A Pattern Language for Tool Construction and Integration Based on the Tools&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 those 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&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&Materials Metaphor is an approach with an underlying view of software development and 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 out that software developed with this goal in mind makes people feel comfortable in using the system, which improves 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&Materials Metaphor characterizes our overall approach. We express the underlying idea by the four design-metaphors *Tool*, *Material*, *Aspect* and *Environment*, each representing a different but related concept. We present several design-patterns for the implementation of the metaphors and we show how we 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 division 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 professionals (called users in this paper) for whom we develop software. They provide a starting point for discussion between users and software professionals (called developers), because most people have a sufficient understanding of what tools and materials are.

When we actually develop software in projects, 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 among all parties concerned; thus, there are various tasks to do: 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 and extend it by new concepts necessary for the envisioned system. We make heavy use of scenarios of current work and system visions about work with the future system.

We have followed this approach successfully in commercial and industrial projects.

To lay a thorough foundation for further research, one of the authors designed and implemented an application framework for the Tools&Materials Metaphor.

Its purpose was to enhance and improve the implementation techniques used in the industrial projects. After the application framework reached maturity and was heavily used, he reinterpreted the framework as a pattern-language which laid the foundations for this paper.

In the following we'll present the design-metaphors as well as the design-patterns from the framework as a pattern-language. We present the patterns for tool construction and integration.

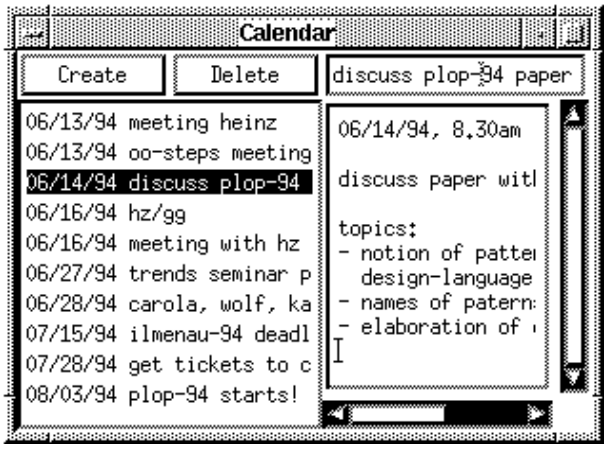
3 The Metaphors

The following description of the Tools&Materials Metaphor may be regarded as high-level analysis and design-patterns. As we could not structure our description into a problem-context-solution pattern we use simple prose.

Materials

Materials are those objects of the application domain that we work upon. They 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, like the different forms laid out systematically on a desktop, ready for the work to be done. During our work, we mainly focus on materials and get the impression of directly accessing and manipulating them. Though we use tools like a typewriter or a pen to fill a form, these tools seem to disappear and only the form is in our focus. We often use our hands as tools for working with tangible materials. But in software systems we can never access materials "directly" but only by using appropriate tools. In order to maintain a 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.



As an example, we will design a small time planning system (TPS) to keep track of our dates, appointments, seminars and so on. The figure presents our first tool, a simple calendar. It consists of a list of appointments on the left and a selected appointment on the right. The list of appointments and the annotation of an appointment are both materials. The whole calendar is a tool that shows the materials.

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

Tools are means of work. Often grown in 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 their current 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.

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 never "disappear" in a work situation. In unclear or erroneous situations, users need to look at a tool itself.

The distinction between tools and materials 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 other tools in an erroneous or unclear state. Later on 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 textually editing an appointment, an Editor.

We normally use many different tools for working upon one material. So, besides the calendar there will a tool to give an overview of our weekly dates and other tools to give statistics about appointments.

Aspects

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. To make this explicit, we introduce the notion of *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 away from any specific materials. The appointments for the calendar must be listable for the task of listing them and they must be editable for the task of editing them. The operations necessary to perform these both tasks are expressed through two different aspects respectively. interfaces: *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.

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**

We perform tasks in a work environment, e.g. in a workshop or on a desktop. Within this environment, we physically and mentally organize 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 mostly do this in an implicit and intuitive way, based on our qualification and experience. Transferring the underlying concepts of environment and location to a software system is important as it provides us with the means to organize our work. For office-work, a good concept is the computerized 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 establishing integration relationships between them.

4 Tool Construction

First, we will look at a single tool and provide patterns for its construction based on our notions of tools, materials and aspects.

We will talk about separating *tools* from *materials* through *aspect classes* and about tools *being composed* from *subtools*. Any *tool* is *constructed* out of an *interaction* and a *functional part* which provide the means for *plugging in* other *tools* as *subtools* and for *being plugged in* as a *subtool itself*.

Couple Tools with Materials through Aspects

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 tools are developed to work with their specific materials. But we wish to develop reusable tools that are not tailored to specific materials. Several tools should work on the same material and one tool should work on several materials.

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

```
// 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;
};
```

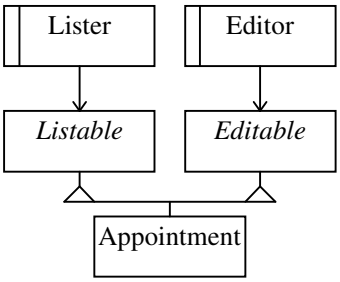
A *Lister(-tool)* visually displays items from a container as a list that can be selected from. An *Editor(-tool)* provides the functionality for textually editing materials. To work properly, both tools need the functionality of a certain aspect, expressed through Listable or Editable respectively. The appointment objects for the calendar will have to offer both aspects, because they can be listed and edited.

Thus, all different aspects have eventually to be part of the material's interface. If realized by single inheritance there will either be no hierarchy or we have to introduce precedences between the aspects. As both tools have to be independent of each other, the Lister should accept materials even if they are not editable and the Editor should accept materials even if they cannot be listed by the Lister. This cannot be achieved by single inheritance in a statically typed language.

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, they can use any material that offers the characteristics of the aspect class. The appointment

class implements the operations given in Listable and Editable, which are usually abstract classes. The single line that the appointment returns via its Listable interface may consist of a string with the date and the name of the person to meet.



A new material's interface is built through inheriting from all aspect classes that provide sensible ways of working with the material. An aspect class can be seen as a partial type.

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.

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.

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

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, it is achieved by the materials having to implement only what is specified by their aspect classes. No further assumptions have to be made about the tool, in particular none about how the material is presented or handled at the user interface. Thus, if a common understanding of an aspect class has been reached in a software team, tools and materials can be developed by different persons. Aspect classes can thus serve as a basis for cooperation and separation of work within in a software team.

We will show that tools working on aspect classes are an important step towards easy tool composition.

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

Compose Tools out of Subtools

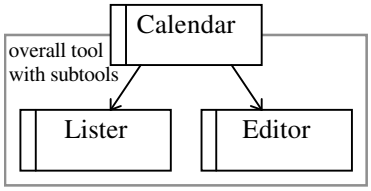
Purpose Compose tools from independent subtools according to a task – subtask division to allow for maximum reuse.

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 thus must adequately relate to the user's understanding of the tool and the working task.

Solution Whenever possible a new tool is built 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.

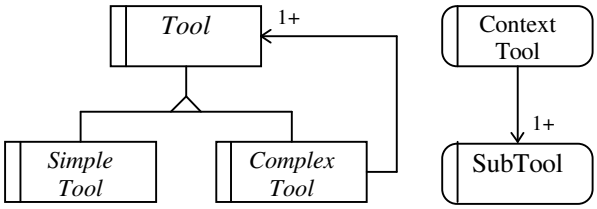


Tools can be either *simple* or *complex*. Simple tools are self-contained; they realize their task without the help of other tools. Complex tools, in contrast, rely on other tools to achieve their tasks. Lister and Editor are simple tools, the Calendar is a complex tool.

Every complex 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 a supertask that provides the context for interpretation. Accordingly, a complex tool is called the *context tool* for its subtools.

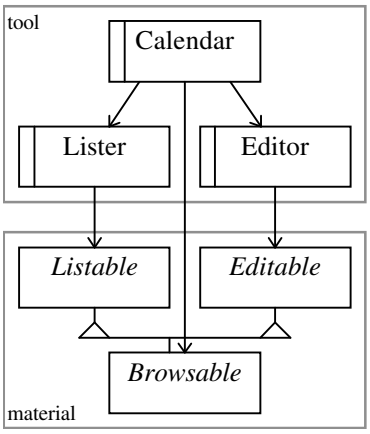
A context tool creates its subtools, delegates and coordinates subtasks and deletes subtools on demand. It integrates the results of its subtools in order to provide its own result. Any subtool may perform its task by

becoming a context tool for the subtools it creates. The resulting structure is a tree of tools that might be seen as a specialization of the Composite pattern as shown in the figure.



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.

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 has to know both the aspect classes Listable and Editable in order to supply Lister and Editor with materials. Both subtools need only their respective aspect classes to work properly. So, aspect classes can be composed to build complex aspect classes in order to satisfy complex tool requirements.



The figure shows the complex aspect class Browsable as a composition of Listable and Editable. The appointment class then would inherit from Browsable.

```
class Browsable : public Listable, Editable {
  // doesn't necessarily need new operations.
  // it is just the combination of both aspects
  // needed by a browser(-tool).
};
```

The introduction of complex aspect classes prevents complex tools from being tailored for specific materials. The reason is the same as for normal aspect classes.

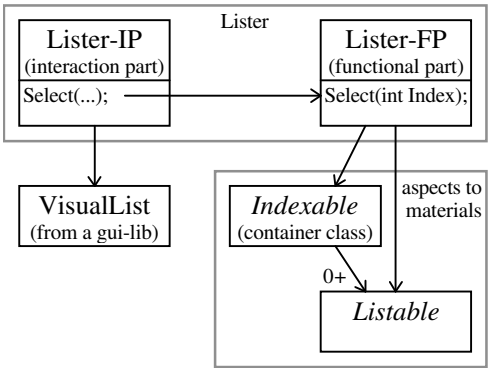
Separate Interaction from Function

Purpose Divide a tool into an interaction and a functional part which separate handling and presentation from functionality. This will allow tools to be adapted to changing requirements more easily.

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 make use of 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 as well as functionality that reflects the task supported by that tool. Assuming that the user interface is graphical it should present the various features of the tool and a tool- and aspect specific view on the material. The user interface has to offer reactive behavior and as little sequencing of activities as possible. It should enable 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. We have adapted the paradigm for the Tools&Materials Metaphor. A tool consists of one or more interaction parts (IP) and exactly one functional part (FP). The interaction part manages user actions and presents materials, thereby providing 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.

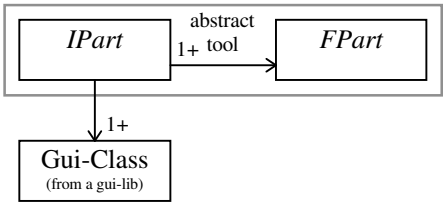
Compared to the MVC paradigm the interaction part of a tool combines the responsibilities of View and

Controller. The tasks of the Model on the other hand are divided among 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 elaboration of the Model concept is one of the major achievements of the tools and materials dichotomy compared to MVC.

Structure Each tool may have more than one interaction part, but each tool has exactly one functional part.

Any interaction part uses several interface elements. The IP coordinates these interface elements in order to realize the actual user interface for presentation and handling of a tool. The interface elements encapsulate the concrete GUI elements (widgets) of the chosen window system. The IP controls the interaction between user and tool and calls its interface elements according to the interaction semantics. It also controls the FP.

The functional part captures the tool's application semantics.



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

Provide Objects with Immediate Feedback

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.

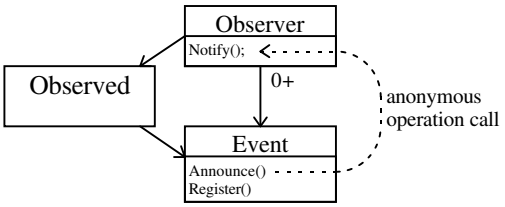
Context Any FP should make as little assumptions about its use context as possible. Thus, we ensure maximum reusability. This would be simple, if it were sufficient for the IP and Context-FP to simply use the FP. But in reactive and complex interactive systems it is not feasible to permanently ask the 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 describes what the FP wishes to tell its observers, i.e. the kind of change that has happened. Observers are the IP and Context-FP which look upon the FP. They receive an event from the FP, the observed object.

If the user selects an item from the Lister's visual list, the Lister's FP is informed about this. It then announces an event ItemSelected that 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 Our solution makes events first class objects, observers can register to. Each potentially observed object, i.e. IP or FP, offers these events in its interface. Observers register by giving an anonymous reference and an operation to be called to the event.



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 given to it by its observers. The code for our example should look like:

```
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 );
  };
};
```

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

An observer should react to an event only through probing operations and should not modify the observed object. This avoids the danger of entering an infinite Event-Change-loop.

Compare Change-update as in Smalltalk [GR83], implicit invocation [NNG+93] or callback as in various windowing systems.

Plug-In Tools via its IP's and FP's

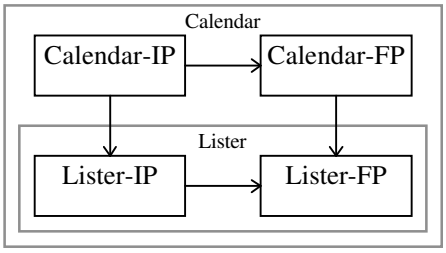
Purpose Reconcile tool composition with tool construction by plugging in Sub-IPs into Context-IPs and Sub-FPs into Context-FPs.

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 a functional part. Creating a subtool poses the question of how to connect the subtool's IPs and FP to the respective component 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.

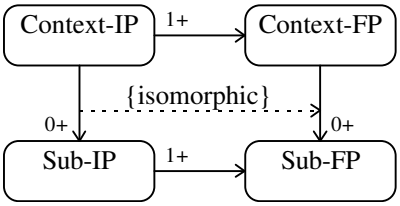
Solution First, we'll show the static structure of class relationships. For the simple case of the Calendar with its Lister subtool the following structure results.



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 means also observation of the used object. So, the Lister-FP is observed by the Lister-IP as well as 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 subtools 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.

Now, we'll 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 event about this. Informed by the 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 following code shows this interplay between the Calendar's IP and FP while creating the Lister subtool.

```
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 );
  };
};
```

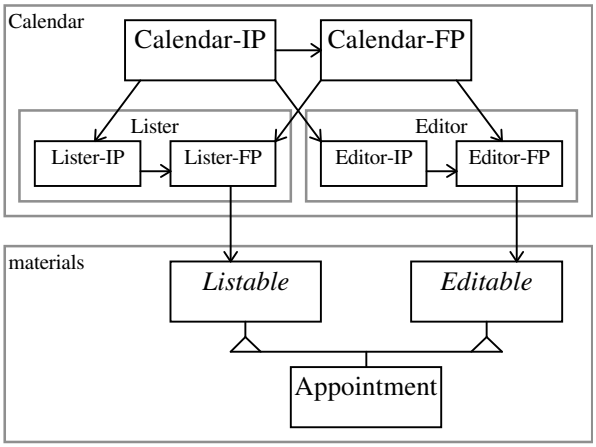
Whenever a Sub-FP has to be deleted, the Context-IP deletes all Sub-IPs belonging to this Sub-FP. Then, the Sub-FP will be deleted by its Context-FP.

Application of patterns for a single tool

How do we apply the different patterns in a uniform way like using a coherent language? Looking at the figure, 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. It is shown that each *tool* is *constructed out of an IP and a FP with the FP notifying its IP* about relevant changes of its state. We could argue that we actually have not built just a calendar but a more *general browser-tool that works with any listable and editable material*. We could add

another *aspect class* FormEditable that allows for structured editing of materials using fields of specific data types.

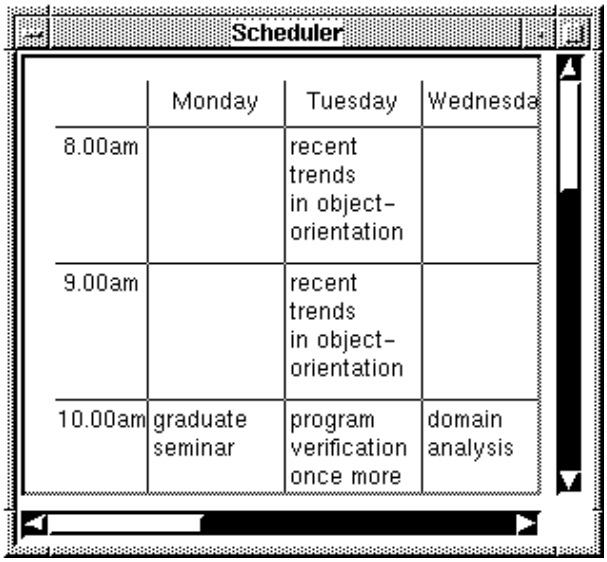


5 Tool Integration

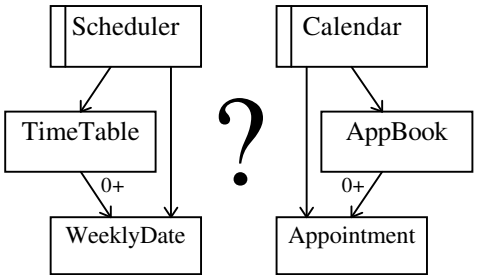
Now we want to combine the Calendar with a second tool. The patterns we provide have to conform to the metaphors of tool, material and environment.

Speaking about 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' closure* through an *Environment object*.



The above figure shows a second tool which we use together with the Calendar. It is a Scheduler for a whole week presenting all periodic dates allowing for direct manipulation. 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 the above figure 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 in a WeeklyDate object, a conflicting individual appointment exists. If the flag is set in an Appointment object, there is an overlapping periodic date.

This flag is necessary for the tools to signal a date clash to the user.

Group Dependent Materials in a Single Container

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 has got to lead to checks and updates of the Conflicts flag of the WeeklyDate and Appointment objects.

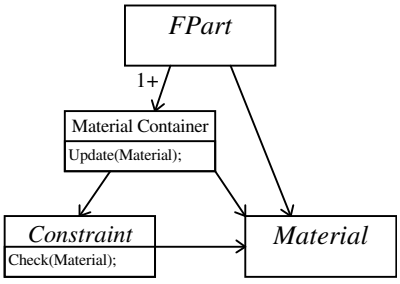
Adding a new tool to our time planning system working on the same materials should not make maintenance of constraints more difficult. If the tools were responsible for maintaining material constraints, these constraints would have to be implemented with any new tool. Each tool then would have to know any related materials – an undesirable situation. Thus, constraints have to be maintained independently of tools.

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

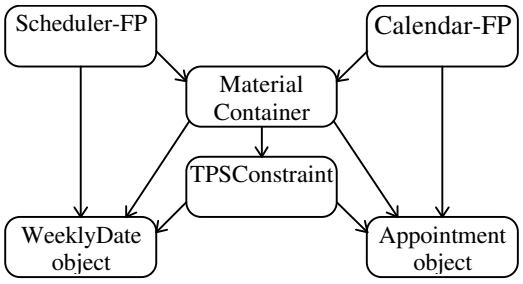
rearrangements of object relationships. By now we implement constraints within our normal programming language, but as constraint languages are becoming more popular this may change in future.

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 within the container). The Container in turn triggers a Constraint object which it parameterizes with the changed material.



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 query them for WeeklyDate and Appointment objects. It may 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 any tools. If a new material is added to the time planning system only the integrating relationship realized through the TPSConstraint has to be updated.

Dispatch Update Events between Related Tools

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

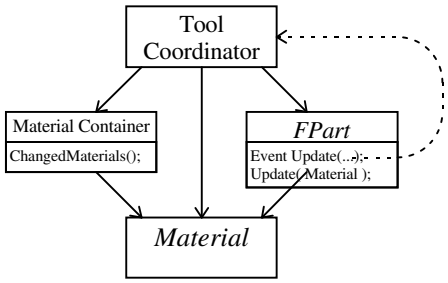
Problem If a constraint changes a material's state, the tool's state and the material's visual presentation can become inconsistent and thus have to be updated.

Context Each tool whose material is changed without its knowledge, needs to be informed. 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, or the constraint or the container, which in turn will notify it about changes.

We feel that using the notification mechanism this way is highly problematic. Notification should be used as sparse as possible. Our experience shows that otherwise the system's architecture and dynamics become harder to understand. Luckily, there is an elegant and efficient solution.

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

The dispatch tables can be build 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 build independently of any specific material or tool.

Compare Compare our concept of tool coordinators and material containers to mediators as in [SN92].

The System's Boundaries (Outlook)

Here we end the presentation of our pattern-language by giving a brief outlook on two more patterns. The patterns of *Material Administration* and *Environment object* define the system boundaries needed to put 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.

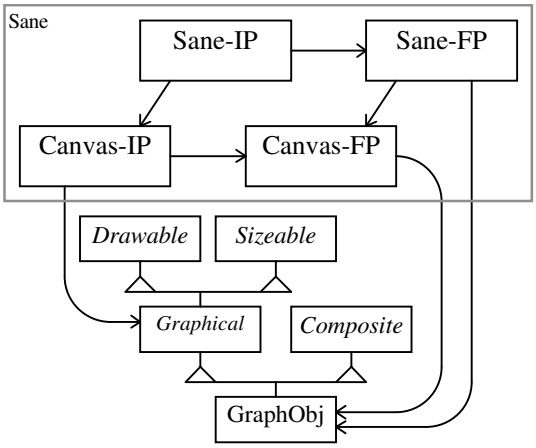
The Environment object is the first object to be created. After system startup, the Environment object creates material providers, each of them encapsulating a database service and the material administration which receives the material providers. After this, it opens the desktop and waits for users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

6 Tools&Materials at Work

We'll now give a short but full example to see the pattern-language at work. We'll design a system for *task oriented requirements analysis* (TORA), which is used at the University of Hamburg for teaching purposes. It consists of the graphical editor Sane for visually working with *activity net* materials and a glossary browser for textually documenting the objects of the editor.

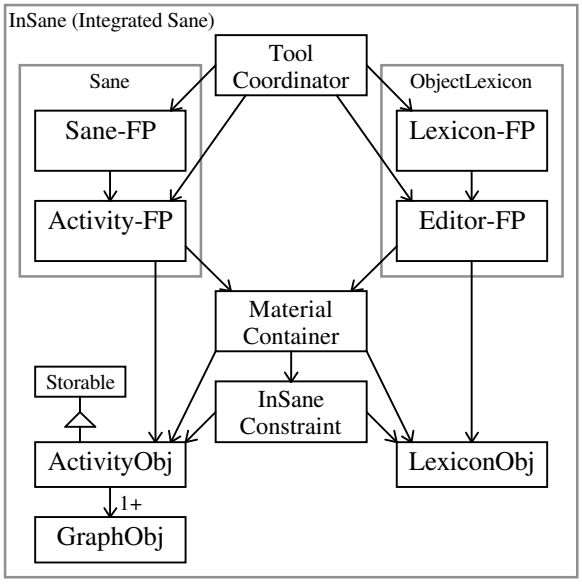
The *tool* Sane is *decomposed into a main tool with two subtools*. Each tool has an *interaction and a functional part*. The *main tool* establishes the frame for the usual services of an application like file-handling, clipboard access etc. The *Canvas subtool* allows for direct manipulation of the *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 a different approach to working with graphical objects as found 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 ResizeWrappers 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) helped very much to separate different functionalities from each other and made the evaluation of needed functionality easier.

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



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

7 Outlook

The pattern-language presented in this paper will be used to teach the Tools&Materials Metaphor to both professional developers and students. We'll carry the gained experience from the application framework to the industrial projects and will carefully analyze how the elaboration of metaphors and implementation techniques as design-patterns from a pattern-language enhance comprehensibility of the overall approach, the developers' language 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.

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*. Ed. 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*. Ed. 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.

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*. Ed. 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.

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*. Ed. 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.

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.

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*. Ed. 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.

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.