *ECOOP2000 - Workshop on Advanced Separation of Concerns*, 2000.

[BMD02] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specifc languages with logic metaprogramming. Submitted to the GSCE/SAIG 2002 Conference., 2002.

[Boa00] IEEE Standards Board. Ieee standards board. recommended practice for architectural description of software-intensive systems, ieee std 1471-2000. IEEE Standards, September 2000.

[Bol99] K. Bollert. On weaving aspects. In *Aspect-Oriented Programming Workshop at ECOOP99*, 1999.

[DD99] K. De Volder and T. D'Hondt. Aspect-oriented logic meta programming. *Lecture Notes in Computer Science*, 1616:250–272, 1999.

[Dec02] Diego Gomez Deck. Remote smalltalk. Available from http://minnow.cc.gatech.edu/squeak/2288, March 2002.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[FBLL02] Robert E. Filman, Stuart Barrett, Diana D. Lee, and Ted Linden. Inserting ilities by controlling communications. *Communications of the ACM*, 45(1):116–122, 2002.

[Fla94] Peter Flach. *Simply Logical: Intelligent Reasoning by Example*. John Wiley, 1994.

[FS97] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison Wesley Longman, Inc., 1997.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 411–28. IEEE Comput. Soc, Los Alamitos, CA, USA, October 1993.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.

[Kru95] Philippe B. Kruchten. The 4 + 1 view model of architecture. *IEEE Software*, 12(6):42–50, November 1995.

[LK97] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Xerox Palo Alto Research Center, Palo Alto , CA , USA, February 1997.

[LLM99] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.

[LOO01] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, 2001.

[Mae87] Pattie Maes. Computational reflection. *Technical Report, Intelligence Laboratory, Vrije Universiteit Brussel*, 87(2), 1987.

[ML98] Mira Mezini and Karl J. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Conference on Object-Oriented*, pages 97–116, 1998.

[OT99] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717)16APR99, IBM Thomas J. Watson Research Center, Yorktown Heights, NY., 1999.

[OT01] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[PSDF01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC: A flexible solution for aspect-oriented programming in Java. *Lecture Notes in Computer Science*, 2192:1–??, 2001.

[PTC00] A. Diaz Pace, F. Trilnik, and M. Campo. How to handle interacting concerns? In *ASOC2000, Advanced Techniques for Separation of Concerns, OOPSLA2000*, 2000.

[Pul00] E. Pulvermller. Aspect composition applying the design by contract principle. In *Generative and Component-Based Software Engineering (GCSE)*, 2000.

[Roy90] Peter Lodewijk Van Roy. Can logic programming execute as fast as imperative programming? Technical Report CSD-90-600, Universit Catholique de Louvain, Belgium, 1990.

[RV97] B. Robben and P. Verbaeten. Aspects should not die. In *Position paper at the ECOOP '97 workshop on Aspect-Oriented Programming*, 1997.

[Str97] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley, 1997.

[Sul01] Gregory T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings TOOLS USA '98*, pages 112–124, 1998.