Abstract

Interactive software systems can now be found on many desktops. They are used by their users to reach a level of productivity that would not have been possible without them. The development of these systems, more specifically the development of interactive software applications, is a difficult task. Software developers are confronted with two main questions, namely what to build and how to build it. The question of what to build can be addressed successfully by leitmotifs and metaphors. The Tools and Materials Metaphor is a design methodology which has been developed for the domain of office work. It works best for users who are knowledgeable with respect to their tasks and can take over the responsibility for it. Systems designed according to this methodology offer a flexibility and degree of freedom seldom found in traditional software systems. How to build it, that is the software design and implementation of such systems, is best supported by an application framework. A framework represents the key abstractions from a domain as reusable classes. Furthermore, it sets up the right relationships between these classes, so that users may not only reuse single classes but also full designs consisting of dependent and collaborating classes. The development of such frameworks is a highly demanding task requiring much experience and thoughtful design.

This report presents shortly the design metaphors from the Tools and Material Metaphor. It then focuses on the design of a Smalltalk Framework that implements the most important aspects from the metaphors. The framework is thoroughly based on prior experience in its conventional parts on how to design tools, aspects, materials and their environment. In addition it comprises some innovative parts, most notably the Aspect Browser, a tool to adequately emulate multiple inheritance in Smalltalk. This report goes down to the level of single methods of a class and thus provides to our knowledge the first detailed technical documentation of such a framework for the Tools and Materials Metaphor.

A concise summary of this report will appear as [RSS96].

Dirk Riehle and Martin Schnyder

Hamburg, Germany and Zürich, Switzerland, July 1995

Copyright 1995 Dirk Riehle and UBS/UBILAB

Table of Contents

1 Introduction and Overview **5**
1.1 Introduction 5
1.2 Overview 6
1.3 Graphical notation 6

2 The Tools and Materials Metaphor **9**
2.1 Context of the metaphors 9
2.2 The metaphors 9
2.3 Framework overview 10

3 Basic Framework Support **15**
3.1 Overview of the basic framework cluster 15
3.2 Enhanced change/update protocol 16
3.3 Class Retrieval and Late Creation 17
3.4 Class specifications 19
3.5 The class manager 21

4 Materials and Aspects **25**
4.1 Overview of the material cluster 25
4.2 Aspects in general and in Smalltalk in particular 25
4.3 The material classes 26
4.4 The aspect classes 27
4.5 The aspect clause class 29

5 Tools in an Environment **31**
5.1 Overview 31
5.2 Tools – rationale and structure 32
5.3 Tool components 35
5.4 Tool composition 37
5.5 Available tools 40
5.6 The environment 40

6 The Aspect Browser **43**
6.1 Emulating multiple inheritance using tool support 43
6.2 The design of the Aspect Browser 45
6.3 Using the Aspect Browser 47
6.4 Method dispatch for multiple inheritance in Smalltalk 52

7 Conclusions **55**
7.1 Observations on Smalltalk 55
7.2 Summary 57
7.3 Future work 57

Bibliography **57**

1 Introduction and Overview

This report presents the design and implementation of a Smalltalk application framework for the Tools and Materials Metaphor. The Tools and Materials Metaphor is used to build interactive software systems and comprises some advanced features not available in current application frameworks. The framework offers a metalevel architecture for class specifications that enhances the expressive power for programming. It introduces a rationale and a tool for dealing with aspect classes in the context of a single inheritance system. Finally, it presents a design for constructing software tools based on the experience gained with previous frameworks. This chapter introduces the background and motivation for the framework and gives an overview of the report.

1.1 Introduction

The Tools and Materials Metaphor is a methodology for developing interactive software systems. Next to several methods addressing the different aspects of a software development process, it particularly provides developers with an elaborate set of metaphors which serve as an efficient means to both interpret the application domain and design a software system for it. The metaphors of tool, material and others more help developers to envision the future system and discuss its properties.

This report presents the design and implementation of an application framework which is used to implement software systems according to the Tools and Materials Metaphor. The framework has been developed using IBM Smalltalk running on Microsoft Windows PCs.

Application frameworks have been acknowledged as a major aid in developing object-oriented software systems [WG94, GOP90, Boo94, Bis95]. Frameworks are developed by experienced software developers of the application domain. They use their expertise to set up a software architecture for the application domain which they implement using, for example, object-oriented design. Users of the framework reuse the design and thus the domain expertise captured within. Their software directly profits from the usually well thought-out design of the framework.

The application framework is based on prior experience gained during the design of a C++ framework for the Tools and Materials Metaphor. The C++ framework was designed over a period of three years at the University of Hamburg [Rie95a, RZ95]. The Smalltalk framework, however, is not a simple port but comprises several new aspects.

1.2 Overview

Chapter 2 introduces the relevant concepts from the Tools and Materials Metaphor. It stresses the aspects of the presented metaphors from a software design point of view. Discussed are the notion of tool, material, aspect and environment as well as some general implementation techniques that are used to implement them. It further gives an overview of the framework.

Chapter 3 discusses the basic framework support derived from the Smalltalk system classes. It became apparent very early that some extensions to the fundamental classes of the underlying Smalltalk system had to be done. This comprises enhancements of the basic change/update mechanism as well as extensions to the metalevel architecture. Representation of class properties as first class objects and general ways of working with them (Class Retrieval, Late Creation [Rie95b, Rie95c]) are presented.

Chapter 4 deals with materials and aspect classes. Materials are usually application specific classes and can be supported by a framework only on a very general level. Aspect classes, which represent a tool's view on a material, were designed from experience. In the standard Tools and Material Metaphor design orthodoxy material class interfaces are built by inheriting from several aspect classes. We shortly introduce the rationale behind this and open up the way to the Aspect Browser in chapter 6 which is used to handle aspect classes for the framework.

Chapter 5 presents the framework classes for software tools and their environment. Software tools all have a common structure which makes them a fine target for framework support. Next to tools we discuss their embedding into the system context performed by the environment and desktop classes.

Chapter 6 presents the Aspect Browser, a tool to model multiple inheritance by copying and maintaining aspect classes within material class interfaces.

Chapter 7 finally summarizes the report, discusses strengths and weaknesses of the framework, points out open problems and presents some conclusions. An outlook on future work closes the report.

1.3 Graphical notation

Three kinds of diagrams appear in this report. Cluster diagrams show the structuring of systems in the large. They consist of rounded and shaded rectangles which represent clusters of classes. A cluster is a grouping of classes based on some common theme shared by them, for example all classes used for tool construction.

Class hierarchy diagrams show a class hierarchy, for example all classes appearing in a certain cluster. These diagrams omit use relationships and consist just of the class names and a line showing the inheritance relation. Figure 1-1 shows such a diagram. It is to be read from the left to the right, showing the superclass on the left and all subclasses to the right of it.

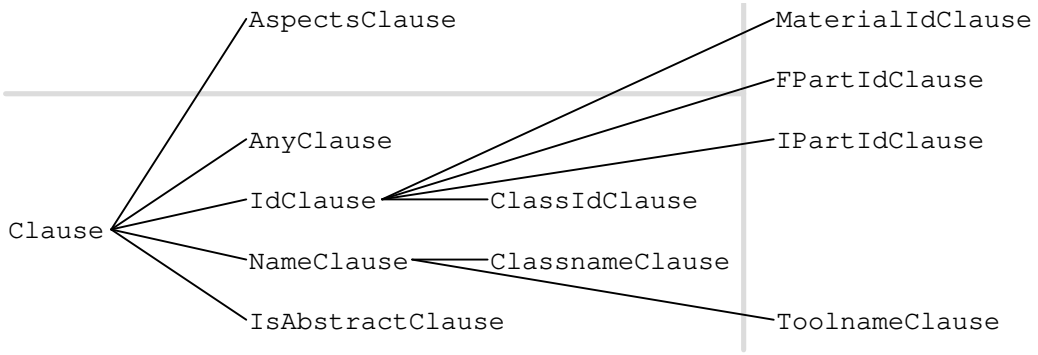

Figure 1-1: A figure from chapter 3: The clause classes from the basic framework cluster.

Class and object diagrams are based on the object-oriented Booch notation explained in detail in [Boo94]. Classes are depicted as clouds. Inheritance relationships are depicted by arrows and use relationships by circles at the end of an interconnecting line between two classes. Figure 1-2 shows an example of this kind of diagram.

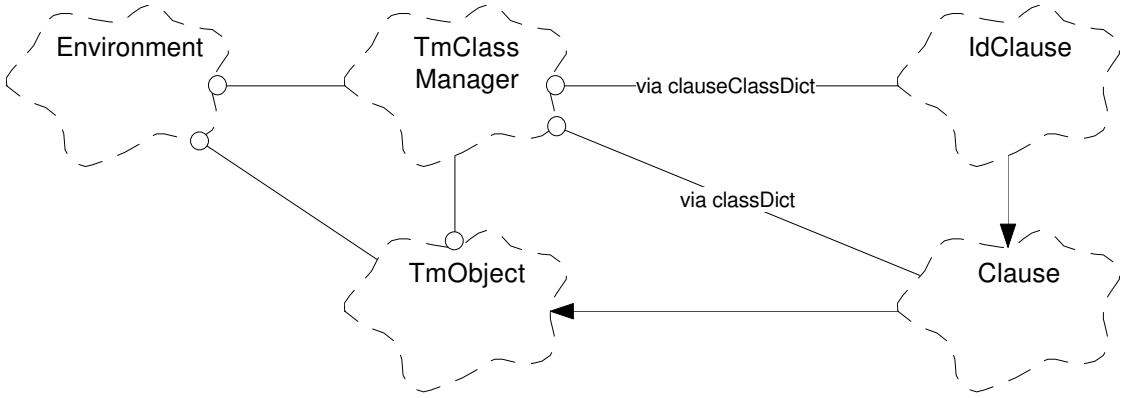

Figure 1-2: Design and implementation structure of *TmClassManager*, its instance creation and the tables maintained by it.

2 The Tools and Materials Metaphor

The Tools and Materials Metaphor offers a number of so called metaphors which serve as a guideline to software developers both for the interpretation of the application domain and the construction of a software system. By spanning the full range of analysis, design and implementation activities, the metaphors have proved to be a unifying approach for developing interactive software systems. They significantly help to bridge the gap between largely informal application domains and a formal software system. This chapter introduces the metaphors of tool, aspect, material and environment which provide the needed background to understand the design of the application framework. Based on this, a short overview of the framework is presented.

2.1 Context of the metaphors

The application domains of the Tools and Materials Metaphor are office work and workshop like settings. It is assumed that users are knowledgeable about their work and perform it in skillful ways. Thus, they need no restrictive system but have to be in control of what they are doing. Furthermore, users are acknowledged to be responsible for their work and its outcome. We call this understanding of a user's skills and responsibilities the *leitmotif* guiding our design of software systems.

We make this understanding of human work concrete by using design metaphors. Examples of design metaphors are tools, aspects, materials and environments which provide both a way of perceiving and interpreting application domains as well as constructing software systems for it. The design metaphors are used as a communication vehicle both to discuss the application domain with users and to discuss software designs amongst software developers.

The application of the metaphors takes place within an evolutionary and participatory process which is not discussed here. The literature offers several starting points to delve into process issues [KM93, KGZ93, BGZ95, FG92].

In the following we will introduce the metaphors, discuss their meaning for software design and shortly hint at standard implementation techniques for them. Finally, we give an overview of the framework presented in the subsequent chapters.

2.2 The metaphors

Users use tools to work on materials. A tool is a means of work used to analyze and manipulate a material which is the outcome of work. When working on a material, a tool makes use of a specific aspect offered by this material. An aspect represents a possible way of working with a material needed to perform a certain task. Thus, it is a view on a material required by a tool.

The trinity of tool, aspect and material, each one being called a metaphor, is the fundamental idea behind the overall approach of the Tools and Materials Metaphor. The distinction between tools and materials is a powerful way of interpreting human work.

When looking at an application domain, we first have to make a distinction between those objects which are used as tools, for example pencils, typewriters, folders and so forth, and those objects which are used as materials, for example plain paper, forms, address books and many more. Having made this distinction, we analyze how tools are used to work on materials and which characteristics of these work tasks can be captured formally as aspects. On a technical level, aspects are turned into aspect classes representing the interface of materials to tools.

Aspects clarify the context of use. They are used to determine whether an object is a tool or a material. A pencil is a tool when writing on a paper but it is a material when being sharpened by a pencil sharpener. The object which uses an aspect is a tool while the object which offers an aspect is a material in the context of a particular object constellation.

Tools and materials have a state of their own. A tool is used to implement idiomatic ways of handling a material. It represents the material to the user dependent on the tasks the tool has been designed for. Its state can be subdivided into presentation state, used to visually present the material and task based state which holds the objects describing the current use situation of the material from the tool's point of view.

A well designed tool is unobtrusive and stays in the background. The user's focus rests on the material which it accesses and manipulates according to its needs while using the facilities offered by a tool. A material can usually be used by a number of different tools using different aspects. Therefore a material's interface is built from aspect classes which are class interfaces representing the different aspects as the formalized context of use of the material. Aspect classes are usually abstract classes which leave the implementation of behavior and the needed implementation state to the material.

Next to the metaphors of tool, aspect and material several other metaphors are needed to successfully interpret an application domain, for example archives, automata, environments, media and so forth. Here, we shortly introduce the metaphor of environment.

Whatever we do, we do it with an understanding of space within which we arrange our things. This space, within which we organize our tools and materials, may have several dimensions, spatial as well as logical. It is called the environment. We need this multidimensional space to make sense of the perception of our work place and to relate to other peoples work places. It furthermore represents a closure to our world of work so that we have clear boundaries and procedures with other people's environments.

There is much more to the metaphors than can be presented here. Some of it can be found in the literature, for example in [BCS92, Rie95a, RZ95, BGZ95].

2.3 Framework overview

We will now give a short overview of the framework. Roughly speaking, for each metaphor there is a grouping of classes that are used to implement designs adhering to the metaphor.

The framework so far can be separated into four main clusters. A cluster is a set of loosely grouped classes which are informally put together according to a common theme like being basic framework classes, tool classes or by being part of a certain application. Clusters can be realized through the IBM Smalltalk concept of Application, which is essentially a synonym for cluster. Our notion of cluster stems from Eiffel [Mey92]

Figure 2-1 shows an overview of the framework. It is hierarchically structured with the clusters being represented as rounded rectangles. A dependency between two clusters is expressed as an arrow going from the dependent cluster to the cluster on which it depends.

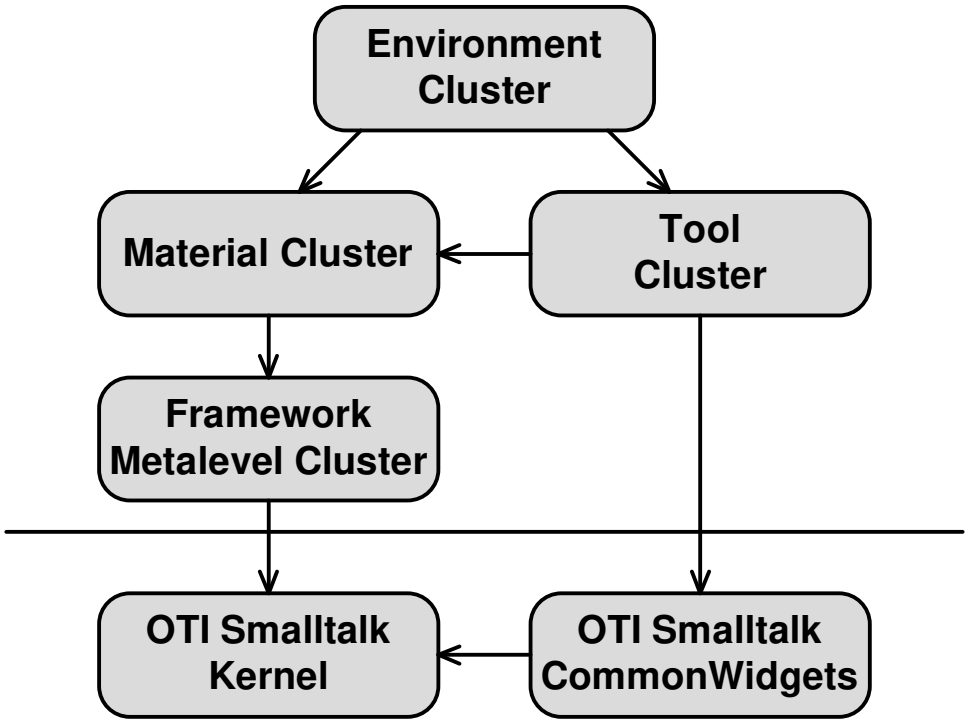

Figure 2-1: Framework overview based on the grouping of classes into clusters (applications in IBM Smalltalk terminology).

The two clusters below the line are the system provided *Kernel* and *CommonWidgets* clusters. They are part of OTI Smalltalk on which IBM Smalltalk is built.

On top of the Kernel the *FBaseCluster* is built, a cluster which comprises basic framework classes which offer an enhanced change/update mechanism and some new meta-level facilities. The change/update mechanism is tied to the class *TmObject* which is the root class of almost all framework classes.

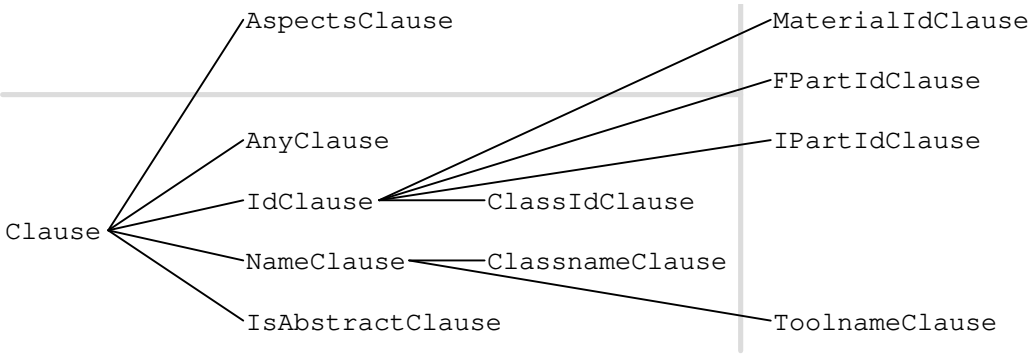

Figure 2-2: The clause classes from the basic framework cluster.

The class *Clause* and its subclasses viewed in figure 2-2 represent simple class properties as first class objects. They are used for a simple representation of class semantics which is needed to determine classes based on given specifications. The needed facilities for looking up classes are located as *TmObject* class methods in turn. The most important benefit of such a facility is to retrieve classes and create objects without having to determine the classes directly which in turn makes systems much easier to change [Rie95b].

Figure 2-3 shows the material cluster comprising the general superclass *Material* and some basic material (*Folder*, *Document*, *SimpleDocument*) and aspect classes. Materials can only be partially captured by a framework since they are usually application domain dependent.

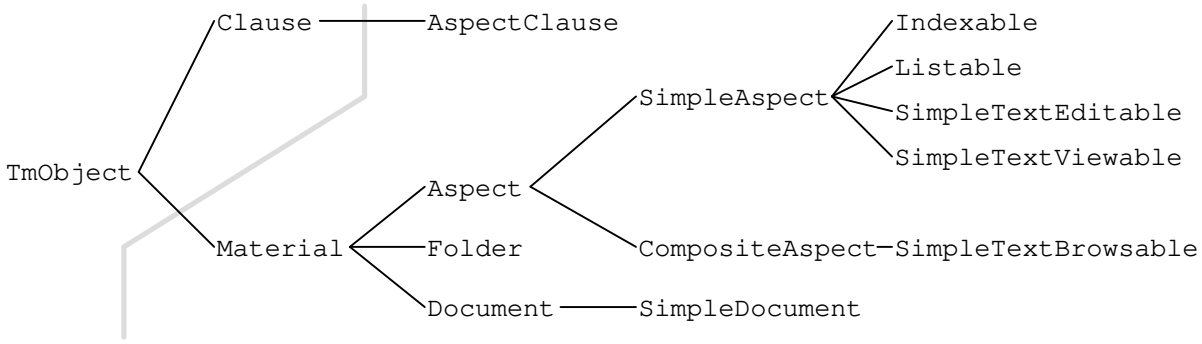

Figure 2-3: The class hierarchy of classes from the material cluster. It groups together all aspect and material classes comprised by the framework

Figure 2-4 shows the basic classes from the tool cluster which in turn is viewed in figure 2-5. These classes comprise much of the design experience for software tools according to the Tools and Material Metaphor known today. Tools are structured horizontally according to the pattern of separation of interaction from functionality and vertically according to tool composition rules from tools and subtools. Objects are linked via the change/update mechanism. The underlying patterns are explained in more detail in [Rie95a, RZ95].

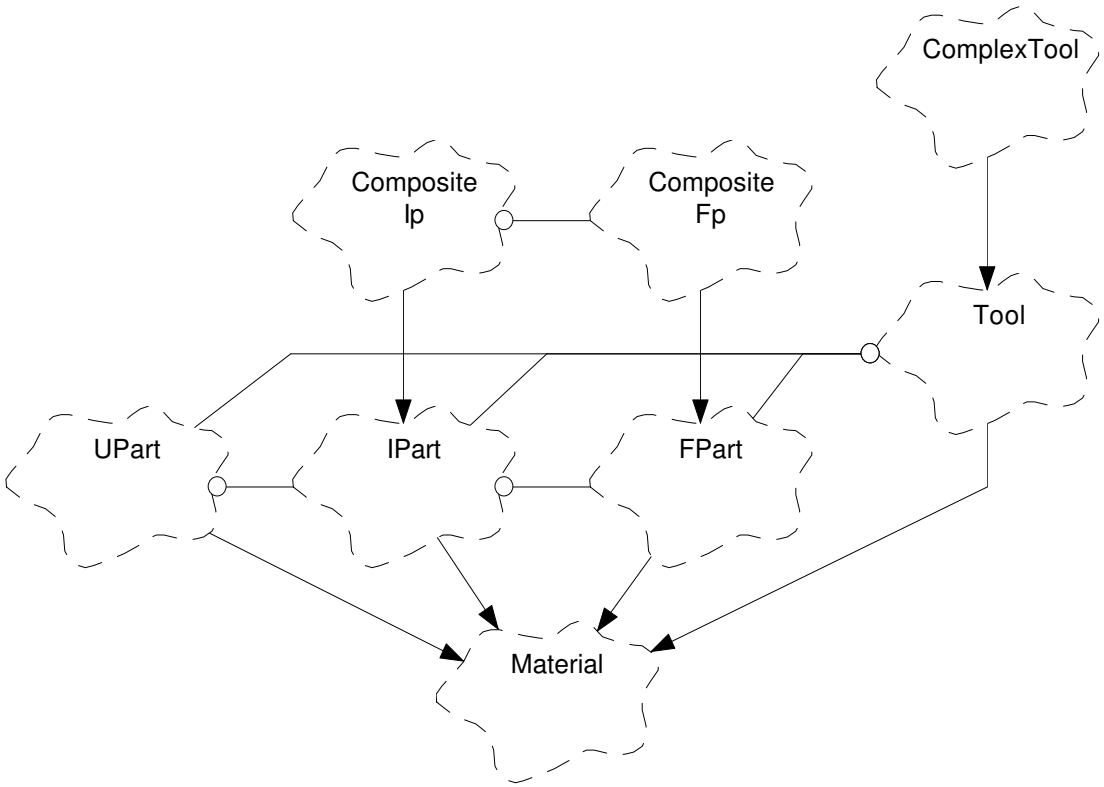

Figure 2-4: The basic tool classes for composing tools vertically out of user interface, interaction and functional parts, and horizontally out of tools and subtools through composite classes.

The full set of classes from the tool cluster are shown in figure 2-5. It comprises the framework classes *UPart*, *IPart*, *FPart* and *Tool* as well as some specific tools like a lister, a simple editor, viewer and browser.

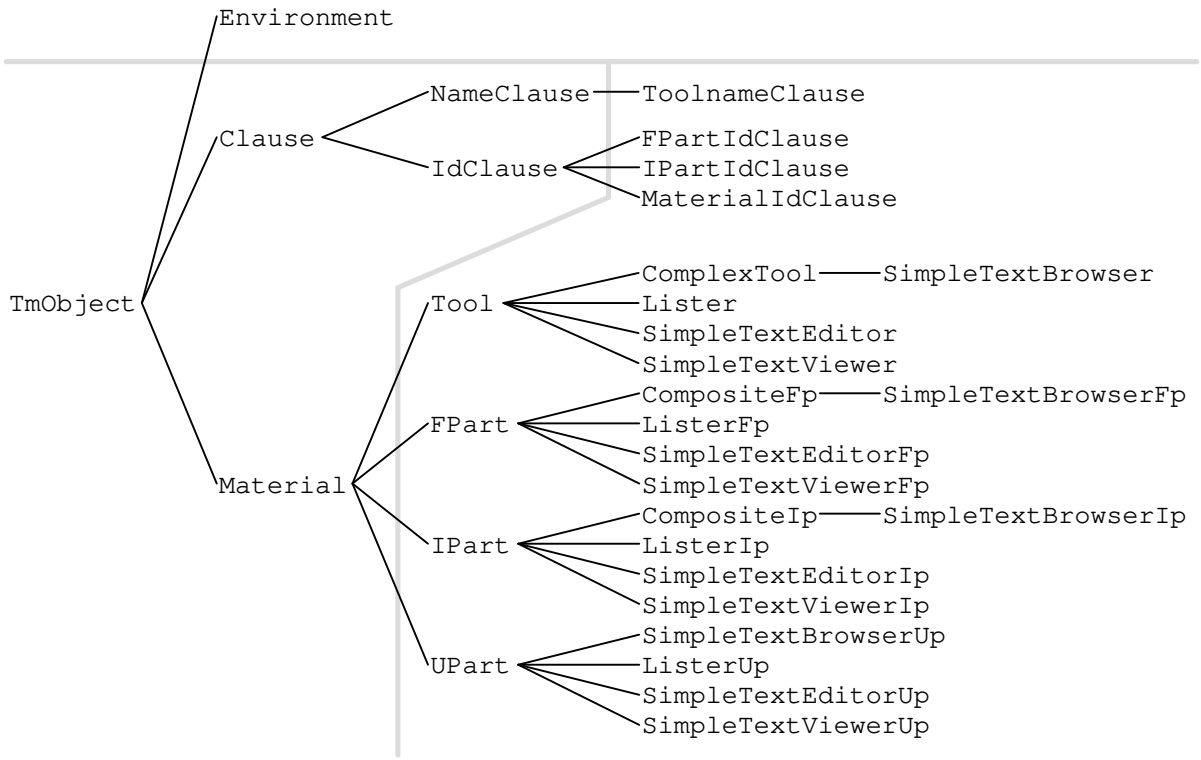

Figure 2-5: Overview of the classes from the tool cluster (on the right).

Finally, figure 2-6 shows the environment class which ties all the clusters together and is the first class to be instantiated during system startup. It creates and manages all tools.

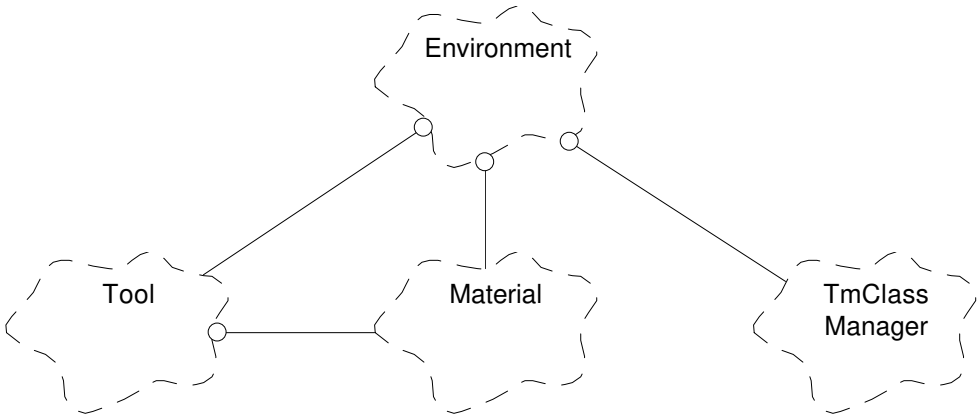

Figure 2-6: The environment cluster and its relationships to classes of other clusters.

This report presents the clusters successively. It discusses its class interfaces in detail. It may serve as a technical documentation, however, it is not a cookbook for using the framework.

3 Basic Framework Support

Very early it became apparent that several enhancements to the basic class *Object* had to be done. Instead of changing the vendor provided class we chose to introduce an intermediate superclass, called *TmObject*. It is the root class of all Tools and Material classes (prefixed with *Tm*) of the framework. *TmObject* offers an enhanced support for the general change/update protocol as well as metalevel facilities that let clients retrieve classes and create objects using simple specifications. We will take a look at these facilities in turn now.

3.1 Overview of the basic framework cluster

FBaseCluster, called the basic framework cluster from now on, comprises the classes *TmObject*, *TmClassManager*, *Clause* and some other classes as well. The classes are listed in figure 3-1. The three classes just named are the most relevant abstractions offered by the cluster.

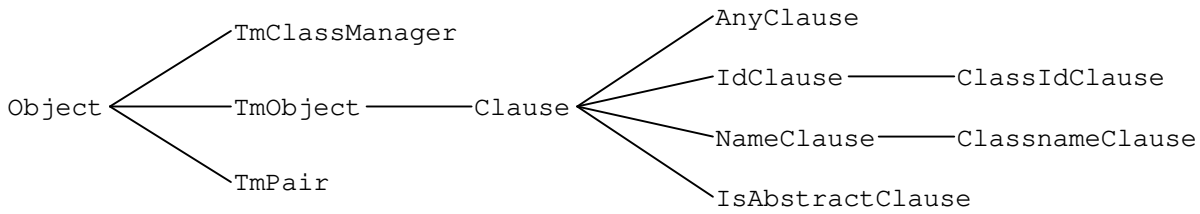

Figure 3-1: Class hierarchy of the basic framework cluster.

The class *TmObject* offers both new instance methods for an enhanced change/update protocol and new class methods for retrieving classes and creating objects based on class specifications. Section 3.2 presents the enhanced change/update protocol and section 3.3 to 3.5 introduce class specifications and a rationale for using them.

The class *Clause* and all its subclasses are the basic constituents of class specifications. They represent simple properties of a class, for example whether the class is abstract or not, has a certain name or id or whether its instances can work with specific other classes instances. The class tree is discussed in section 3.3.

The class *TmClassManager* serves as a managing facility to keep track of the properties of the different classes from the framework. It holds a list of clause instances for each class with each of this clauses representing a specific property of a class. It further maintains tables for fast lookup of those classes based on clause instances. This is part of the mentioned specification mechanism and is dealt with in section 3.4 and 3.5.

3.2 Enhanced change/update protocol

Object-oriented software design and implementation is very much about managing dependencies between objects. Each object that uses another object becomes dependent on it. Most of the time, the dependency of a superordinate object on a subordinate object is uncritical. The superordinate object controls the subordinate object and takes care that its state is synchronized with the subordinate object's state.

Some of these dependencies, however, are critical and require feedback from the subordinate object to the superordinate object. The classical example is a view object which uses a certain model object and thus depends on the model's state. Since there may be several views on a single model, the model might change without all of the views taking notice of it. Thus they might loose synchronization. These problems are discussed in more depth in [KP88] and as the Observer pattern in [GHJV95].

This lead to the introduction of the change/update mechanism which is available in Smalltalk as the operations *changed:* and *update:* of class *Object*. *Changed:* is called by the subordinate object to inform all dependent objects about a change of its state. Dependents register and unregister using *addDependent:* and *removeDependent:*. On the superordinate object's side, *update:* is called. The subordinate object is called subject and the superordinate objects are called observers.

IBM Smalltalk offers only the most basic change/update protocol by passing a single object to its observers. This single object is usually a constant denoting a certain event, that is a state transition of the subject. An observer, however, can potentially observe more than a single subject. How can it make a distinction between the same event received from a number of different subjects? This is an important question that shows up, for example, if a tool tries to reuse several subtools of the same class. Thus, an observer must be enabled to distinguish between the subjects that sent an event by calling *changed:*.

Therefore, the class *TmObject* was enhanced with a new *update:from:* operation which passes in an additional parameter. *From:* names the subject that originated the event.

```
update: aSymbol from: aSubject
  "do nothing by default"
```

The operations *changed:* and *update:* have been specialized in *TmObject* to support this enhancement. *Changed:* builds a simple pair using class *TmPair* to put together the event with the subject, and *update:* deconstructs the pair and calls *update:from:*. This way, the already existing change/update mechanism was reused. It should be said that this enhancement is available in other Smalltalk implementations like VisualWorks right from the beginning.

Based on our experience with this change/update protocol we suggest a certain policy of using it which is not mentioned in the otherwise excellent discussion in [GHJV95]. The *update:from:* operation should be designed all over the application in a similar fashion. Its only purpose is to delegate the received event to an operation responsible for reacting to this event. None of this code should be directly incorporated into the *update:from:* operation. Next to delegating, the *update:from:* operation may also gather the parameters for the event specific operation it is going to call so that this hasn't to be done by the operation itself.

The following code was taken from class *SimpleBrowser* which has to react to the selection of a new item in the *Lister* subtool by switching the currently displayed item.

```
update: aSymbol from: anObject
  aSymbol == #cEvItemSelected ifTrue: [
    ^self switchItem: ( self listerFp selectedItem ) ].
  super update: aSymbol from: anObject.
```

It shows the standard conventions of naming events (*cEvItemSelected*) and dispatching the event to the proper operation (*switchItem:* with appropriate arguments). The naming of events should state what has happened and not what the subject assumes the observers have to do, that is *cEvItemSelected* instead of *cEvItemSelect* or *cEvSwitchItem*. It is left to the observer how to interpret the event.

At the end of the *update:from:* operation one should not forget to call the superclass's *update:from:* operation (if needed) so that no events to be handled by the superclasses are forgotten. See Chain of Responsibility in [GHJV95] for applications and further implications.

3.3 Class Retrieval and Late Creation

Systems designed according to the Tools and Material Metaphor usually require a major configuration management overhead, since each customer might require a slightly different version of the system. For example, each customer might want to choose a different subset of the overall set of possible tools to be delivered. Instead of hardcoding a desktop class which brings all tools to the users eyes it is preferable to specify on a meta level which classes are to be incorporated into the final image. The system then has to determine at runtime which tools can be accessed. On the technical level, this boils down to retrieving all classes from a certain class tree, for example the tool class tree, which are available in the system. This process was coined "Class Retrieval."

As another example suppose that the user double clicked on a material icon on the desktop. This is to be interpreted as the user's wish to startup a default tool for the material. How can the system determine what the default tool is? The material itself should not know since materials never know about specific tools designed for them. However, a tool class can state that it has been designed for a certain material and that it would like to be the default tool for this material. On the technical level the environment handling the double click on the material has to determine the tool class and then create an instance of it. This has been coined "Late Creation" (resembling late binding), because the class of a new object is determined not at compile time but only at runtime.

Both class retrieval and late creation serve a common purpose. They let clients solve class retrieval and instance creation tasks on a general level without directly referencing classes by name. Thereby, the clients get independent from concrete classes which makes the system easier to change. Essentially, the class trees can be encapsulated and the mentioned tasks can be solved on a general level using only the abstract classes of the class tree. These issues and the implications were explored and discussed in detail in [Rie95b, Rie95c]. The following discussion and the implementation in the framework are based on the concepts elaborated in the referenced works.

The tasks of class retrieval and late creation have been generalized and implemented as class methods of class *TmObject* so that users of the framework may use them but don't have to bother with the details of its implementation.

The two examples mentioned above require only little coding effort on the client side. The environment class uses the following piece of code to retrieve all classes from the tool class tree:

```
initToolClassCol
   toolClassCol := IndexableWrapper new:
      ( OrderedCollection new ).
   Tool getClassCol: toolClassCol byClause:
      ( IsAbstractClause new: false )
```

The first statement creates a collection in which the tool classes are to be put. The second statement asks class *Tool* by calling *getClassCol:byClause:* to put all concrete subclasses into the collection. The key part of this operation call is the *byClause:* parameter which gives a specification for those classes that are to be put into the collection. The specification here is an instance of class *IsAbstractClause* which denotes whether a class has to be abstract or concrete.

The late creation example is implemented in a similar fashion in class *Environment*. Here, the class method of *createByIdClause:* serves to replace the ubiquitous factory methods [GHJV95] that are present in so many class interfaces. It is assumed that a single default tool class exists for every concrete material class, so that the following mapping is unambiguous:

```
createToolFor: aMaterial
   | aClause aTool |
   aClause := MaterialIdClause new:
      ( aMaterial class symbol ).
   aTool := Tool createByIdClause: aClause.
   aTool ~~ nil ifTrue:
      [ self addTool: aTool ].
```

In the third line an instance of *MaterialIdClause* is created which holds the symbol of the material class the new tool class has to fit. In the fourth line, the tool is created using *Tool>>createByIdClause:*.

A similar example can be taken from the implementation of class *CompositeIp* which has to create the interaction part for the functional part of a new subtool:

```
createSubIpFor: anFPart
   | aClause anIPartClass |
   aClause := FPartIdClause new: ( anFPart class symbol ).
   anIPartClass := IPart getClassByIdClause: aClause.
   self addSubIp: ( anIPartClass new: anFPart ).
```

This time, however, the late creation process is broken up into two steps. First, in line 3, the functional part class matching the previously created clause is retrieved, and then, in line 4, the actual instance of the interaction part class is created parameterizing it with its functional part.

In previous versions of our frameworks [Rie93a, Rie93b] these problems were originally solved using factory methods. Each user of the framework was required to specialize the factory methods for the single purpose of creating a fitting object.

TmObject offers currently five operations dealing with these issues, namely:

createByIdClause:	Creates an object of a class specified by the *IdClause* instance that has been passed in.
getClassByIdClause:	Looks up the class which has been denoted by the given *IdClause* instance.
getClassCol: byClause:	Puts all classes which match the given specification (second argument) into the collection (first argument).
getConcreteClassCol:	Puts all concrete classes in the given class tree into the collection which has been passed in as an argument.
getFullClassCol:	Puts every class from the class tree into the collection.

These operations are available as class methods of class *TmObject*. If called on a subclass of *TmObject*, all specification matching processes are restricted to this subclass and its subclasses in turn. Calling *getConcreteClassCol:* on *Tool* reveals only concrete classes derived from *Tool*.

3.4 Class specifications

We have just touted the general idea of retrieving classes and creating objects using its superclasses only. Systems become more easier to change since we don't have to directly name classes anymore. Naming classes has always to be done by providing additional code which has to be changed if class names or classes change.

To make up for the information loss of class names, specifications were introduced. The specifications we use avoid the overhead that usually comes with such approaches by providing an easy to use object-oriented scheme of expressing class properties. All class retrieval and late creation operations are based on these specifications. The class methods of *TmObject* expect a specification to be passed in. It evaluates based on these specification which classes the client is interested in.

Each simple property of a class is represented by an instance of a subclass of class *Clause*. An instance of clause class *IsAbstractClause* indicates whether a class is abstract or not. It is implemented by a simple flag. Several more clause classes exist as depicted in the overview given by figure 3-2.

It has to be said that a simple property of a class is only a very restricted kind of specification. However, the representation of simple properties as objects might be the very foundation of more enhanced specifications than just the simple standalone properties. It can easily be imagined to use propositional calculus or even predicate calculus based on clauses [Rie95b, Rie95c]. However, these advanced kinds of specifications are not available in the current version of our framework. In the following discussion we therefore assume a specification to consist of a single clause.

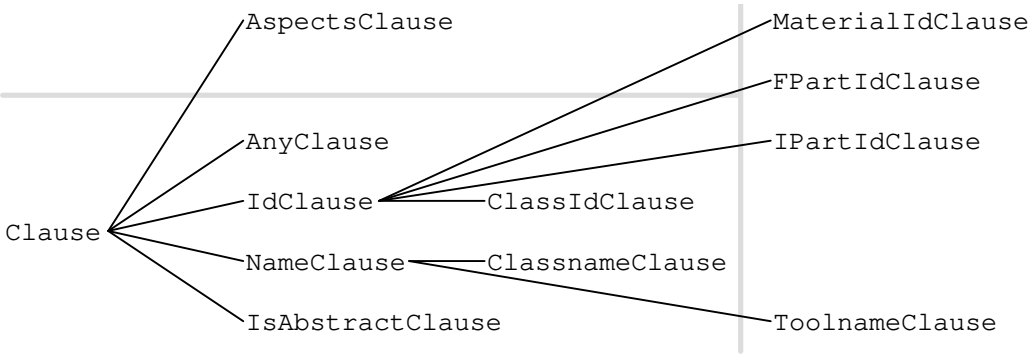

Figure 3-2: Overview of all clause classes in the framework. The lower left part are classes from the basic framework cluster, the upper left clause class is from the material cluster and the clause classes on the right are from the tool cluster.

We will now discuss the basic interface of class *Clause*, the properties represented by each clause class and how a clause instance retrieves its information.

What is relevant about a clause instance? It has to be comparable with other clause instances of the same class, so that the matching processes described in the next section can be carried out easily. Each matching process is carried out between a clause from a specification and a list of clauses maintained for each class. The clauses maintained for each class represent the properties of that class.

Furthermore, clause instances have to be initialized in one of two different ways. First, they have to receive the properties they represent from the client. The client knows explicitly which classes it is interested in and can therefore directly express this using clauses. Second, clause instances are to be created for a certain class. Thus, they have to retrieve the data for the properties they stand for directly from the class. In the current implementation they do so by using a so called property method which offers the data needed by the clause.

Finally, it has to be taken into account that a clause class may not be applied to every class in the system but only to a subset of classes. This subset is indicated by a root class. The clause class works for this root class and all its subclasses. While *IsAbstractClause* works for any class from *TmObject* on, *MaterialIdClause* works only from the root class *Tool* on.

Summarizing, three main methods of class *Clause* have to be implemented:

adaptToClass:	Adapts clause instance to a specific class. The clause instance analyzes the passed in class and sets its attributes according to the class's properties.
matches:	Compares two clause instances. This is by default a type check followed by an equality check.
root	Returns the root class which the clause class has been designed for. This is both an instance and a class method. It should be specialized as a class method.

The *adaptToClass:* and *root* methods, are specific to each clause class and thus have to be reimplemented.

The following is a list of the clause classes, their meaning and implementation.

AnyClause	An instance of *AnyClause* matches with any other *AnyClause* instance and can be used to retrieve all classes from a class tree. It works from class *TmObject* on. It can be adapted to any class and does nothing by default.
NameClause	A name clause holds a string which has to be matched by a class as its name. *NameClause* is abstract and intended to be reused by subclasses which specify which name of a class is relevant.
ClassnameClause	A class name clause holds the class name as a string and is the most obvious example of a specialization of *NameClause*. It works from *TmObject* on and extracts the class name using the standard class methods of a class.
ToolnameClause	A tool name clause holds a string indicating the tool name of a specific tool class. Its root class is *Tool* which offers a class method called *toolName* tool name clauses rely on.
IsAbstractClause	An *IsAbstractClause* instance denotes whether the class is abstract or concrete. It's root class is *TmObject* and it relies on a class method called *isAbstract* to determine whether the class it stands for is abstract or concrete.
AspectClause	An aspect clause instance groups a number of aspects together to identify a material that supports these aspects. It holds a list of aspects either requested by the client or offered by the class the aspect clause stands for. The root class for *AspectClause* is *Material*. Aspect clauses use the class method *aspectClasses* of *Material*.

So far, the *IdClause* class tree has been left out. *IdClause* is an abstract class that serves as a superclass to several kinds of id clause classes. The key idea behind the abstraction *IdClause* is to have clause instances which denote a single class. They do so by giving an unambiguous identifier through the method id. *IdClause* enhances *Clause* by this single attribute and the corresponding access method.

Id clauses are needed if an unambiguous mapping is required, for example from material to default tool class, or from functional part to default interaction part, or from interaction part to default user interface part (see section 5-2 and 5-3). On the implementation side, id clauses can be used for a fast table lookup scheme of denoted classes, so that class retrieval and late creation can be implemented in constant time. Thus, they consume time in the order of factory methods which they serve to replace.

FPartIdClause — *FPartIdClause* has the root class *IPart* and denotes the *FPart* the interaction part has been chosen to be the default for. It is built using the class method *fPartClass* of *IPart* if instantiated for the class. If instantiated by a client, it directly receives the functional part the interaction part of which it is used to look up.

IPartIdClause — *IPartIdClause* does the same for user interface parts what *FPartIdClause* does for interaction parts. Root class is *UPart*. An interaction part id clause uses the class method *iPartClass* of class *UPart*.

MaterialIdClause — A material id clause identifies a material which a default tool is to be looked up for. It's root class is *Tool* and it is built using the class method *materialClass* of class *Tool*.

Each of the clause classes forced us to introduce a single operation for the clauses' root class. *IsAbstractClause* relies on *isAbstract*, *FPartIdClause* relies on *fPartClass*, *MaterialIdClause* relies on *materialClass* and so on. These operations are a simple way of specifying class properties, so that a clause can retrieve them and make them explicit as an object (instead of the operation). Subclassing a class requires specializing these operations according to the new class's properties, if we want it to participate in the class retrieval and late creation facilities.

At first glance it seems that we have given up the advantage of code reduction we gained by omitting factory methods. However, factory methods serve only a single purpose, while the property methods may be reused for all kinds of specifications. Furthermore, we don't have to specialize every property method but rely on their default implementation. Given an *isAbstract* implementation of class *TmObject* which returns true, we only have to specialize this operation for those classes which are concrete.

3.5 The class manager

Section 3.3 discussed the functionality available to users of the framework. Section 3.4 discussed clauses and specifications, the clause classes available and how they are implemented. This section in turn presents the specification matching and class lookup functionality behind the scenes.

Summarizing the previous sections, two major questions remain unsolved: How and where are class properties maintained and how are they used to match and lookup classes? This section addresses both questions.

The implementation of the tasks to be done has been factored out into the class *TmClassManager* of which a single instance is created during runtime. This instance holds tables for each regular class and its properties as well as tables for all kinds of id clause classes. It provides the

basic functionality to match a clause with a specific class and to lookup a class based on a given id clause instance.

At present, the first object which is created during system startup, the *Environment* instance, also creates the single instance of *TmClassManager* and sets it to class *TmObject*. All class methods of *TmObject* are implemented in terms of the instance methods of the *TmClassManager* instance they receive. We have chosen this approach of creating an instance anew each time instead of having the class maintain the tables persistently because it is not clear to us yet how system startup behavior will be defined in the target system. By calculating the tables from scratch we also avoid update problems that come with system changes. The startup time didn't yet lead to perceivable delays.

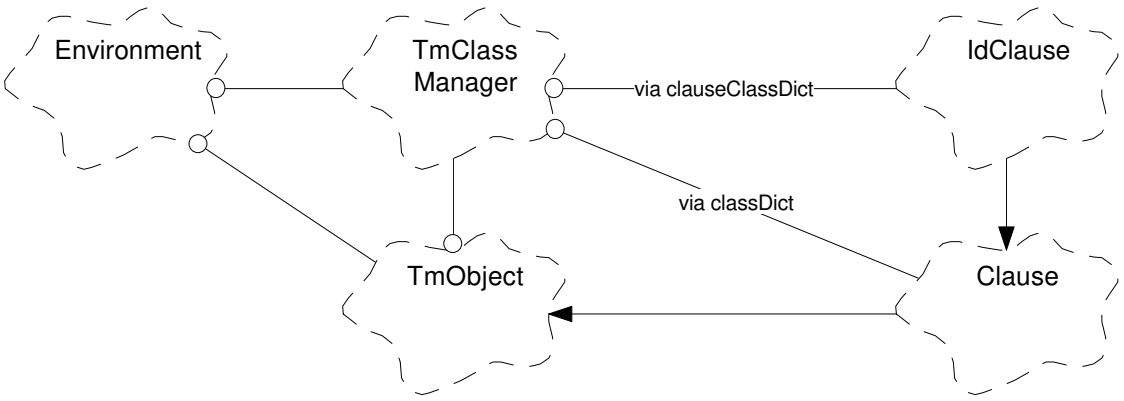

Figure 3-3: Design and implementation structure of `TmClassManager`, its instance creation and the tables maintained by it.

The class methods of *TmObject*, mentioned in section 3.3 are implemented in terms of two simple operations of *TmClassManager*, namely *lookupByIdClause:* and *matchesClause:for:*. These two instance methods in turn rely on two internal dictionaries, *classDict* and *clauseClassDict*.

MatchesClause:for: receives a clause instance as its first and a class as its second parameter. It returns true, if the class offers the property specified in the clause instance.

```
matchesClause: aClause for: aClass
   | aClauseDict bClause |
   aClauseDict := classDict at: ( aClass symbol ).
   ( aClauseDict includesKey: ( aClause class symbol ) ) ifFalse:
      [ ^false ].
   bClause := aClauseDict at: ( aClause class symbol ).
   ^ aClause matches: bClause
```

As shown in the above implementation, *matchesClause:for:* first retrieves a dictionary of clause instances all instantiated for a specific class by indexing the *classDict* with exactly this class. It then looks up the class specific clause instance using the class of the external clause as an index. Finally the retrieved clause and the external clause are matched.

The second method, *lookupByIdClause:*, returns the class which is denoted by the received id clause. It is implemented using a dictionary indexed by the id clause class, which leads to another dictionary that is indexed by the id retrieved from the id clause itself.

```
lookupByIdClause: anIdClause
   | aClassDict |
   aClassDict := clauseClassDict at:
      ( anIdClause class symbol ).
   ( aClassDict at: ( anIdClause id ) ifAbsent:
      [ ^nil ].
   ^aClassDict at: ( anIdClause id )
```

The initialization procedures of the dictionaries are more complicated and not explained in detail here. They are implemented in a straightforward way by using *getFullClassCol:* and *getConcreteClassCol:* which are implemented independently of the mechanism explained here. Essentially, these are simple traversal, gather and build methods.

4 Materials and Aspects

This chapter discusses the classes *Material*, *Aspect* and their subclasses. We discuss questions of multiple inheritance and how to model aspect classes in Smalltalk. We thus prepare the way to the Aspect Browser presented in chapter 6.

4.1 Overview of the material cluster

MaterialCluster, the material cluster from now on, comprises the basic material and aspect classes of the overall framework. The class hierarchy in figure 4-1 shows the classes from the cluster. It introduces a new clause class, *AspectClause*, which groups a number of aspect classes together so that material classes can be matched with it. The main part is the material class tree starting with class *Material*. It comprises the *Aspect* class tree and some general material classes (*Folder*, *Document*, etc.).

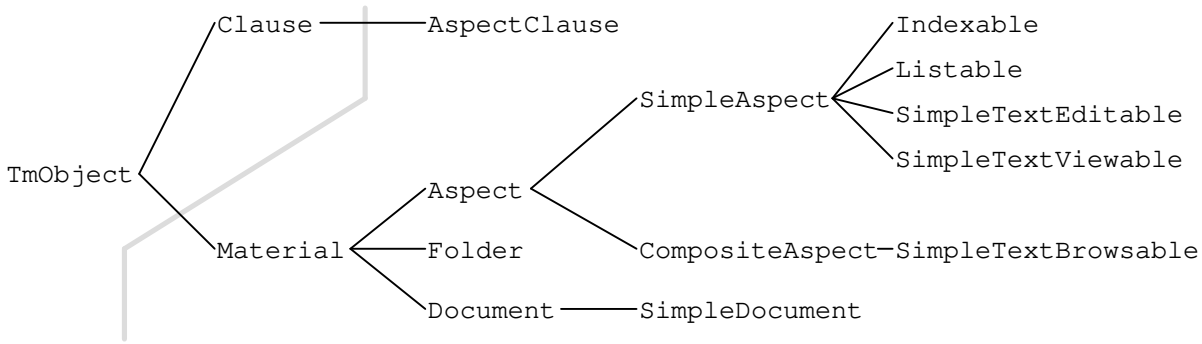

Figure 4-1: Main part of the material cluster's class hierarchy.

Materials, aspect classes and their subclasses are maintained in the same cluster. We think that this is justified, since the aspect classes to be discussed are very general. It is important to point out, however, that tool, aspect and material classes form a trinity which can only understood well enough when considered as a whole. More specialized tools, aspects and materials will probably be put together into a cluster of their own.

We will take a look now on the class *Material*, the simple material subclasses offered, the *Aspect* class tree and the class *AspectClause*. Before doing so, however, we have to reconsider the question of implementing aspects in Smalltalk.

4.2 Aspects in general and in Smalltalk in particular

Generally speaking, aspects represent ways of handling materials from a tool's point of view. They are called aspects, because they are confined to a specific task the required functionality

of which they make explicit. They are formalized as aspect classes which express their functionality as a class interface.

A material has several aspects, since there are usually several ways of handling it. On a software design level this means that we have to provide access to a material via each aspect class that represents an aspect the material offers.

There are at least two well-known techniques for handling such a situation: multiple inheritance and wrapper technology. In the case of multiple inheritance, we model a material as the subclass of all those aspect classes which represent the aspects the material offers. Thus, the material offers all the methods defined by the aspect classes directly as part of its interface. This solution is chosen, if the aspect is considered to be a permanent way of handling a material that cannot be attached and removed over time. Such an aspect is usually a very general way of working with a material like listing or indexing.

Wrapper technology is a different approach. A wrapper object adapts a given interface, the aspect class, to the interface of a specific material class. The wrapper works as a representative hiding the material from the tool. However, it acts on behalf of it and is implemented in terms of it. Modeled this way, interfaces can be attached to and removed over time from a material. This approach is usually chosen if an aspect is considered to be application specific and using multiple inheritance would interfere with other applications.

The obvious problem is that we can't use multiple inheritance to design a material's interface since Smalltalk is a single inheritance system. We also prefer not to temper with the metalevel architecture of IBM Smalltalk. Furthermore, it is not sensible to rely only on wrapper technology since it poses several problems: The number of wrapper classes tends to explode, the state of a material is spread over several (wrapper) classes and no distinction between regular clients and aspect emulating clients is possible anymore [OH92, HO93].

We therefore chose to emulate multiple inheritance in an aspect specific way: We write aspect classes as standard class interfaces and use a tool to copy the methods from an aspect class into the material class interface. Essentially, aspect classes specify protocols which are then attached to material classes.

This tool, the Aspect Browser described in chapter 6, is used to deal with maintenance and evolution. Modeling a material's aspects this way works, because Smalltalk is dynamically typed. Thus, if a material is passed to a tool it doesn't have to offer a specific aspect class as a superclass but only the methods defined by that aspect class.

These questions are elaborated in detail in chapter 6 which presents the Aspect Browser. Here we confine the discussion to the class hierarchy, its functionality and how a user of the framework makes use of them.

4.3 The material classes

Materials are the intermediate and final products of work processes. They are the outcome of work performed by using tools. Users perceive materials and manipulate them only indirectly through the mediating tools. The possible ways of handling a material from a tool's point of view are captured as aspects which in turn are formalized as aspect classes (see previous and next section).

The notion of material is expressed formally by one of the central abstractions of the framework, the class *Material*. Almost any class is a subclass of it. This applies equally to abstractions that are easily recognized as materials, like folders and documents, and abstractions that are not so obviously recognized as materials like aspects and tools. They are discussed in their

respective section. In this section we focus on the class *Material* and its subclasses *Folder*, *Document* and *SimpleDocument*.

Material has no instance methods but a number of class methods that let users as well as the Aspect Browser access and mutate the aspect classes offered by a certain material. The basic idea is that each material holds a collection of aspect classes. Since aspect classes are classes and thus objects themselves, this can be done easily. The aspect classes in the collection are exactly those aspect classes which correspond with the aspects of the material. The aspect classes are implicitly present in the material's interface since it comprises all the methods offered by these aspect classes.

The following list shortly presents only the most obvious *Material* class methods, since they are discussed in more detail in chapter 6.

addAspectClass: Adds an aspect class to the material's internal collection of aspect classes. No implicit update of the class interface is done.

removeAspectClass: Removes the given aspect from the internal aspect class collection. Again, no implicit updates are connected with this method.

hasAspectClass: Returns true if the material class offers the given aspect class both in its collection and its interface.

aspectClasses: Returns the collection of aspect classes offered by the material class.

Next to the general superclass *Material*, the subclasses *Folder*, *Document* and *SimpleDocument* are available. However, they have been used for demonstration purposes and contain only the most obvious functionality. In an extension of the framework for a specific application domain, for example in a customer support system, they have to be extended or replaced with similar but more enhanced classes.

The class *Folder* is a material class that represents the basic folder: it has a table of contents and contains some documents. From this the aspects of the folder become obvious: It has to be indexable and listable. It has to be indexable because the documents in the folder are assumed be maintained according to a given order, that is a table of contents. As an alternative to being indexable it might have a separate table of contents which may be requested by clients. It has to be listable because we assume that a folder can be contained in another folder. The folder class is implemented in terms of an ordered collection. It offers the aspect classes *Indexable* and *Listable* respectively which are discussed in the next section.

A folder may contain documents, so there is a quite general document class called *Document* and a subclass called *SimpleDocument*. The general abstraction of *Document* is only expected to be a listable kind of something, so currently no more assumptions about a document are made than just being listable. This is expressed by the aspect class *Listable*.

Class *SimpleDocument* represents a concrete though basic document which can next to being listable edited as a simple text. Thus it offers the aspect class *SimpleTextEditable*.

4.4 The aspect classes

An aspect class expresses the functionality needed by a tool to work on a material. This functionality is derived from an aspect of the material which captures domain and tool specific ways of handling it. An aspect is confined to a single task.

Thus, an aspect class is expressed as a regular class. An aspect class has no implementation state of its own, however, it declares through its interface an abstract state which clients can rely on. It would be desirable to have specification like features that make dependencies, traces

and the abstract state space explicit. However, Smalltalk doesn't offer these facilities and thus we didn't take further steps into that direction.

As shown in figure 4-1, we introduced a general superclass *Aspect* which in turn is a subclass of *Material*. From a conceptual point of view, any object is a material as long as we can sensibly think of a tool to work on it. For example, an aspect is both a way of handling a material and a material for the Aspect Browser. Even a tool is a material for some kind of meta tool that can be used to manipulate or fine tune it.

We therefore chose to make *Aspect* a subclass of *Material*. So far, no general functionality is tied to this inheritance relationship. However, we can easily think of providing class *Aspect* with the aspect *Listable*, since any aspect class as a material of the Aspect Browser appears in some lists. The only reason why this hasn't been done yet is because the Aspect Browser was developed in parallel to the framework and is not implemented based on it. The next version of the Aspect Browser will be based on the framework.

Since an aspect is also a material, we introduced aspects as classes in a classification hierarchy as shown in figure 4-2. An aspect class can either be a simple aspect class or a complex aspect class (made explicit by the superclasses *SimpleAspect* or *CompositeAspect*). A complex aspect class is based on more than one aspect class and thus holds a collection of "super aspect classes" which it would inherit from in case of multiple inheritance. A simple aspect class is an aspect class which is independent of other aspect classes. *Listable* and *Indexable* are simple aspect classes, while *SimpleTextBrowsable* is a complex aspect class composed out of the simple aspect classes *Listable* and *SimpleTextEditable*.

The classes *Aspect*, *CompositeAspect* and the simple aspect classes are an implementation of the Composite pattern [GHJV95]. Essentially, a single rooted tree of aspect classes is realized which, if turned upside down, is a representation of the desired multiple inheritance structure between aspect classes.

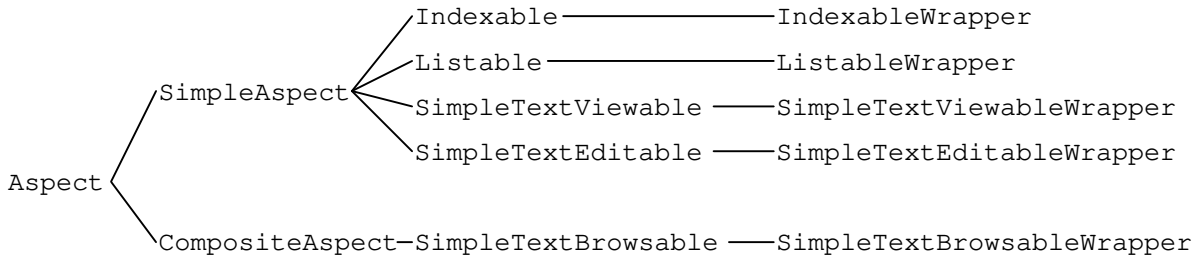

Figure 4-2: The aspect classes from the material cluster. Not shown is the superclass of `Aspect`, class `Material`. New are the wrapper classes on the right.

The chapter about the Aspect Browsers explains the structure of simple and complex aspect classes in more detail. Here we focus on the contents of the aspect classes. In the following, we discuss the aspect classes which are currently provided by the framework. The overview of figure 4-2 shows them and some wrapper classes.

Listable The aspect class Listable offers the functionality required from a material in order to be represented by a string in any kind of list like structure. The operations offered are *listName*, = and <=. *ListName* returns a material specific string that can be used as a name for the material. By default it is implemented using *printString*. The comparison operators realizes a *Listable* specific ordering relation of the material so that it can be presented using a default ordering scheme.

Indexable — *Indexable* lets tools access a material using an index. It lets them iterate over the material. Thus, it represents the interface to an ordered collection and indeed, in its current state just replicates this protocol. It should be noted that prior experience shows that collection classes are problematic to use directly as a material. They should only be used to implement application specific materials or collection classes which have a suitable interface for the tasks at hand.

SimpleTextEditable — *SimpleTextEditable* presents the interface to a plain text structure consisting of simple paragraphs without further formatting functionality. The access and mutation methods work on the abstract model of an indexable collection of strings, each on representing a paragraph. The basic methods are *text* and *text:* to get and set the full text.

SimpleTextViewable — *SimpleTextViewable* declares a simple string representation of the material. Thus it offers only the operation text to retrieve a string.

SimpleTextBrowsable — The complex aspect class *SimpleTextBrowsable* is the combination of *Listable* and *SimpleTextEditable*. It offers no additional methods.

The presented aspect classes are obviously the most basic ones one can think of. The framework has still a long way to go here, since we haven't yet incorporated domain specific classes. However, these domain specific classes are the meat that make a framework usable for a domain like customer support systems in a banking context. Once we will focus on these domain specific classes, the needed tools, aspect classes and materials will come into existence and evolve.

One final word with regard to the wrapper classes depicted in figure 4-2. They are subclasses of their respective aspect classes and are parameterized with blocks so that a *ListableWrapper* or a *SimpleTextEditableWrapper* instance can work on any material. The adaptation functionality is contained within the blocks.

These wrapper classes are intended for use in situations where it is not sensible to manipulate the original class. Thus, it is wrapped for a specific aspect. As a consequence, the original class can't offer the blocks itself to provide a wrapper with the adaptation blocks. They have to be maintained by clients or in a new class. This either increases the number of classes or spreads blocks in client code. Both situations should be avoided. Thus, these wrapper classes exist but aren't used in the current framework. So we don't discuss them any further.

4.5 The aspect clause class

Finally, the aspect and material cluster offers the class *AspectClause*. This clause class holds a list of aspect classes. It is used to either represent the aspect classes a material offers or to retrieve a material which offers the aspects indicated by the clause.

AspectClause — An aspect clause instance is created for each material and holds a collection of its aspect classes. Its root class is *Material*. It is built using a material's class method *aspectClasses*. Clients supply a new clause instance with a collection of aspect classes a material to be looked up has to conform to.

5 Tools in an Environment

This chapter presents the tool and environment cluster. The tool cluster offers several classes for building tools. These classes are by far the most mature classes of the framework since tools are built from a regular structure that can be captured by well-known patterns. The environment cluster contains a single class of which only a single instance is created at runtime: the environment. It is the first object created during startup time.

5.1 Overview

The tool and environment cluster (*ToolCluster* and *EnvCluster*) comprise the classes depicted in figure 5-1. The environment cluster offers a single class, *Environment*, which is used to startup the overall system in the first place. The tool cluster comprises several subtrees based each one dedicated to the implementation of a specific part of an overall tool. The subclass trees are the *Tool*, *FPart*, *IPart* and *UPart* class tree.

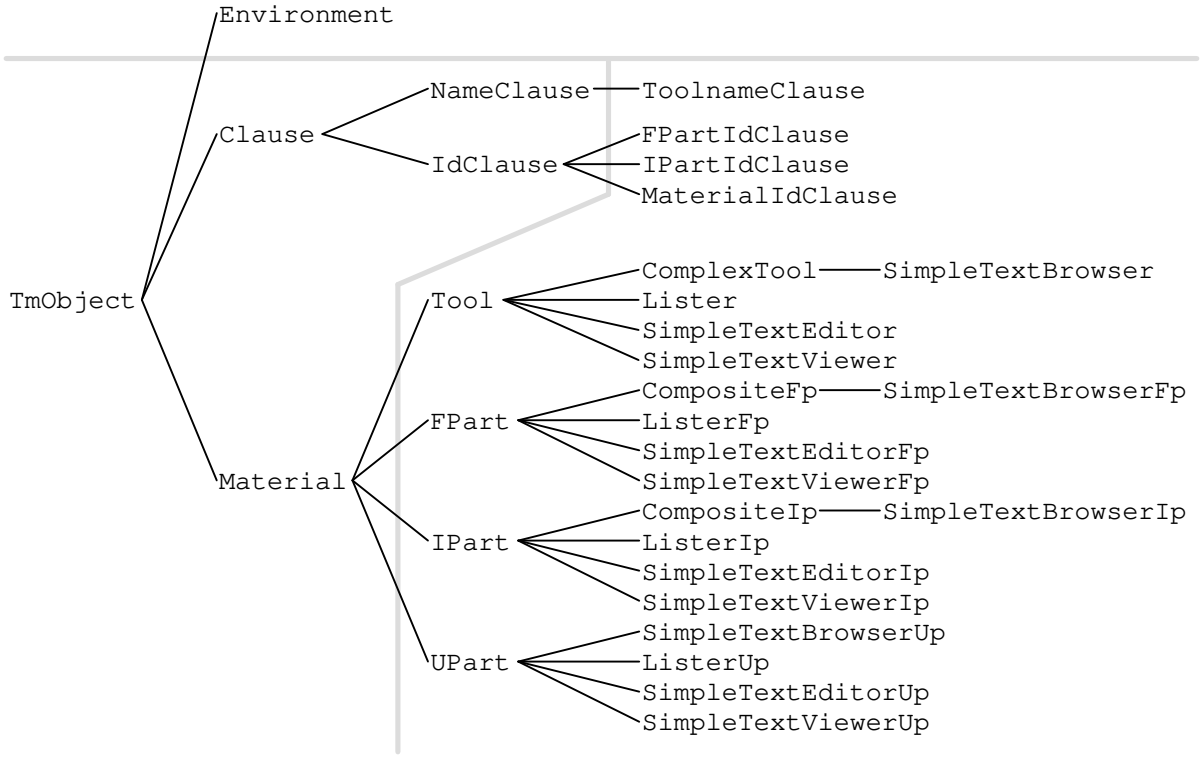

Figure 5-1: The environment cluster is shown at the top and the tool cluster is shown on the right. On the left are the superclasses from the other clusters.

We will first discuss the overall structure of a tool, the idea of tool composition from components, the structure of a single component and the basic tools provided by the framework. After this we will introduce the environment and show how it handles tools and materials.

5.2 Tools – rationale and structure

A tool is used to perform work on materials. It presents materials in a graphical user interface and lets users manipulate them. A tool is connected with a material via aspect classes which confine the possible actions a user might perform on a material. Each tool has a handling and state of its own which make up the character of the tool. Well designed tools focus on a single task and don't try to be overly general.

A tool exhibits a regular structure. It is built from tool components, which are organized in a single rooted component tree. Each component consists of three different objects, called the functional part, interaction part and user interface part of the component. A tool may be simple, consisting of a single tool component, or complex, consisting of several tool components.

A tool component is designed to serve a certain well-defined task, for example to present a listable material visually and to let users select from the visual presentation. A tool component may either be a simple or composite tool component. A simple tool component needs no other tool components to implement its task, that is it works stand-alone.

A composite tool component relies on further tool components to implement its task. This leads to the single rooted tool component tree mentioned above. The browser tool consists of a supervising browser tool component with two subordinate tool components, a lister tool component and a simple text editor tool component.

Each part of a tool component serves a different purpose. The functional part implements the functionality offered by the tool component, the interaction part defines the handling of the tool and the user interface part creates and manages the widgets that are used to realize the user interface for the interaction part.

The closure of a tool is realized by a single object called the tool object. This tool object is the first object of the overall tool to be created. It comprises all relevant data about the tool and creates the first tool component in turn. This tool component becomes the root tool component in case of a complex tool. It creates subordinate tool components according to its needs.

Figures 5-2 and 5-3 show a simple and a complex tool respectively.

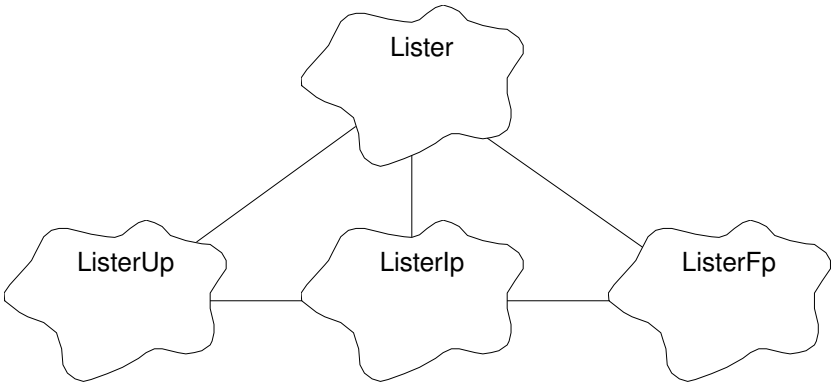

Figure 5-2: The object diagram of the lister tool. The tool object creates the functional part, the interaction part and the user interface part in this order. The three parts form the one (and only) tool component of the lister tool.

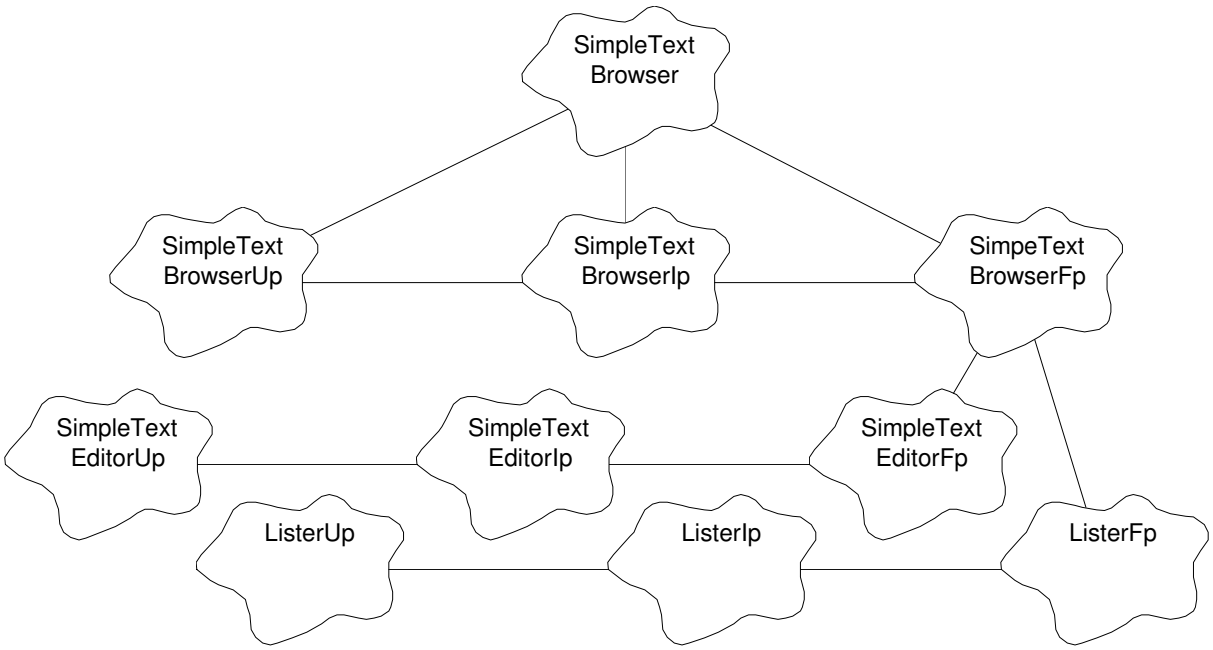

Figure 5-3: The object diagram of a complex tool, here the simple text browser tool. It is built from the lister and simple text editor tool components.

The framework classes *Tool*, *FPart*, *IPart* and *UPart* represent the central abstractions and superclasses of the classes in the figures 5-3 and 5-4. To ease implementation of complex tools, intermediate subclasses of *Tool*, *FPart* and *IPart* were introduced, called *ComplexTool*, *CompositeFp* and *CompositeIp* respectively. This is another incarnation of the Composite pattern [GHJV95].

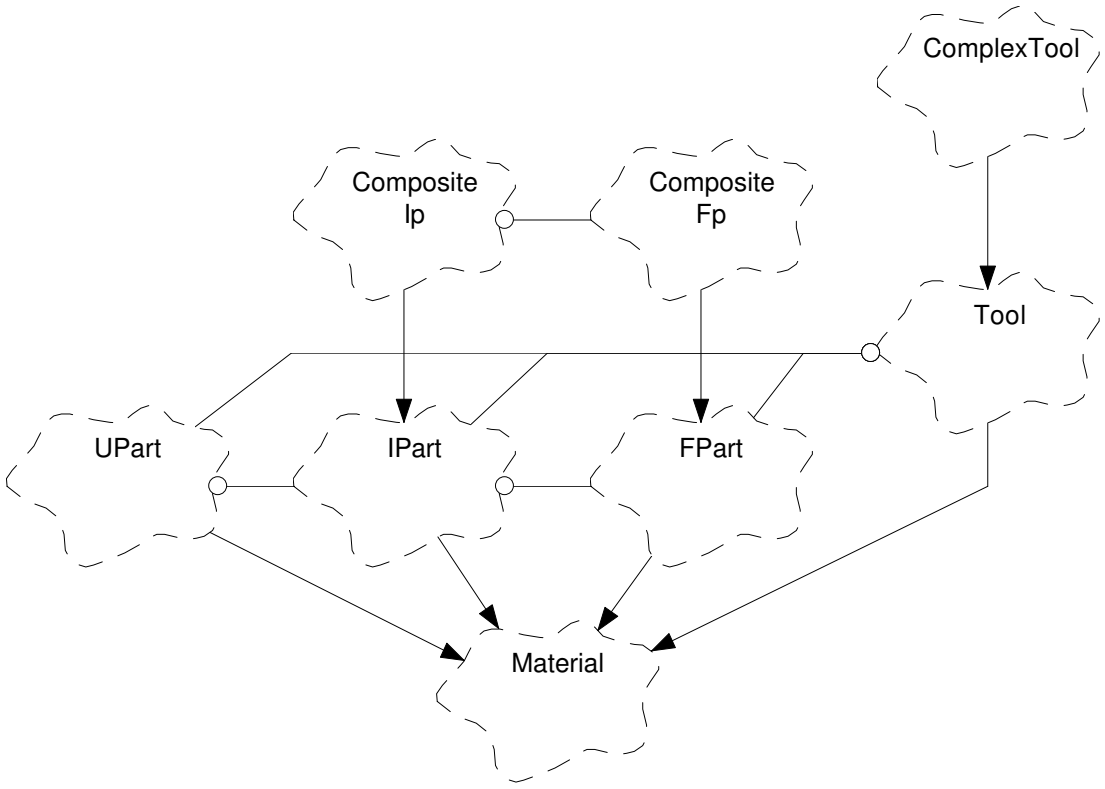

Figure 5-4: The framework classes for tools and tool components. The classes *IPart*, *FPart* and *Tool* have a composite subclass called *CompositeIp*, *CompositeFp* and *ComplexTool*. Leaf classes are not shown.

The next section presents the structure of a single tool component and the composition strategy of a whole tool from its possibly several components. Before doing so, however, we have to discuss class *Tool* itself. Instances of (subclasses of) class *Tool* serve as a closure of the overall tool from the environment's point of view. Tool components or the different parts of a component are hidden behind the tool object. Thus, it serves as a closure to encapsulate the implementation of a given tool.

Class *Tool* offers two class methods and several instance methods. The class methods *materialClass* and *toolName* offer some data about the tool which are used by the clause classes *MaterialIdClause* and *ToolnameClause*. The instance methods are listed in the following:

fPart, fPart:	Get and set the functional part of the single or root tool component of the overall tool.
iPart, iPart:	Get and set the interaction part of the single or root tool component of the overall tool.
uPart, uPart:	Get and set the user interface part of the single or root tool component of the overall tool.
material, material:	Get and set the main material of the overall tool (assuming that there is a single exposed material of this kind).
toolName, toolName:	Get and set the tool name instance wise. If no particular tool name was given, the name defined in the class method *toolName* is returned.
initMainComponent: iPart: fPart:	This method creates instances from the three parameters which are passed in. The parameters have to be concrete subclasses of *UPart*, *IPart* and *FPart*. First the functional part, then the interaction part and finally the user interface part is created. Together they form the first (and possibly only) tool component of the tool.
initUiClosure	This method is directly called from the otherwise void method *initialize*. This operation creates the user interface elements of a shell widget and a mainwindow widget. These two widgets form the root of the widget tree that finally presents the user interface of the whole tool. Essentially, they look like a window. They have been factored out into the tool class so that composition of tool components is not compromised by layout problems of the user interface parts of a component (see section about complex tools).
realize	After initialization the tool is still invisible. This method realizes the shell and mainwindow widget and thus makes it appear on the screen.
shellWidget, shellWidget:	Get and set the shell widget, the first widget that provides the root of the whole widget tree for the user interface of the tool.
mainWidget, mainWidget:	Get and set the main window widget. This widget defines the main window of the user interface of the tool.
windowClose: clientData: callData:	This method is called whenever the user double-clicks on the close button provided by the system menu of the main window. It is used to gracefully close down the tool.

The most basic tool specializes the initialize operation to call *initMainComponent:iPart:fPart:* to create its main tool component. In case of the lister tool, the following lines were written for method *Lister>>initialize:*

```
initialize
   super initialize.
   self initMainComponent: ListerUp iPart: ListerIp fPart: ListerFp.
```

5.3 Tool components

As discussed, a tool is built from one or more tool components. Each tool component focuses on a single well-defined task. It implements the tool state and handling for this task. It is linked to its materials via the corresponding aspect classes. The aspect classes confine the capabilities of the tool component and thus help it to focus on its task. Figure 5-5 shows the participating classes. This might be called the horizontal structure of a tool while tool composition discussed in the next section deals with the vertical structure.

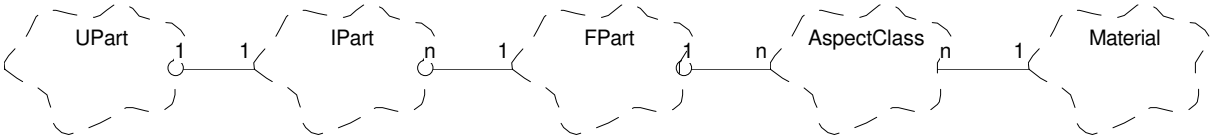

Figure 5-5: The horizontal use relationships within a tool component. The relationship between an aspect class and a material is left unspecified (denoted by an association without further specifying its semantics).

The different components in figure 5-5 have been introduced to serve different purposes.

5.3.1 Functional part of a tool component

The functional part (fpart or fp for short) of a tool component implements the tool component's state and its functionality, that is what you can use the tool component for. It is linked with its materials via adequate aspect classes which confine its possible functionality. An fp just uses its materials, and no observation in the sense of the change/update mechanism exists. If more than one tool is to work on a material, other update mechanisms than the Observer pattern have to be used, for example those discussed in [RZ95, Rie95a].

The framework superclass of all fparts of a tool component is called *FPart*. It has the instance method initialize which should be specialized by subclasses. A corresponding class method new calls this initialization method.

5.3.2 Interaction part of a tool component

The interaction part (ipart of ip for short) of a tool component realizes the handling of a tool component. To do so, it has a state of its own which is distinct from the state of the functional part. It furthermore presents both the material's state and the tool component's state as provided by the component's functional part in a user interface. This it just a "conceptual" user interface since the actual presentation through user interface elements like widgets is left to the user interface part discussed below.

An interaction part sits on top of a functional part. The ip directly makes use of the fp. Furthermore it observes the fp, so that is immediately informed about changes in the material's or fp's state. The application of the Observer pattern [GHJV95] introduces a tighter coupling than a regular use relationship. Therefore, interaction parts and functional parts as the main participants in a tool component are often designed hand in hand.

A functional part may have one or more interaction parts each one dealing with a possible view on the tool. Usually, there is only one interaction part for each functional part. All tool components presented in this report have just one interaction part working on its functional part.

Any interaction part should be a subclass of *IPart*, which is the framework supplied superclass. It offers the following class and instance methods:

fPartClass	This class method returns the functional part class, the current interaction part class has been written for.
new:	This is the usual *new* class method which takes an instantiated fpart and forwards it to *initialize:*.
fPart	This instance method returns the fpart instance the ip is working on.
initialize	This instance method initializes the interaction part.
initialize:	This instance method takes a fpart instance as its parameter, initializes itself via *initialize* and adds itself as a dependent to its fpart.

5.3.3 User interface part of a tool component

A user interface part (upart or up) realizes a specific user interface implementation via window system widgets. It sits on top of an ip and implements a concrete user interface that lets users perceive and handle the tool component as realized by the interaction part. The upart observes its ip and immediately realizes the changes of the interaction part as changes in the concrete user interface. Changes in the user interface through user interaction in turn are immediately forwarded to the interaction part.

An upart's role is more a technical than a conceptual one. It implements a specific user interface for a given platform using the available graphical user interface elements, called widgets in Motif and IBM Smalltalk as well. It can take advantage of peculiar dependencies between certain widgets by having an overview of all widgets needed for its implementation.

An upart decouples the interaction part fully from any system specific user interface behavior. It is our hope that we don't have to code user interface parts by hand in the near future. Since it is separated from any conceptual part dealing with the tool component's state, functionality handling and presentation, the upart is the part to address with user interface builders.

The class *UPart* is the superclass of all user interface parts and offers the following methods:

iPartClass	This class method returns the interaction part class this user interface part class has been written for.
new:with:	This is a creation class method which takes a widget as its first and an ipart instance as its second parameter and forwards them to *initialize:*.
iPart	This instance method returns the ipart the upart instance is working on.
initialize	Instance initialization method.
initialize:with:	This instance method takes a widget as its first and a ipart as its second parameter. The widget is used as the parent widget for all widgets created by this upart. The ipart is the ipart the up is going to present visually. Furthermore, this method introduces the upart as a dependent to the ipart.
createSubUpFor:	This instance method creates a subordinate upart (in the context of tool composition, see below) depending on an ipart passed in as a parameter. Several user interface specific considerations have to be carried out and the new upart has to be chosen on an instance by instance level based on the given ipart.

update:from:	This method dispatches the events the upart receives from its ipart. In the superclass, only the wish for a new subordinate upart is expected and dispatched.
create	This method actually creates the widgets for the user interface. It furthermore sets the *rootWidget*, so that *manage* and *unmanage* work correctly.
manage, unmanage	These two messages *manage* and *unmanage* the whole user interface created by the upart according to the rules set up by Motif.
parentWidget	This method returns the parent widget passed into the initialization method in the first place. Usually, it is a form widget.
rootWidget	Returns the root widget for all widgets of the upart. It is created by the upart itself.

Those who know prior work on implementing the Tools and Materials Metaphor will wonder what has happened to the so called interaction types. Interaction types are wrapper classes for widgets. They are the classes of your average cross platform window system library. In our prior framework interaction parts implemented the user interface itself. They worked with interaction type classes only and thus became portable.

First, interaction types aren't needed anymore, since IBM Smalltalk provides a Motif API for any graphical user interface its Smalltalk runs on anyway. Second, writing a cross platform library is really much work since so many platform specifics and peculiarities have to be taken into account. It was not feasible trying to reimplement it in Smalltalk.

We think that user interface parts should be fully handled on a tool level, that is a graphical user interface builder.

Recent work on interaction forms as higher level abstractions of interaction types has not been addressed.

5.4 Tool composition

Tools, as has been pointed out, may consist of several tool components. These tool components are organized hierarchically in a tree like structure, so that a tool component may have several subordinate tool components. Since each tool component focuses on its own task, any superordinate provides the context of interpretation of the subordinate tool components. This superordinate tool component, called the context tool component for its subcomponents, uses the subordinate components to implement its task. Thereby, an efficient way of breaking up complex tasks into smaller tasks as well as reusing classes and components is achieved.

Structurally speaking, we face the Composite pattern once more. This is made explicit by the classes *ComplexTool*, *CompositeFp* and *CompositeIp* which are the superclasses for a complex tool and the iparts and fparts of a complex tool component. We talk about a complex tool, since it doesn't have a composite tree like structure, but of composite iparts and fparts, since these objects are organized in a composite tree. The class diagram was already shown in figure 5-4.

Every composite ipart or fpart class should be subclass of either *CompositeIp* or *CompositeFp*, because these classes provide the functionality of maintaining a list of subordinate iparts or fparts. Reusing these classes leads to a dual class and object hierarchy of iparts and fparts. User interface parts have not been provided with a *CompositeUp* class since we hope to tackle upart classes fully with tools. Furthermore, uparts usually don't need to know about embedding or

subordinate uparts, since no relationship other than creating and passing widgets should exist here.

5.4.1 Composite functional parts

The logic of creating and maintaining tool component with subtool component relationships resides at the fpart of a tool component. It decides whether to create or delete subordinate components, it makes sense of them by embedding and interpreting them. Therefore, an fpart knows the full interface of all subordinate fparts. In addition, all subordinate fparts are maintained in a collection of sub fparts provided by the intermediate superclass *CompositeFp*. This class offers the following functionality in addition to the one offered by *FPart*:

addSubFp: A new, already created, sub fpart is added to the internal collection of sub fparts. Furthermore, the current fpart is added as a dependent of the subordinate fpart and an event is issued, which denotes that a new fpart has been registered.

removeSubFp: A subordinate fpart is removed from the collection, the current fpart is removed as a dependent from it and an event is issued that denotes that a subordinate fpart has been removed.

subFp For its communication with the ipart, each fpart offers this access method to the last new fpart. It is used when creating or deleting the right ipart for a new fpart.

createSubFps This method is the place were to put the creation code for new fparts that are to be build right from the beginning with the overall tool.

subFpCol Returns a collection containing all the subordinate fparts.

initialize Initializes the composite fpart.

The dynamics of the creation and deletion procedures are discussed below. If an fpart creates or removes a subordinate fpart, it has to call the appropriate methods *addSubFp:* and *removeSubFp:* afterwards.

5.4.2 Composite interaction parts

The interaction parts are created and deleted according to the creation and deletion of their fparts. Subordinate iparts are hold in a collection offered by *CompositeIp*. If an ipart needs access to the subordinate iparts, it may store additional named references. Though this might be sensible in case of complicated user interfaces, most of the data and control flow is handled on the fpart side. Thus, only a small number of iparts should directly use their subordinate iparts.

The instance methods offered by *CompositeIp* are similar to those offered by *CompositeFp*.

addSubIp: A freshly created subordinate ipart is put into the sub ipart collection and an event is issued, which denotes that a new ipart has been registered.

removeSubFp: An old ipart is removed and an event is issued that denotes that a subordinate fpart has been removed.

subIp For the communication with its upart, each ipart offers an access method to the last new ipart. It is used when creating or deleting an ipart.

createSubIpFor: This method creates a new subordinate ipart for the given fpart. It

	uses an *FPartIdClause* instance to do so. The rationale for the technique has been discussed in chapter 3.
update:from:	This method dispatches the incoming events from the composite fpart to its appropriate method. It might catch a *cEvSubFpAdded* which it dispatches to *createSubIpFor:*.
subIpCol:	Returns the internal collection of all subordinate iparts.
initialize	Initializes the composite fpart.

The dynamics of creating and deleting a subtool component can now be discussed, using the *SimpleTextBrowser* as an example of a complex tool. Given that the fpart of the simple text browser has been created, the tool calls *createSubFps* on it:

```
createSubFps
   super createSubFps.
   listerFp := ListerFp new.
   self addSubFp: listerFp.
   editorFp := SimpleTextEditorFp new.
   self addSubFp: editorFp.
```

This method subsequently creates the subordinate fparts and adds to its collection of sub fparts by calling *addSubFp:*. This operation adds the new fpart to an internal collection, registers itself as a dependent of the new fpart and announces an event for the new fpart.

This event is received by the simple text browser ipart and gets dispatched to *createSubIpFor:* which creates the appropriate sub iparts using late creation. Since the new ipart is added to the collection of subordinate iparts, the same creation procedure takes place between the upart and the ipart.

The distinction between interaction parts and functional parts, their clear separation of concerns, their focus on single well defined tasks and easy embedding make them a powerful means for reuse.

5.4.3 Composite user interface parts

Though well thought out classes adhering to the given guidelines should be easy to reuse we haven't yet addressed, however, one of the major obstacles for reusing whole tool components: the reuse of user interface parts.

More precisely, we're less interested in reuse of user interface parts since they should be tool generated, but more in composability. Composability means that we can easily put uparts in a composite tree without being screwed up by technical problems due to widget layout and widget composition.

The discussion of tool components and their user interface part has, however, prepared the scene for successfully tackling this problem. It can be overcome by adhering to three simple guidelines:

1. Never let a user interface part create a window itself, unless you're very sure that your upart will never be reused.
2. Make no expectations about your embedding widget but only assume it to be a general form widget.
3. Use the object bound tool names of a tool component to give names to the widgets, so that a widget from a reused upart can have a different layout than the same widget from another instance of this upart.

The first guideline is successfully realized by the tool class, which factors out the window creation procedure and provides the upart of the root tool component with a mainwindow widget.

The second guideline can only be ensured by carrying out a clean design. No assumptions about an embedding widget context should be made unless one really wants to constrain the possibility of reuse for some good reason.

The third guideline is less important on the programming level, since widgets can be identified as objects very well. However, if tools are used to manipulate user interface parts, very often the only identification of widgets is by name. These names should be different, even if the widget is created by an instance of the same class.

5.5 Available tools

The framework offers some general tools, mainly for demonstration purposes. It comprises the tool classes *Lister*, *SimpleTextEditor*, *SimpleTextViewer* and *SimpleTextBrowser*. The only tool which is sensibly reusable is the lister tool, since the other tools are based on too simple aspect classes representing materials only as strings.

The structure of the lister tool has been depicted in figure 5-2 and its embedding and reuse within a complex tool has been depicted in figure 5-3. We therefore point out to the discussion of tool composition in the previous subsections and go on the present the environment class.

5.6 The environment

The *EnvCluster*, called the environment cluster from now on, offers a single class, that is *Environment*. It is the first class which is instantiated during system startup. Only a single instance will be created which is the root of the object graph that emerges when the system is initialized.

The purpose of this first object is to startup the whole system. This usually means connecting the system to all required external services, building up the desktop and starting and managing tools. Tools are linked to external services only by way of the environment which controls which services are made available to which tools and which service representatives they receive. Here, however, we will only focus on tool startup from a desktop since this is the only functionality that has been implemented so far.

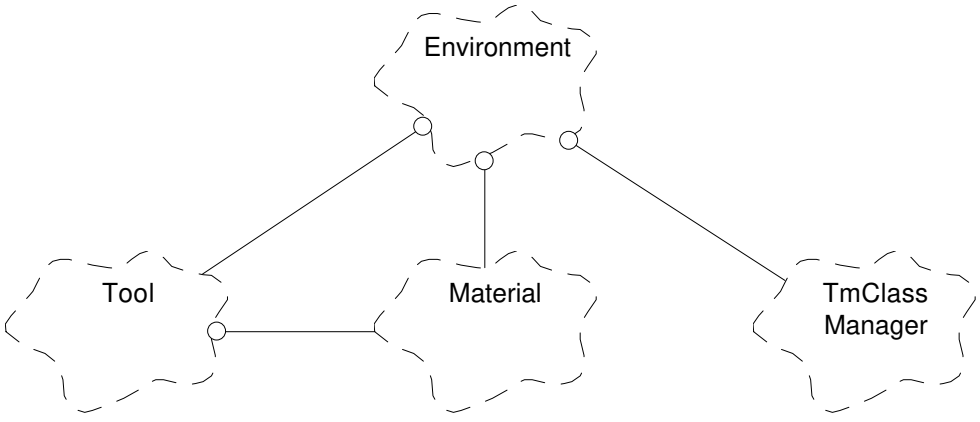

Figure 5-6: The environment cluster and its relationships to other classes.

Currently the environment uses a simple lister tool to show all possible tools within a system to the user. This will be replaced in future versions by a full blown desktop of its own or the inte-

gration with MS-Windows. The environment therefore receives notifications if the user wants to startup a tool. Class *Environment* relies on extensive initializations. This comprises the following attributes of the class.

toolCol	This collection comprises all currently activated tools in the local environment.
toolClassCol	This collections contains all instantiable tools classes. Only for these tools icons are represented on the desktop or listed in the tool menu of a material icon.
materialCol	This collection comprises all currently activated materials that are used by tools. This is a preliminary implementation which be replaced in future versions if adequate material supply services become available.
materialClassCol	This collections contains all material classes of the system that can be instantiated using *classManager*. This is a single instance of class *TmClassManager* which is used to retrieve classes based on specifications (see chapter 3).

The corresponding initialization methods relating to desktop initialization are listed below:

initDesktop	Initializes the desktop.
initialize	Calls subsequently all required initialization methods.
initMaterialClassCol	Collects all concrete material classes using the facilities of *TmObject*.
materialClassCol	Returns the corresponding instance variable.
initMaterialCol	Creates a collection for *materialCol*.
materialCol	Returns the corresponding instance variable.
initToolClassCol	Collects all concrete tool classes using the method *getConcrete-Classes:byClause:* of class *TmObject* starting with class *Tool*.
toolClassCol	Returns the corresponding instance variable.
initToolCol	Creates a collection for *toolCol*.
toolCol	Returns the corresponding instance variable.

For the creation and management of tools the following operations are used.

addTool:	Receives a tool, puts it into the tool collection and realizes it causing it to appear on the desktop.
createTool:	Receives a tool class and instantiates it. The new tool is passed to *addTool:*.
createToolFor:	Receives a material class, determines the default tool class for it and creates a tool for the material using *createTool:*.
removeTool:	Removes a tool from the internal collections.

The environment object receives notifications through its *update:from:* method which indicate whether a new tool is to be created or a tool is to be removed.

Further issues on the environment object can be found in [Rie95a, RZ95].

6 The Aspect Browser

The interface of a material class is defined by one or more aspect classes. The interface that an aspect class defines can itself be composed of one or more aspect classes. In programming languages like C++ that offer multiple inheritance the resulting aspect class tree can be implemented using inheritance [ES90]. In Smalltalk, which only allows single inheritance, we have to find a way to imitate multiple inheritance.

In this chapter we describe two pragmatic solutions to the multiple inheritance problem. The first solution makes use of tool support to accomplish the task. The design and the usage of this tool, the so-called Aspect Browser, is explained. A second solution, which was not used in the framework but could be of interest, is described at the end of the chapter. It presents a simple implementation of method dispatching needed for multiple inheritance. A third solution to this problem, using wrapper classes, has shortly been described in chapter 4.

6.1 Emulating multiple inheritance using tool support

Aspect classes normally don't define any behavior of the material classes, which are derived from them. They only define their interfaces. Therefore, it would be sufficient if we found another way to make sure that the material classes have the interface that is given by the aspect classes which the material has assigned. This lead to a quite pragmatic solution of the multiple inheritance problem.

In IBM Smalltalk, as in other Smalltalk systems, methods can be grouped into categories. Our idea was to use method categories for the protocols of the particular aspects, one category for each aspect. We don't give the material classes the interfaces of the aspect classes through inheritance, but provide tool support to copy the methods of the aspect classes to the material classes.

With this approach, the default behavior of an aspect class can still be implemented as regular methods. The implementation will be copied to the material class. The programmer of the material class can textually overwrite the copied methods, that is he or she replaces the default implementation of the methods with a material specific implementation. The source code of the aspect methods acts only as a template for the implementation of the material classes.

The aspect classes used in a system don't have to build a flat structure. They can also build hierarchies of general and more specific aspect classes. Thus, we need multiple inheritance not only between material classes and possibly several aspect classes, but also between aspect classes and several aspect superclasses.

We applied the Composite pattern [GHJV95] to address this problem. In hierarchies of aspect classes, messages don't have to be dispatched to the right superclass. Since methods of aspect classes are just copied to the material classes, this simply has to be done for all aspect classes in

the hierarchy. The material class finally builds a flat view of the interfaces of the aspect classes in the hierarchy (see figure 6-2).

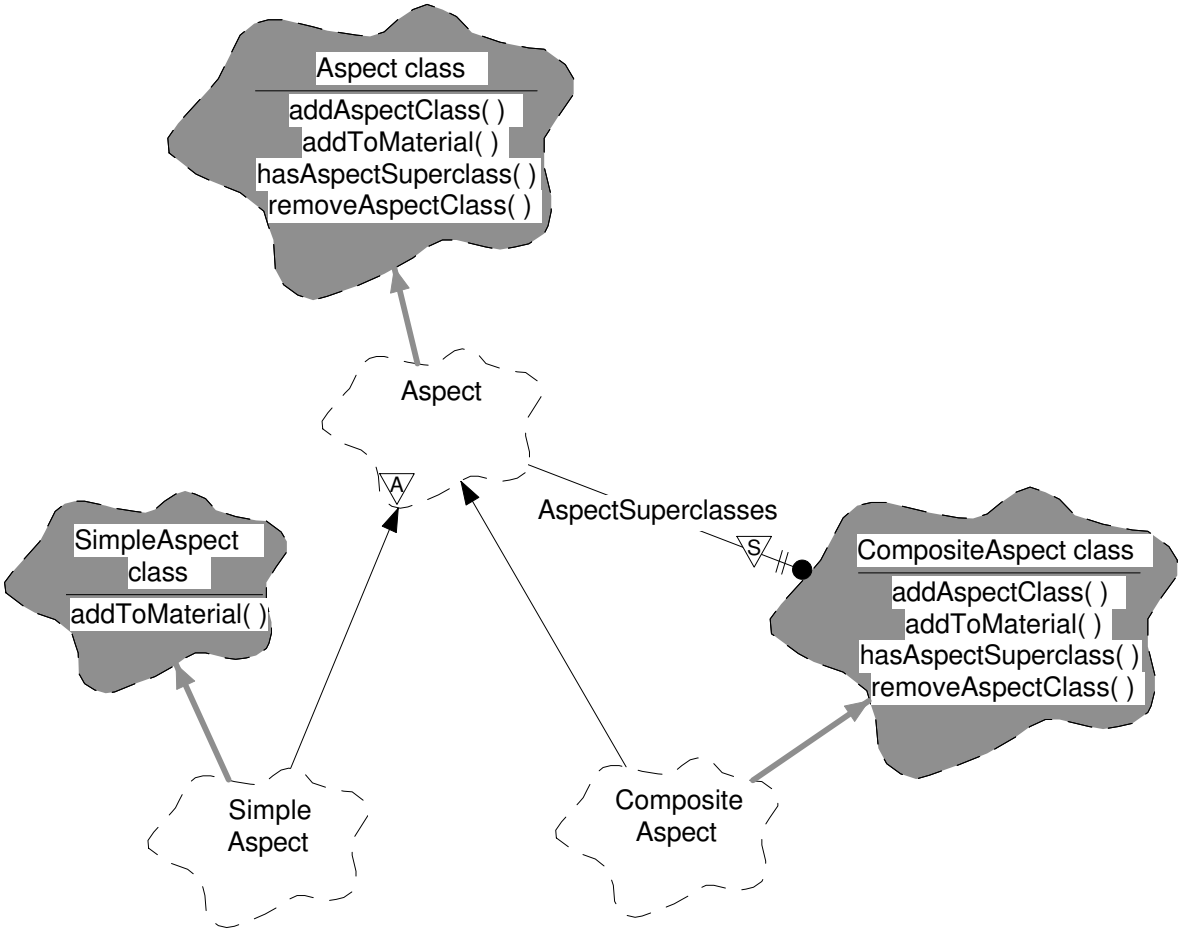

Figure 6-1: Structure for managing composite aspect classes.

Figure 6-1 shows a simplified diagram of our application of the Composite pattern. The gray metaclass objects visualize that the methods to handle the composite structure are all located in the metaclass interface. This static structure is used because our composite structure deals with aspect classes, not instances of aspect classes.

Aspect classes are subclasses either of *SimpleAspect* or of *CompositeAspect*. The classes *Aspect*, *SimpleAspect* and *CompositeAspect* itself are not counted as aspect classes. Their purpose is to structure the aspect classes and to provide common functionality for all aspect classes. The Aspect Browser tool, which is described in the next chapter, uses several methods of their metaclass interface to organize the aspect classes and assign them to material classes. The public methods of these classes are:

addToMaterial:	Adds the aspect to the given material class. Copies all methods of the aspect class to this material. The inverse operation *removeAspect:* is implemented in the class *Material*.
addAspectClass:	Adds the given aspect class to the list of aspect superclasses. For simple aspect classes, all methods related to aspect superclasses don't do anything.
removeAspectClass:	Removes the given aspect from the list of aspect superclasses.
aspectSuperclasses	Returns a list of all direct aspect superclasses.
allAspectSuperclasses	Returns a list of all aspect superclasses in the hierarchy.

hasAspectSuperclass: Returns true, if the given aspect class is part of the list of all aspect superclasses.

isSimple Returns true, if the aspect class is a subclass of *SimpleAspect* and false if it is a subclass of *CompositeAspect*. Although the Composite pattern normally makes this kind of type tests superfluous, there were some situations where this method significantly simplified the implementation.

Aspect classes, which are only the subclasses of *SimpleAspect* and *CompositeAspect*, contain exactly one method category with the same name as the class. This category contains all the methods, that the aspect provides. Neither the user of the framework, nor the framework itself has ever to create instances of aspect classes. Their methods are only copied by the Aspect Browser but never called directly. Only their class interface inherited from *SimpleAspect* or *CompositeAspect* is used. So, their main purpose is to serve as containers for the aspect methods.

The class *Material* maintains a list of the aspect classes that are assigned to it. The list is contained in a dictionary in the class variable *Aspects*. The dictionary has an entry with the actual list of aspect classes for every subclass of *Material*. The Aspect Browser uses the following class methods of the class Material to manage the list of aspects classes:

addAspectClass: Adds the given aspect class to the list of aspects.

removeAspectClass: Removes the given aspect class from the list, but only if this aspect class is not an aspect superclass of an other aspect class in the list. If the aspect class can be removed, the private method *removeAspectMethods:* is called afterwards, which deletes all methods for this aspect from the material class.

aspectClasses: Returns the list of aspect classes.

hasAspect: Returns true if the given aspect class is a member of the list of aspects and false otherwise.

hasAspectCategory: Returns true, if the material class has a method category with the name of the given aspect class. This means, to a certain extent, that the aspects methods are contained in the material class.

These class methods could be used manually, if the Aspect Browser would fail to handle the list correctly for some reason.

6.2 The design of the Aspect Browser

Chapter 4 explained our solution to the multiple inheritance problem with aspect classes and the need for tool support resulting from this solution.

To follow the various browsers in the Smalltalk environment, we named our tool Aspect Browser. The Aspect Browser provides the following basic functionality:

- Creation and editing functionality for aspect classes (simple and composite aspects).
- Assignment of aspect classes to material classes. Removal of aspect classes from material classes.
- Providing consistency between material classes and aspect classes.

- Editing functionality for methods of the material classes, after assigning aspect classes to the material class.

The main disadvantage of our approach to the multiple inheritance problem is, that it is difficult to preserve consistency between the material classes and the aspect classes assigned to them. The implementor of a material class is free to add methods to the material, which are not part of an aspect class. He could also delete aspect methods from the material class instead of implementing it. The Aspect Browser has to provide support for synchronizing aspect classes and material classes, after changing the interface of one of them.

Another important function of the Aspect Browser is the assignment of aspect classes to a material class and the removal of them. As explained in section 6.1, assigning an aspect class to a material class means copying all the methods of the aspect class with their enclosing category to the material class. In the case of a composite aspect class, this process has to be repeated for all aspect superclasses. Since the aspect superclasses can themselves be composite aspects, this is a recursive process.

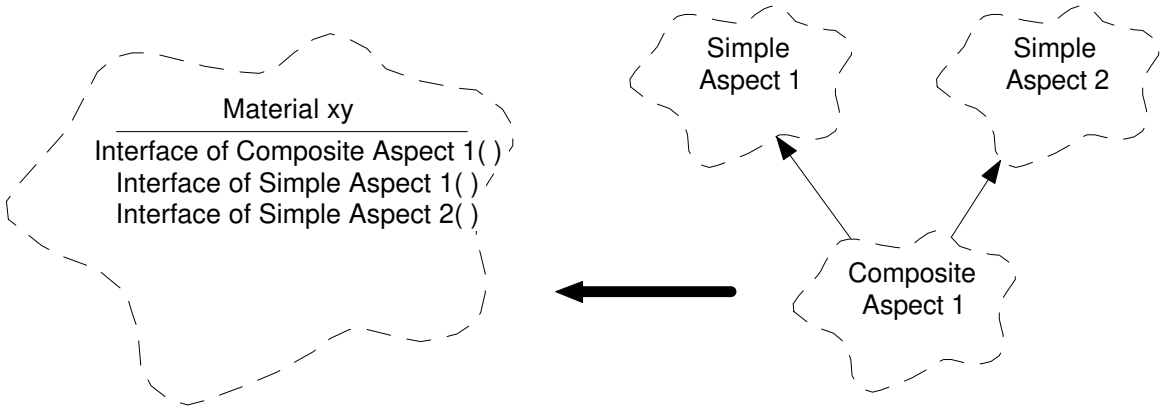

Figure 6-2: Assigning a composite aspect class to a material class.

After assigning a composite aspect class to a material class, the material class builds a flat view of the interfaces of the composite aspect structure. The composite aspect structure in figure 6-2 shows the logical inheritance structure, not the real structure based on the Composite pattern.

The removal of an aspect class from a material class simply means deleting all the methods from the material class, that have the same selector as a method of the aspect class.

The class and application browsers in the IBM Smalltalk environment are implemented in a class hierarchy with the root class *EtWindow*. We could have derived our Aspect Browser from one of the classes in this hierarchy, since the Aspect Browser has some similarities with the Smalltalk browsers. One of our ideas for future work is to build the Aspect Browser on top of the Tools and Materials framework. If we had implemented the Aspect Browser using the Smalltalk browser classes, a later switch to our framework would have been hindered. Therefor, we implemented the first version of the Aspect Browser in a straightforward way and copied some useful code from the Smalltalk browser classes. We could not develop the Aspect Browser directly on the framework, because the Aspect Browser and the framework have been developed at the same time. Developing an application on a framework that still runs through significant changes is a daunting task.

A big part of the Aspect Browsers functionality is contained in the class methods of the classes *Aspect*, *SimpleAspect*, *CompositeAspect* and *Material*. The copying of methods between aspect classes and material classes and their synchronization is implemented in these classes. The user interface of the Aspect Browser is implemented in a class *AspectBrowser*. The class *As-*

pectBrowser is contained in a IBM Smalltalk application of the same name together with several prompter classes for the application specific dialogs.

6.3 Using the Aspect Browser

This chapter gives an introduction to using the Aspect Browser. It explains the purpose of the widgets in the Aspect Browser window and the various menu entries.

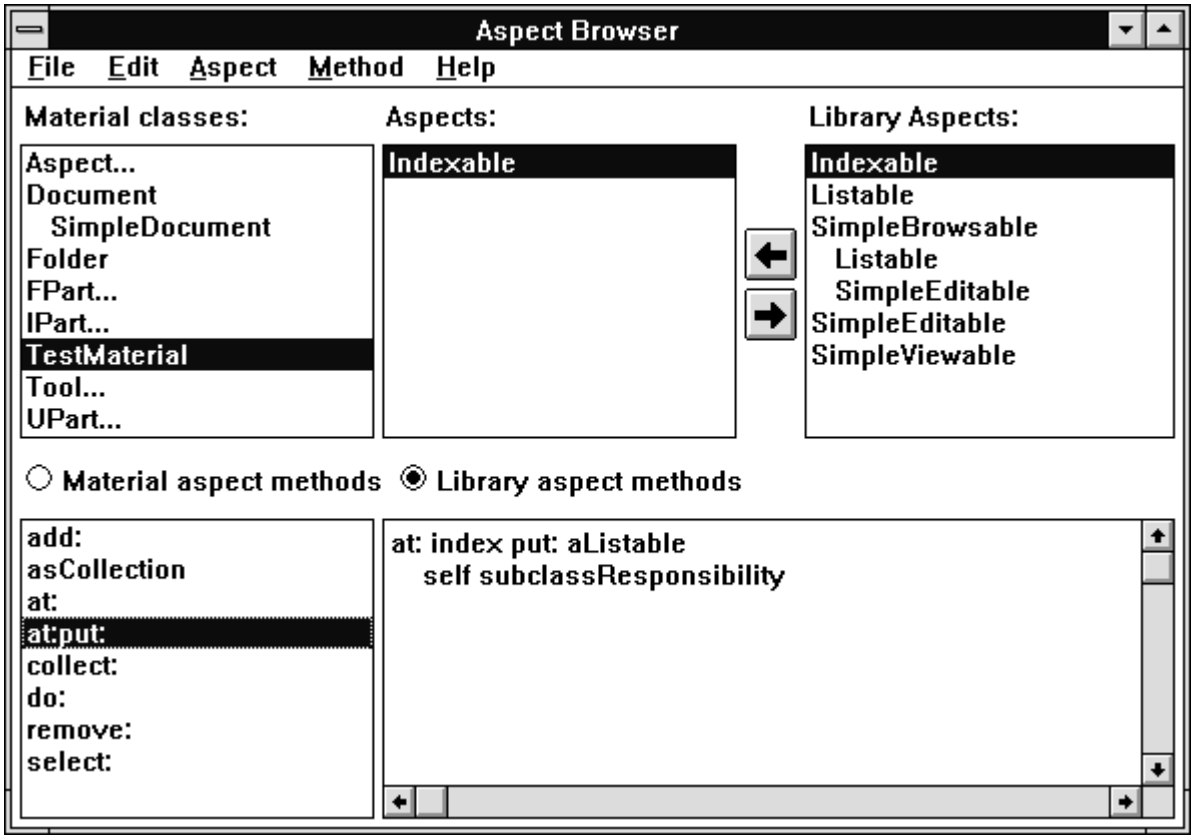

Figure 6-3: The Aspect Browser window.

6.3.1 Elements of the Aspect Browser window

The Aspect Browser window contains several list boxes and an edit field. The list boxes in the upper half of the window show the material classes, their assigned aspect classes and the available library aspect classes. Library aspect classes are those aspect classes which the user can assign to material classes.

With the two (arrow) buttons between the list boxes, aspect classes can be added to a material class or removed from it. The list box and the edit field in the lower half of the window are used to edit the methods of the aspect classes. The list box shows either the methods of the library aspect classes or the methods of the aspect classes assigned to the material classes, depending on the setting of the radio buttons in the middle of the window. In the edit field, the method source code can be edited. It behaves similar to the source code editor in the classes and applications browser of the Smalltalk environment.

The list boxes showing the material classes and the aspect classes are hierarchical list boxes. They visualize the tree structure of the material class hierarchy and the aspect hierarchy through indention. Collapsed entries in these list boxes are marked with three dots '...'. They can be expanded and collapsed by double-clicking it.

The edit field and most of the list boxes have context menus, which can be opened with the right mouse button.

6.3.2 Creating and deleting library aspect classes

New aspect classes can be created from the Aspect menu in the menu bar or the context menu of the Library Aspects list box. There are separate menu entries for creating simple aspect classes and composite aspect classes. For simple aspect classes, the only information the user has to enter is the name of the new aspect class. Figure 6-4 shows the dialog, which is opened for the creation of a composite aspect class.

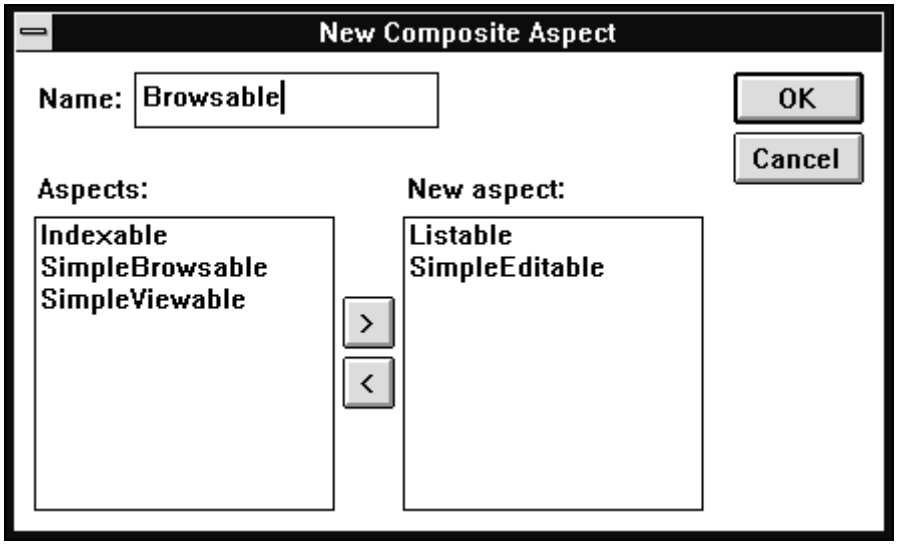

Figure 6-4: Dialog for the creation of new composite aspect classes.

Here, the user has to specify, which of the list of available aspect classes should be the aspect superclasses for the new composite aspect class. With the two (arrow) buttons, aspect classes can be added to or removed from the list of aspect superclasses.

The creation of a simple or a composite aspect class results in the creation of a new aspect class, which is derived from the corresponding class *SimpleAspect* or *CompositeAspect*.

Aspect classes can be deleted with the menu entry ‘Delete’ from the Aspect menu or the context menu of the library aspects list box. The Aspect Browser tests first, if the aspect class that is going to be deleted is a part of a composite aspect class. If it is a part, a message is displayed, stating that the composite aspect class has to be deleted first. A following dialog presents a list of all composite aspect classes, in which the aspect class, which is to be deleted, takes part.

6.3.3 Editing methods of aspect and material classes

The lower half of the Aspect Browser window is used to edit the methods of aspect and material classes. The two radio buttons in the middle of the window serve to switch the visible methods, either the methods of the selected aspect of the material class or the selected library aspect class.

The methods list box and the editor basically function the same way as their counterparts in the Smalltalk browsers. The context menu of the methods list box has menu entries for inserting a new method template in the editor and for deleting the selected method. The context menu of the editor field corresponds to the one in the Smalltalk browsers, excepting the missing ‘File In’ option.

6.3.4 Assigning aspect classes to material classes

Aspect classes can be assigned to material classes using the ‘left arrow’ button between the aspects list boxes or with the menu entry ‘Add to Material’ from the Aspect menu or the context menu. With the assignment of an aspect class to a material class, a new method category is created within the material class and the methods of the aspect class are copied into this category. If the assigned aspect class is a composite aspect class, all the aspect superclasses are also copied, if they are not already assigned to the material.

Removing an aspect class from a material works similar to the assignment. The ‘right arrow’ button or the Menu entry ‘Remove from Material’ can be used to remove an aspect class.

6.3.5 Updating/synchronizing aspect classes

Aspects can be updated in two directions. After a change to a library aspect class, all material classes, that have this aspect assigned, can be updated. After a change to an aspect of a material class, the corresponding library aspect class can be adapted.

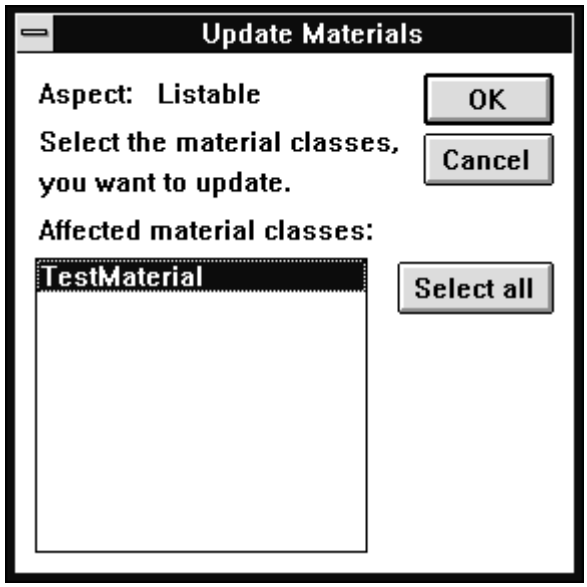

Figure 6-5: Dialog for the update of the material classes

Material classes can be synchronized with a changed library aspect class with the menu entry ‘Update Materials’ from the Aspect menu or the context menu.

Figure 6-5 shows the dialog, which is opened for the update of the material classes. The list box displays all material classes, which have the selected aspect class assigned. From the list, the user can select which material classes should be updated.

During the update of a material class, every additional method of the library aspect class is copied to the material class. In the current implementation, every method missing in the library aspect class is also deleted from the material. We will remove that for the next release, because it is too dangerous to just delete methods from material classes. The Aspect Browser should instead either inform the user in advance or just present a list of methods, which the user has to delete manually.

The update of a library aspect class, based on a changed aspect in a material class, can be invoked using the menu entry ‘Update Library Aspect’ either from the Aspect menu or the context menu.

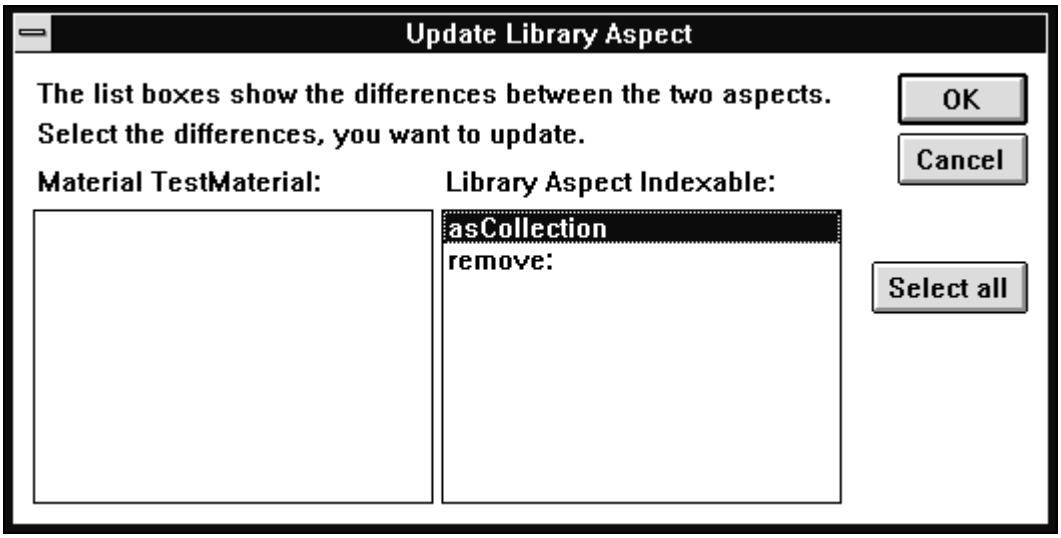

Figure 6-6: Dialog for the update of an aspect class, based on a changed material class.

A dialog displays all the differences between the interface of the aspect in the material class and the interface of the library aspect class. The listed methods are those, which are only present in that aspect. The user can select, which methods are to be updated. A selected method in the material list box means, that this method should be copied to the library aspect class. A selected method in the library aspect list box means, that this method has to be deleted from the library aspect class. This gives the user complete control over the process of updating, which is necessary for such a significant operation as the change of a library aspect class. Updating a library aspect class also has an impact on the other material classes, which have this aspect class assigned.

6.3.6 Menu reference

The menu reference gives a short explanation to every menu entry in the menus of the Aspect Browser. The reference is ordered after the menus in the menu bar. The menu entries in the context menus are implicitly explained, since they build a subset of the menu entries accessible from the menu bar.

6.3.6.1 Menu File

New, Open, Save, Revert, Save Image, Save Image As, Exit Smalltalk: These entries all correspond to the equally named entries in the Smalltalk browsers file menus. They have the same functionality.

Update Lists: Updates the list of material classes and aspect classes in the Aspect Browser window. This is necessary if for example material classes are created or deleted outside the Aspect Browser.

6.3.6.2 Menu Edit

Cut, Copy, Paste, Execute, Display, Inspect, Select All, Search/Replace: These entries all correspond to the equally named entries in the Smalltalk browsers edit menus.

6.3.6.3 Menu Aspect

New:	Creates a new simple aspect class.
New Composite:	Creates a new composite aspect class.
Delete:	Deletes the selected library aspect class.
Add to Material:	Adds the selected library aspect class to the selected material class. Copies all the methods of the aspect class to the material class.
Remove from Material:	Removes the selected aspect class from the material class. Deletes all the methods, which belong to this aspect from the material class.
Update Materials:	Updates the materials, which have the selected library aspect class assigned, after a change to this library aspect class. The user can restrict the list of material classes, which are updated.
Update Library Aspect:	Updates the library aspect class, based on the selected aspect of the selected material class.

6.3.6.4 Menu Method

New Method Template:	Replaces the content of the editor field with a method template.
Delete Method:	Deletes the selected method.

6.3.6.5 Menu Help

About Aspect Browser:	Displays the version and copyright information to the Aspect Browser.

6.3.7 Implications of using the Aspect Browser

Materials and aspects are ordinary Smalltalk classes that could be edited with the standard classes and applications browser of the Smalltalk environment instead of the Aspect Browser. The additional functionality of the Aspect Browser is, that it maintains the list of aspects in the material classes and the composition structure of aspect classes. It further helps to provide consistency between material classes and aspect classes. These management tasks can't be performed with the standard browsers.

Several problems could arise when the Aspect Browser is used in combination with classes or applications browsers to edit aspect or material classes. Some of the possible problems are:

If material classes are edited with the Smalltalk browsers:

- Methods of an aspect can be removed from a material class. That leaves the material with an uncompleted aspect.
- Additional methods could be entered that have the same selector as a method of an aspect. This will lead to problems, if one wants to assign this specific aspect to the material class. However, the Aspect Browser informs the user of methods that are already in the material and thus can't be copied.

If aspect classes are edited with the Smalltalk browsers:

- The danger of forgetting to update the material classes would be higher.

- A future version of the Aspect Browser will possibly include a version management for aspect classes. The Aspect Browser would have to actualize the version of the aspects classes after making changes to them. Editing the aspect classes with the Smalltalk browsers could corrupt the version management.

Despite all these arguments against editing material and aspect classes with the Smalltalk browsers, this is still possible if one is aware of the implications and knows about the behavior of the Aspect Browser.

6.4 Method dispatch for multiple inheritance in Smalltalk

During the development of the framework we examined another way to implement multiple inheritance in Smalltalk. This approach is described shortly together with an application of the same technique for the implementation of a generic wrapper class.

In order to realize multiple inheritance in Smalltalk we have to find a solution for the binding of a message sent to an object to one of the objects methods or to a method in one of the inherited classes. One way to achieve this is to use a dispatcher object, which dispatches every message it receives to one of a list of objects. The job of the dispatcher is to find the object, which can respond to the received message. If more then one of the tested objects can respond, it can't be decided which method the sender wants to call and thus an error condition should be raised.

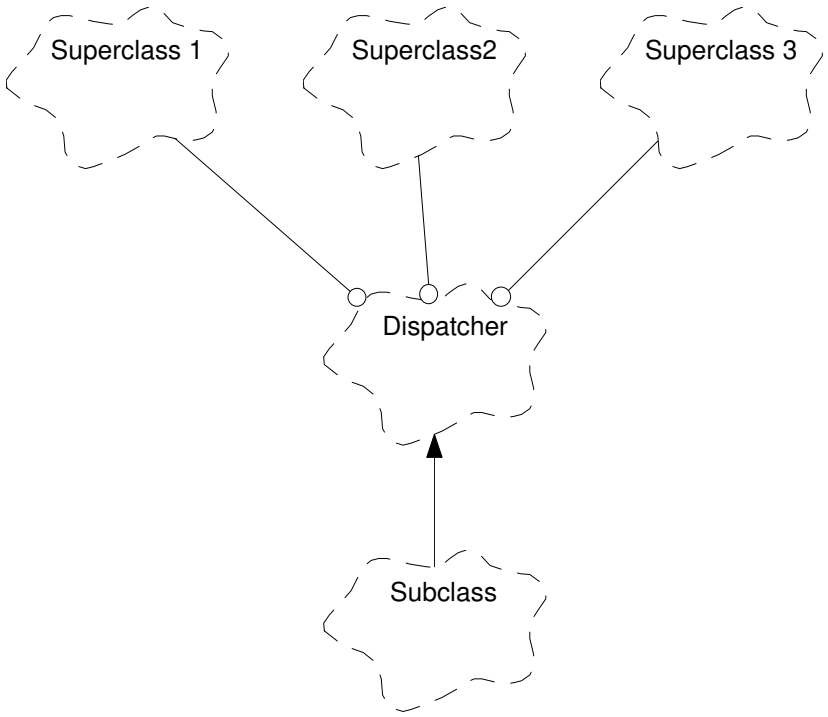

Figure 6-7: Dispatcher class with its participant classes.

The subclass in the multiple inheritance relationship should inherit from the dispatcher class. A message to an instance of the subclass is passed to the dispatcher, if the subclass doesn't implement a method that can be bound to the message. The dispatcher can be implemented using the Facade pattern described in [GHJV95]. For our specific problem, the dispatcher can be made more generic. This can be achieved by using a Smalltalk specific feature. The dispatcher class overrides the method *Object>>doesNotUnderstand:* to do the dispatching. The dispatcher class is derived from the class *Object*, so that it responds directly to all messages, that can be bound to a method in the interface of *Object*. Any other messages result in a *doesNot-*

Understand: message. In the method *doesNotUnderstand:*, we select the right receiver object and forward the message to it.

```
MethodDispatcher>>doesNotUnderstand: aMessage
   | targets selector |
   selector := aMessage selector.
   targets := superclassInstances select: [ :superClass |
     superClass respondsTo: selector].
   targets size = 1 ifTrue: [
     ^aMessage sendTo: targets first].
   targets size = 0 ifTrue: [
     ^self error: self class name,' does not understand ', selector].
   targets size > 1 ifTrue: [
     ^self error: 'More than one superclass responds to ', selector].
```

The instance variable *superclassInstances* holds a list of instances of the superclasses. These instances can be created in the method *initialize* of the dispatcher class.

```
MethodDispatcher>>initialize
   superclassInstances := self class superClasses collect: [:each |
     each new ].
```

superClasses is an accessor method to a class variable *SuperClasses*, which holds the references to the classes that act as superclasses. Since every class derived from the dispatcher class wants to define it's own superclasses in the class variable *SuperClasses*, *SuperClasses* is a dictionary with an entry for every subclass. The keys are the names of the subclasses.

In order to allow multiple inheritance on more then one level the method *respondsTo:* has also to be overridden. This method has to test if the class itself or one of its superclasses can respond to a message with the given selector.

```
MethodDispatcher>>respondsTo: selector
   (super respondsTo: selector) ifTrue: [ ^true ].
   ^(superclassInstances
     detect: [ :each | each respondsTo: selector ]
     ifNone: [ nil ]) notNil.
```

To use the dispatcher class for multiple inheritance one only has to derive a subclass from the dispatcher class and set the superclasses once.

```
SubClass1 superClasses: (Array with: SuperClass1 with: SuperClass2)
```

The described simple implementation can't solve all the problems that arise with multiple inheritance. One problem is that if one of the superclasses sends a message to self, the method dispatching starts at this superclass and not at the subclass as one would expect.

The technique used for method dispatching to realize multiple inheritance is also useful for the implementation of a generic wrapper class. Instead of dispatching a message to one of a list of objects, the wrapper only has to forward the message to the wrapped object.

```
ObjectWrapper>>doesNotUnderstand: aMessage
   self messagePreProcessing: aMessage.
   ^aMessage sendTo: content
```

One usage of this kind of wrapper class is for the pre-processing of every message sent to the wrapped object. For this purpose, the wrapper class should not be derived from class *Object*, so that every message is forwarded to the wrapped object.

Specific wrapper classes can subclass *ObjectWrapper* and override the method *messagePreProcessing:*. One possible application would be a message Tracer.

```
Tracer>>messagePreProcessing: aMessage
  "Prints each message, the wrapped object receives, to Transcript."
  | messageText |
  messageText := self content printString, '>>',
     aMessage selector asString, aMessage arguments printString.
  Transcript cr; show: messageText.
```

The wrapper class can provide a creation method *ObjectWrapper class>>forObject:*, which after creation of a new instance sets the instance variable *content* to the wrapped object. In the case of the Tracer, the wrapped object could then be created together with the wrapper object:

```
Tracer forObject: ClassXYZ new.
```

Other applications of this wrapper class could be the queuing of messages, profiling of classes or various kinds of proxy classes, e.g. for access protection or locking of an object before accessing it. In IBM VisualAge, this technique is used in the implementation of part wrappers.

7 Conclusions

This chapter gives a short summary reflecting on what has been achieved by the framework so far. It summarizes some of our experiences with using Smalltalk in three main respects. It then points out the next steps to extend the framework.

7.1 Observations on Smalltalk

This section summarizes some of our experiences of using Smalltalk compared with our experiences of using C++ and the software development environment Sniff+. The issues are separated into the subsections (1) language issues, (2) environment issues and (3) system issues. These experiences are our personal impressions. We do not claim to be very experienced Smalltalk programmers, so some of the experiences reported about below may not be appropriate for generalization.

7.1.1 Language issues

The following is a list of common themes leading to mistakes which we did and which we think are typically done by others as well (depending on the degree of a developer's experience with Smalltalk). They are based on the lack of static typing which otherwise would have enabled the compiler to catch the errors.

- *Wrong return values.* It is easy to forget the ^ token which corresponds to *return* in C++. Since each method has an implicit return value this cannot be indicated as an error. The most popular version of this mistake is to forget the ^ in the class method *SomeConcept class>>new* implemented as *^super new initialize*.
- *Simple typing mistakes.* It is easy to make simple typing mistakes which cannot be indicated as such since the error is a typing error. A common variant of this mistake is to forget the trailing dot at the end of a statement which may or may not be caught at compile time.

These mistakes are only discovered at runtime and through testing. They could have been caught easily in a typed system.

The ease of using the language and its purity in terms of objects, however, is well-known and provided us with a clean model of using Smalltalk.

7.1.2 Environment issues

The IBM Smalltalk environment offers several browsers, the complexity of which is probably overwhelming for most of the inexperienced Smalltalk developers. During development we

switched to VisualAge which provided us with even more confusing browsers. In the following, we list our impressions and include some early experiences with VisualAge as well.

- *Number of browsers.* The number of browsers is high. There are many slightly differing browsers which all can serve as simple class browser, but pop up in different settings and have slightly differing menus. We do not feel that the rationale behind the browsers and their functionality is always intuitively clear.
- *The problem of text versus higher level abstractions.* Traditional compilers take arbitrary text as an input. So does Smalltalk, but any change to a method is directly incorporated into the system and is presented according to the browsers logic.
  - One cannot easily work with annotations as comments (since there is no simple text retrieval facility). Thus, traditional methods of organizing one's work processes by the help of annotations aren't applicable.
  - One has to accept the presentation of methods in an alphabetically sorted way. The only way of further organizing the methods is to put them into method categories. This is sometimes insufficient, since method categories are either too course grained or not appropriate.
  - It is difficult to get an overview of a class's implementation, since one has to browse all the methods one by one. We felt that our perception of a system became much more fragmented since we had to get the details of a class in a more sequentialized order.
- *Silent messages to the Transcript.* Each new Smalltalk programmer learns very soon that he or she has to pay attention to the silent messages appearing on the Transcript. It is only there that it is indicated, for example, that some methods of a class have been removed because they cannot be executed since they are based on a previously removed instance variable.
- *Modality of editing methods.* All browsers are modal with respect to editing a method. It is not possible to browse another method if the current method has been edited and is in an inconsistent state. Having edited a method, one can only leave that browser state by either successfully compiling this method or by discarding the changes. To look at another method, a new browser has to be opened.
- *VisualAge composition editor.* We have tried the composition editor to build user interfaces by drag and drop (GUI builder) and think that it still needs further refinement. Some example problems include (but are not limited to) receiving unmotivated messages from a VisualAge part, problems with using German Umlauts and difficulties with composing parts (changing a part that has been incorporated into another part does not update that other part - so much for aggregation and inheritance between parts).

Summarizing, we feel that the IBM Smalltalk or VisualAge browsers need a phase of consolidation and better integration. Programmers which rely on text layout to logically and visually structure their source code, for example class definitions, or who use text as a basic means for organizing their work processes with source code most likely will have to give up on this.

However, despite the resulting fragmentation of our perception the elegance of point-and-click browsing through the system has been well apparent to us.

7.1.3 System issues

The only relevant system issue we encountered is a problem with Smalltalk's garbage collection. When interfacing to Smalltalk external resources like a window system it may not be

possible to coherently integrate these into Smalltalk. In particular, in IBM Smalltalk window system resources, for example widgets, have to be freed by hand and are not subject to garbage collection.

This section has summarized some of our experiences with Smalltalk in general and IBM Smalltalk in particular. At present, we cannot say how difficult we think issues like missing static type checking will turn out to be. In general, however, we are confident and have experienced the system as powerful means for developing the framework.

7.2 Summary

We have presented the design of a Smalltalk Framework for the Tools and Materials Metaphor. The design was thoroughly based on prior experience with a similar framework. Otherwise it would not have been possible (nor sensible) to develop the framework within the given time.

We enhanced the basic framework classes, set up classes for tool construction, identified some aspect classes and used example material classes. The framework classes for tools stem from our experience. Based on our previous experience we assume that they have already reached maturity.

We introduced a tool for conceptually dealing with multiple inheritance. Essentially, this tool in the tradition of the Tools and Materials Metaphor provides a specific view on some of the software developers' materials, that is classes and class interfaces, and specific ways of handling theses materials based on the requirements for the task of dealing with multiple inheritance. The Aspect Browser will certainly evolve.

This report has been written on the detailed level of class interfaces. We have therefore presented a detailed technical description of the framework going beyond current framework descriptions which usually stopped at a more abstract level.

Due to its detailed description, the framework is open to be criticized. However, this is just what we are hoping for. We are looking forward for such critique and hope to gain new insights from it which we can use then to enhance, refine and improve the framework.

7.3 Future work

The future of the framework lies in the hand of an industrial project which will use it as a basis for further development. The framework will certainly evolve. Here we shortly list which kind of future enhancements could be considered:

- The change/update mechanism might be replaced by advanced techniques for dealing with object dependencies, for example techniques discussed as implicit invocation [SN92, NGGS93]. Events might be made explicit as event objects in a class interface to which clients might link themselves. In case of events not a general list of observers is notified but only those clients which registered for a certain event.
- The Aspect Browser will continue to evolve. It has to deal with issues that arise in the context of large projects and team development, for example if classes are not available in a single image but stored in a repository. System consistency and versioning will require several extensions so that it can be used as a real aid in system design.

- The environment will definitely evolve when external services are connected to the system. This will most likely lead to the introduction of material providers and suppliers and other concepts as presented in [Rie95a, RZ95].

This Tools and Materials Metaphor framework is the first Smalltalk framework of its kind. We found that developing and working with it was fun. However, this is just the beginning. The next year will reveal further observations and findings for such an undertaking.

Bibliography

BCS92 Reinhard Budde, Marie-Luise Christ-Neumann and Karl-Heinz Sylla. "Tools And Materials: An Analysis and Design Metaphor." Tools-7, *Conference Proceedings*. Edited by Georg Heeg, Boris Magnusson and Bertrand Meyer. Prentice-Hall, 1992. 135-146.

BGZ95 Ute Bürkle, Guido Gryczan and Heinz Züllighoven. "Object-Oriented System Development in a Banking Project: Methodology, Experiences, and Conclusions." *Human Computer Interaction 10*, 2&3 (1995): 293-336.

Bis95 Walter R. Bischofberger. "Frameworkbasierte Softwareentwicklung." OOP München '95, *Conference Proceedings*. SIGS Publications, 1995.

Boo94 Grady Booch. *Object-Oriented Analysis and Design with Applications.* 2nd Edition. Redwood City, California: Benjamin/Cummings, 1994.

ES90 Margaret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Reading, Massachusetts: Addison-Wesley, 1990.

FG92 Christiane Floyd and Guido Gryczan. "STEPS – a Methodological Framework for Cooperative Software Development with Users." EWHCI '92, *Conference Proceedings*. 1992.

GHJV95 Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Design.* Reading, Massachusetts: Addison-Wesley, 1995.

GOP90 Keith E. Gorlen, Sanford M. Orlow and Perry S. Plexiko. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons Ltd., 1990.

HO93 William Harrison and Harold Ossher. "Subject-Oriented Programming (A Critique of Pure Objects)." OOPSLA-93, *Conference Proceedings.* See also: *ACM SigPlan Notices* 28, 10 (October 1993): 411-428.

KGZ93 Klaus Kilberth, Guido Gryczan and Heinz Züllighoven. *Anwendungsorientierte Softwareentwicklung.* Vieweg, 1993.

KM93 Sarah Kuhn and Michael J. Muller. "Participatory Design." *Communications of the ACM* 36, 4 (June 1993).

KP88 Glenn E. Krasner and Stephen T. Pope. "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80." *Journal of Object-Oriented Programming* 1, 3 (August/September 1988): 26-49.

Mey92 Bertrand Meyer. *Eiffel – The Language*. McGraw/Hill, 1992.

NGGS93 David Notkin, David Garlan, William G. Griswold and Kevin Sullivan. "Adding

Implicit Invocation to Languages: Three Approaches." ISOTAS-93, LNCS-742, *Conference Proceedings*. Edited by Shojiro Nishio and Akinori Yonezawa. New York: Springer-Verlag, 1993. 489-510.

OH92 Harold Ossher and William Harrison. "Combination of Inheritance Hierarchies." OOPSLA-92, *Conference Proceedings.* See also: *ACM SigPlan Notices* 27, 10 (October 1992): 25-40.

Rie93a Dirk Riehle. *Dokumentation zur IATMotif-Bibliothek.* Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1993.

Rie93b Dirk Riehle. *Dokumentation zur FIAK-Bibliothek.* Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1993.

Rie95a Dirk Riehle. *Muster am Beispiel der Werkzeug und Material Metapher*. (Masters Thesis, in German). UBILAB Technical Report 95.6.1. Zürich, Switzerland: Union Bank of Switzerland, 1995.

Rie95b Dirk Riehle. "How and Why to Encapsulate Class Trees." OOPSLA '95, *Conference Proceedings*. To appear, 1995.

Rie95c Dirk Riehle. "Patterns for Encapsulating Class Trees." PLoP '95, *Conference Proceedings*. To appear, 1995.

RSS95 Dirk Riehle, Bruno Schäffer and Martin Schnyder. "Design and Implementation of a Smalltalk Framework for the Tools and Materials Metaphor." *Informatik/ Informatique* (February 1996). To appear.

RZ95 Dirk Riehle and Heinz Züllighoven. "A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor." *Pattern Languages of Program Design.* Edited by James O. Coplien and Douglas C. Schmidt. Reading, Massachusetts: Addison-Wesley, 1995. 9-42.

SN92 Kevin J. Sullivan and David Notkin. "Reconciling Environment Integration and Software Evolution." *ACM Transactions on Software Engineering and Methodology* 1, 3 (July 1992): 229-268.

WG94 André Weinand and Erich Gamma. "ET++ – a Portable, Homogenous Class Library and Application Framework." *Computer Science Research at UBILAB*. Edited by Walter R. Bischofberger and Hans-Peter Frei. Konstanz: Universitätsverlag Konstanz, 1994. 66-92.