A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor

Dirk Riehle and Heinz Züllighoven

{riehle, zuelligh}@informatik.uni-hamburg.de
University of Hamburg, Dept. of Computer Science, Software Engineering Group
Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany

1 Background and Motivation

Why do people prefer to use certain software systems and why do they have problems using others? What is the quality within certain software that makes people soon feel familiar using it and lets them work efficiently? These are questions that we, like a lot of people who develop and use application systems in their everyday work, asked ourselves.

We believe that the key to this quality (in the sense of [Pir74]) is found in systems that allow people to work according to their qualification and needs while using their skills and competence.

In order to develop software with this quality we, as a group, have put together many of the things which have proved useful in software engineering over the last decades and have integrated these methods and techniques into a unifying approach – the Tools and Materials Metaphor [BCS92, BZ92]. It has guided us and other developers during analysis and design and has helped to envision and finally build systems of quality.

The Tools and Materials Metaphor is an approach with a specific underlying view of human work:

- People have the necessary competence and skills for their work.
- There is no need to define a fixed work flow, because people know what they do and can cope adequately with changing situations.
- People should be able to decide on their own how to organize their work and their (software) environment according to their tasks.

We have found that software developed with this goal in mind makes people feel comfortable in using the system while improving both the working process and its outcome. This holds at least in the field we are familiar with, which is work in office-like environments and workshops.

But why “Tools and Materials”? From the long tradition of craftsmanship we have learned that human work has often found a physical embodiment through *tools* that craftsmen use to work on *materials*. We have taken this basic notion as a starting point for understanding what tools and materials are and how we can extend this concept to software tools and materials. Thereby we try to bridge the gap between the “soft” social requirements and a “hard” software system.

The term Tools and Materials Metaphor characterizes our overall approach. We explain the approach and its underlying idea by using a pattern language. We start by outlining this language on its “highest” conceptual level which we call design metaphors. Design metaphors are *Tool*, *Material*, *Aspect* and *Environment*, each representing a different but related concept. We then present the next lower level of abstraction which are so-called design patterns for the implementation of the

In: Pattern Languages of Program Design. Edited by James O. Coplien and Douglas Schmidt.
Reading, Massachusetts: Addison-Wesley, 1995. Pages 9-42.

metaphors and we show how to use them as a coherent language. Our notion of patterns is based on [GHJ+93] and [Joh92].

2 The overall approach

The central notions of our approach are *tools* and *materials*, which we call design metaphors. In many work situations an intuitive conceptual distinction can be made between those things that are worked upon (materials) and those that are means of work (tools). We use these metaphors to describe the work of the application domain experts (called users in this paper) for who we develop software. Metaphors provide a conceptual framework for discussion between users and software professionals (called developers), because most people have a sufficient understanding of what tools and materials are.

When actually developing software in a project, we follow an evolutionary and participatory approach using prototyping [Flo84, BKK+92]. Thus, we organize software development as a mutual learning and design process where both users and developers cooperate.

Evolutionary software development means designing software in fast feedback cycles involving all parties concerned. Thus, there are various tasks to perform: We analyze the application domain by observing how users actually do their daily work, focusing on the tools and materials they use and we discuss our findings with them. Looking at everyday work, we try to understand the professional language in use in order to build a model based on these concepts and terms. We extend this language by new concepts necessary for the envisioned system. While doing this we make heavy use of scenarios of current work and system visions about anticipated work with the future system.

Having outlined the *process model* of our approach, we come to our *leitmotif* of humans as skilled and trained experts of their domain. To make the goals of our leitmotif concrete, we use the *design metaphors* of tools, materials, aspects and environment. These metaphors are realized as objects and classes on the design level by a *construction technique*, which we describe as design patterns of our pattern language.

As a foundation for further research one of the authors designed and implemented an application framework for the Tools and Materials Metaphor. Its purpose was to extend and improve implementation techniques used in industrial projects. After the application framework reached a sufficient degree of maturity and was heavily used in students’ projects, he reinterpreted the framework as a pattern language which became the main part of this paper.

3 Leitmotif and design metaphors

The context of the design metaphors to be described in this chapter is our leitmotif of skilled experts whose work we wish to support by software. The metaphors have to conform to the given objectives of enabling skilled human work, no fixed work flow and knowledgeable interaction with software tools.

3.1 The distinction between design metaphors and design patterns

Before elaborating our pattern language, we will clarify the difference between the two levels of this language, i.e. the design metaphors and the design patterns. In short: Design metaphors are patterns which govern our perception of the application domain and guide us when designing the future system, while design patterns are used in the technical construction process as a kind of “micro architecture”.

The *design metaphors* we propose are *tools*, *materials*, *aspects* and *environment*. They express our understanding of human work in many areas as four different but interrelated concepts. These work situations will frequently include the use of computers. Design metaphors provide a guideline and a perspective on how software systems for a given application domain should be designed. They are used for understanding and analyzing the work of others. Thus they provide an approach as well as a world view.

The well-known MVC paradigm is based on the metaphor of direct manipulation and a general concept of domain modeling. This is quite general, lacking any specific leitmotif. Our metaphors are more specifically aimed at skilled human work supported by computers.

Design patterns are based on design metaphors and are used to relate them to the technical level of designing software systems. We follow [GHJ+93] seeing design patterns as a set of related classes and objects interacting in a specific way to achieve a well-defined goal. Design patterns are described by naming the components, their collaborations and responsibilities. They emerge out of experience with recurring solutions in designing software systems which they are abstracted from. They describe the solution to a problem usually making use of a problem-context-solution schema. Design patterns are used in specific contexts they were invented for as “micro architectures”, i.e. they build the actual programs.

The MVC paradigm is an example of a classical design pattern consisting of the three components Model, View and Controller interacting to build a software tool with an underlying domain model. In line with Gamma et al. we wish to stress that patterns are not merely technical but also capture the professional’s domain specific understanding guided by the underlying design metaphors.

Grown from experience, patterns as well as metaphors form the developers’ language and become powerful tools for analyzing and designing software systems. We use them to communicate, document and develop our software.

In actual projects we need a third level of our pattern language, which is the level of programming patterns – they may also be called atoms, idioms or fundamental patterns as on the patterns mailing list. Programming patterns are basic ways of realizing a software design in terms of a programming language and with respect to the principles of software engineering. As an example take the distinction between object and value and different ways to realize this in an object-oriented language [Cun95]. Pipes and filters may also be shown as basic patterns of software [Sha95]. This shows that the distinction between design patterns and programming patterns is not sharp.

To sum up, we have three levels within our pattern language for analyzing, designing and constructing software systems:

1. The design metaphors within the realm of a leitmotif guiding our perception and our thinking;
2. The design patterns helping us to transform our design ideas into a concrete software design;
3. The programming patterns as basic means and forms for expressing software building blocks.

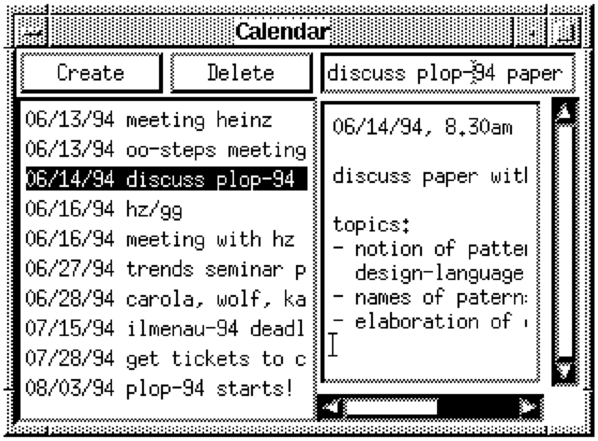
In the following we will explain the first two levels of our language assuming that the third level is familiar to an experienced software engineer.

3.2 The Tools and Materials Metaphor

The design metaphors of our approach are based on the underlying separation between tools as means of work and materials as outcome of work. This so-called tools and materials dichotomy leads to the metaphors of *material, tool, aspect* and *environment*.

The presentation of the design metaphors and patterns is illustrated by the following example: We will design a time planning system to keep track of our dates, appointments, seminars and so on.

Fig. 1 shows our first tool, a simple calendar. It consists of a list of appointments on the left and the contents of a selected appointment on the right. Looking at the calendar we will encounter the tool (the calendar) and its materials (the appointments) as well as aspects of the material (the listable and editable properties).



discuss plop-94 paper

```
06/13/94 meeting heinz
06/13/94 oo-steps meeting
06/14/94 discuss plop-94
06/16/94 hz/gg
06/16/94 meeting with hz
06/27/94 trends seminar p
06/28/94 carola, wolf, ka
07/15/94 ilmenau-94 deadl
07/28/94 get tickets to c
08/03/94 plop-94 starts!
```

```
06/14/94, 8.30am

discuss paper witl

topics:
- notion of patte
  design-language
- names of patern
- elaboration of
```

Fig. 1: A calendar (tool) that was built from the application framework.

Materials Design Metaphor

Problem Looking at the skilled professional, we have to identify what he or she is actually working on, i.e. what the relevant objects of work are. In line with our leitmotif and as a basis for adequate modeling the focus of the experts' work has to be analyzed and made explicit.

Solution The "things", professionals work on are *materials*. Materials are examined, manipulated or incorporated into other materials in order to become part of the work results. Examples of materials are forms, files and folders.

Materials are passive entities which are taken and worked upon when appropriate, like the different forms laid out systematically on a desktop. In our work, we mainly concentrate on materials and get the impression of directly accessing and manipulating them. Though we use tools like a typewriter or a pen to fill in a form, these tools seem to disappear and only the form is in our focus. We often use our hands as tools while working with tangible materials. In software systems, however, we can never access materials "directly" but only by using appropriate tools. In order to maintain an understandable system model, we design "active" tools and "passive" materials. So, in a software system we need a browser tool to look at the contents of a folder.

The calendar tool from fig. 1 shows some appointment materials presented by that tool.

When designing materials we are not concerned with the graphical or textual representation or interactive manipulation. Tools use additional graphical objects to display materials. They also provide the context for working with materials both graphically and logically. Despite the fact that materials are always represented and manipulated by tools, they exist in their own right and should not be subordinate to a specific tool.

Tools Design Metaphor

Problem In order to accomplish a task it is obviously not sufficient to just look at materials and wait for something to happen. Thus, we need the means to both organize and perform our work on materials.

Solution When manipulating materials, we use tools as means of work. Often grown out of a long tradition, they physically embody the experience of how to work efficiently with materials. They are manifestations of the ways and means materials can be examined and manipulated in a given application domain. Tools are our example calendar, all kinds of form editors, browsers for folders and so on.

Tools present a permanent view on materials and give users feedback about their activities. Tools have their own state which relates to the respective work situation. For example, the calendar consists of objects that display the appointments, of objects receiving user input and objects which transform input into manipulation of an appointment. We expect a calendar to "remember" the last selected appointment.

Well-designed tools become transparent when handled by an experienced user, i.e. they are non-distracting and give the user the impression of directly manipulating the materials. Despite this, tools should be marginally present and never "disappear" in a work situation. In unclear or erroneous situations, users need to look at a tool itself.

Being a tool or a material is not an objective property of an item as such. The distinction depends on the task at hand. A pencil, e.g., is a tool when we use it to write on paper; it becomes a material when we sharpen it with a pencil sharpener. As a consequence, we design a help system as a tool whose materials are tools to be inspected for retrieving help about them. We will see that *Aspects* will establish the context that makes clear which is which.

The notion of tools and materials is compatible with the concept of direct manipulation but underpins the role of tools in human work.

Often a task can be divided into subtasks that may be performed independently. The results of these subtasks are integrated to form the overall result. We can organize tools in a similar way. The calendar allows for selecting appointments from a list and editing them. Thus, we build a tool for handling a list of items, a lister, and a tool for processing the text of an appointment, an editor.

We normally use different tools for working upon a single material. Besides the calendar a scheduler tool will give an overview of our weekly dates and other tools will do bookkeeping with appointments.

Aspects — Design Metaphor

Problem We never work with a tool on a material in a generic way, but use our knowledge to select the right tool and handle it in a *specific way* suitable for the combination of tool, material and work at hand. Because there may be a variety of intentions of working with the same tool and material, we have to sort out the different ways of work and their meaning.

Context From traditional crafts, we know that not every tool is suitable for every material. We will hardly use a pencil sharpener to sharpen a ball-point pen. Furthermore, we find that in many areas the relations between tools and materials are governed by standards, e.g. the relation between spanners and nuts. Each tool that fits a material in a specific way does so, because humans designed it that way to perform a specific task. This task is relevant for understanding the relationship between tool and material.

Solution We make the relationship between a tool and a material explicit by introducing *aspects*. An aspect defines a single interface that provides all necessary operations a tool needs to work properly with materials. The interface will contain exactly those operations that a tool will need to work with a material in *a specific task*. At the same time aspects abstract from specific materials. The appointments must be listable and editable for the calendar to display and edit them. The operations necessary to perform these tasks are expressed by two different aspects: *Listable* and *Editable*.

Aspects are of equal importance as the distinction of tools and materials, because they define the necessary operations and thus the context that is needed to perform a specific task. Doing this, they formally establish *the contract which connects a tool with a material* [Mey91]. As a consequence, neither the tool knows its specific materials nor does a material know its tool; they only know the contract that connects them. This decoupling is a major aid in making our systems more flexible (see pattern *Tool and Material Coupling*).

Aspects reflect what work psychologists call usability. Usability means the usefulness of an item with respect to an intention or a purposeful task or activity. It relies on objective or non-objective characteristics within a use situation that can be assessed on the background of individual needs (derived from [DWA93]).

Environment **Design Metaphor**

Problem A tool or a material is never found in isolation. We always work in well-organized places equipped with the things we need. In a computer system there is no “natural” environment of this kind. In order to support skilled work the notions of spatial and logical ordering and relations are of crucial importance to software systems.

Context We perform tasks in a work environment, e.g. in a workshop or on a desktop. Within this environment, we physically and mentally organize and arrange our tools and materials of work according to the needs of the respective tasks and our working habits. Tools and materials therefore have their place, their location and order. We generally do this in an implicit and intuitive way, based on our qualification and experience.

Solution We transfer this concept of environment to the computer, because it provides the means to organize our work. For office work, a familiar concept is the electronic desktop as a place of planning, working and arranging. It provides a space and several logical dimensions for arranging things.

The notion of environment allows us to think about constraints between tools and materials. The environment must provide means for ensuring consistency between them.

4 Design patterns for tool construction and integration

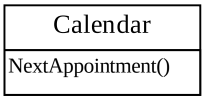
Having introduced the design metaphors of our approach, we will now explain the design patterns in detail. These patterns conform to the metaphors emphasizing certain characteristics and showing ways to efficiently implement them. Design patterns *do not* introduce new concepts or views distinct from those given by the metaphors but carry over their meaning into detailled software design.

4.1 Graphical Notation

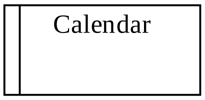
The following figure sketches the graphical notation used throughout the paper. It is based on Rumbaugh’s work [RBP+91] with small deviations.

Appointment

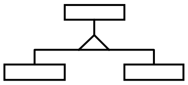
A Rectangle represents a class. If its label is written in *Italics*, the class is abstract.



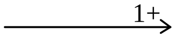
This class has an operation called NextAppointment().



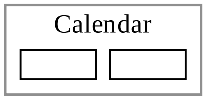
The line on the left shows that this class is essentially a cluster.



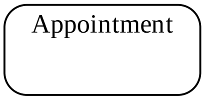
Inheritance (here from top to bottom) is shown by a triangle.



An arrow shows a use-relationship with potential multiplicity at each end.



This is a cluster called Calendar with some classes in it.



This is an object called Appointment.

4.2 Roadmap to the design patterns

This section introduces the design patterns for the construction of tools and their relation to materials, starting with patterns for integrating tools with other tools and materials in an environment. First, however, we will give a short roadmap to these design patterns.

Fig. 2 and 3 show the patterns for tool construction and tool integration respectively. The patterns are ordered according to their range with the largest one being the border of the overall layers of each figure. Some patterns break the simple ordering by range, namely *Separation of Powers*, *Event Mechanism* (Observation) and *Material Container* which overlap.

Each rectangle represents a design pattern except some special classes named *Interaction Part*, *Functional Part*, *Aspect Class* and *Material* whose meaning can be guessed from the metaphors and will be discussed later.

Probably the most important design pattern for tool construction is *Tool and Material Coupling*, which structures the system in the large. *Tools and materials* known from the metaphors become objects that are linked by *aspect classes representing aspects*. Tools use materials via specific aspect classes; materials are subtypes of several aspect classes. Tools are composed from simpler tools using the mechanisms of *Tool Composition*. Each simple tool consists of *an interaction part and a functional part separating the powers* of both domains. To connect the interaction and functional parts of tools, an *Event Mechanism* is used. It allows dependent components to be informed about changes in order to react appropriately. So, the interaction part of a tool depends on the state of its functional part and is informed by the functional part about relevant changes. This is done by *announcing an event*.

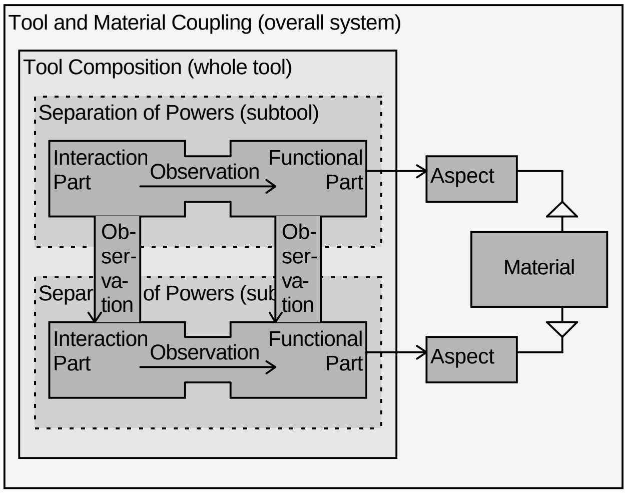


Fig. 2: Patterns for tool construction.

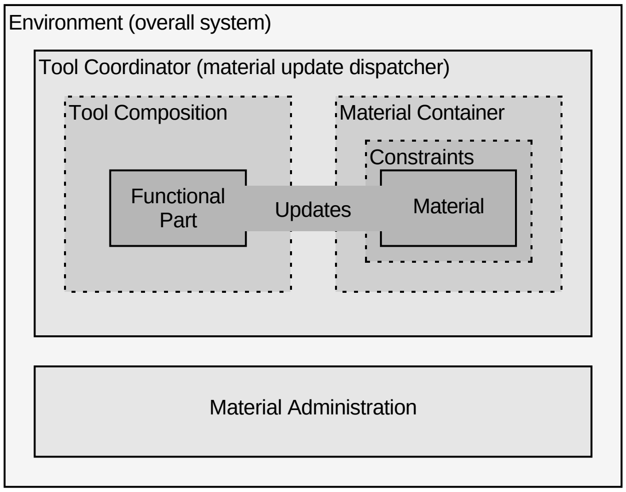


Fig. 3: Patterns for tool integration.

The design patterns for tool integration are based on the patterns for tool construction, especially on *Tool Composition* and *Separation of Powers*, which structure a compound tool. As the boundary of the overall system we use *Environment* which represents the closure for the single (computer) workplace. The environment creates *Tool Coordinators* to inform tools about changes of their materials that might have occurred due to side effects. These side effects are controlled by the *Material Container*, which *groups dependent materials* and *provides constraints* for ensuring consistency.

The *functional parts of tools* are made available on an abstract level to the *Tool Coordinator*. Linked to the tool and material patterns is the *Material Administration* which provides an abstract way of accessing *Material Providers*, i.e. databases. The *Material Administration* and patterns for integrating *Material Providers* are a pattern sub-language of its own, which goes beyond the scope of this paper.

4.3 Design patterns for tool construction

We will first look at a single tool and introduce patterns for its construction based on the metaphors tool, material and aspect.

Tool and Material Coupling — Design Pattern

Purpose Couple tools with materials through aspect classes which implement aspects. This pattern captures the way we work with tools on materials and represents the smallest reusable interface to materials.

Problem Usually a tool is developed to work with a single type of material. But we wish to develop reusable tools that are not tied to specific materials. Several tools should work on the same material and one tool should be usable for several materials.

Context Tools and materials are represented by different objects. The interface of a material should offer all the operations a tool needs but no more. This interface is the aspect a tool uses to work with materials. Aspects should be independent of each other. The code in figure 5 shows the aspects Listable and Editable as C++ class interfaces.

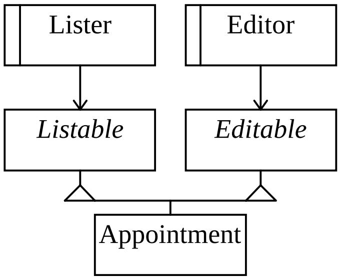


Fig. 4: The lister and editor tool access the same material through different aspect classes.

```
// simple interface for listable objects
class Listable {
  String GetDescription() = 0;
  bool   isEqual( Listable& ) = 0;
  bool   isLower( Listable& ) = 0;
};

// simple textually editable objects
class Editable {
  String GetParagraph( int ) = 0;
  void   SetParagraph( String, int ) = 0;
};
```

Fig. 5: C++ Interface of `Listable` and `Editable`. Their functionality is independent of each other.

A *lister (tool)* displays items from a container used as a list to select from. An *editor (tool)* provides the means for textually editing materials. In order to work properly, both tools need a part of the functionality of their materials expressed through the aspects Listable or Editable. As discussed under the metaphor of aspect, both classes should be treated as a contract established between a tool and a material.

From a tool’s point of view, aspects can be seen as properties of a material. Thus, the appointment objects for the calendar should offer both aspects Listable and Editable, because they can be listed and edited.

Solution Each aspect becomes a class interface in its own right, called an aspect class. An aspect class determines the operations necessary to make a material usable for a specific tool. The aspects Listable and Editable given above are aspect classes.

Tools are restricted to work with aspect classes. As a consequence, tools can use any material that offers the respective characteristics of the aspect class, i.e. fulfills the contract formalized through the aspect class interface. The appointment class implements the operations specified in Listable and Editable. The single line returned by the appointment via its Listable interface may consist of a string with the date and the name of the person to meet. Figure 4 shows the resulting class diagram for our example.

Introducing a new material to a system is done by identifying which tools should be working upon that material. Then the appropriate aspect classes are inherited and the specified operations are implemented. An aspect class can be seen as a partial type. Material interfaces are built out of the different aspect classes they inherit from.

An aspect class is normally an abstract class that establishes the context in which to interpret a class as a tool class and other classes as appropriate materials for this tool. We state: A class is a tool class in context α, if it uses aspect class α, and a class is a material class in context α, if it inherits from aspect class α.

Aspect classes make the dependencies between materials and tools explicit. They structure a material’s interface into different sections, each representing a certain way of usage. Tools working on aspect classes present this interface as the work context to users. Thus, materials cannot be active (nor “self-representing” or “self-editable”). This indirection provided by tools makes our systems more flexible.

Aspect classes are our rationale for dealing with multiple inheritance in a constrained but efficient way. Applied properly, no diamond inheritance structures result and thus no name or repeated inheritance conflicts emerge.

Additionally, they are a step toward independence. From the point of view of the tools, this is achieved by the tools’ ignorance of the concrete materials they work upon. They only know their respective aspects. From the point of view of the materials, no assumptions have to be made about the tool, in particular none about how materials are presented or handled at the user interface. This independence is achieved by the materials having to implement only what is specified by their aspect classes.

If a common understanding of an aspect class has been reached in a software team, tools and materials can be developed by different groups or persons. Aspect classes can thus serve as a basis for cooperation and separation of work within in a software team.

Compare Such classes of characteristics are known as Interface Classes [CCH+89]. According to [WJ90], they also represent responsibilities.

Tool Composition — Design Pattern

Purpose Compose tools from independent subtools according to a task – subtask division to allow for maximum reuse.

Problem Complex tools often consist of similar parts. We will frequently find elementary tasks that can be captured as basic building blocks. Examples are the lister and editor just presented. We wish to reuse these building blocks and thus need a guideline for building complex tools from simpler ones.

Context We have already said that the calendar consists of simpler tools, i.e. the lister and the editor. A composition guideline for tools will have a major impact on the tools presentation and handling; therefore it must adequately relate to the users' understanding of the tools and the working tasks.

Solution When possible, we build a new tool using available tools. These component tools are called subtools. As each tool realizes a well-defined task, the decision which tool to reuse and how to embed it must conform to the overall task of the tool. Thus, each subtool's task must be a subtask of the embedding tool's task.

The calendar displays materials in a list and allows editing of a selected material. It embeds the lister and the editor as subtools, because each subtool realizes a subtask, i.e. listing and editing, of the calendar's overall task. In this example the calendar's task is simply to connect both subtools in order to realize the way we want to work with a calendar: After an item is selected from the lister, the editor must be informed and receive the new selected material (fig. 6).

Tools can be either *simple* or *compound*. Simple tools are self-contained; they realize their task without the help of other tools. Compound tools, in contrast, rely on other tools to achieve their tasks. Lister and editor are simple tools, the calendar is a compound tool.

Every compound tool embeds some subtools that are arranged according to the so-called principle of delegation. This principle divides tasks into subtasks which can be performed with minimal information about their context. The result of a subtask is integrated by the overall task that provides the context for interpretation. Accordingly, a compound tool is called the *context tool* for its subtools.

A context tool creates its subtools, delegates and coordinates subtasks and deletes subtools on demand. It integrates the results of its subtools in order to provide its own result. Any subtool may perform its task by becoming a context tool for the subtools it creates. The resulting object structure is a tree of tools (fig. 7).

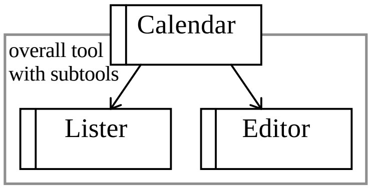


Fig. 6: The calendar is a compound tool built from the two simple tools lister and editor.

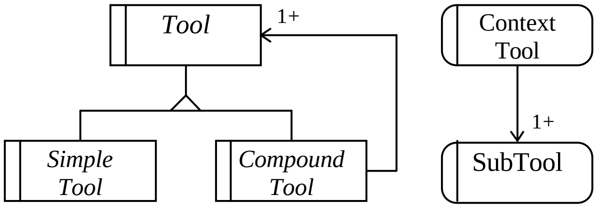


Fig. 7: Class (left) and object diagram (right) for tool composition. Each context tool may have 1..n subtools.

This structure of tools and subtools should not be confused with functional decomposition of e.g. structured design. Normally, each tool is visible to and thus accessible by the user. No predefined control flow from the root to the leaves of the tool hierarchy is determined, but the user may interact freely with any tool.

Tools are not a collection of functions but have a state of their own distinct from their materials state. A tool's state captures the way a material is currently used and thus preserves vital information for the user.

Comment Though each tool can work on its own material, frequently a subtool will work on the same material as its context tool, but with fewer aspect classes. The calendar uses the aspect classes

Listable and Editable in order to supply the lister and editor with materials. Each subtool needs its respective aspect class to work properly. The calendar, however, needs both aspect classes combined in order to be able to pass materials between the subtools. Thus, aspect classes can be composed to build complex aspect classes to satisfy complex tool requirements.

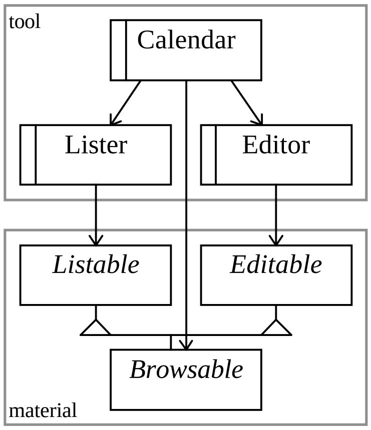


Fig. 8: Full class diagram for the calendar, its subtools and the simple and complex aspect classes.

Figure 8 shows the complex aspect class Browsable as a composition of Listable and Editable. The appointment class will then inherit from Browsable. The introduction of complex aspect classes prevents complex tools from being tailored to specific materials. The reason is the same as with normal aspect classes. Tools build a tool – subtool hierarchy and so do complex aspect classes.

Separation of Powers Design Pattern

Purpose Divide a tool into an interaction and a functional part to separate handling and presentation from functionality. This facilitates tools to be adapted to changing requirements.

Problem Conceptually, any tool can be divided into two parts: one that deals with presentation and handling of materials and one that provides the necessary functionality for manipulating materials. We wish to use this distinction, also known as separation of interaction from function, when building tools.

Context Each tool has to provide both a user interface for presentation and handling and the functionality reflecting the task supported by that tool. Assuming a graphical user interface it should present the various features of the tool as well as a tool and aspect specific view of the material. The user interface has to offer reactive behavior and as little sequencing of activities as possible. It should provide modeless interaction. The functionality of each tool should be closely related to a task or subtask within the application domain.

Solution The general technical solution of separating functionality from presentation and handling has been introduced by the MVC paradigm of Smalltalk. Adapting the paradigm to our approach we compose a tool out of one or more interaction parts (IP) and exactly one functional part (FP). The interaction part manages user actions and presents materials, thereby allowing for complex interactive handling. It translates user actions either into a mere change of presentation at the interface or into calls of the functional part. The functional part interprets all actions that examine or manipulate the materials at hand. It knows the aspect classes of its materials and incorporates the application-oriented knowledge of the tool. The interaction part should be replaceable without affecting the functional part.

The lister can be divided into two objects implementing the interaction part and the functional part. The interaction part uses graphical user interface elements to present a list and to receive notification about a selection from the list. It transforms this notification into a call to a selection operation of the functional part which is parameterized with an index.

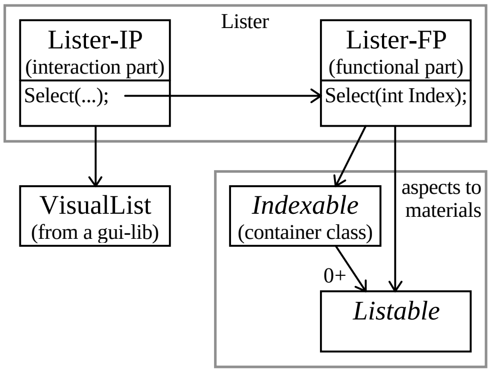


Fig. 9: The lister tool consisting of an interaction and a functional part.

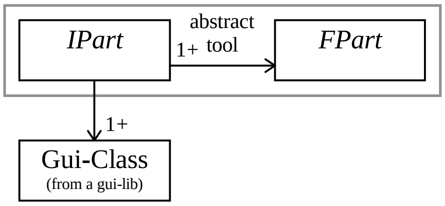


Fig. 10: General class diagram for a tool with relation to GUI objects.

Compared to the MVC paradigm the interaction part of a tool combines the responsibilities of View and Controller. The tasks of the Model on the other hand are divided between the functional part, the aspects and the material. So the Model combines material behavior as seen from the special viewpoint of an aspect with the tool's functionality of working with materials. We believe that this extension of the Model concept is one of the major achievements of the tools and materials dichotomy compared to MVC.

Compare See also MVC [KP88], ALV [Hil92] and CommonInteract [SP93] as well as [DHM89] about the basic concepts.

Event Mechanism Design Pattern

Purpose Provide a mechanism that automatically updates the dependencies of IP's and Context-FP's on their FP's state and also preserves the hierarchy between the components.

Problem If the state of a material or the respective FP changes, the FP's IP and its Context-FP have to be informed about these changes. This has to be done anonymously in order to avoid making FPs dependent on specific IPs or Context-FPs.

Context Any FP should make as little assumptions about its use context as possible. Thus, we ensure maximum reusability. This would be easy if it were sufficient to restrict the IP and Context-FP to using the FP. But in reactive and complex interactive systems it is not feasible to permanently ask the FP (i.e. poll) about changes. Thus, the FP has to take action and notify its IP and Context-FP about relevant changes.

This means that in case of relevant changes, the FP has to issue an *event to notify its observers*. An event is an announcement from the FP to its observers about a change and its kind that has happened. Observers are the IP and Context-FP which look upon the FP. They receive an event from the FP, their observed object.

If, for example, the user selects an item from the lister's visual list, the lister's FP is informed about this through the invocation of Select() as given in fig. 12. It then announces an event

ItemSelected which is received by its observer, the Calendar-FP. In the following, the Editor-FP will receive the new selected appointment to be edited. It will then issue the event TextHasChanged that is received by its IP which in turn updates its display.

It is important that the observed object knows its observers only anonymously. Otherwise, the observed object could not be used with other observers than those it has been tailored for.

Solution In our solution events are first class objects, observers can register to. Each observable object, i.e. IP or FP, offers these events in its interface. Observers register by passing an anonymous reference to the event and an operation to be called.

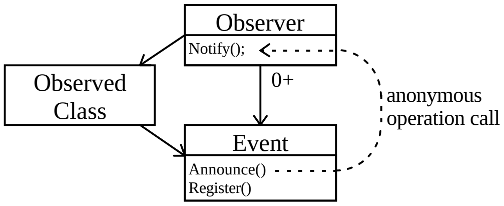


Fig. 11: Class diagram for decoupled observer and observed objects.

```
class ListerFP : public FPart {
  class Event {
    void Register( ... );
    void Unregister( ... );
    void Announce( FPart*, Listable* );
  } ItemSelected;
  void Select( int Index ) {
    CurrentItem = Container->Get( Index );
    ItemSelected.Announce( this, CurrentItem );
  };
};
```

Fig. 12: The ItemSelected event in the lister’s interface is available to clients.

The observed object decides when to announce an event according to changes of its state. It then calls the Announce Operation of an event which in turn calls the operations which have been passed to it by its observers.

The parameters for Announce() will differ from event to event. Usually 2 to 3 parameters will suffice to inform the observers about specific changes. The first parameter is almost always a reference to the observed object itself; otherwise, the observer has to find out which of its observed objects has announced the event itself.

An observer should react to an event only through probing operations and should never modify the observed object. This avoids the danger of entering into an infinite Event-Change-loop.

Compare Change-update as in Smalltalk [GR83] or Observer/Subject in [GHJ+94], implicit invocation [NGG+93] or callback as in various windowing systems.

IP/FP Plug In — Design Pattern

Purpose Reconcile tool composition with tool construction by plugging in Sub-IPs into Context-IPs and Sub-FPs into Context-FPs.

Problem Tools are structured vertically by composition out of subtools and horizontally by separating interaction from function. If we wish to dynamically create a subtool, this leads to two diverging forces that have to be reconciled on the object level.

Context Any tool consists of at least one interaction and one functional part. Creating a subtool poses the question of how to connect the subtool‘s IPs and FP to the IPs and FP of the context tool.

Calls and events have to be routed in a disciplined way in order to keep the structure and dynamics of complex tools clear and understandable. This means that a technique for dynamically plugging subtools into context tools as well as putting interaction parts on functional parts has to be provided. Creation and deletion have to be considered, too.

Solution First, we show the static structure of class relationships. For the simple case of the calendar with its lister subtool the structure of fig. 13 results.

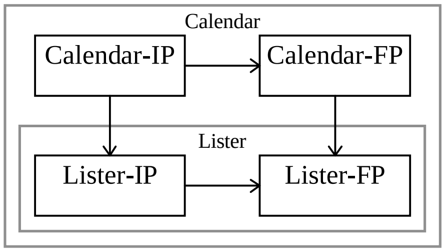


Fig. 13: IP and FP coupling between calendar and lister (context and subtool).

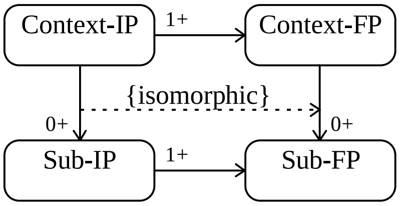


Fig. 14: General object diagram between IPs and FPs of context and subtool.

The calendar's IP works on the lister's IP and the calendar's FP works on the lister's FP. Additionally, each IP works on its FP. Each relationship in the figure indicates also an observation of the used object. So, the Lister-FP is observed by the Lister-IP and by the Calendar-FP.

While notifying an IP about relevant changes usually means that the display has to be updated, notifying the FP of an embedding context tool means that something relevant to the subtool's task has happened. If a user selects an item of the list, the lister-FP will notify the calendar-FP, which in turn takes appropriate action (by handing over the newly selected appointment to the editor).

The general structure is more complicated, because a tool may have more than one interaction part. The FP of the context tool, the *Context-FP*, accesses the FPs of its subtools, the *Sub-FPs*. Every IP of the context tool (*Context-IP*) may put several *Sub-IPs* onto each Sub-FP. The Sub-IPs work on their Sub-FP like the Context-IPs work on their Context-FP.

Again, each relationship means using as well as observing the used object. The IP is mainly interested in events that deal with presentation. The Context-FP of an FP is normally interested in logical issues related to the material or the tool's state.

We will now look at the dynamics. The decision to create a subtool is made by the Context-FP. It simply creates the Sub-FP and announces an appropriate event. Informed by this event, the IPs of the Context-FP decide which objects for the Sub-IPs are needed for the new Sub-FP and therefore have to be created. Thus, each Context-IP creates zero, one or more Sub-IP objects for every new Sub-FP. The Context-IP introduces the new Sub-FP to its newly created Sub-IPs. The code of fig. 16 shows this interplay between the calendar's IP and FP while creating the lister subtool.

Whenever a Sub-FP has to be deleted, the Context-IP deletes all Sub-IPs belonging to this Sub-FP. Then, the Sub-FP will be deleted by its Context-FP.

4.3 Application of design patterns for a single tool

How do we apply the different patterns in a uniform way like using a coherent language? Looking at fig. 15, we see a *calendar tool that is composed out of the subtools lister and editor both working on materials via the aspect classes Listable and Editable, respectively*.

The Lister-FP and the Editor-FP have been plugged into the Calendar-FP as well as its IPs. The figure shows that each tool is constructed out of an IP and a FP with the FP notifying its IP about relevant changes of its state. On closer examination, we actually have not just built a calendar but a general browser tool that works with any listable and editable material. We could enhance this design by adding another aspect class FormEditable for the structured editing of materials using fields of specific data types.

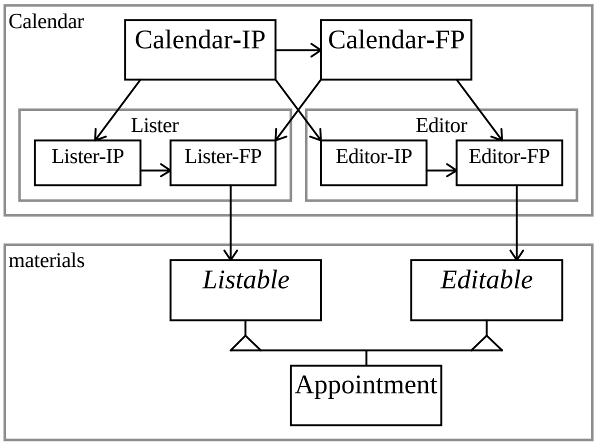


Fig. 15: Detailed class diagram for the calendar tool and the appointment with its aspect classes.

```
class CalendarFP : public ComplexFP {
  class Event {
    Announce( FPart* SubFP };
  } SubFPCreated;
  void CreateSubFPs() {
    ListerFP = new ListerFP( this );
    SubFPCreated.Announce( ListerFP );
  };
};
class CalendarIP : public ComplexIP {
  void CreateSubIP( FPart* NewSubFP ) {
    if ( NewSubFP->IsA( ListerFP ) ) {
      ListerIP = new ListerIP( NewSubFP );
    }
  };
  CalendarIP( CalendarFP* MyFP ) {
    MyFP->SubFPCreated.Register( CreateSubIP );
  };
};
```

Fig. 16: C++ code showing the interplay between events and operation calls for creating a IP for a given FP.

5 Tool integration

Next, we will combine the calendar with a second tool. The additional patterns again have to conform to the metaphors of tool, material and environment.

5.1 Extending the example

Fig. 17 shows a second tool which we want use together with the calendar. It is a scheduler for a whole week that presents all periodic dates and allows to directly manipulate them. While the calendar shows the individual appointments, the scheduler will only show dates like group meetings and seminars. The materials of the scheduler are a TimeTable object based on WeeklyDate objects. WeeklyDate objects are distinct from Appointments as they have no fixed date.

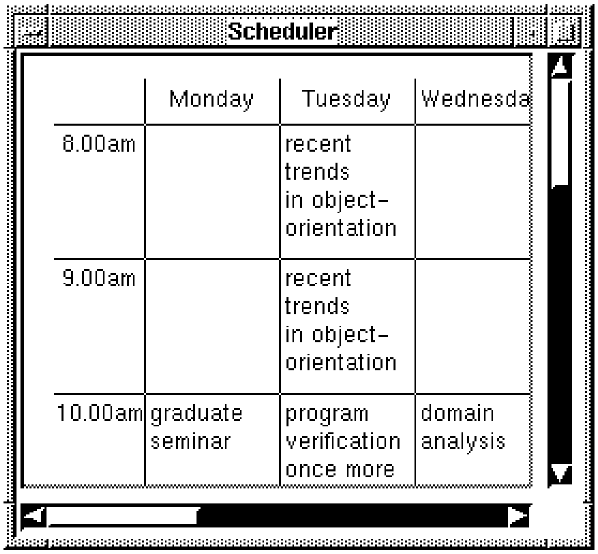
In fig. 18 we simplify both tool structures. Thereby we can concentrate on the problems of tool integration. The scheduler has a TimeTable containing WeeklyDate objects and the calendar has an AppBook (appointment book) containing Appointments.

Obviously, weekly dates can clash with individual appointments. In order to focus the discussion, a date clash will be expressed by a boolean flag Conflicts of both WeeklyDate and Appointment objects. If the flag is set for a WeeklyDate object, a conflicting individual appointment exists. If the flag is set in an Appointment object, there are overlapping periodic dates. The flag is necessary for the tools to signal a date clash to the user.

5.2 Design patterns for tool integration

Designing tools there is no need for a supertool controlling all other tools, but a fair selection of interrelated but independent tools ready at hand for their users. Dependencies may exist among materials, not tools.

We collect dependent materials into a single Material Container that maintains constraints among these materials. We use tool coordinators to propagate change notifications among interrelated tools. Materials are retrieved from the Material Administration which is a kind of Object Request Broker. Finally, we realize the workplace’s closure through an Environment object.



	Monday	Tuesday	Wednesda
8.00am		recent trends in object-orientation	
9.00am		recent trends in object-orientation	
10.00am	graduate seminar	program verification once more	domain analysis

Fig. 17: A scheduler (tool) extending our example to examine integration problems.

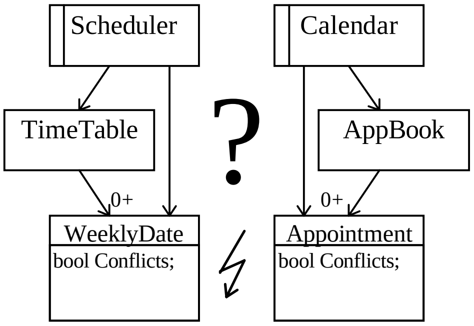


Fig. 18: Class diagram of the two tools and their materials and an integration relationship.

Material Container Design Pattern

Purpose Group dependent materials into a single container acting as a closure, so that constraints can be maintained independently from tools and in one place.

Problem Different materials are often related and mutually dependent. Maintaining such constraints has to be independent of tools in order to allow for easy addition of new tools.

Context Materials like WeeklyDate and Appointment depend on each other; this has to be expressed and maintained as constraints. For example, adding a new appointment leads to checks and updates of the Conflicts flag of the WeeklyDate and Appointment objects.

Adding a new tool to our time planning system that works on the same materials should not make maintenance of constraints more difficult. If the tools were responsible for maintaining material constraints, these constraints would have to be re-implemented for each new tool. Each tool then would have to know any related materials – an undesirable situation. Thus, constraints have to be maintained independently of tools. All constraints will be localized in a single place.

The notion of constraints can be captured formally, e.g. as mathematical equations relating object attributes. Our experience shows that dependencies often have structural implications on materials that force rearrangements of object relationships. For the time being, we implement constraints within standard programming languages, but as constraint languages are becoming more popular this may change in future.

The constraints treated here take immediate effect. They are restricted to a single workplace. We call them short-term constraints.

Solution We enclose all materials that are mutually dependent through short-term constraints into a single container called a *material container*. This container provides an Update() operation which is called by a tool, if a relevant change has taken place (a stronger solution controls each access to the materials of a container). The container in turn triggers a constraint object which it supplies with the changed material.

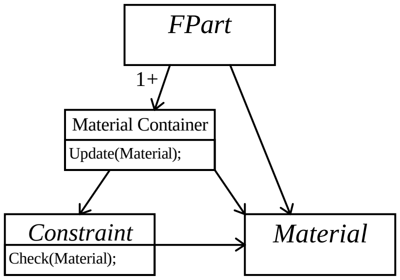


Fig. 19: The material container hides the constraints between materials and thus localizes the dependencies.

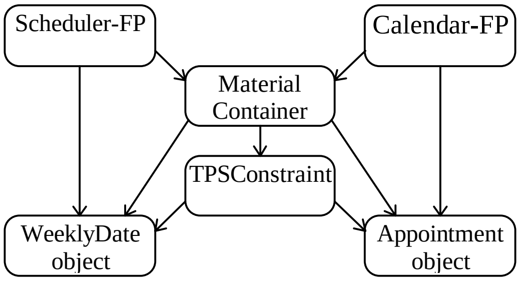


Fig. 20: Our example has two materials (WeeklyDate and Appointment) that are constrained by the TPSConstraint.

In our example, WeeklyDate and Appointment are subclasses of Material. A tailored constraint has to be written which takes care of these specific materials, called TPSConstraint (time planning system constraint). The following object diagram shows the result.

The TimeTable and AppBook objects have been omitted for clarity. The TPSConstraint will ask them for WeeklyDate and Appointment objects. It can be seen as a strategy class encapsulating an algorithm for constraint maintenance.

Introducing a new tool will cause no further changes as constraints are taken care of in the TPSConstraint class independently of tools. If a new material is added to the time planning system only the integrating relationship realized through the TPSConstraint has to be updated.

Tool Coordinator — Design Pattern

Purpose Notify tools about changes to their materials due to constraints.

Problem If a constraint changes a material's state, the tool's state and the material's visual presentation can become inconsistent and thus have to be updated.

Context Each tool whose material is changed by a third party needs to be informed about this change. As each container may hold several materials, usually more than one tool will be affected. Entering a new weekly date might lead to clashes with several appointments. Thus, both tools, scheduler and calendar are affected and have to be informed.

A number of possible solutions come to mind, all based on the notification mechanism. Each tool can observe its materials, the constraint or the container that in turn will notify it about changes.

We feel that using the notification mechanism this way is highly problematic. Notification should be used as sparsely as possible. Our experience shows that otherwise the system's architecture and dynamics become harder to understand.

Solution For each material container we create a *tool coordinator*. This is an object that observes all functional parts working on materials in the container. If a functional part changes a material, it will issue an Update event to inform its IP or Context-FP. Additionally, this event is received by the tool coordinator.

The tool coordinator requests a list of materials that were affected by the last manipulation from the container. From this list and its internal dispatch tables, the tool coordinator derives which FPs' Update operations have to be called.

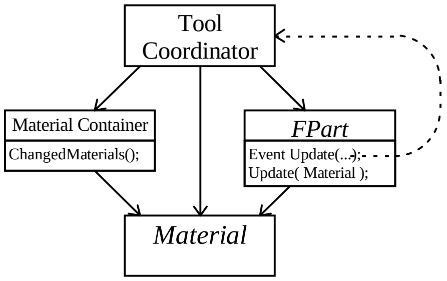


Fig. 21: The Tool Coordinator dispatches an event received from an FP to the FPs of other tools.

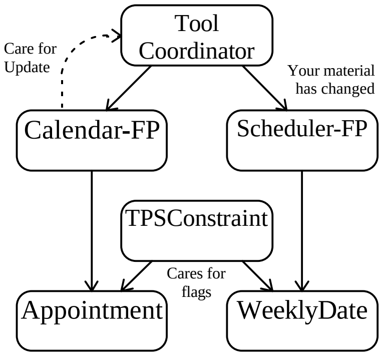


Fig. 22: The runtime relationships between the participating objects and the resulting control flow.

The dispatch tables can be built efficiently. The tool coordinator knows all tools working on materials in the container. Each tool offers a list of its FPs under their abstract superclass FPart. Each FPart object offers the material it works on under the abstract superclass Material. They are held in a dictionary. The tool coordinator uses the list of affected materials it gets from the container and the dictionary in order to decide which FP has to be informed. With these superclasses, the tool coordinator can be built independently of any specific material or tool.

Compare Compare our concept of tool coordinators and material containers with the widespread notion of mediators as e.g. in [SN92] or [GHJ+94].

5.3 Outlook: The system’s boundaries

We end the presentation of our pattern language by giving a brief outlook on two more design patterns. The patterns of *Material Administration* and *Environment object* define the system boundaries necessary for integrating the patterns into a work environment.

Material Administration We need a Material Administration for retrieving and storing materials, for controlling access to originals and copies and for grouping materials into material containers. The Material Administration subsystem is accessible by any tool. As the result of a request, tools receive iterators on a set of materials conforming to the query.

The Material Administration works with several material providers, each of them encapsulating a database service, e.g. an OO-DBMS or a RDBMS. In addition, non-persistent material providers may be used.

The Material Administration may be seen as a combination of an Object Request Broker (ORB) and a Portable Common Tool Environment (PCTE) enhanced to fit our specific needs.

Environment object The Environment object sets up the whole system. It shows accessible tools and materials on a desktop. For each new material container it creates the corresponding tool coordinator and takes care of technical issues during initialization like providing screen and database services.

The Environment object is the first object to be created. After system startup, it creates material providers, each of them encapsulating a database service and the material administration which receives the material providers. After this, it opens the desktop and waits for users to launch a tool.

6 Tools and Materials at work

We will now give a short but complete example in order to see the pattern language at work. We will design a system for *task oriented requirements analysis* (TORA) [Kei87], which is used at the University of Hamburg for teaching purposes. It consists of the graphical editor Sane for interactive manipulation of *activity net* materials and a glossary browser for textually documenting the objects of the editor.

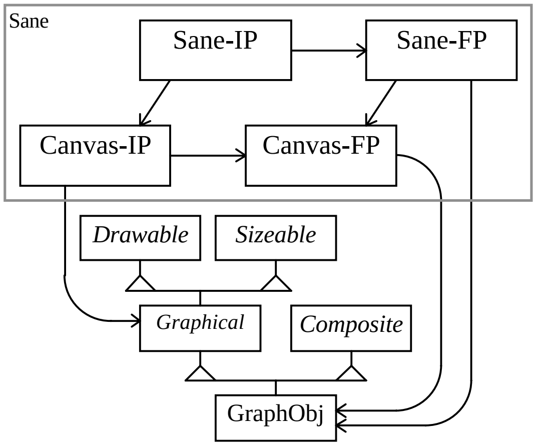


Fig. 23: The Sane (tool) and its materials. The materials' interface heavily relies on aspect classes separating different functionality.

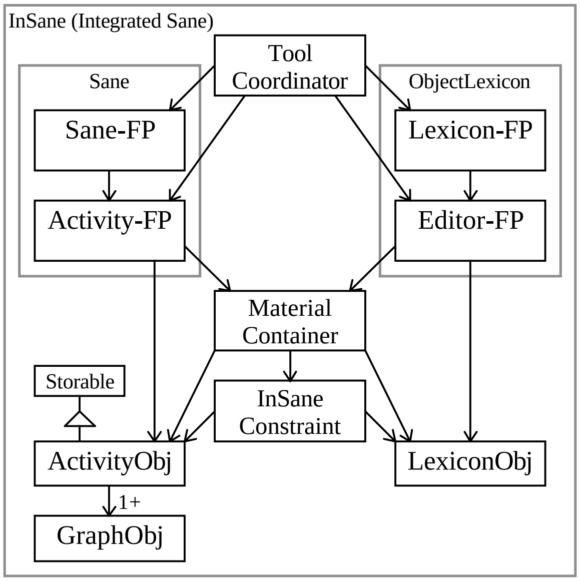


Fig. 24: The integrated system shows that tools are tied together at the top (Tool Coordinator) and at the bottom (Material Container).

The *tool* Sane *consists of a compound tool with two subtools*. Each tool has an *interaction and a functional part*. The *compound tool* establishes the frame for the usual services of an application like file handling, clipboard access etc. The *Canvas subtool* allows for direct manipulation of the *editor's materials*, which are graphical objects (GraphObjs). The *ActivityNet subtool* works with the *logical materials ActivityObj*. Depending on the number of contexts it is presented in, an ActivityObj has one or more graphical presentations through GraphObjs.

Access to these *materials is mediated through aspect classes establishing the context of use.* They provide an approach for working with graphical objects that differs from what we find in most editors. The *main tool works only with the aspect class Storable* to store and retrieve objects from files or the clipboard. The *Canvas-IP* uses drag and resize wrappers to *manipulate the graphical objects via its aspect classes Drawable and Sizeable.* More complex functionality requires knowledge of the *Composite structure of ActivityObjs* introduced through the *Composite aspect class* available to the *Canvas-FP through the complex Graphical aspect class*.

Thinking in terms of aspect (classes) helps very much to separate different functionality from each other and makes the evaluation of the necessary functionality easier.

Each logical ActivityObj can be documented using the *tool ObjectLexicon*. For each ActivityObj there exists a LexiconObj which users textually edit to document the ActivityObj and its GraphObj. *Both materials are kept inside a Material Container which uses the InSane Constraint* that e.g. ensures that the label of the ActivityObj is always the same as that of the LexiconObj.

If an ActivityObj is created or deleted or its label is changed the *tool coordinator will be notified* and in turn *informs the related tool*.

7 Discussion of the Tools and Materials Metaphor

The application domain of the Tools and Materials Metaphor are environments that fit naturally with the notion of tools and materials used by skilled human workers. It focuses on the work of people in workshops and offices. The individual craftsman, the software developer as well as the office worker fall into this category and may be supported adequately by a software system designed according to the Tools and Materials Metaphor.

To complete the Tools and Materials Metaphor, additional metaphors like the automaton and the material administration for embedding databases address other issues that are not discussed in this paper but are needed in every major project.

When it comes to cooperative work, however, these metaphors have to be extended – a topic currently under discussion. For office work a possible solution seems to be obvious, namely introducing mailboxes for incoming and outgoing materials - a concept that used in offices for many decades. Similar ideas are currently discussed with different outcomes under the label of workflow management.

Still more ambitious tasks remain to be solved. Close and intense cooperation on an electronic whiteboard, working on commonly shared artifacts in parallel are tasks that are not fully addressed by the Tools and Materials Metaphor by now. Nevertheless, we believe to have a fruitful starting point that can be developed further towards more and advanced metaphors which either integrate or compete with communication and media metaphors or agents [MO92, Mae94].

8 Outlook

The pattern language presented in this paper will be used to teach the Tools and Materials Metaphor to both professional developers and students. We want to transfer the experience from the application framework to industrial projects. In the industrial settings we will carefully analyze how the metaphors and implementation techniques of our pattern language enhance comprehensibility of the overall approach, the communication in a team and the resulting design quality.

Acknowledgments

This paper was written in close contact with Karl-Heinz Sylla at GMD St. Augustin, who proposed essential changes and amendments. We wish to thank our reviewer Bruce Anderson for his substantial help in making this paper readable and Ralph Johnson for inspiring comments on early drafts. The paper profited a lot from the discussions in the writer’s workshop at PLoP-94; thus, we wish to thank all those who by commenting helped improving it.

Bibliography

BCS92 Reinhard Budde, Marie-Luise Christ-Neumann and Karl-Heinz Sylla. “Tools And Materials: An Analysis and Design Metaphor”. Tools-7, *Technology of Object-Oriented Languages and Systems, Europe-92*. Edited by G. Heeg, B. Magnusson and B. Meyer. Prentice-Hall, 1992. 135-146.

BKK+92 Reinhard Budde, Karl-Heinz Kautz, Karin Kuhlenkamp and Heinz Züllighoven. *Prototyping*. Berlin, Heidelberg: Springer Verlag, 1992.

BZ92 Reinhard Budde and Heinz Züllighoven. “Software Tools in a Programming Workshop”. *Software Development and Reality Construction*. Edited by Christiane Floyd, Heinz Züllighoven, Reinhard Budde and Reinhard Keil-Slawik. Berlin, Heidelberg: Springer-Verlag, 1992. 252-268.

CCH+89 Peter S. Canning, William R. Cook, Walter L. Hill and Walter G. Olthoff. “Interfaces for Strongly-Typed Object-Oriented Programming”. OOPSLA-89, *ACM SigPlan Notices* 24, 10 (October 1989): 457-467.

Cun95 Ward Cunningham. “The CHECKS Pattern-Language of Information Integrity”. *This Volume*.

DHM89 Mahesh H. Dodani, Charles E. Hughes and J. Michael Moshell. “Separation of Powers”. *Byte* (März 1989): 255-262.

DWA93 Wolfgang Dzida, Marion Wiethoff and Albert G. Arnold. *ERGOguide – The Quality Assurance Guide to Ergonomic Software*. GMD, Schloß Birlinghoven, Germany, 1993.

Flo84 Christiane Floyd. “A Systematic Look at Prototyping”. *Approaches to Prototyping*. Edited by Reinhard Budde, Karin Kuhlenkamp, Lars Matthiassen and Heinz Züllighoven. Berlin, Heidelberg: Springer-Verlag, 1984.

GHJ+93 Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. “Design Patterns: Abstraction and Reuse of Object-Oriented Design”. ECOOP-93, *Lecture Notes in Computer Science No. 707*, 1993. 406-431.

GHJ+94 Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1994.

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

Hil92 Ralph D. Hill. “The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications”. CHI-92, *SIGCHI Conference Proceedings*. Edited by Penny Bauersfeld, John Bennet and Gene Lynch. Reading, Massachusetts: Addison-Wesley, 1992. 335-342.

Joh92 Ralph E. Johnson. “Documenting Frameworks using Patterns”. OOPSLA-92, *ACM SigPlan Notices* 27, 10 (October 1992): 63-70.

Kei87 Reinhard Keil-Slawik. “Supporting Participative Systems Development: Task-Oriented Requirements Analysis”. *System Design for Human Development and Productivity: Participation and Beyond*. Edited by Klaus Fuchs-Kittowsky and D. Gertenbach. Berlin, DDR: Akademie der Wissenschaften der DDR, 1987.

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.

Mae94 Pattie Maes. “Agents that Reduce Work and Information Overload”. *Communications of the ACM* 37, 7 (July 1994): 31-41.

Mey91 Bertrand Meyer. “Design by Contract”. *Advances in Object-Oriented Software Engineering*. Edited by Dino Mandrioli und Bertrand Meyer. London: Prentice-Hall, 1991. 1-50.

MO92 Susanne Maaß and Heinz Oberquelle. “Perspectives and Metaphors for Human-Computer Interaction”. *Software Development and Reality Construction*. Edited by Christiane Floyd, Heinz Züllighoven, Reinhard Budde and Reinhard Keil-Slawik. Berlin, Heidelberg: Springer-Verlag, 1992. 233-251.

NGG+93 David Notkin, David Garlan, William G. Griswold and Kevin Sullivan. “Adding Implicit Invocation to Languages: Three Approaches”. JSSST-93, LNCS-742, *Object Technology for Advanced Software*. Edited by Shojiro Nishio and Akinori Yonezawa. New York: Springer-Verlag, 1993. 489-510.

Pir74 Robert M. Pirsig. *Zen and the Art of Motorcycle Maintenance*. London: Corgi Books, 1974.

RBP+91 James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen. *Object-Oriented Modeling and Design*. London: Prentice-Hall, 1991.

Sha95 Mary Shaw. “Patterns for Software Architecture”. *This Volume*.

SP93 Bernhard Strassl and Franz Penz. “CommonInteract: An Object-Oriented Architecture for Portable Direct Manipulative User Interfaces”. *Journal of Object-Oriented Programming* 6, 3 (June 1993): 33-39.

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.

WJ90 Rebecca Wirfs-Brock and Ralph E. Johnson. “Surveying Current Research in Object-Oriented Design”. *Communications of the ACM* 33, 9 (September 1990): 104-124.