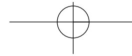
Zu11-07.qxd 20/8/04 7:38 AM Page 176



Domain service providers that are used as collections of functions are normally *stateless* and not based on a session model. Ideally, they have pure value semantics. However, if the calculations produced by a service provider build one on the other, then we often have to store intermediate results or subtotals. In this case, a collection of functions could tend to have a session character that is similar to the service providers used for transaction processing.

Example

Let's look briefly at a practical *example*. Assume that the financial mathematics of a bank can be implemented as a collection of functions, where all functions have to be verified. It appears meaningful to implement a centralized collection of functions across the organization, which can then be used in tools or automatons at different workplaces.

We define the following T&M design guideline:

Design guideline

A centralized collection of functions, to which we pass all parameters as values, and which calculates result values without side effects, does not represent a domain service provider. Rather, it is a mathematical collection of functions that do not encapsulate interactions with materials.

RATIONALE

The introduced concept of domain service providers has become a central element of our software systems. They provide the means to encapsulate domain-specific logic from the specific frontend technology or workplace type. They complete the picture outlined by tools, materials, and automatons.

WHAT NEXT

The construction of domain services is described as a design pattern in the next chapter.

7.11 THE TECHNICAL AUTOMATON PATTERN

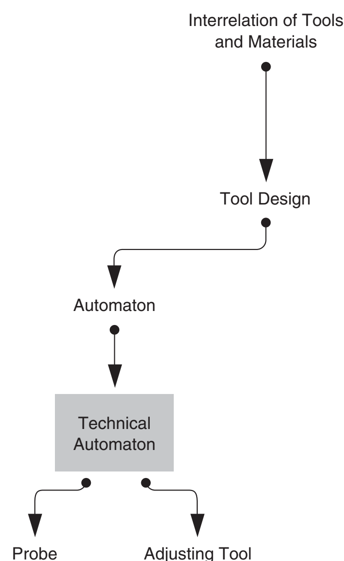
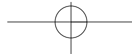


FIGURE 7.16 The *Technical Automaton* pattern.

Zu11-07.qxd 20/8/04 7:38 AM Page 177



Intent

The patterns of automatons and tools described in the previous sections are not sufficient for use in embedded systems because they do not meet all technical requirements. This section describes the additional concepts that are relevant—technical automaton and related elements—in the context of embedded systems.

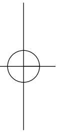
Problem

So far, we have limited the concept of an automaton to reactive devices that are activated by user interaction, such as the device park checker automaton described earlier. Technical automatons, on the other hand, in our application system model must include components that can be active without user intervention. This special property of embedded systems has to be considered both in design and construction.

What we need is a concept to control an embedded software system. This application component has to be able to control or represent technical components under hard or soft real-time conditions.

Relate To

This pattern is directly related to the concepts discussed in the *automaton* pattern (see Figure 7.16).



Solution

We design a *technical automaton* that maps domain-specific states of (real) technical devices to integrate them into the application system model. It informs other components of the application system about state changes and represents an additional source for events.

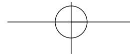
Technical automatons represent a conceptual pattern for real, physical devices, such as a lab system, a telephony system, or any other machine. Naturally, technical automatons are also real, but they are a model implemented in software that replaces physical devices. Technical automatons are a more specific variant of the generic notion of an *automaton* used in the T&M approach for use in embedded application systems.

We encapsulate the characteristics of embedded systems in technical automatons. In an embedded application system, a technical automaton represents an additional source of events, which can trigger actions on an equal rank with user-enabled actions.

Discussion

The state of the technical context is extremely important for embedded application systems, so it must be explicitly modeled. This state model has a purely conceptual nature. It reflects the domain view that a software developer has of the technical configuration, that is, the embedded hardware components. The state model includes all aspects of the technical context that the developer deemed relevant for the design

Zu11-07.qxd 20/8/04 7:38 AM Page 178



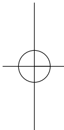
and construction of the application system. This means that this state model can be compared with the domain model when we develop interactive application systems (see Section 6.4).

From the purely technical perspective, the state model of a context contains mainly the interfaces available between hardware and software (e.g., readers or barcode scanners), and the events and state changes of that technical context, which are visible at these interfaces. In a medical lab, for example, this could include sample trays or sample flasks (see Section 11.4.2). In contrast, in the state model we normally abstract from the technical details of hardware components.

Embedded application systems are used to operate and monitor or control technical equipment. The state model of such an equipment is the *equipment model.*

> **The *equipment model* of an embedded application system is the software developer's abstract view of the embedded hardware configuration. At runtime, the equipment model maps the state of the application system's actual technical context, that is, the state of the equipment to be operated. From the technical perspective, the equipment model includes the interfaces between hardware and software, and the events and state changes of the equipment visible at these interfaces. The equipment model is a conceptual model and is not implemented in software.**

Obviously, we cannot analyze and create the equipment model and the application system model independently of one another, because they interact at runtime.



BACKGROUND

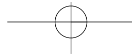
Representing technical processes and components

Technical devices in combination with application software are increasingly used outside the manufacturing industry, for example, analytical automatons in medical laboratories or call centers in telephony systems. These processes or technical components are usually characterized by the fact that they are used under continuous operation and real-time requirements, widely independent of individual workplaces. For example, the telephony system behind a call center or automatons in a medical lab should ideally be available all the time and at an optimum throughput of materials, such as calls or lab samples.

At the same time, there are normally workplaces where these devices are controlled and that request services or information about these devices. For example, a call center may have functional workplaces for the marketing staff and an expert workplace for the group manager who supervises and controls the call workplaces. Or the lab workplaces in a medical lab may be used to monitor analytical automatons and run special analyses.

Automatons handle technical processes

Real-world projects have shown that it is often meaningful to encapsulate the technical processes or components in independent automatons. These automatons differ from the automatons described earlier in that they basically have to be modeled as distributed or concurrent systems. They are independently active in a high availability rate, and the only way to address them is through asynchronous communication mechanisms. At first, this concept of a so-called "technical automaton" appears to conflict with the reactive character of workplaces developed by the T&M approach. To refresh your memory, these workplaces depend on their users' activities. However, the simple registry presented as an example in Section 7.10 represents a component that can be modified by other users. At this point, we have to consider a second characteristic. The technical components of an embedded system exist "outside" of our application software.


Zu11-07.qxd 20/8/04 7:38 AM Page 179

In fact, they are generally independent hardware and software systems, using sensors and actuators to interact within their environment. This effect on the environment is not within reach of the "direct access" of our application software. This is reflected in the fact that the application software would not be able to detect or influence that a lab sample flask tipped over or that a phone handset is off the hook.

Telephony example

Let's look at an example of a complex technical automaton for a telephony system. Obviously, a device of this telephony system changes its state when it receives an incoming call. We can describe this status change by implementing a "ringing" state for a phone extension. Accordingly, the "telephony system" automaton, that is, the control model we implemented in the software, also has to change its state. To better understand this process, imagine this telephony system in an interactive application software. The "ringing" state would have to be displayed by an interaction component that, in turn, is represented to the user by a suitable icon. This means that we have two equal-ranking sources of events: the user and the telephony system automaton.

WHAT NEXT

The subsequent patterns *Probe* and *Adjusting Tool* complement this pattern.

7.12 THE PROBE PATTERN

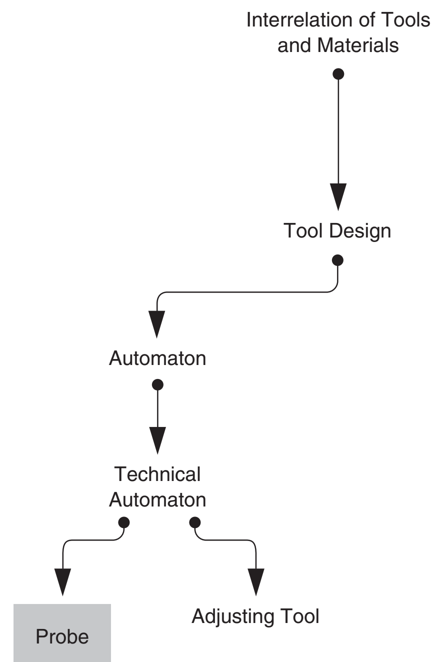
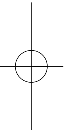


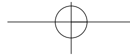
FIGURE 7.17 The Probe Pattern.



PROBLEM

Clients of a technical automaton are not always interested in the entire state of an automaton. In many application cases, it is sufficient to know a partial state of the

Zu11-07.qxd 20/8/04 7:38 AM Page 180



technical automaton. This is also reflected in our design principle that the model of an application system should include only states motivated by the application domain. In addition, we should bear the software-specific rule in mind that unnecessary communication traffic between the technical automaton and its clients would slow down the system's runtime or performance.

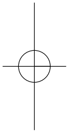
We want to design a unit related to a technical automaton where the client of an automaton can select an interesting section of the state space mapped by the automaton. We want to integrate this unit so that it will interfere as little as possible with the automaton.

Relate To

This pattern is directly related to the concepts discussed in the *technical automaton* pattern (see Figure 7.17).

Background

The domain-specific states of a technical automaton can be grouped into two categories. Characteristic properties of an automaton that never change fall into the first category. These properties may be read directly at the automaton. The domain-specific states of an automaton that change in regular or irregular intervals fall into the second category; these are variable reading values.



Unfortunately, it contradicts the very idea of a technical automaton to interfere with its operation to read variable reading values. Instead, we want to see it operating without interruption.

We are familiar with this kind of determining reading value from technical systems, where we normally use probes. Like sensors, probes sense or detect physical conditions for reading purposes, without interfering in such conditions. We take this concept and, using it as a metaphor, transfer it to the T&M approach. In doing this, we show clearly that probes exist only on the application software level, and that, unlike sensors, they are never part of a physical equipment.

Solution

We design a *probe* as a component that can determine a measurable value of a technical automaton with high accuracy. A probe can be set so that it reads and updates this value in specific time intervals and with specified tolerances.

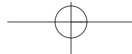
We attach probes to a technical automaton to read domain values (see Section 2.6.5) based on typed reading values specified in the automaton, and make these values available in the application system. In the design of our technical automaton, we will specify the types of probes that can be attached. The automaton can be queried for available probes.

Discussion

In the T&M approach, a probe is an object that reads a defined reading value type from an automaton and returns these values in specific intervals, for example, in domain-specific intervals, or when specific values change.

A probe allows us to represent single reading values, relevant for the application domain, from an automaton. A probe can aggregate the states of a physical device and

Zull-07.qxd 20/8/04 7:38 AM Page 181



use them to calculate domain values. The automaton knows the basic requirements of the probe. Accordingly, it returns a result to the specified probe according to its state changes. The probe is loosely coupled to its clients, and informs these clients when new values are available.

RATIONALE

Technical automatons with probes are a useful conceptual pattern to design embedded application systems. Technical automatons and probes form a conceptual unit. This unit represents a source of events in the application system.

WHAT NEXT

The subsequent pattern *Adjusting Tool* complements this pattern. Technical constructions can be found in the related design pattern of Section 8.13.

7.13 THE ADJUSTING TOOL PATTERN

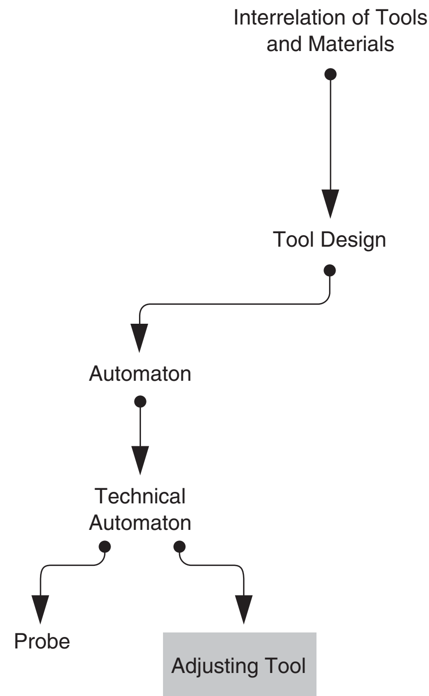
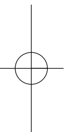


FIGURE 7.18 The *Adjusting Tool* pattern.

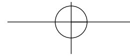


PROBLEM

With technical automatons and probes, we have useful concepts to port data from embedded systems to a T&M environment. What we also need is a tool to be able to modify a technical system.

The components of an embedded system have to be set in regular intervals, in addition to other tasks such as control and maintenance work. However, these technical

Zull-07.qxd 20/8/04 7:38 AM Page 182



components are normally not installed in the immediate neighborhood of the workplaces supported by the underlying application system. What we need are interactive tools in the work environment to be able to do these setting, control, and maintenance jobs within an embedded application system.

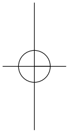
Relate To

This pattern is directly related to the concepts discussed in the *technical automaton* and the *probe* patterns (see Figure 7.18).

Background

Though automatons *can* have an interactive component, it is not always *possible* in technical automatons. The technical device or equipment, represented by an automaton, may be in a location outside the work environment in which this automaton is used. In many cases, security reasons may require that the technical automaton, which controls a technical device and maps it to the application model, runs on a separate computer, independently of the application system. Nevertheless, we need a way to modify it from within the application system.

We observed in many real-world projects that all of the automatons used in these projects had to be set in more or less frequent intervals. For example, the technical automatons behind a telephony system supporting workplaces in a call center have to be continually maintained. A user has to be able to configure this automaton to the number and distribution of incoming calls over the call center staff.



In out search for a suitable concept, we oriented ourselves to the daily work with "real" equipment and technical devices, which are normally maintained and set by qualified engineers in regular intervals. The maintenance staff normally uses special tools to set various parameters for these technical devices.

Solution

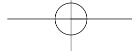
We design an *adjusting tool*, which is a special tool used to maintain and control technical automatons. Based on domain-specific motivation, it shows a segment of the state of one or more technical automatons, allowing us to set parameters for these automatons.

Basically, a technical automaton can have a domain interface for that purpose. The adjusting tools interact between the user and the technical automaton. The user can use these tools to visualize and change the parameters of a technical automaton.

Discussion

To be able to represent the states of an automaton, we connect the adjusting tool to the automaton, for instance, by the use of a probe, so that it can be fed with state information. We can use such an adjusting tool to specify both the type of information and the interval in which these state descriptions should be returned. Depending on the application domain, we can display a subset of the states of an automaton or the entire range of domain states. In addition, we cannot generally specify how state information should be updated in the adjusting tool. Based on typical requirements, state information is refreshed upon each change of the automaton's state or in fixed time intervals.

Zull-07.qxd 20/8/04 7:38 AM Page 183



Thus far in our real-world projects, we have rarely seen a case where a user operates the adjusting tool explicitly to request state information.

Modifying the parameters of a technical automaton requires a slightly different concept than what was described earlier for the design of tools and materials. In interactive application systems, the user normally has full control over everything that he or she does with tools and materials. In contrast, technical automatons have a certain "life of their own," because they map independently running technical devices.

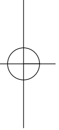
For this reason, adjusting tools cannot directly access a technical device; instead, they send requests to the technical automaton. How or to what extent the automaton will then be able to change state as a result of such a request, or whether it rejects a requested state change, should be based on domain and software requirements.

RATIONALE

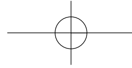
Technical automatons with probes and adjusting tools are a useful conceptual pattern to design embedded application systems. Adjusting tools frequently will be used together with probes; they manage this source of events in the application system.

WHAT NEXT

Technical constructions may be found in the related design patterns of Section 8.13.



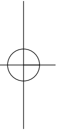
Zull-08.qxd 31/8/04 2:35 PM Page 185



T&M Design Patterns

8.1 INTRODUCTION

Chapter 7 described T&M conceptual patterns. In this chapter, we introduce their matching T&M design patterns. These build on the conceptual patterns and are a major constructive step toward actual software implementation. The major portion of these design patterns deals with technically designing and implementing tools and materials. In addition, we introduce patterns that complement the core elements of our approach. This chapter discusses the following concepts, which have been introduced in previous chapters:

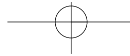


- tools and materials;
- tool construction and composition;
- domain values;
- containers;
- forms;
- automatons;
- domain services; and
- work environments.

Each concept is described as one or more separate patterns (cf. Figure 8.1). We take up the pattern structure from Chapter 7 and complement it with a few items based on the character of design patterns:

- Pattern name:
- *Intent*: What is this pattern good for?
- *Relate to*: Which other conceptual or design patterns precede this pattern?
- *Problem*: Which problem does the pattern solve?
- *Solution*: The central solution concepts.
- *Schema*: The elements of the pattern and their interrelation.
- *Background*: What has led us to this pattern? (A section that can be skipped on first reading.)
- *Trade-offs*: Pros and cons using this pattern

Zull-08.qxd 31/8/04 2:35 PM Page 186



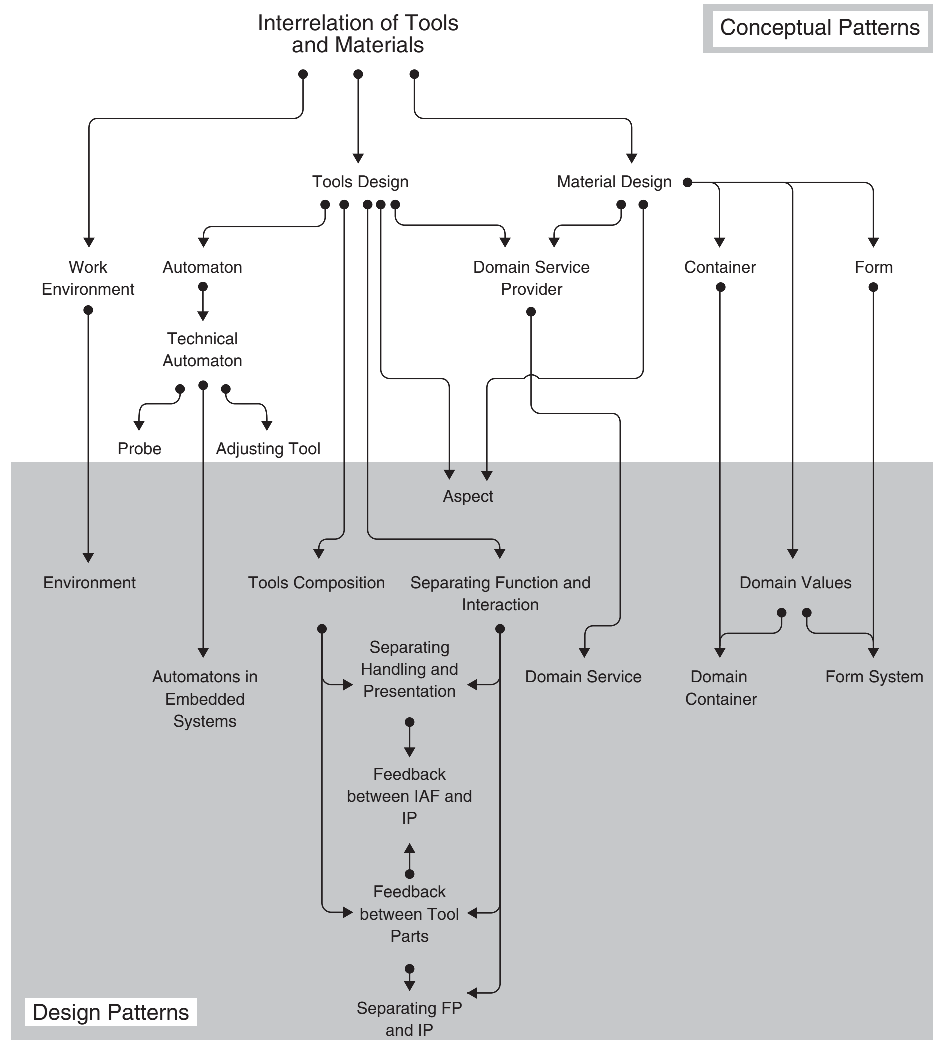
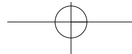


FIGURE 8.1 Hierarchy of T&M design patterns.

- *Example*: An example, usually with a short discussion.
- *Construction part*: How can this pattern be implemented? Here we use short examples from different programming languages (Java, C++, Smalltalk).
- *What next*: Which patterns will be useful next?

We begin this chapter with an overview of the T&M design patterns as a guided tour. The reader will thus understand how the patterns of this chapter belong together. In the final section we close with a discussion of the details of implementing the patterns discussed in our JWAM framework. This section should serve both as another example showing the interplay of the patterns and as a sketch of how to build a framework that supports the constructive ideas of the T&M approach.

Zu11-08.qxd 31/8/04 2:35 PM Page 187



8.2 A GUIDED TOUR OF THE T&M DESIGN PATTERNS

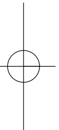
Based on the conceptual patterns discussed in Chapter 7, this section describes T&M design patterns. We will now show how conceptual patterns and design patterns are related. We then discuss which design patterns belong together. In this way we present something like a roadmap, with comments for the reader, that continues the guided tour to the T&M patterns.

The following guided tour is based on the T&M design patterns illustrated in Figure 8.1.

The Aspect Pattern (Section 8.3)

The central conceptual pattern is called *interrelation of tools and materials*. It was introduced in Chapter 7.3 and shows how tools and materials complement each other in the human work process. This chapter starts with a design pattern called *aspect* that shows how the two components match.[1] The aspect pattern explicitly represents the interplay between tools and materials. We will discuss two construction parts, aspect classes and interfaces, in some detail, because they are suitable for most cases. In addition, we will briefly describe a number of alternatives for special cases.

The Separating Function and Interaction Pattern (Section 8.4)



This pattern is directly related to the conceptual pattern *tool design* (Section 7.5). Designing and implementing tools is a major challenge to software developers as here everything comes together: domain functionality, handling and presentation that is based on a clear usage model, and the implemention of interactive graphic components.

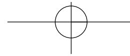
In order to deal with the complexity of this task, the separation of concerns is the prime aim. This is addressed at a general design level by the pattern *separating function and interaction*. It shows a proven method to implement the two fundamental characteristics of a tool as two separate modeling and construction units. This pattern is based on the idea of dividing tools internally into a function and an interactive part (IP). A similar form of this separation appears in the Model View Controller concept well-known in Smalltalk systems.

The Tool Composition Pattern (Section 8.5)

Similar to the pattern *separating function and interaction*, the pattern *tool composition* directly relates to the conceptual pattern *tool design* (Section 7.5). It addresses the problem of scaling up tools by showing how we can build simple tools and then how we can combine them to build complex tools. In this connection, we will think of tools as components that can be implemented internally in different ways. Three construction parts give detailed guidelines on how to solve various construction problems. Several other patterns introduced in this chapter ensure that tools may be combined

1. In this context, we use "aspect" as a separate concept, not related to "aspect-oriented programming."

Zull-08.qxd 31/8/04 2:35 PM Page 188



into complex tools even when we implement the single tools differently. This includes the following patterns:

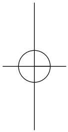
- Feedback between Tool Parts
- Separating Handling and Presentation
- Feedback between Interaction Forms and IP

THE FEEDBACK BETWEEN TOOL PARTS PATTERN (SECTION 8.6)

This pattern directly follows the patterns *tool composition* and *separating function and interaction*. With respect to tool composition and separation of function and interaction, we have to answer the question how the single parts involved should interact. The so-called reaction patterns play an important role in answering this question. They control part of the communication between dependent components and are described in the pattern *feedback between tool parts*, which is used, for example, in the pattern *separating FP and IP* (Section 8.7). The central idea is again the separation of concerns: this time we make sure that two or more construction units, which interact, know as little of each other as possible.

THE SEPARATING HANDLING AND PRESENTATION PATTERN (SECTION 8.8)

Regardless of whether or not tools are divided into functional and interactive parts, we can use another pattern called *separating handling and presentation*. Together with the concept of interaction forms, we will introduce a proven and easy-to-implement concept that allows us to abstract GUI design from the concrete user interface and GUI toolkit. Some readers may think that this abstraction is superfluous in the days of Java and Swing, but our project experience shows that independence from a specific GUI library will pay in the long run. Of course, the trade-offs between direct GUI access and another abstraction layer must be discussed.



THE FEEDBACK BETWEEN INTERACTION FORMS AND INTERACTION PART PATTERN (SECTION 8.9)

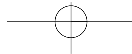
This pattern is the logical consequence of the pattern called *separating handling and presentation*. Once we have separated the concrete GUI classes from the general interaction of a tool, we need to bring together these two construction units. Showing another application for the reaction pattern, this pattern describes a *feedback between IAF and IP*. It concludes the patterns explicitly addressing tool construction.

THE DOMAIN VALUES PATTERN (SECTION 8.10)

As materials objectify pure domain functionality, little can be said about the concrete construction of materials. So, although there is a conceptual pattern called *material design*, there is no directly matched design pattern called *material construction*. On the construction level, there is, however, a fundamental T&M concept called *domain values*. We addressed that concept in Section 2.6.5, when we discussed the fundamental elements and characteristics of the object-oriented programming model.

Domain values form the basic components that we use to compose materials. The pattern *domain values* introduces different implementation techniques as construction parts. In addition, we will show how special domain-value interaction forms (see also *separating handling and presentation* in Section 8.8) or special domain-value widgets can be used to simplify the development of sophisticate user interfaces.

Zu11-08.qxd 31/8/04 2:35 PM Page 189



THE DOMAIN CONTAINER PATTERN (SECTION 8.11)

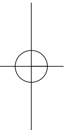
We have introduced *containers* as a conceptual pattern directly related to materials. On the constructive level the pattern *domain containers* shows how containers can be implemented. Although we have said that it is of little value to have a dedicated design pattern for materials in general, we must solve special design problems for domain containers. Today, developing technical containers, sometimes still called dynamic data structures, is no longer a problem. There are good standard libraries for all common programming languages. For this reason, we deal with this issue only briefly, simply explain the relationship between domain and technical containers.

THE FORM SYSTEM PATTERN (SECTION 8.12)

On the conceptual level, we have established forms as another materials category, in addition to containers. Forms are suitable to implement simple materials with only few domain-specific operations. The directly related design pattern *form system* shows how we can implement forms and how they can be combined with a small set of standard tools to speed up our application development process. Forms cannot exist "independently" from materials. For this reason, we will also explain how forms can evolve toward more complex materials with their own domain functionality.

THE AUTOMATONS IN EMBEDDED SYSTEMS PATTERN (SECTION 8.13)

The concept pattern *automatons* discusses the main characteristics of automatons. We have already said that an automaton usually runs in the background. This reduces an automaton almost entirely to the implementation of a domain or technical algorithm, and there is little left for a general design pattern.



The matter is different with technical automatons in a distributed environment. Therefore we discuss the design and construction of software within embedded application systems as the design pattern *automatons in embedded systems*. More specifically, we will explain how automatons can be built to fit into the T&M approach, taking asynchronism and process distribution into account.

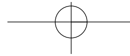
THE DOMAIN SERVICES PATTERN (SECTION 8.14)

The concept pattern *domain service provider* deals with collections of materials and the domain interactions with these collections. As a direct logical consequence, the design pattern *domain service* offers a way to represent the notion of service providers as software units, resulting particularly in a possibility for flexible configuration of client-server systems. For example, we could support totally different workplace types with frontends (rich clients or thin clients on PC, Webtop clients, laptop clients, etc.) and easily connect back-end systems (e.g., host, SAP). We will use the *domain service* pattern to show how domain service providers may be implemented to realize these ideas.

THE ENVIRONMENT PATTERN (SECTION 8.15)

The concept pattern *work environment* discusses the characteristics of the workplaces and their environments according to the T&M approach. It defines the boundaries of a workplace and embeds tools and materials in this workplace. In addition, it allows the workplaces of this environment to communicate with other environments. The design pattern *environment* explains how we can implement this generic concept and what features we would normally need.

Zull-08.qxd 31/8/04 2:35 PM Page 190



USING THE T&M DESIGN PATTERNS FOR THE JWAM FRAMEWORK (SECTION 8.16)

The last section of this chapter describes how design patterns interact. Based on our EMS example, we show how the design problems discussed in this chapter can be solved by a framework approach. We show the general elements of an interactive T&M application that we have moved into our JWAM framework in Java. Besides seeing the interrelation of the T&M design patterns, the reader can get an idea about designing a generic application framework that is domain-independent but still provides a good basis for implementing T&M applications.

The References at the end of this chapter includes useful sources for further study of the material discussed in this chapter.

8.3 THE ASPECT PATTERN

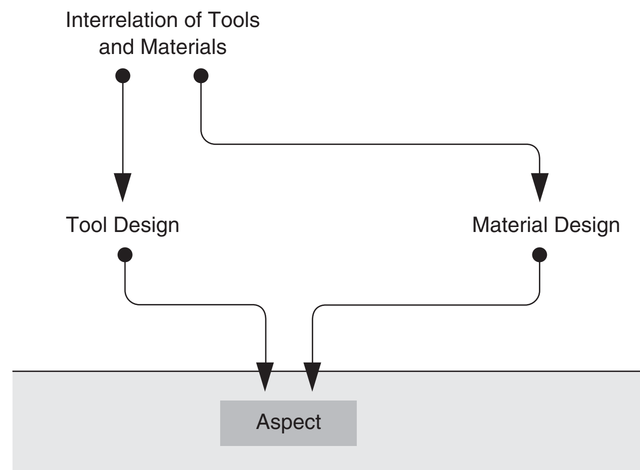


FIGURE 8.2 The Aspect pattern.

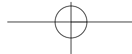
INTENT

Users handle tools to work with materials in completing their tasks. Tools and materials should not be combined arbitrarily, because a tool matches some materials but not others, and vice versa. Aspects link matching tools and materials.

PROBLEM

To identify a construction principle, ensure that tools and appropriate materials match.

This principle should apply to the design, before we get to writing code. It appears to be meaningful to identify more than a single material for a tool (and vice versa). The domain-specific relations between tools and materials should emerge in the technical design.



RELATE TO

The *aspect* pattern is related to the conceptual pattern *interrelation between tools and materials* (see Figure 8.2).

SOLUTION

We represent the interplay between tools and materials by the use of aspects. Such an *aspect* objectifies the interrelated use of a tool and the materials it operates on.

The characteristics of an aspect are:

- It represents both the syntactic interface and as much of the semantics of the relation as possible.
- It ensures that a material meets the requirements specified in the aspect for its use by a tool.
- In our construction, we pay careful attention that the tool uses its allocated materials exclusively over the interface specified in the aspect.
- An aspect explicitly solves the technical problem of combining different protocols or interfaces in our design.

To better understand the solution, consider the example shown in Figure 8.3. In this example, a `Tool` uses an `Aspect`. This `Aspect` specifies the services that a

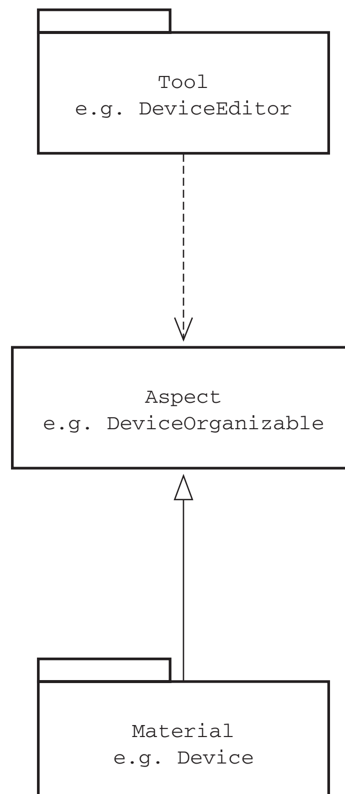
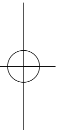
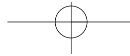


FIGURE 8.3

Pattern schema to bind tools and materials.



Zull-08.qxd 31/8/04 2:35 PM Page 192



Material has to provide for a Tool. The Material meets the interface promised by the Aspect and the behavior defined in the interface. Note that we are not saying at this point that the material always must inherit from the aspect, because several construction approaches for the pattern will use inheritance between Material and Aspect while others will not, depend on the programming language we use.

EXAMPLE

Let's return to our EMS example for a moment to better understand how we can use an aspect to bind tools and materials (see Figure 8.4).

In the EMS example, DeviceOrganizer is a tool that uses materials, including the RoomPlan material. However, our DeviceOrganizer needs only part of the features offered by RoomPlan. The DeviceOrganizer is responsible for allocating Devices and Employees to rooms, giving the user an overview of the distribution of devices and persons over the rooms. Figure 8.5 shows the interface of a material, RoomPlan, and the aspect DeviceOrganizable, representing that part of the interface the DeviceOrganizer is interested in.

BACKGROUND: MATCHING TOOLS AND MATERIALS

The way tools and materials interrelate within a work process is extremely important for understanding human work. Handicraft traditions and underlying standards ensure that tools match materials. The correct handling of tools and materials is part of every crafts person's training. Standards (e.g., the ISO standards) specify the appropriate match of many tools and materials, such as screwdrivers, screws, and nuts. Unfortunately, there are very few such solid traditions and standards for software tools and materials. One of these few examples is the CORBA-IDL interface definition for services. For this reason, we have to ensure in our construction that software tools and materials match and that they can be combined.

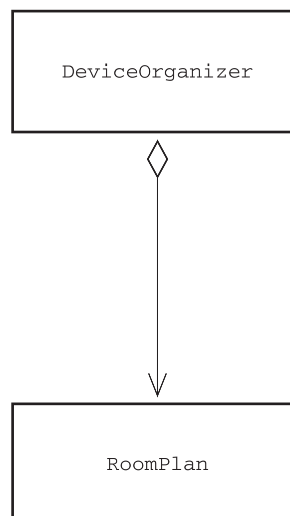
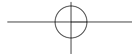


FIGURE 8.4 Using the Device Organizer tool to edit the Room Plan Material.

Zu11-08.qxd 31/8/04 2:35 PM Page 193



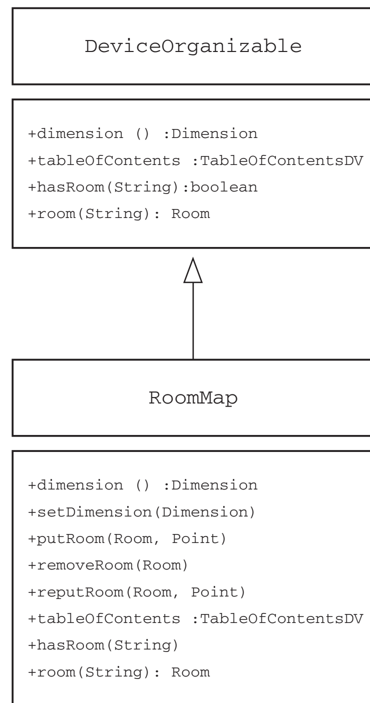
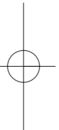


FIGURE 8.5

Material and aspect interfaces.



If we think of tools and materials as components for our application systems, then this match means that we need appropriate interfaces. Based on the usage model, users are supposed to use tools to work with materials; we obviously have to add an interface to a material, so that it can be used by a tool.

Component models and their interfaces

Most component models currently available define generic component interfaces, including a set of operations that can be used to request actual domain-specific operations at runtime. These approaches are too generic for the purpose of matching tools and materials. What we want is an explicit domain-specific abstraction that describes the match of tools and materials. This also tells us explicitly which conditions a material has to meet to be suitable for a tool.

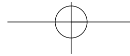
A single tool for different materials

Our conceptual pattern *interrelation of tools and materials* (see Section 7.3) shows that a tool should ideally be suitable for different materials. This means that a tool's interface to a material is narrower or more abstract than the full interface of a specific material. If no explicit interface between a tool and a material is specified, then the question is what segment of a material's interface will be used by a specific tool. This information cannot be easily derived from the design, because the interface that a tool expects cannot simply be extracted from the material; it is determined by that tool's requirements. Unless we introduce additional components to the design, then analyzing the program text of that tool is the only way to determine the interface between a tool and a material.

Different tools for a single material

This basic problem becomes more serious if several tools use a single material. The concrete interface of a material becomes "wider," while the reference between tools and materials drifts further apart. If we want to introduce new materials to an existing application system, we must solve the following problem: To be able to handle the new

Zull-08.qxd 31/8/04 2:35 PM Page 194



material using existing tools, we have to identify the interface that materials should have. Unfortunately, we won't find this information either in the tool nor in other materials used by that tool.

TRADE-OFFS

A material is responsible for providing a set of coordinated operations that a tool can use to fulfill a specific task. For this purpose, an aspect should specify features beyond the purely syntactic interface and describe the behavior of materials.

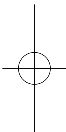
Thanks to these features, aspects are also suitable to define the relationship between automatons and the materials these automatons require. This means that automatons and tools can be equally combined with materials.

Specifying the behavior of aspects

The means available in object-oriented programming languages to specify the behavior of aspects are limited. We want to use an aspect at least to *specify* an interface. We want to define the interface as a type to ensure that its requirements are met, which is basically supported by a static type concept available in all languages. In statically typed languages, you can use an abstract class to implement an aspect. Our use of inheritance to link tools and materials is based on our experience with practical projects, where we used C++ and Eiffel. Considering that Java and similar languages offer a named interface concept, these languages are even better for this kind of construction.

Aspects and single inheritance

In contrast, we will have to deal with certain problems when trying to use a type to implement aspects in languages that know only single inheritance and no interfaces. If we want to use classes to implement aspects, and if there is no one-to-one relationship between tools and materials, then a material has to inherit from several aspect classes. For example, Smalltalk knows only single inheritance, and the fact that Smalltalk does not support static type checking represents an additional problem. In that case, we have to test the interfaces of materials for compliance either at runtime, or we check and change the program text in the course of our development process.



Aspects and materials

We can also use aspects to specify the behavior of materials in different "strengths." For example, one solution uses aspect classes with abstract implementations and appropriate hook operations. If we do not want abstract implementations for the specification of behavior, then we could use the contract model described in Section 2.3, at least in part, to define the conditions for operation calls and legal states for a material. Note that these solutions require the use of class types to implement aspects; otherwise, all that remains is the pure interface test.

Construction approaches for aspects

When discussing Java and its way of defining interfaces, we have to deal with the following questions. Should we use classes to define aspects at all? Or would it be better to define aspects as named interfaces?

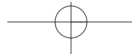
RATIONALE

If you have to design and implement complex or generic tools using more than one type of material, then you should explore the different constructions realizing the aspect pattern. For simple tools or those working on one material, subtyping of the used material will do.

WHAT NEXT

There are no specific T&M design patterns for materials as they implement "pure" application logic.

Zull-08.qxd 31/8/04 2:35 PM Page 195



8.3.1 Construction Part: Using Inheritance or Interfaces to Implement Aspects

If you use a class to model an aspect, then a material can inherit this aspect directly. This approach normally uses inheritance such that all operations of an aspect class are visible at the material's interface.

In contrast, languages like Java let you use a language construct directly to define named interfaces. In fact, this is necessary in Java, because the language does not support multiple inheritance between classes.

Note that there is a real type-subtype relationship between an aspect and a material, because a material inherits and implements the full interface of an aspect. In all places where you need an aspect type, you can use any object of a material class that is conforming to the type of the aspect.

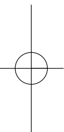
EXAMPLE

The EMS example

Let's look at an aspect interface in the context of our EMS example (see Figure 8.6). The aspect interface shows the operations of a material expected by the `DeviceOrganizer`.

TRADE-OFFS

Aspect classes allow you to specify the aspect interface between a tool and its material in an abstract class. This means that, although aspect classes can be used as types, you cannot use them to directly create objects, as when using interfaces in Java. This construction should be used so that tool classes never operate directly on a material class; instead, they use abstract aspect classes (see Figure 8.7).



If classes implement aspects, then the material classes inherit from these aspect classes and implement the features they inherited. Using polymorphism, tool classes call operations implemented in materials.

Aspects and multiple inheritance

Materials are normally not used by a single tool. Figure 8.7 shows that doing so would cause a material to inherit from several aspect classes. However, a tool can also operate on several materials. This means that the entire interface of a material includes the individual interfaces of each aspect class. In summary, this case requires multiple inheritance.

Note that the schema shown in Figure 8.7 is simplified, as each tool uses one aspect class to access one material. In the real world, complex tools would normally use more than one material, and thus more than one aspect. Also, it is customary to combine elementary aspect classes with more complex ones by the use of multiple inheritance.

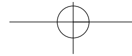
Aspect classes and standard implementations

By modeling aspects in classes, we can define standard implementations for operations, which complies with the basic idea of aspects. In fact, the more accurately the behavior of materials is described, the more probable it is that a tool will be able to

```
public interface DeviceOrganizable
{
public Dimension dimension ();
public TableOfContentsDV tableOfContents();
public boolean hasRoom(String name);
public Room room (String name);
}
```

FIGURE 8.6
Example of an aspect interface.

Zull-08.qxd 31/8/04 2:35 PM Page 196



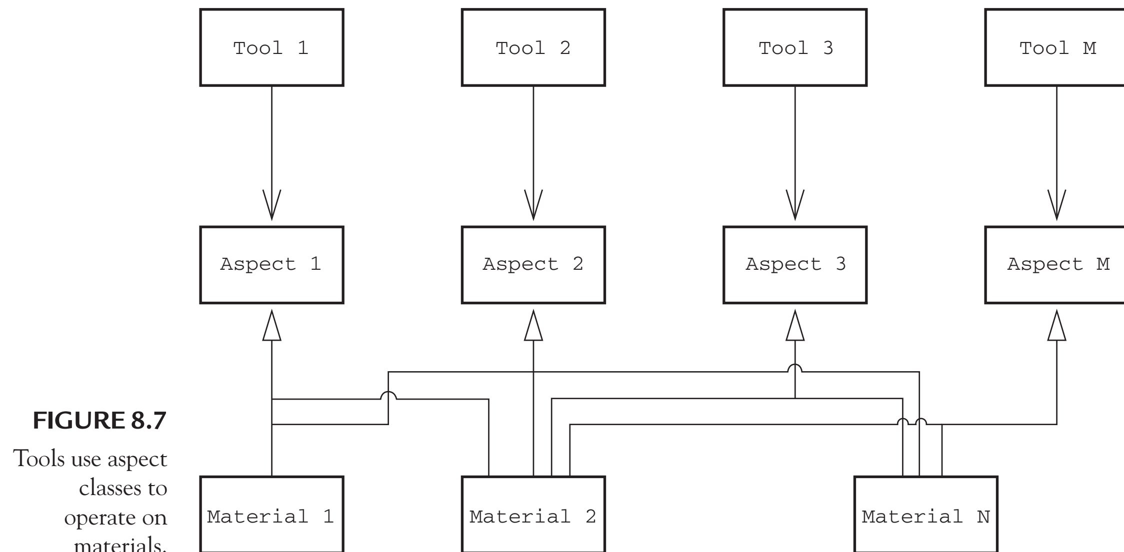


FIGURE 8.7
Tools use aspect classes to operate on materials.

work on a material, both in terms of syntax and semantics. In addition, these standard implementations allow us to model the material state that a tool requires early on, that is, in the aspect class. Material classes can override the defined operations or add attributes to expand a state.

However, a construction that also defines attributes in aspect classes should be handled carefully. The reason is that such a construction conflicts with the idea that aspects are interfaces, as well as with the object-oriented design principle that abstract classes should be lightweight. As a consequence, all material classes we derive are "burdened" with these attributes. Even if this may be justified in the original design of an application system it does not necessarily mean that future material classes under an aspect class should actually be defined with these attributes.

If we use interfaces to implement aspects, as in Java, then the pattern schema would look like the one in Figure 8.8.

In this case, however, we could not provide for a standard implementation in the aspects. This limitation caused by the use of interfaces is offset by the fact that we will not erroneously define attributes for our aspects that would burden all derived material classes. In addition, the problem of many tools using many materials is solved, because Java supports multiple inheritance for interfaces.

Consequences of Static Aspect Typing

One major benefit of static typing by means of an aspect class or an aspect interface is that the compliance of a material with an aspect is checked at translation compile time, which is like asking: "Does the material implement the type defined in the aspect?" In addition, it ensures that a tool can use "its" materials only under that aspect, which is like asking: "Does a tool exclusively use materials of that aspect type?" In a class-based solution, we can use standard implementations or template methods to specify behavior. And finally, if we are careful, we can implement a state for materials at that early stage.

Zu11-08.qxd 31/8/04 2:35 PM Page 197

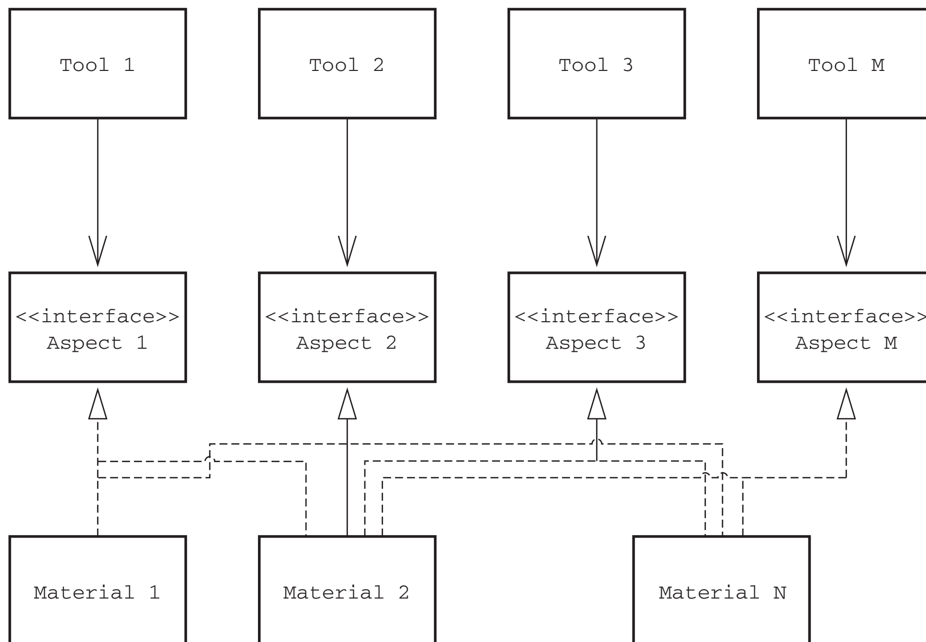


FIGURE 8.8
Using interfaces to implement aspects.

Generalizing aspects

The idea of aspects does not have to be limited to coupling tools and materials. In fact, we have extended it to the relationship between automatons and materials. If we think of aspects as protocols that a class should meet, then we can find additional applications for this modeling type. Examples include a protocol for the use of container classes, or between containers and managed objects, or to distribute objects. In all cases, objects expect a specific protocol, or generally more than one operation. At the same time, we have to answer the question whether separate classes for these protocols would eventually be too expensive and confusing.

Problems with the construction approach

Unfortunately, using abstract superclasses to model aspects for materials has some drawbacks.

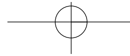
When aspects are bound to materials, the *inheritance mechanism is used differently* than in domain modeling. To better understand the relation between domain and software design, we use inheritance primarily to model domain concept hierarchies, where single inheritance is an essential composition rule. When using inheritance to implement aspect classes, we describe the match of tools and materials in one single work context. This is a totally different notion, because an aspect can unite a set of materials that have no similarity from the domain view. They are all totally different, apart from the fact that they can be manipulated by the same tool. For example, a printer could print the set of different materials and a trash could delete them.

Different ways to use inheritance can make your design harder to understand. One solution for solving this problem is name conventions. For example, we name all classes that model domain objects with nouns (e.g., `Folder`, `Form`, `Account`), while using adjectives for aspect classes (e.g., `Editable`, `Storable`).

Interfaces used for aspects

This problem does not initially occur when using interfaces to implement aspects. However, it is meaningful to use interfaces for other things, in addition to aspects, in

Zull-08.qxd 31/8/04 2:35 PM Page 198



languages that don't support multiple inheritance for classes, such as Java. For this reason, it is always a good idea to use the name convention proposed previously.

Another question is of a more general nature: Will inherited aspects cause materials to be too closely coupled to tools? After all, inheritence means that a material has to implement static interfaces that may be required only temporarily or in specific use.

Another important thing to remember when working with aspect classes and multiple inheritance in large systems are the costs for compiling and linking, especially in languages like C++. In addition, we may have to deal with an enormous amount of superclasses, resulting in confusing code.

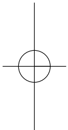
Aspect classes and subsystems

If we develop a complex framework based on the T&M approach and use appropriate mechanisms to structure these frameworks (see Section 8.15), we will have to deal with problems when attempting to allocate aspect classes to subsystems.

Considering that aspect classes describe the interface of a tool to one or more materials, they actually belong to the tool classes that use them. On the other hand, material classes must inherit from aspect classes. This means that material subsystems would depend on tool subsystems, because a material subsystem inherits from classes belonging to the tool subsystem.

Another thing is that we cannot simply allocate an aspect to a material class, because we want the aspect to be basically valid for many materials.

On the other hand, if we build an independent subsystem for aspect classes, then we will have to go through a tiresome amount of work to reconstruct the domain context; and the material subsystems would depend on the aspect subsystems. Despite all these problems, this is the road we will take, for example to avoid relationships between material packages and tool packages in a Java environment. In summary, we explicitly create aspect packages.



RATIONALE

We would use this solution of adapter classes for large and complex systems implemented in languages that provide multiple inheritance.

8.3.2 Construction Part: Using Object Adapters to Implement Aspects

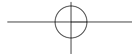
Languages based on single inheritance do not let you use superclasses of materials to build aspects. We propose the use of an adapter, as described by Gamma et al., as an alternative solution.

You can use the *adapter pattern* to let classes collaborate, which they could not do otherwise due to incompatible interfaces. In general, an adapter matches a specific interface (the interface of the object to be adapted) to an interface expected by a client. The adapter pattern comes in two flavors: the *class adapter* is based on multiple inheritance (and corresponds to our aspect class), and the *object adapter*, which is the one we want to use here (see Figure 8.9).

TRADE-OFFS

The obvious benefit of using object adapters is that the aspects can be clearly identified as independent classes in the design. Another important benefit is that this solution also works for languages with single inheritance.

Zull-08.qxd 31/8/04 2:35 PM Page 199



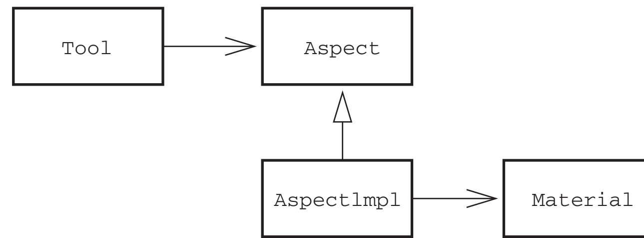
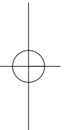


FIGURE 8.9
Using object adapters to implement aspects.

In addition, you can also use object adapters if the interface of a material does not comply with the interface required by a tool. In this case, you can use operations of that material when implementing the aspect so that it emulates the required interface. This method of implementing aspect classes so that the material interface is adapted to a tool can also be useful where aspect inheritance is supported. The operations declared in the aspect class will not be visible in the materials. And the materials remain limited to their pure domain interface, while tool interactions are defined in special adapters.

Adapting materials to different use contexts

Another important benefit of the object adapter transpires when you need materials with different functionality in different use contexts. In such a case, the object adapter will prevent a material object from being loaded with functionality that is required in only one place.



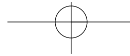
When using adapters, you can statically check for aspect compliance, and the aspects will be visible as independent classes in the design. The material interface has to be suitable basically for an aspect and can be adapted in the aspect adapter class, so that the material remains "slim" (i.e., with a minimum of operations and attributes). Consequently, another benefit of this approach is that you can incrementally improve or expand your project.

Finally, you can link aspects to tools without destroying the overall architecture, as discussed in Section 8.3.1.

Despite all these benefits, the adapter pattern also has drawbacks. One major drawback is that you have to implement a concrete adapter subclass for each material. When implementing generic tools, such as editors or list generators to edit many different materials, you will soon observe an *inflation* of concrete adapter classes. This adds complexity to your design, reducing the benefit of having explicit aspect classes.

More serious problems arise in many contexts resulting from the two objects, such as an adapter object and a material object, that actually make up the material. One of these serious problems is a loss of identity: from a tool's perspective, a material has a different technical identity than its adapter. However, we need the domain identity of materials in many situations. For this reason, we have to ensure that the domain identity of a material is maintained after we have attached an adapter, despite a different technical identity. A feasible though rather complex solution uses the role pattern described in Section 9.4.1. In any event, we cannot directly compare objects, but have to define an additional identifier for comparison purposes. For example, in C++ you can elegantly solve this problem by overloading the comparison (relational)

Zull-08.qxd 31/8/04 2:35 PM Page 200



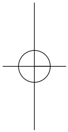
operator. The additional identifier for objects is required anyway in complex application systems to ensure unique identifiers in connection with persistence and cooperation support.

Finally, we have to solve a construction problem. How and from where should we create aspect objects? If we use direct aspect inheritance to build them, we can simply pass a material to a tool, but the additional adapter objects for each tool-material binding have to be available. The tool should not be able to know and create a special adapter object. It should merely use the abstract aspect class. One possible solution would use a "class object" of the abstract aspect class for the late creation of the concrete adapters. To this aspect class object we would then pass the respective material as a specification for the concrete adapter (see also the pattern *product trader* in Section 9.4.2).

Another more general solution would be the use of the *factory pattern* (see Gamma et al.) for creating adapter objects.

RATIONALE

We recommend that the adapter object solution for languages that use single inheritance without explicit interface construct and for large applications with different workplace types, where material objects migrate between these workplaces. This solution may also be combined with the aspect inheritance solution.



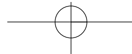
8.3.3 Construction Part: Using Development Tools to Realize Aspects

The previous sections introduce solutions that use aspect objects to build aspects. These solutions check at runtime whether or not there is an adapter object for a material in compliance with the underlying aspect. More specifically, each material offers operations for use by its aspects, and each tool uses only the material operations declared in the aspect. The part of this test that is really critical at runtime has to be done only once, namely when a tool or material is created. Once we can rest assured that a material offers all required aspect operations, or that a tool uses only aspect interfaces, then this situation will never change during that component's life cycle. All we have to do in addition is to ensure that tools and materials are bound correctly.

When using a language like Smalltalk, we can move expensive checks from the time an object is created forward to its class definition. Aspect compliance is then tested during the programming phase rather than at the program's runtime. This solution requires appropriate development tools, which shouldn't be a problem in Smalltalk, because all popular Smalltalk versions support an open development environment. Such an environment lets you modify browsers and program text editors so that aspects appear as independent categories that can be included in parsing.

A development tool for aspect editing purposes, say `AspectBrowser`, can be used to copy the operations of an aspect to material classes. This means that you can both use abstract superclasses to implement aspects and to define standard implementations for operations.

You can also use the `AspectBrowser` to define new aspect classes. Subsequently, you can implement the operations you copied, if they are available only in textual form, or replace a standard implementation from the aspect class by a material-specific one.



Finally, you can use the `AspectBrowser` to check that all operations have been implemented.

TRADE-OFFS

For this solution to work, the language you use should allow you to expand it during runtime. To take full benefit of the solution, the language should offer a powerful metaobject protocol (see Section 2.7). Notice that, even with the appropriate support by the programming environment, this solution is insecure, because the standard tools of the development environment do not know aspects, which can lead to an inconsistent system.

RATIONALE

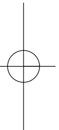
We recommend this solution only for Smalltalk or comparable languages that can be extended at runtime as an alternative to interface objects.

8.3.4 Construction Part: Alternatives to Using Aspects

Sometimes, tools use their materials directly instead of using aspects. In this case, a tool always knows the full interface of a material. Polymorphism is used only if the materials themselves form an inheritance hierarchy.

TRADE-OFFS

One major benefit of using this alternative to aspects is naturally that working without aspects will simplify the coupling of tools and materials, because no extra work is required conceptually or constructively to let a *single* tool use a *single* material. In addition, the coupling will be typesafe.



One drawback of this approach emerges when we try to let one tool use more than one material. In this case, we have to make some constructive effort and maintain several interfaces. If several tools are to operate on one material, it might become difficult to see which parts of the material's interface are suitable for which tool. Notice that the coupling of tools and materials is not made explicit in the design. Consequently, working without aspects will make it more difficult for us to change or expand our design.

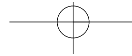
Nevertheless, there is enough justification for small and "young" projects to do without aspects. Designs that have not yet reached their maturity often show a one-to-one relationship between tools and materials. Languages that support dynamic typing, such as Smalltalk, often develop prototypes or initial pilot systems without aspects. In addition, extremely specialized materials may require a close tool coupling, that is the tool must know the full material interface. This applies also to special tools used for a single material.

And finally, aspects represent an abstraction of the work relationships between tools and materials, so that often they can be implemented only once we have a clear picture of these work relationships.

RATIONALE

We recommend using this solution when starting a new project and then creating aspects successively as required in the course of the project. This solution works equally well for simple tools and materials or very specialized ones.

Zu11-08.qxd 31/8/04 2:35 PM Page 202



8.4 THE SEPARATING FUNCTION AND INTERACTION PATTERN

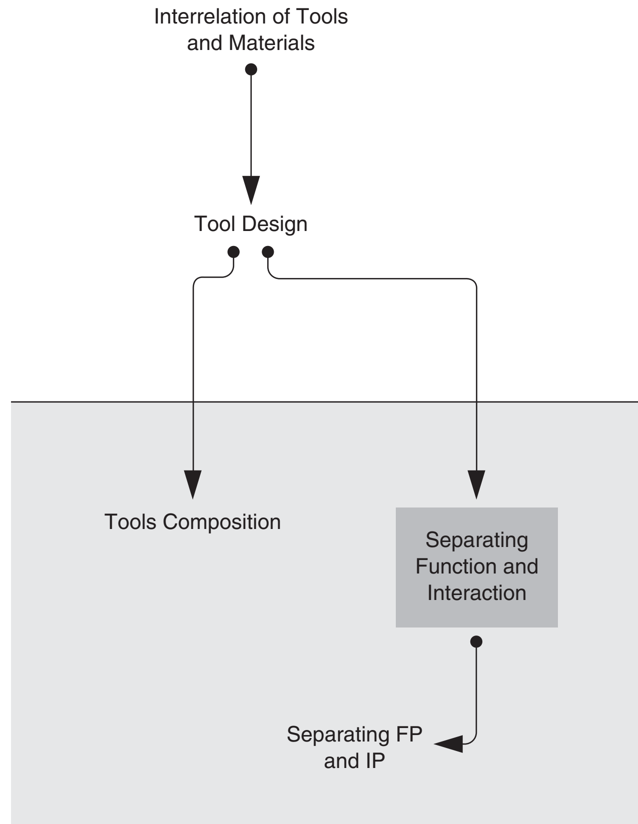


FIGURE 8.10 Separating function and interaction.

INTENT

This pattern describes how to utilize an important architectural principle of interactive software systems, that is, the separation of interaction and function as one essential design principle for interactive tools.

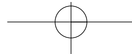
PROBLEM

We want to maintain the separation of concerns between a tool's handling and presentation on the one hand and on the other hand its functionality in software. We want to be able to modify the interactive parts of the tool without having also to adapt its functionality.

RELATE TO

The conceptual patterns *tool design* and *interrelation between tools and materials* show how to design tools and materials on a conceptual level (see Figure 8.10). When we have understood what a tool is good for and how users should be able to interact with it, we can deal with the problems addressed in the current pattern.

Zull-08.qxd 31/8/04 2:35 PM Page 203



SOLUTION

Design guidelines for tools

When implementing tools, we observe the following conceptual division:

- **Define the characteristic domain functionality for each tool independently of its shape. We call this its *function*.**
- **Define a specific shape and a characteristic handling and presentation for each tool based on its function. We call this its *interaction*.**

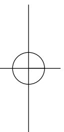
BACKGROUND: TOOL CONSTRUCTION

We said that we define software tools to be interactive. For this purpose, we assign a domain functionality as well as handling and presentation to each tool. To elaborate on these guidelines, we search for answers to the following questions:

- *Function*: What domain purpose does a software tool have?
- *Handling*: How can a software tool be used?
- *Presentation*: How can a software tool and the material it works on be represented?

Obviously, a tool must be good for *something*. It lets us handle tasks in that we can use it to work on suitable materials. This means that a tool has a domain functionality. The conceptual pattern *relating tools and materials* indicate how to design the functionality of a tool.

A tool never becomes active on its own; it is always handled by a user, so that handling is essential for a tool. The domain functionality of a tool is accessible only by handling it. The conceptual pattern *relating tools and materials* shows that the domain functionality can be utilized by different usage forms or interactions. This means that, although handling refers to functionality, it can also be implemented independently of the functionality.



For a tool to be usable, it has to have a representation of its own. Considering that a tool shows the state of the material it manipulates, the tool has to represent the material. This representation is the only way that a tool can give feedback for the user needed for a reliable working with tools and materials. Handling and presentation are closely related.

TRADE-OFFS

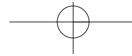
Responsibilities of an interactive tool

There are several ways to separate the function and interaction of tools. First, we can implement both concepts in a single construction unit by representing these two concepts as distinct interfaces (see design pattern *tool composition* in Section 8.5).

An alternative would be to further divide tools along the basic concepts. This will initially produce two construction units; we call them *functional part* (FP) and *interactive part* (IP). We have described the appropriate design pattern *separating FP and IP* in Section 8.7.

When further dividing a tool, we take the three fundamental *responsibilities* of an interactive tool into account. As mentioned earlier, each tool has a GUI representation that can be manipulated by users, in addition to its domain functionality. The original model-view-controller (MVC) paradigm found in Smalltalk systems uses separate classes for each of the three responsibilities. Considering that presentation and user input are closely coupled in modern GUIs, the *separating FP and IP* pattern combines the presentation and manipulation of user inputs, originally separated as *view* and *controller*, in a class—the interactive part.

We can identify a number of arguments in favor of each of the solutions introduced here for internal tool construction. For prototypes, simple tools, and small or young software projects, a separation based on the MVC paradigm or FP-IP model can



introduce too much overhead. In these cases, we suggest the use of monolithic tools instead (see Section 8.16.2).

However, the latter solution should be handled with care, because there is a tendency to miss the right point when a tool with its different responsibilities should be divided into separate classes. If this job is done late, it could become expensive.

For complex tools or large software systems, tools that were divided from the outset are much easier to maintain and reuse, compared to monolithic tools.

RATIONALE

Always conceptually separate a tool's interaction and function. For complex tools there should be distinct construction units; also, if the interaction is likely to change frequently. Otherwise, you could just represent interaction and function as different interfaces.

WHAT NEXT

The construction part using *components to build tools* of the design pattern *tools composition* shows how to represent interaction and function as two interfaces.

The design pattern *separating FP and IP* shows how to implement interaction and function as two construction units.

8.5 THE TOOLS COMPOSITION PATTERN

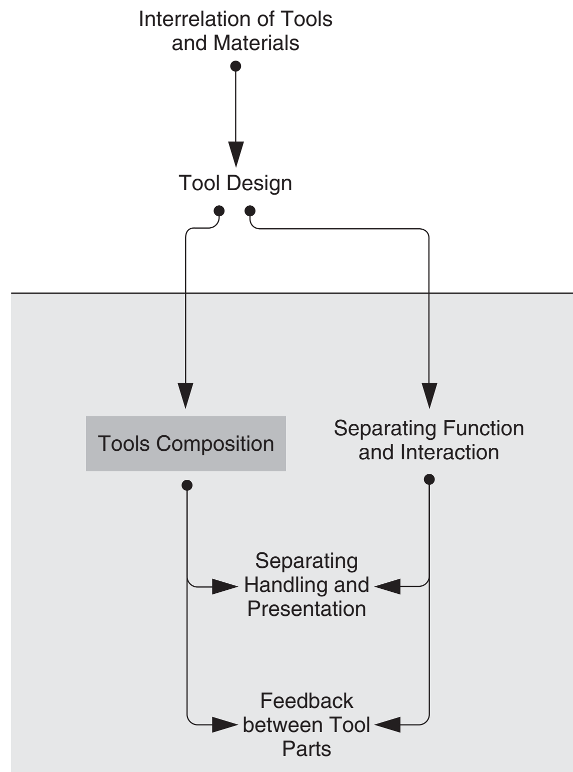


FIGURE 8.11 *The Tools Composition* pattern.

Zull-08.qxd 31/8/04 2:35 PM Page 205

INTENT

This pattern shows how you can build complex tools by composing subtools that are responsible for handling well-defined tasks.

PROBLEM

A tool, including its manipulation and presentation and its domain functionality, can be very complex, depending on the work and tasks. Using a few construction units to map this complexity to one single tool is normally not a feasible software solution. In addition, each tool has to be developed from scratch, even when we implement similar subsets of functions. For this reason, we should try to build tools on the basis of existing tool components to reduce the complexity of each component and make them reusable. Therefore

How can we divide a tool into subtools, and how should these subtools be integrated into one tool?

RELATE TO

The conceptual patterns *tool design* and *interrelation between tools and materials* show how to design tools and materials on a conceptual level (see Figure 8.11). Once we have understood how a tool is related to the individual tasks in the application domain, we can address the problem of how complex tasks or entire workflows should be handled by a composition of tools.

SOLUTION

We develop tools so that they are responsible for a defined task or set of activities. We integrate the subtasks of these simple tools to form a combination tool. We build a context tool that integrates several subtools. Each tool can basically be embedded in the context of an enveloping tool, thus becoming a subtool. For this purpose, we design a generic component interface.

The schema in Figure 8.12 shows the pattern and technical relations.

In our tools composition, we will ensure that context tools know their subtools and can call their operations directly.

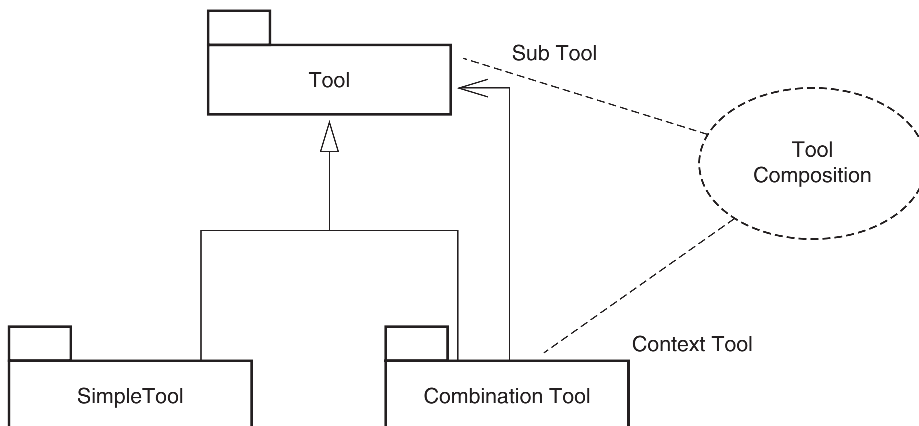
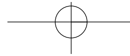


FIGURE 8.12
Building tools by composition.

Zull-08.qxd 31/8/04 2:35 PM Page 206



BACKGROUND: BUILDING SUBTOOLS

We design tools for the domain tasks on hand, that is, we normally design one tool for each task. We want to be able to combine tools for simple tasks to complete more complex interrelated tasks. Therefore, new tools should be built on the basis of existing ones. This allows us to reduce the complexity of single construction units and reuse existing components.

Tools designed to support complex activities are normally built so that they can manipulate one material and even material containers. The work that a tool is designed for can normally be divided into subtasks, that is, working with single materials and containers.

A taxonomy of tools

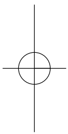
To combine tools from components, we first need to define several terms.

A tool used as a technical construction unit within another tool is referred to as a *subtool*.

A *context tool* embeds subtools, both from the technical and conceptual views, that is, it implements a domain combination tool.

A *combination tool* combines different domain services to complete a complex task. It is composed of subtools in software, and it is a domain element of the usage model.

A *simple tool* is domain-specific and represents one elementary task or functionality. On the software side, it has no subtools. Simple tools are also domain elements of the usage model.



The term functionality of simple tools has to be strictly separated from the term function. The reason is that simple tools do not implement functions the way we know from the structured design (e.g., create a record). Simple tools normally also have states.

Since we are talking about simple and combination tools in our domain design, we are now interested in how subtools and context tools relate. This composition can be recursive. For this reason, a tool can occur as a context tool for its subtools and be itself a subtool. In addition, we look for a view of tools as components to facilitate the composition.

EXAMPLE

The EMS example

Figure 8.13 shows an example for a complex tool—the `DeviceOrganizer` we know from previous sections. This example shows each room by a subtool. The actual room plan embeds these tools. The right-hand part of this figure shows a special room—the storage room, which is also represented by a subtool.

We could build this tool in different ways:

- We design separate tools for each task (a `DeviceHandler`), allocating employees to rooms (a `RoomHandler`) and managing the storage room (a `StorageHandler`).
- We build a complex tool that can display rooms and the storage room and run operations on these rooms and devices.
- We build one complex tool by combining the single components.

Zull-08.qxd 31/8/04 2:35 PM Page 207

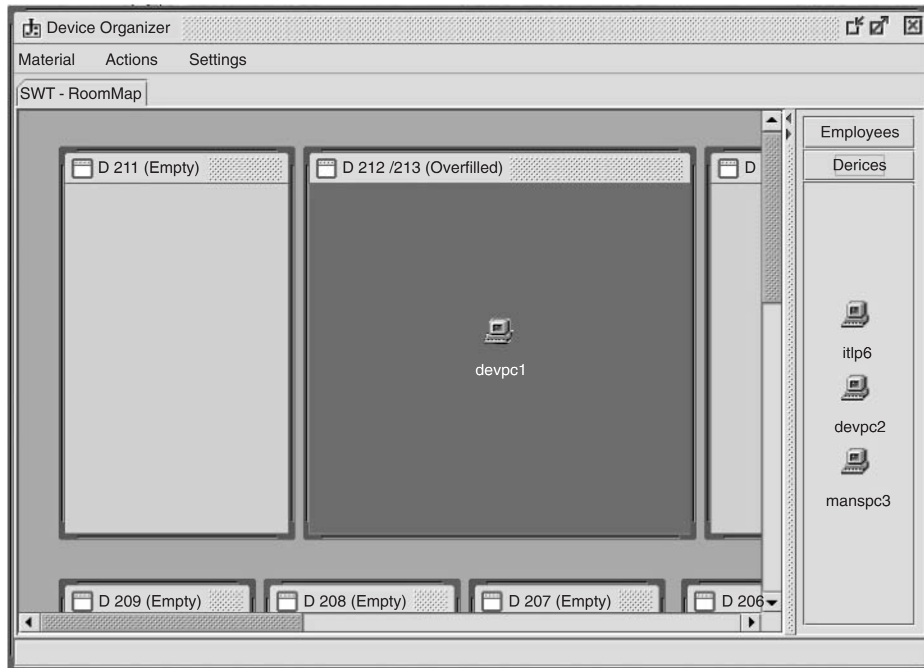


FIGURE 8.13 Example of a complex tool: the `DeviceOrganizer`.

In our `DeviceOrganizer` example, we have chosen the following solution. The `DeviceOrganizer` is a combination tool implemented by one context tool and several subtools, where these subtools know nothing about each other. They are controlled by the context tool so that the entire task of the combination tool, such as organizing devices in rooms, is fulfilled.

TRADE-OFFS

Design alternatives for related tools

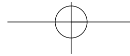
When building separate tools, we often find it difficult to establish a relationship between these tools. For example, when we move a device from storage to a room, then this involves both the room and the storage. To produce this relationship, we would either have to link the two tools or use their material to link them.

Linking the two tools directly means that one of the two tools (e.g., the storage handler) has to know the other tool. This is not a good solution, because the storage handler would lose its independence; it could be used only in combination with the `DeviceOrganizer`.

Using the material to link the tools is also ruled out by our construction principles, because the material would then know about the existence of the tools, or at least have to have some notification mechanism.

Though we could build one single complex tool, this solution is not appropriate from the software view, because we would not be able to use the single parts, that is the `DeviceOrganizer`, room handler, and storage handler, separately. For example, we would have to write a new room handler for the room planner to create and arrange rooms.

Zull-08.qxd 31/8/04 2:35 PM Page 208



For these reasons, it appears meaningful to encapsulate all subtasks in independent tool components. We could then compose tools based on the building block principle. This means that existing tools should

- be usable as *components* in new tools, that is, we can reuse their functionality for subtools in extended contexts; and
- whenever sensible, be able to act as *combination tools*, using the functionality of new components.

We are looking for a construction part where simple tools can be composed into combination tools based on a uniform schema.

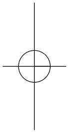
Rationale

Whenever you have to design and implement a tool you should think about reusing existing ones. Thus, for every tool design and construction, this pattern should be checked for applicability.

What Next

The design pattern *separating handling and presentation* shows how to further subdivide tools with an elaborated user interface.

The design pattern *feedback between tool parts* shows the basic principles and constructions of how to couple tool components in tool hierarchies.



8.5.1 Construction Part: Using Components to Build Tools

To enable an integration of tools as components, all tools have to support a common tool interface. Figure 8.14 shows a minimal tool interface.

In this example, the tool interface has to let us initialize the tool (`equip`). Subsequently, the tool can be activated (`activate`) and deactivated (`deactivate`) or closed (`close`). The tool's `Interaction` interface allows us to show or hide the tool representation. The `Functionality` interface supports all domain-specific operations on the tool and in addition, those operations required for tools and materials to interact—mainly setting and probing the material. Specific tools specialize this interface.

Trade-Offs

Building tools like components

We build tools so that they behave like components. This means in particular that their internal representation is hidden. This approach allows us to combine tools with different internal construction to one single system. It even allows us to combine complex tools from different other tools, which do not have to be built by the same internal schema.

On the other hand, this construction approach leads to a generic interface that may not fit perfectly in all situations. If, for example, we want to work with multiple materials, then we have to use `setMaterial()` for changing the materials. This is not very elegant and leaves the tool in an undefined state during the changes.

Zu11-08.qxd 31/8/04 2:35 PM Page 209

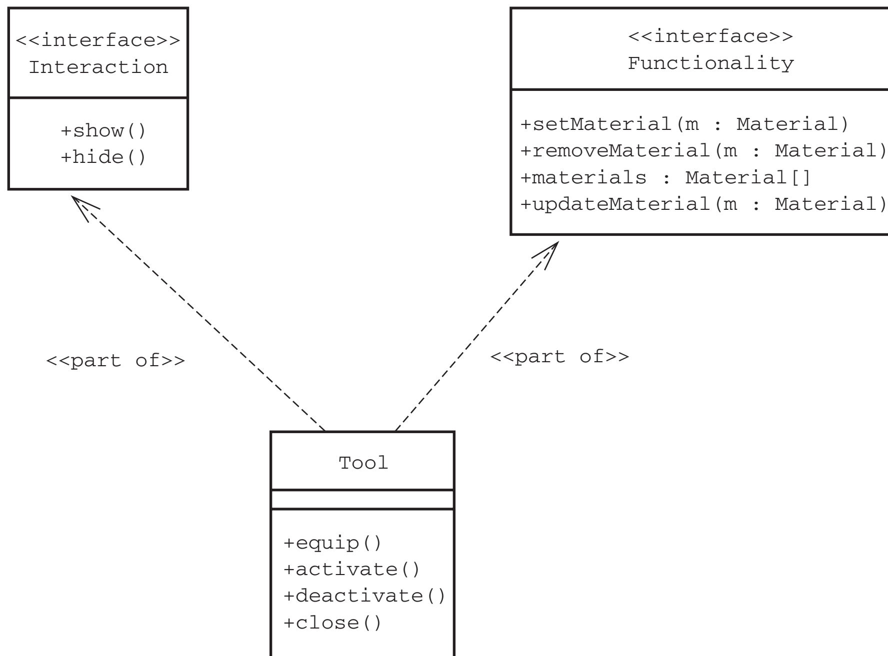


FIGURE 8.14
Minimal tool interface.

RATIONALE

Use this pattern whenever you have to combine subtools with different internal structures or reuse existing tools that are built without a clear separation of function and interaction.

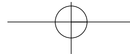
8.5.2 Construction Part: Using Components to Build Combination Tools

When building subtools and context tools that, together, form a combination tool, we have to answer the question of how the functionality of each of the above simple tools should behave within that combination tool.

Constructing combination tools

To implement a combination tool, we use simple tools that provide the functionality we need. Accordingly, the functionality of each simple tool is used as a subfunctionality. To combine several subfunctionalities, we build an additional context tool. This context tool implements the interaction of the subfunctionalities and delegates subtasks to the responsible subfunctionality. Remember that we are here talking of the conceptual functionality of a tool, which does not necessarily mean that there has to be an independent class for this functionality. The responsibility defined by a functionality can also be assumed by the tool class. In this case, the tool class would directly provide this functionality (see the discussion of monolithic tools in Section 8.16.2).

Delegating subtasks to subfunctionalities means that we clearly distribute tasks over components. This means that the context tool does not have to handle all subtasks itself. In one direction, from the context functionality to the subfunctionalities,



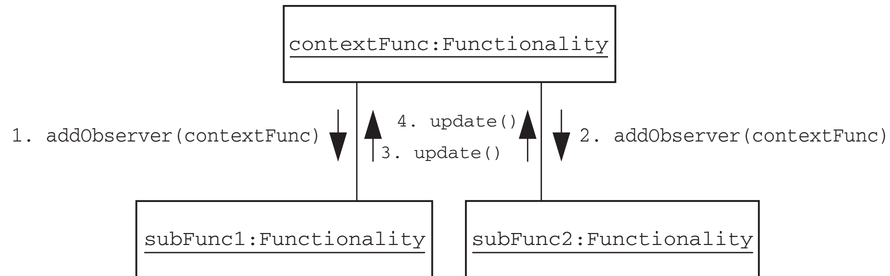


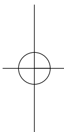
FIGURE 8.15 Feedback between a context functionally and its subfunctionalities.

there should be tight coupling, because the context functionality delegates specific tasks to the subfunctionalities, so it must know their interfaces.

Designing a context tool

For the context tool to be able to assume its coordinating function, it needs information about relevant changes in the subfunctionalities. On the other hand, we don't want subtools to know their context tools, which would cause cyclic dependencies and eventually make the system more difficult to understand and maintain. For this reason, we use loose coupling in this direction.

In general, subfunctionalities should be built to be independent of their context functionality, so that we can use them in different contexts. Since as we want to build reusable tool components, it will eventually not be clear when building a tool that it will be used as a subtool and integrated in a specific context tool.



The feedback problem

We develop single tools so that they can also be used as subtools. For this purpose, the tool functionality has to support a defined subtask. We use a context functionality to integrate subfunctionalities into a combination tool. The context functionality will then delegate subtasks to its subfunctionalities and coordinate them. In doing this, we use the event pattern or observer mechanism to solve feedback problems.

Figure 8.15 shows how the observer mechanism is used. In this example, the context functionality is the observer and the subfunctionalities are the observed.

EXAMPLE

The EMS example

In our EMS example with the `DeviceOrganizer`, the context functionality `DeviceOrganizerFunctionality` coordinates a subfunctionality, `RoomEditorFunctionality`.

When a user drags and drops a device from one room onto another room, then the `DragDropManager` removes the device from the source room and adds it to the target room, as shown in Figure 8.16. This causes the source room editor and the target room editor to send an event (using the `announce` operation) saying that their device sets have changed. The `DeviceOrganizer` has registered for this event and obtains the new device sets from the source and target rooms. The `DeviceOrganizer` can now have the screen representation updated.

RATIONALE

This pattern should be used to combine well-designed subtools into combination tools.

8.5.3 Construction Part: Identifying Tool Boundaries

We have seen how tools can be recursively combined to form complex tools. However, when looking at a tool component, this construction does not tell us automatically

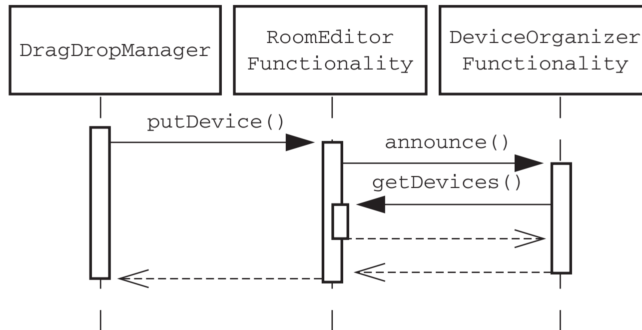


FIGURE 8.16
Control flow when adding a device.

whether or not it is the top-level context tool or an embedded subtool. But we can always decide at runtime whether or not a tool component marks the tool boundary. This means that this is the only tool component in this combination tool that has no higher-level context tool.

A tool has to fulfill certain software and domain responsibilities, which we want to manage in a dedicated instance. A good example for such responsibilities would be information about tool names, presentation icons, vendor names, and tool versions, or a list of aspects that the tool can manipulate. This information could be accommodated in the context functionality of a tool. On the other hand, we could argue that this information rather refers to the tool as a whole.

EXAMPLE

The EMS example

Figure 8.17 shows a class diagram for our combination tool, the `DeviceOrganizer`. The `DeviceOrganizer` has a functionality, `DeviceOrganizerFunctionality`, which formulates the domain handling of that tool. The `DeviceOrganizer` can have an arbitrary number of subtools of type `RoomEditor`. In turn, each `RoomEditor` has a functionality, `RoomEditorFunctionality`. The `DeviceOrganizerFunctionality` knows the `RoomEditorFunctionality` objects, while the context tool, `DeviceOrganizer`, does not know the other parts of the `RoomEditor's` tool interface.

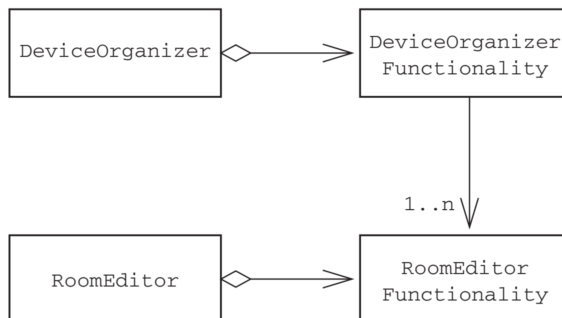
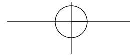


FIGURE 8.17
Functionalities in the Device Organizer.

Zull-08.qxd 31/8/04 2:35 PM Page 212

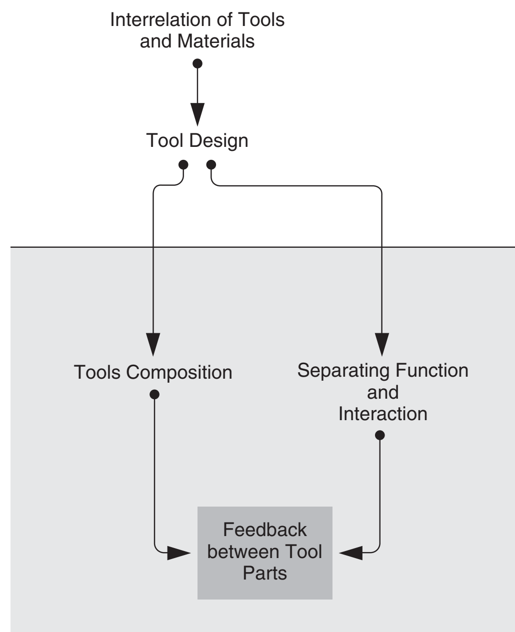


RATIONALE

Use this pattern whenever tools have to be represented by a single instance in your system and when a tool as a whole has to offer specific services that cannot be allocated to a tool part.

8.6 THE FEEDBACK BETWEEN TOOL PARTS PATTERN

FIGURE 8.18
The Feedback between Tool Parts pattern.



INTENT

This pattern tells you how to realize a feedback mechanism between tool parts, so that one part knows as little as possible about the other.

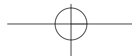
PROBLEM

When composing tools from single components, we often find that there is an asymmetric relationship between these components. A component, such as the context tool, knows its subtools from their interfaces. On the other hand, a subtool should know as little as possible about its context tool when state change messages are exchanged.

We can formulate the general problem that this pattern solves as follows:

Two components should be linked together so that one component is the service provider and the other is the client. The client is responsible for reacting to the

Zu11-08.qxd 31/8/04 2:35 PM Page 213



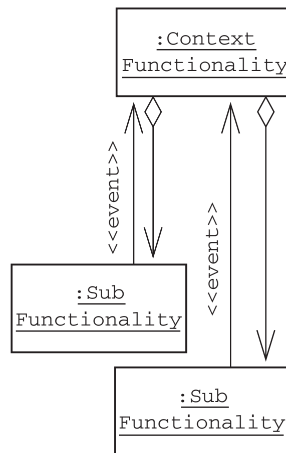
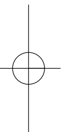


FIGURE 8.19
Feedback problem of a combination tool.

results of service requests. The client then needs to know how the provider's state has changed once its services have been used.

RELATE TO

This pattern solves a problem that emerges when you use the patterns *separation of function and interaction* or *tool composition* (see Figure 8.18).



BACKGROUND: FEEDBACK MECHANISMS

Let us look at a subfunctionality that should have a way to inform its context functionality about state changes. As mentioned earlier, we want to use loose coupling in this direction to avoid cyclic dependencies. This inversion of the control flow is called *feedback problem* (see Figure 8.19).

The literature describes several construction approaches and patterns to implement the required feedback mechanism. All of these construction approaches are either based on their use contexts or are bound to specific language mechanisms.

Feedback mechanisms can be distinguished by whether the observer is generally informed about the changes of the subject, or whether the subject informs its observer about state changes in specific ways. In addition, a feedback mechanism may be able to address a specific observer or an unknown number of observers arranged in a hierarchy.

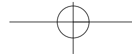
Event patterns initially appear suitable for our purpose of using and combining tool components. A similar solution would be the *observer pattern* proposed by Gamma et al., but if we have to deal with increasingly complex dependencies, it is better to use a variant of the event pattern, which includes explicit event objects. In contrast, we use a *chain of responsibilities* (Gamma et al.) for hierarchies of potential handlers. A component can then send a request to these unknown handlers.

SOLUTION

We build a feedback mechanism, which informs observers about changes to a subject in an abstract way so that these observers can respond to such a change.

The following construction parts spell out different concepts and related construction approaches to solve this problem on a more concrete level.

Zull-08.qxd 31/8/04 2:35 PM Page 214



8.6.1 Construction Part: Event Pattern

This construction part uses combination tools, composed of tool components. The *event* pattern will serve us as feedback mechanism (see Figure 8.20).

A subfunctionality informs its context functionality anonymously about each relevant state change. In this example, the context functionality assumes the role of an observer, while the subfunctionality acts as subject. If a tool is internally built from separate functional and interactive parts, then the event pattern can also be used by functional parts to inform interactive parts (see *separating FP and IP* in Section 8.7). In this case, the interactive part is the observer and the functional part is the subject.

An observer knows its subject and calls operations that change or probe it. The feedback mechanism works in the opposite direction, that is, the subject informs the observers about events when a relevant state change has occurred. In order to avoid an uncontrolled "firing" of events, we divide the operations at the interface of a class into *statements*, *requests*, and *tests* (see Section 2.1.8). We apply this rule to the interaction between context functionality and subfunctionalities:

Structuring the interface and interaction between functionality and sub-functionalities

- *Requests* and *tests* are probing operations (functions) with no side-effects. Requests inform the calling context functionality, for example, about the state of a material or about the configuration of a subtool. Tests are often used to show whether or not other operations of a subfunctionality may be called, such as to test pre-conditions for other operations. Probing operations must not cause a visible side-effect at the interface of a subfunctionality. In particular, they must not lead to informing the context functionality in its role as observer. For this reason, requests and tests can be called at any time as a response to a notification of the subfunctionality.
- *Statements* are operations (procedures) that change a state. They are used by the context functionality to have a subfunctionality manipulate a material or change the work state of a subtool. After a statement, the context functionality can call a probing function to check whether or not the last subfunctionality call was executed successfully. Most statements lead to a notification. For this reason, a statement should never be used by a subfunctionality as a response to an event it received.

TRADE-OFFS

Interaction between observer and subject

The example of the FP-IP coupling by means of the *observer* pattern shows a clear benefit of this anonymous notification: It is very easy to have arbitrary observers observe a subject at any later point in time.

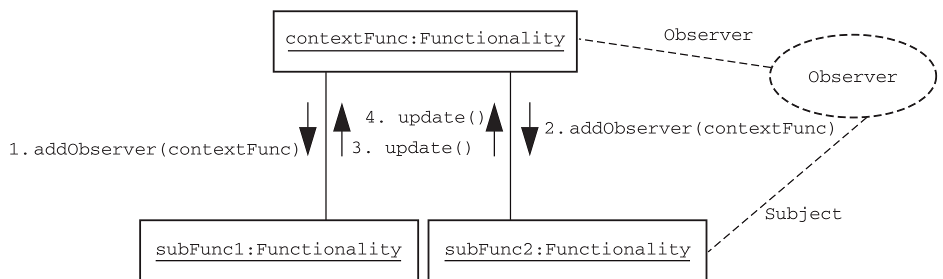
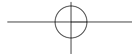


FIGURE 8.20
Using the Event pattern for combination tools.

Z011-08.qxd 31/8/04 2:35 PM Page 215

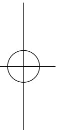


The subject (here the functionality) must not know how its observers respond to changes. All the subject should know is that the observers can potentially respond to changes. The idea is to have the subject inform the observers with minimum knowledge of the interface when their own states or the material state has changed. Considering that we don't deal with languages that have an anonymous signaling mechanism built in (e.g., Events in HyperTalk), we have to use the regular call mechanism for this notification.

Avoiding the "oscillation" problem

We have to take action to prevent an undesired "oscillation" between a subject and observers. The reason is that, when observers cause another state change to the subject as a response to being notified by the subject, then this leads to a notification. Obviously, this process could continue infinitely and never end. Another problem can occur when several observers register for the same event. In this case, observers that get notified later will not find the subject in the state that was signaled to them. They would then take wrong assumptions, such as calling operations on the subject that can be called in the signaled subject state, but no longer in the current subject state. We call these problems collectively "reactive change" and request that reactive changes be constructively avoided.

If, however, these rules are observed, then the event pattern can be used safely. But note that there are cases where we will combine a procedure with a function, such as for reasons of better performance or understandability. Then, the operation looks like a function but semantically is a procedure with a return value. In order to avoid confusion and reactive changes, we should carefully document this type of procedure and, perhaps, use naming conventions.



RATIONALE

This is the standard mechanism for a feedback mechanism with loose coupling of clients and related service providers. It works well for all simple tools, that is, tools with few different events.

8.6.2 Construction Part: Event Objects

We build an event class and create an independent event object for each relevant state change in the subfunctionality. We extend the subfunctionality's interface so that each event can be polled at the interface. The context functionality can register for events available at the subfunctionality's interface. It can optionally pass an operation together with the event.

The context functionality normally registers directly with each event. In many languages (e.g., C++ and Java), an operation called by an event can be passed together with an event more or less elegantly, that is, typesafe.

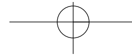
The EMS example

EXAMPLE

This section describes how we can implement the event pattern, extended to include event objects, for the `DeviceOrganizerFunctionality` and `RoomEditorFunctionality` in our EMS example (see Figure 8.21).

The class `RoomEditorFunctionality` implements the interface `EventSubject`. `DeviceOrganizerFunctionality` implements two interfaces, that is, `EventSubject` and `EventObserver`, where the latter indicates that `DeviceOrganizerFunctionality` is also an observer. None of the two classes has to implement operations. Note that `EventSubject` and `EventObserver` serve merely for typing and hiding the mechanism used to signal events.

Zull-08.qxd 31/8/04 2:35 PM Page 216



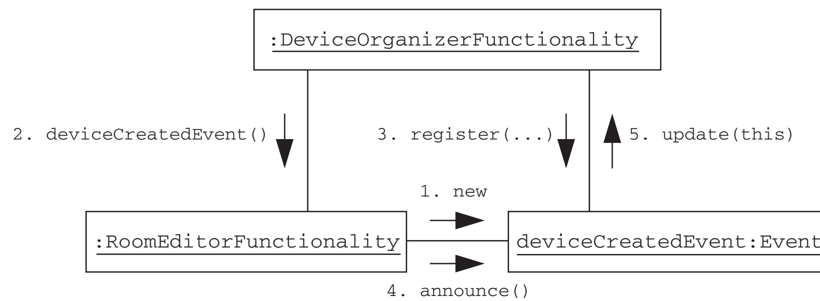


FIGURE 8.21
Even pattern with event objects.

At its interface, the `RoomEditorFunctionality` offers one operation for each event that it announces; in our example we only have the operation `deviceCreated()`. This operation returns a corresponding event object, which is created during the initialization of the `RoomEditorFunctionality` (step 1). The `DeviceOrganizerFunctionality` registers directly with this special event, `deviceCreatedEvent`.

The `DeviceOrganizerFunctionality` requests an event object for a specific change that it is interested in from `RoomEditorFunctionality` (step 2). The `DeviceOrganizerFunctionality` uses `register()` to register one of its methods with this event object, `deviceCreatedEvent` (step 3). As this is a Java example, the `DeviceOrganizerFunctionality` registers with the event using an anonymous inner class.

If `RoomEditorFunctionality` calls the operation `putDeviceProxy`, then the appropriate state change (e.g., adding a device to a room) will be executed. Next, `RoomEditorFunctionality` calls the `announce()` operation at the event object, `deviceCreatedEvent` (step 4). This means that, rather than informing all observers of `RoomEditorFunctionality` about an effected change, only the observers registered for this specific event object are informed (step 5).

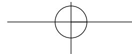
Once the control flow has reached the `DeviceOrganizerFunctionality` after calling `reactOnDeviceCreated`, `DeviceOrganizerFunctionality` can then respond to this event. It uses the operation that has been passed as a call back. This means that it knows the state change effected in `RoomEditorFunctionality`, so that it can poll the new values from `devices` and update its representation.

TRADE-OFFS

Complex tools and the observer mechanism

Complex tools show particularly well that the simple signaling of an observer can cause noticeable runtime effects. A considerable number of possible state changes need to be requested from a subfunctionality, or detected in the context functionality. This is the reason why different state changes to the subfunctionality should lead to different events, which can be distinguished early on, that is, in the subfunctionality. The context functionality can then register for specific events.

The different states of a subfunctionality need to be observable at its interface. We could develop a dedicated observer that monitors changes specified at a subfunctionality's interface.


Zull-08.qxd 31/8/04 2:35 PM Page 217

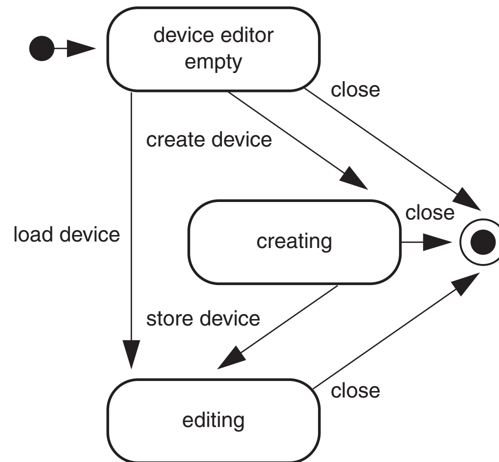
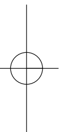


FIGURE 8.22

State model for the Device Editor in *the* EMS.

In practice, we often find that subfunctionalities and their events are strongly tailored to existing observers. This means that subfunctionalities offer only events needed by existing observers. On the other hand, it also means that a subfunctionality is more strongly coupled to observers, so that it may be difficult to replace observers later on. To avoid this implicit dependence, we can model the states and events of a subfunctionality solely from the domain view. In this respect, the design of state automatons for subfunctionalities has proven to be a useful technique (see Figure 8.22). For this purpose, we define the domain states that a subfunctionality can take, as well as the changing operations that initiate state transitions. Each state transition will then be an event. For example, we can deduce the events called `deviceCreated, deviceLoaded`, and `deviceStored` from the state diagram shown in Figure 8.22. This is a nice way to design subfunctionalities that are independent of specific observers.



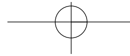
RATIONALE

This is a rather elaborate feedback mechanism for complex tools with several events and a broad probing interface of the observed subject.

8.6.3 Construction Part: Chain of Responsibility

When using the event pattern between subfunctionalities and their context functionality, we often find that there are messages sent from a subfunctionality to its context functionality, which are not caused by a state change of that subfunctionality. These are messages by which the subfunctionality signals that it cannot supply a requested service. In addition, it is often difficult to identify the entity that can actually supply a requested service. To allow sending such messages, we can build a *chain of responsibility* between a subfunctionality and its context functionality; this chain can then be used to send such messages.

Zull-08.qxd 31/8/04 2:35 PM Page 218



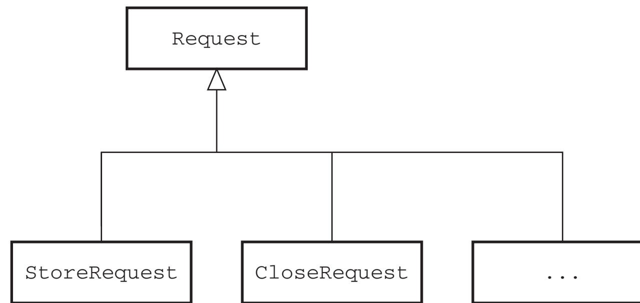


FIGURE 8.23
Often used *Requests*.

A chain of responsibility adds functionalities to a tool tree, in addition to the event pattern. Requests can then be sent along the chain of responsibility. We represent requests in a separate class, say `Request`, and we can then derive more specialized request classes. For example, we could introduce a special request for a closing procedure (see Figure 8.23).

BACKGROUND: CHAIN OF RESPONSIBILITY VERSUS EVENT PATTERN

For example, if a user wants to close a tool, he or she will most likely click a button representing that subtool on the screen. However, the subtool cannot terminate itself or its context tool. Also, it does not know whether or not the context may reject such a closing attempt. Therefore, the subtool has to delegate this task to its context. Depending on the construction, the context can consist of a context functionality, a tool object, or the work environment.

If we select the event pattern to let the subtool's context send a close request, then we will violate several concepts of this pattern:

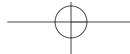
- An event is sent, although the functionality's state has not changed.
- The event pattern should be used exclusively for signaling, but not to request services from the context, in the sense of an operation call.
- If a subtool wants to close, then the observing object will not simply delete the subtool from the memory. It will normally delete the subtool *and* call several cleanup operations. For example, in C++ we would automatically call the destructor. So we would have a reactive change leading to a system behavior that is hard to control.

EXAMPLE

The EMS example

Let's see the above idea in our EMS example. The `DeviceEditor` can edit several devices concurrently. If a user closes a card tab, then the corresponding subtool sends a request announcing that it wants to be closed (see Figure 8.24, step 1). The context tool responds to this request by closing the subtool (step 2). If the user closes the context tool itself, then the context tool will send a request to the environment (step 3), causing the work environment to close the entire tool (step 4).

Zuli-08.qxd 31/8/04 2:35 PM Page 219



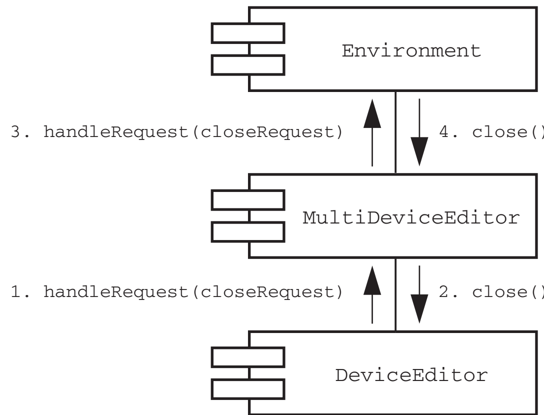


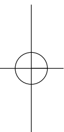
FIGURE 8.24
Chain of responsibility for our EMS example.

TRADE-OFFS

Regardless of whether or not you use a chain of responsibility, deleting objects can cause problems in some programming languages, in particular when there are still operation calls on objects on the callstack that are to be deleted.

Using C++

In languages like C++, where you have to delete objects explicitly, problems often arise while the callstack is being processed. To avoid having to remove the callstack of objects that have already been deleted, we could integrate a trash bin object into the system. When you then delete objects, they will be moved to that trash bin, while the callstack can be further processed, and observers can be deregistered. Objects in the trash bin will be actually deleted from the system when you empty the trash bin. We place the trash bin emptying task before a new request to the event loop of the window system, because we can assume that, at this point of the control flow, there will be no more active objects.



Using Java

In a language like Java, which has its own garbage collector, we don't need a dedicated trash bin to delete tool hierarchies. Java's garbage collector only deletes objects that can no longer be reached by a thread running in the virtual machine. This prevents the deletion of objects that are still referenced in the callstack.

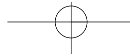
RATIONALE

Use a chain of responsibility for passing requests to the context. Whenever a subfunctionality or a subtool or another subordinate element cannot handle a request for service on its own, this is the right construction approach.

8.6.4 Construction Part: Tool Component with Reaction Mechanisms

The component model for tools allows us to compose combination tools from simple tools. However, for a fully generic tool interface, we have to integrate reaction

Zull-08.qxd 31/8/04 2:35 PM Page 220



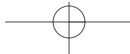
mechanisms. Figure 8.25 shows these interfaces, which support both the event pattern and the chain of responsibility.

RATIONALE

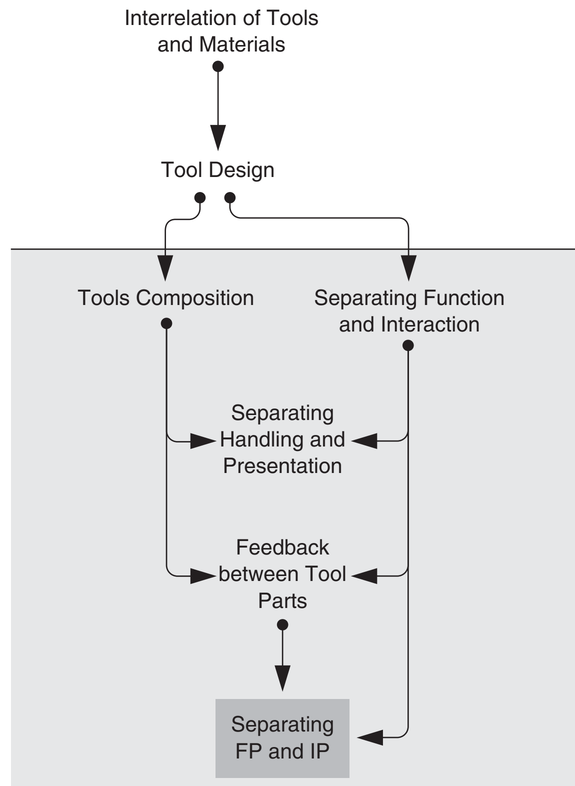
Use this construction part as a reaction mechanism when you assemble combination tools from simple tools.

FIGURE 8.25 A tool component with reaction mechanisms.

Zu11-08.qxd 31/8/04 2:35 PM Page 221



8.7 THE SEPARATING FP AND IP PATTERN



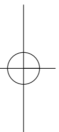


FIGURE 8.26

Separating FP and IP pattern.

INTENT

This pattern details the essential concepts for designing an interactive tool. Even if you decide to implement the function and interaction of a tool as one construction unit, you should understand the principle behind this pattern.

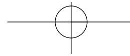
PROBLEM

There are a number of different ways to constructively implement the conceptual division of function and interaction.

The pattern *separating function and interaction* that we previously introduced described the basic tool design principles. When developing a tool, we will not see the usage quality of its handling and presentation before it is actually used. Since requests for changing the handling and presentation of a tool will arise from its use, we want to implement the requirements *for different handling and presentation in software* without having to adapt the domain functionality.

The presentation of a tool depends largely on the GUI elements, so we also have to take the changes in the GUI into account. We want to change the interaction

Zull-08.qxd 31/8/04 2:35 PM Page 222



of a tool by exchanging window systems or GUI frameworks, without affecting the function. Therefore,

We need an internal division of interaction and function into two construction units, which allows us to exchange the tool interaction without impact on the tool functionality.

RELATE TO

This pattern directly follows the pattern *separation of function and interaction* or *tool composition*. It leads to the feedback problem already addressed by the pattern *feedback between tool parts* (see Figure 8.26).

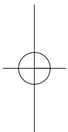
SOLUTION

We divide a tool into a *functional part* (*FP*) and an *interactive part* (*IP*). We combine FP and IP so that an IP knows and uses its FP, while the FP knows as little as possible of its IP. For this purpose, we use a suitable feedback (or reaction) mechanism.

The FP implements the functionality of a tool, while the IP implements the tool's interaction. We loosely couple the two parts, so that the primary direction of the control flow, from the user into the system, is observed, thus maintaining the FP's independence of IP.

Task division between FP and IP

The following division of tasks for FP and IP results from this general separation of function and interaction:



- The *functional part* (FP) is the acting and probing part of a tool. It defines the domain functionality of a tool. The FP handles the material and knows the work context supported by that tool. To handle or manipulate materials, the FP uses operations specified in one or more aspects. To be able to support the work context, the FP manages a work state—the *tool's memory*.
- The *interactive part* (IP) defines the tool's user interface. More specifically, it accepts events, calls the FP, and controls the GUI presentation. To allow the IP to abstract from the concrete interactions and the window system used, it normally uses so-called *interaction forms*.

FP is the starting point

The T&M approach pays much attention to discussing tools and their domain interactions with the future users. In this context, it makes sense to design the FP of a tool first. The next step then defines the way a tool should be handled and the desired presentation. This development approach ensures that all tools are motivated by the application domain, and that the responsibilities of FP and IP are separated.

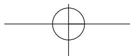
Notice that this guideline does not conflict with our use of prototypes, such as when decisions are taken about how to proceed in the design based on presentation prototypes. The reason is that these prototypes, essentially showing representation and manipulation aspects, are built on the basis of a domain tool design, forming our platform for discussions with future users.

EXAMPLE

The EMS example

In our EMS example, the *material* is the room plan as a collection of rooms. The user wants to add a room to this collection. To do this, the user uses the *RoomMap Designer* (see Figure 8.27). This RoomMap Designer shows the rooms, and its menu can be used to create new rooms and edit existing ones. It creates and displays a new room when the user selects the *NewRoom* option from the *Actions* menu.

Zull-08.qxd 31/8/04 2:35 PM Page 223



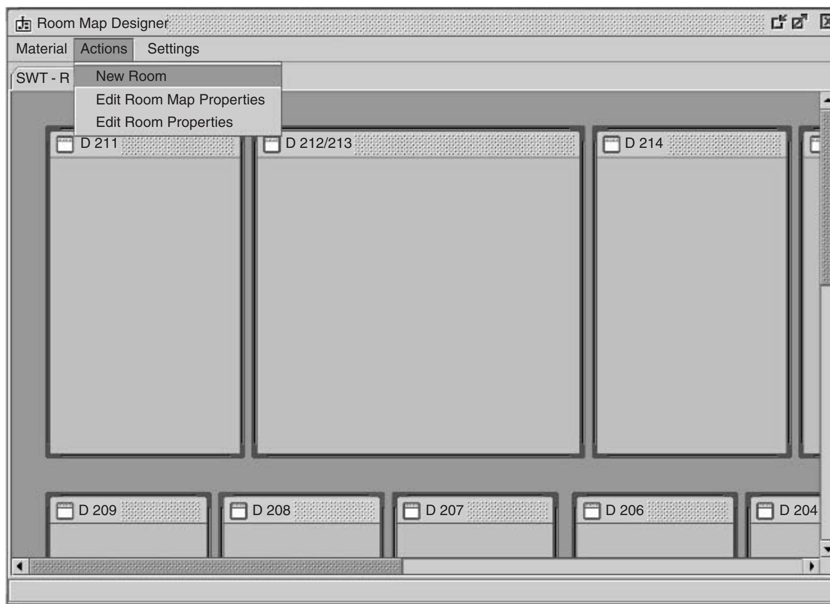
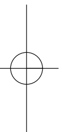


FIGURE 8.27
The RoomMap Designer from our EMS example.



The interaction form, `activator`, converts the system event (e.g., *NewRoom)* into a program event and sends it to the IP. The IP calls the FP's operation, `addRoom`. Next, the FP adds a new room to the room plan. This means that the user's action has led to the desired material manipulation. What's missing now is some feedback about the action's success, where the following construction guidelines are important.

The IP decides whether and how it wants to represent the modified room plan. The IP does not automatically add a new room to the rooms represented on the screen, and it does not represent that change immediately. It needs information that the FP actually created a new room. In addition, it does not know what standard name the FP used for the new room.

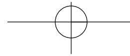
Displaying state changes

On the other hand, the FP does not automatically display the new room when it calls an IP operation, because the IP is responsible for this task. The FP does not have information about the display of a new room. It merely offers a list with information about all the rooms in the room plan. This information is available by calling a probing operation. The list is completely separate from the material, `RoomPlan`. The FP decides about the form it wants to use to create the list with room information from the room plan. Consequently, we need a feedback mechanism. The event pattern has proven useful in these cases, too (see Figure 8.28).

BACKGROUND: A TOOL AS A REACTIVE SYSTEM

If you look at it from the technical perspective, a tool is built as a so-called *reactive* system, that is, each tool activity is triggered by an explicit user action, such as clicking the mouse or pressing a key (see Figure 8.29, step 1). These actions are directed to the tool in the form of a stream of events (step 2). The tool reacts to each user action. To allow the tool to do this, these events are interpreted by the IP. The IP converts these events into FP calls or into a different presentation (step 3). The FP uses appropriate aspects to handle the material (step 4). After manipulating the material the FP anounces the state

Zull-08.qxd 31/8/04 2:35 PM Page 224



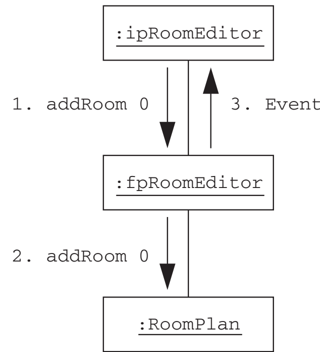


FIGURE 8.28 Feedback problem between FP and IP.

change (step 5). The IP reacts on the announced event and updates the presentation (step 6), which is recognized by the user as a program reaction (step 7).

TRADE-OFFS

The division of FP and IP described so far specifies responsibilities within a tool that improve the legibility of your design. In addition, it supports two important design goals: flexibility and reusability.

Visual programming

A tool's user interface can normally be changed independently of its functionality. Given that the domain components of a system, that is its functional part, aspects, and materials, do not make any assumptions about the interactive environment in which they are embedded, they can easily be ported to different system platforms.

Many development environments offer attractive support for visual programming. GUI tools allow you to simply drag and draw GUI elements to your layout. You can then gradually add more functionality to these GUI elements. Remember that we have already mentioned some risks inherent to this design method (see Section 7.4).

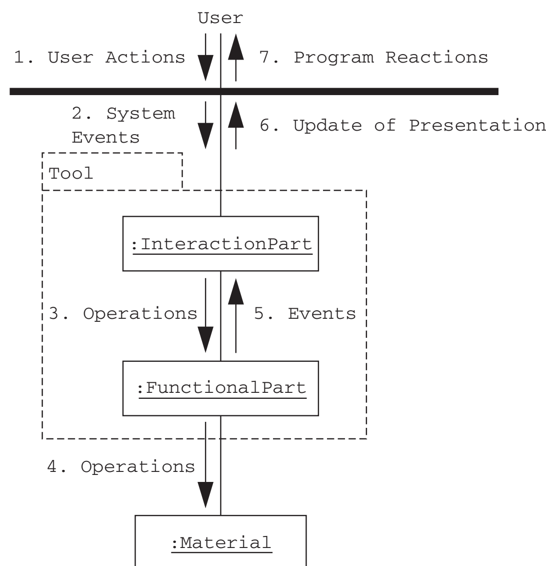
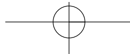


FIGURE 8.29 A tool can be seen as a reactive system.



In connection with tools construction, this method normally leads to tools that are developed with the interaction part (IP) as its basis. Our experience has shown that this does not always take fully into account the domain contents of a tool. In fact, the tool could easily become a "window" on the material, which then appears to be manipulated directly. The tool itself often has no own elaborate domain functionality, beyond that of the material.

In addition, this method leads to a one-to-one relationship between tool and material. In such systems, all tools have the character of an editor. For example, you can use a tool to read attributes from a material, or set such attributes, and save changes to a material. Though these applications use object-oriented technologies for implementation, they do not utilize them to the user's benefit. Basically, such systems are not much easier to use than conventional software systems.

RATIONALE

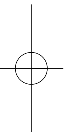
Whenever you have to design and implement a medium-sized complex tool that has a good chance of needing frequent changes to its handling and interaction, this pattern provides the basic design and construction priciples.

WHAT NEXT

The design pattern *separating handling and presentation* shows how to further subdivide a tool's interaction part to encapsulate the actual GUI used.

8.7.1 Construction Part: Interactive Part (IP)

Characteristics of an IP



As already mentioned in the previous Section 8.7, we build a separate class that encapsulates all IP tasks for each specific IP. Before we continue discussing this construction approach, it will be useful to understand a few facts about IP.

An *interactive part* takes a stream of system events, triggered by user actions, as program events and interprets them. In this respect, it converts presentation-specific events (e.g., scrolling in a list or menu), while passing application-specific events to its FP.

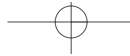
An *interactive part* implements the manipulation and presentation of a software tool. It can call the widgets of the underlying window system or use generic input and output components, the so-called *interaction forms* (e.g., 1:n selection, activator), and call these interaction forms for representation and user input.

Returning to our discussion, the IP's tasks are specified by its responsibility for interaction within a reactive system. Each user action arrives over input channels as a system event (e.g., the user pressed the left mouse button at a specific point on the screen) at the event context of a tool. Modern systems use an event dispatcher, that is, the normal distribution mechanism for system events in window systems, to route system events to the appropriate event context.

TRADE-OFFS: SEPARATING TASKS BETWEEN IP AND INTERACTION FORMS

System events do not directly reach a tool's IP from the outside. As one option, the IP can use building blocks, the interaction forms introduced above, to react to interaction events. Interaction forms encapsulate the specific window system and convert system events into program events. *System events* are events generated by the system base, while *program events* are events generated by the application program. The general idea is to abstract the design of interactions from the specific system base. The IP can be

Zull-08.qxd 31/8/04 2:35 PM Page 226



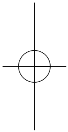
thought of as a shell around interaction form objects. Naturally, the IP can also interact with the widgets of a window system. Even a combination of direct widget calls and the use of interaction forms is feasible.

The main task of an IP is to interpret incoming events and manage the user interface. Interpreting incoming events means that the IP decides whether or not an event is to be passed on as an FP call, or whether it entails a change to the GUI presentation.

A specific implementation of widgets or interaction forms and their arrangement based on a specific layout (horizontal, vertical, proportional, etc.) does not have to be handled by the IP in modern window systems. In fact, this part of a tool design can be specified separately by the use of a GUI builder or similar development tools or generators. The GUI resulting from this method is normally linked with the tool at runtime.

Trade-Offs: Separating Tasks between IP and FP

To maintain a fair division of tasks between IP and FP, the IP must not make any assumptions about the logical relation of FP operations or results from operation calls. This method ensures that the IP will not automatically change the GUI representation of information after an altering FP operation has been called. This is important, because it would mean that the IP knows the effect of an operation call on the FP state, thus breaking the semantic encapsulation of operations in the FP. Again, we must deal with the general feedback problem. An appropriate solution was introduced in our discussion of the event mechanism in Section 8.6.



Rationale

Whenever you have to design a complex tool with an elaborate user interface, it is useful to implement the interaction part as a separate component. When the application is supposed to work over a long period of time, you may consider abstracting interaction design from the actual GUI system by means of interaction forms.

8.7.2 Construction Part: FP

The construction approach discussed in this part uses a separate class that encapsulates FP tasks for each specific functional part (FP). First, let's look at the terminology.

A *functional part* (*FP*) implements the domain functionality of a software tool. It manipulates material via the interfaces of aspects.

An FP uses probing operations to provide information about its own working state and that of a material. It manages its own working state and the tool memory, depending on user actions and material states.

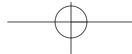
In the construction approach, the FP is the acting and probing part of a tool, where the services of a tool are implemented. The FP defines what application-specific activities can be performed by a tool.

Responsibilites of FP

The FP encapsulates the following design decisions:

- *Access a specific material*: When a material is to be manipulated, the IP always calls the FP, passing information to the latter. In turn, the IP uses a material-independent FP interface to obtain information for presenting a material. The IP does not directly access a material. In some cases (e.g., to handle complex tabular materials), the IP may be granted reading access to materials.

Zu11-08.qxd 31/8/04 2:35 PM Page 227



- *Changes to the material*: The IP can obtain information about the working state of a material to the extent that such information is provided by the FP, which means that a material never causes a representation to change.
- *Managing a work context between different materials and a tool*: The FP ensures for a work context that all participating materials are edited consistently. For this reason, the IP cannot at any time call the FP's altering or proving operations. It can call such operations only provided that the FP is in a suitable work state. Tests can be used at the interface to identify the FP's work state.

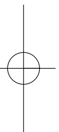
TRADE-OFFS: SEPARATING FP AND IP

In order to ensure the separation of interaction and function, the functional part should not make any assumptions about the handling and presentation of the interaction part. This is in line with our discussion about the responsibilities of the IP. An FP should know nothing about the ways and means by which a user interface and its interactions are realized. This, by the way, facilitates testing the functionality.

According to the general design principle, the FP never calls the IP directly to cause a change of the representation. Here we have another instance of the general feedback problem discussed earlier in Section 8.6.

RATIONALE

When you decide to implement the interaction part and functional part as separate components, you should consider this construction part.



8.8 THE SEPARATING HANDLING AND PRESENTATION PATTERN

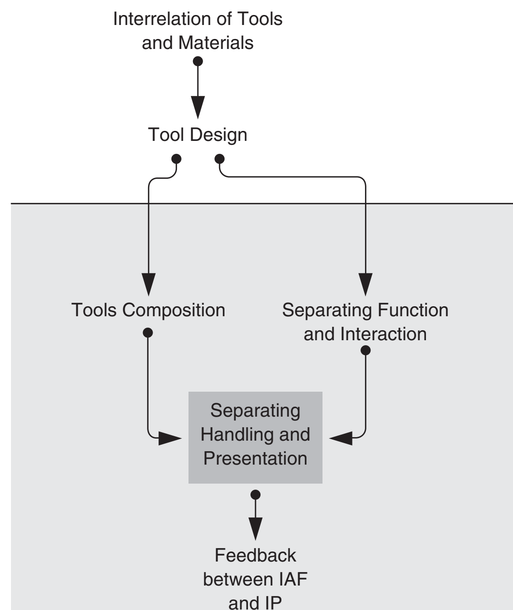
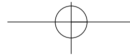


FIGURE 8.30
The *Separating handling and presentation* pattern.

Zull-08.qxd 31/8/04 2:35 PM Page 228



Intent

This pattern details the essential concepts of designing the interactive part of a tool. Even if you should decide to implement the interaction of a tool as one construction unit, you should understand the general principle behind this pattern. This helps clarify the different concerns covered by an interaction.

Problem

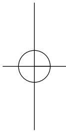
We know from previous discussions on the *separating interaction and function* pattern that it is meaningful to separate the functionality of an interactive tool from its interaction. Since we are dealing primarily with the construction of interactive workplace systems, the interaction will normally be initiated by users and implemented on a graphical interface. There is a large number of commercial and public libraries and development components for the design of graphical user interfaces. We collectively call them *user interface toolkits*, or *toolkits* for short.

How can we encapsulate a user interface toolkit so that we reduce the dependence of our IP on a specific toolkit to a minimum?

Relate To

The design patterns *tool composition* and *separation of function and interaction* provide the conceptual background to understanding this pattern (see Figure 8.30).

Background: Separating Handling and Representation



The most important task of an application developer is to convert the domain functionality of a tool into interactions and then use the latter in combination with graphic components to implement suitable handling and presentation forms. Such graphic components are readily available in many toolkits, such as Motif, TCL/TK, or the Swing framework. These graphic components are often called *widgets*, and they are used to create graphic user interfaces. Using these turnkey components to build your user interface has several benefits. One major benefit is that you can quickly implement complex user interfaces at relatively little programming cost. In addition, turnkey components contribute to a uniform look and feel for your user interfaces, facilitating the use of your application system. For example, Java's Swing framework even let's you replace the look and feel of a GUI without having to change the toolkit.

Let's first see how the components of a toolkit may be used to build tools (see Figure 8.31).

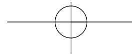
Designing the GUI of an IP

The user interface of an IP is built by combining elements from the toolkit. In doing this, we have to observe the following important points:

- Each toolkit makes assumptions about the control flow within an application. This may conflict with your ideas about the tools to be implemented.
- The way widgets are linked with the IP is defined by the toolkit developers. This means that, depending on the toolkit, different system events may have to be linked with callback operations. These callback operations have to be implemented in the IP.

Another motivation for loose coupling of IPs to the toolkit you use is the latter's volatility. The historical development of toolkits has shown that the interfaces of toolkits have changed, often to the point where they were actually completely redesigned, while other toolkits disappeared from the market. One example is Java: the

Zull-08.qxd 31/8/04 2:35 PM Page 229



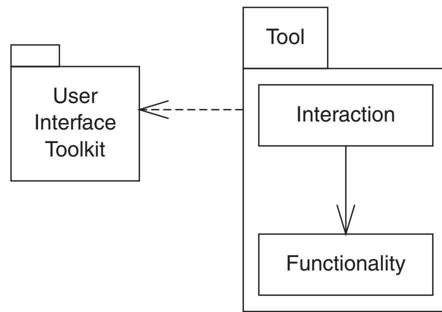


FIGURE 8.31
Linking a tool to a GUI toolkit.

introduction of the Java Foundation Classes and the related Swing toolkit almost entirely replaced the former GUI library, the Abstract Windowing Toolkit (AWT).

Exaple for a toolkit library

Figure 8.32 shows a segment of the *Swing* library's class tree in Java 2. Important features of this library include components arranged in an inheritance tree. Notice that inheritance is used mainly to be able to reuse an implementation, as in other toolkits. Such inheritance hierarchies generally violate the type-subtype hierarchy in many ways. For example, you often find operations that can be called in many but not all subclasses in the root class of the toolkit. In some cases, calling such operations can either lead to a runtime error, or an exception is thrown, or the call does not lead to any result at all.

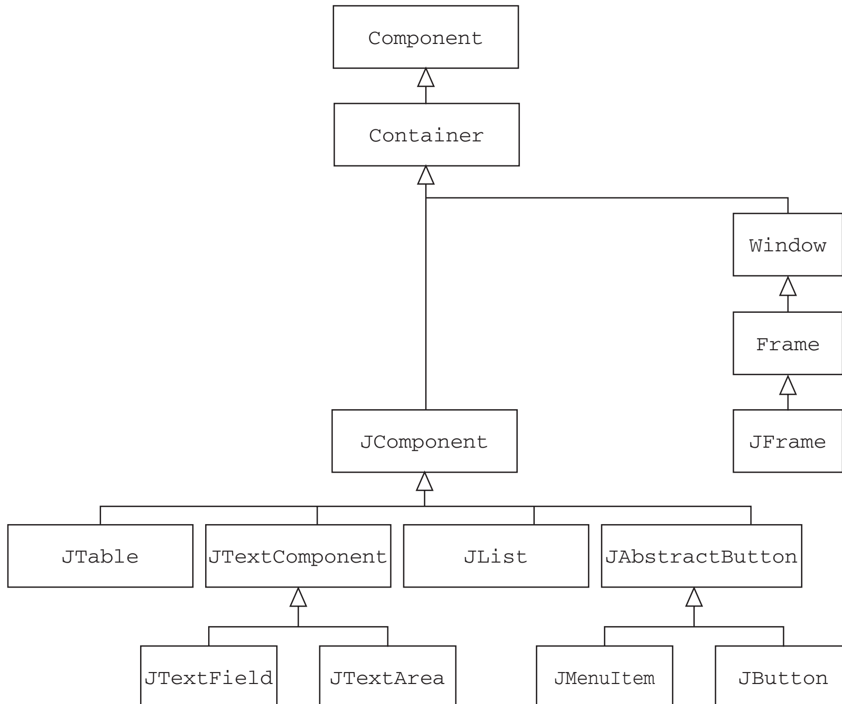
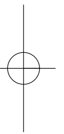
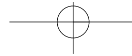


FIGURE 8.32
Components of the Swing library (excerpt).



Zull-08.qxd 31/8/04 2:35 PM Page 230



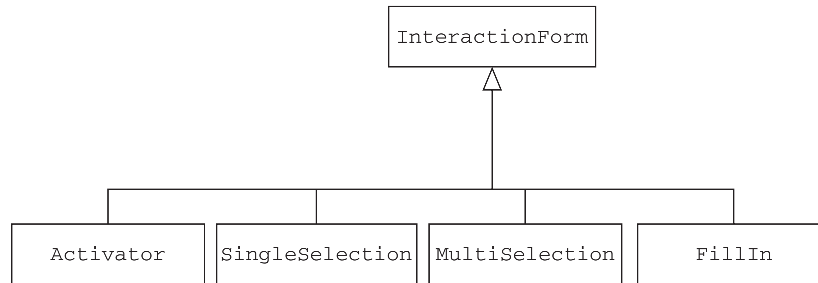


FIGURE 8.33
Examples of common interaction forms (IAFs).

Solution

When building the IP, we want to ensure that it is not dependent on the toolkit we use. Rather, we want to abstract the GUI design from a toolkit. For this purpose, we introduce *interaction forms*.

The solution proposed in this section reduces the dependence of an IP on a specific toolkit.

We encapsulate different types of potential manipulations for a tool in interaction forms. Next, we compose an IP from the instances of our set of interaction forms. And finally, we ensure that the IP exchanges only domain values with its interaction forms.

Let's first look at some terminology to better understand our idea.

- An *interaction form* (IAF) is an abstract form of handling a tool. It represents a domain way of using a tool and has no side-effects on that tool.
- An IAF represents and returns only domain values, which means that is has no global effect. An IAF is used by an IP. An IAF's interface does not make assumptions about a toolkit, which means that the IP is totally independent of a selected toolkit.

Figure 8.33 shows examples of common interaction forms (IAFs).

When building an IP, developers normally select interaction forms based on their decision about how the functionality of the FP should be converted into useful user interactions. Developers then create one object out of the set of available interaction forms for each type of user interaction.

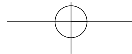
All information represented by an interaction form on the user interface, and the results this IAF returns, are domain values. These domain values are supplied and accepted by the IP. This allows an interaction form to run stateless and without side-effects.

Of course, an interaction form has to be represented at the user interface. On the other hand, the type of widget used from a toolkit to represent an IAF at the GUI is not important for an interaction. Also, the interaction form is not interested in the size and position in which it is represented. This information can be encapsulated in presentation forms. To better understand this, we discuss the following terminology:

A *presentation form* (*PF*) implements the specific representation and handling of a domain interaction form (IAF) at a tool's user interface.

Presentation forms encapsulate widgets of a GUI toolkit, implementing a protocol that allows us to link it with interaction forms.

Zu11-08.qxd 31/8/04 2:35 PM Page 231



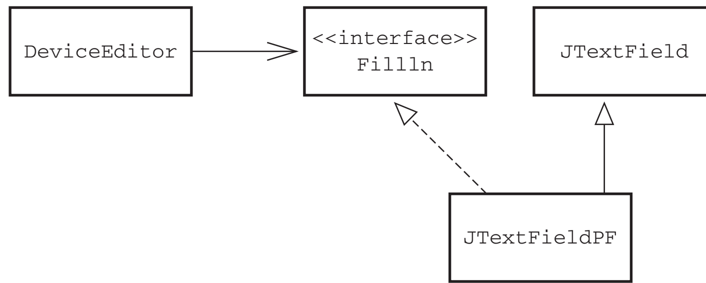


FIGURE 8.34

An interaction form, Fillin, is represented by a JTextFieldPF in the Device Editor.

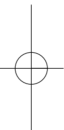
Presentation forms are managed outside a tool, but they are linked with interaction forms inside a tool.

Depending on the GUI toolkit we use, the widgets of the toolkit could also assume the role of presentation forms. In this case, there is no need to program additional presentation form classes.

EXAMPLE

The EMS example

Returning to our EMS example, we want to see how a `DeviceDescription` is filled out in the `DeviceEditor`. Figure 8.34 shows an interaction form used to fill in a value, `FillIn`. We used such an interaction form in the `DeviceEditor` tool, and selected a matching presentation form, `JTextFieldPF`. The value for `DeviceDescription` is passed to the interaction form and read from it after an interaction.



TRADE-OFFS

Interaction forms and presentation forms represent a good way to abstract from concrete toolkits. The tool is completely decoupled from the concrete GUI representation. Presentation forms can be easily adapted to toolkits as they change, without a need to change the tool interaction. In such a case, the interactive part does not have to be changed. In addition, interaction forms decouple the material from its interactive manipulation.

IAFs and domain values

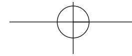
To allow the use of interaction forms independent of a context, they have to exchange domain values or basic data types with the IP. This is the way to ensure that the IP will have full control of what an IAF represents and which results it returns.

The set of interaction forms that can be identified no longer depends on the options available in a specific toolkit. For example, it does not matter for interaction forms whether or not there is a special widget available on the platform used. Most toolkits have to deal with this problem, if they really want to be portable. Either they must reduce their set of widgets to the smallest common set for all supported platforms, or they must reimplement exotic widgets not directly supported by a specific platform, which is expensive.

Encapsulating the system platform

The presentation forms allow us to implement a presentation of the interaction required by a tool for the platform used. For example, Microsoft Windows systems have floating menu bars and toolbars in application windows, while the Macintosh displays menu bars and toolbars for the active application in a generic program bar in the top part of the screen. Similarly, buttons were represented differently in Windows 95 or Windows 3.1. There are, however, still some fundamental problems in implementing complex tree or table interactions independent of the specific GUI toolkit.

Zull-08.qxd 31/8/04 2:35 PM Page 232



By separating the interaction from the presentation form, we overcome the direct coupling to single, specific GUI widgets from within a tool, so that we can respond much more flexibly to changing presentations and toolkits. Of course, the independence of a specific toolkit does not come for free. The cost is that we lose part of the control over the widgets of a GUI toolkit. For example, we can no longer use interaction forms to directly control a specific representation from within a tool (e.g., to set the color of button labels). However, experience has shown that a very detailed control over widgets is normally not necessary for most tools. Representation details, such as font colors, can normally be defined statically in the GUI builder. For tools where exact control is required, we may have to access the presentation forms or widgets directly. In summary, the interaction forms concept shows clearly which tools depend on a specific GUI toolkit and which don't.

Rationale

Whenever you need a decoupling of a tool's interaction and presentation from the concrete GUI toolkit you may use this pattern.

What Next

The design pattern *feedback between interaction forms and IP* shows how to implement the appropriate reaction mechanism between the interactive part and its interaction forms.

8.9 THE FEEDBACK BETWEEN INTERACTION FORMS AND IP PATTERN

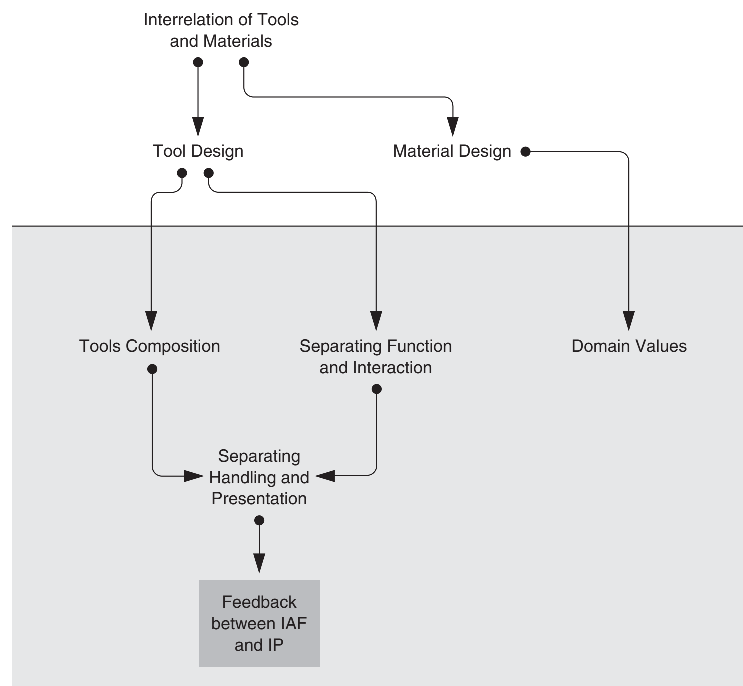
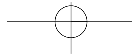


FIGURE 8.35 The Feedback between interaction forms and IP pattern.

Zull-08.qxd 31/8/04 2:35 PM Page 233



Intent

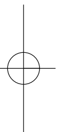
We introduce a solution that uses the command pattern (Gamma et al.) to solve the feedback problem between interaction forms and IP.

Problem

User actions such as mouse or keyboard movements are accepted by the window system. The window system converts these actions into system events, which are initially passed on to the representation and interaction forms. For this purpose, representation forms are normally linked to the appropriate widgets of the window system over a callback (or over subclasses and the bridge pattern). Interaction forms convert system events into program events by alerting the IP. This requires an interaction form to be linked to the IP. The IP can then identify the FP operations that must be called (see Figure 8.36).

Interaction forms, however, must not know the specific type of the IP they collaborate with, otherwise we could not use interaction forms with different IPs. As a consequence, we would have to reimplement all interaction forms for each new IP. As in the feedback between FP and IP (see Section 8.7), we have to solve the problem of loose coupling between interactive components.

How can we implement a suitable feedback mechanism between interaction forms and IPs to allow loose coupling and meet the special requirements of an IP at the same time?



Relate To

The design pattern *separating handling and presentation precedes* this pattern conceptually (see Figure 8.35).

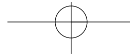
Solution

We use the command pattern to bind the interaction forms to the IP, achieving loose coupling.

One way to solve this problem is to define suitable command classes (e.g., `Command, DropCommand`) for the program events sent by interaction forms to IPs. Another way is to have a generic command class that identifies the appropriate interaction form. We can build a command class so that the interaction form will feed it with appropriate domain values (see Sections 2.6.5 and 8.10). These domain values are required to further process the events in the IP.

If an IP then wants to obtain information about a user action from an interaction form object, it generates a command object. The IP passes a reference to the operation to be called to this command object; this operation will be called as soon as the command object is activated. Subsequently, the IP registers the command object with the corresponding interaction form object. The different interaction forms each accept a set of command objects matching their interactions.

The interaction form object activates the command object registered with it when the corresponding user action occurs. This causes the command object to call the IP's operation registered with it. In Java, a typical implementation of this pattern is based



on an interface that contains an `execute` method. Next, the interaction form calls this operation from the registered command object. This allows the IP to implement this interface and register directly as a command object, using anonymous inner classes of Java.

If a user action means the input or selection of domain values (e.g., a user enters an account number), then the IP requires these values. A command class used for such user actions expects a parameter from the interaction form. The operation the IP passed to the command object has to accept arguments from the command object or probe the IP.

The EMS example

EXAMPLE

Let's look at the `DeviceEditor` tool in our EMS example. This tool allows us to save changes to a device. The `DeviceEditor` tool includes an interaction form, `Activator`, for this user action. This interaction form is implemented by a button, that is, a presentation form, in the GUI. `Activator` is an interaction form that accepts only one type of command objects, namely `Command`. It is a command class that does not expect any parameter.

The sample code in Figure 8.37 was taken from a Java implementation of the `DeviceEditor`. The constructor creates the interaction form object for the button and the command object. In addition, the IP of the `DeviceEditor` tool registers the command object with the button.

When a user clicks the button to save device information, then the button triggers the command object registered with it, and the command object calls the `storeDevice` operation in the IP of the `DeviceEditor` tool.

TRADE-OFFS

Implementation variant: passing operation names

One possible implementation variant to the solution introduced in this pattern would pass only the name of the called operation as a character string to the command object as callback. Java lets you use CoreReflection to call such a named operation at runtime. But this solution is not as performant and less typesafe. In other languages, such as C++, you have to use a method pointer or a "normal" class.

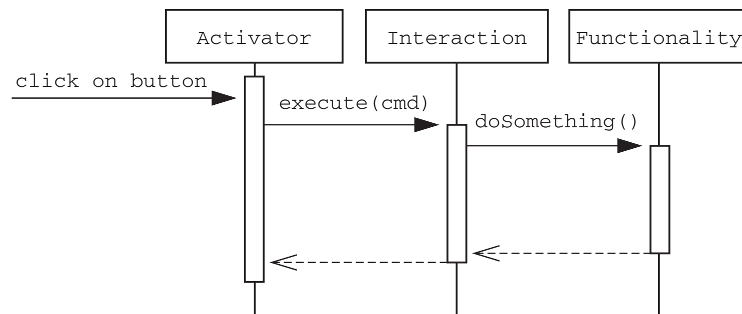
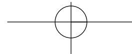


FIGURE 8.36 Flow of events in a tool with interaction forms.




```
public class DeviceEditorIP extends InteractionImpl
{
  //constructor
  public DeviceEditorIP(IAFContext iafsContext, ...)
  {
    // create command object
    _storeCommand=new Command()
     {
       protected void doExecute ()
       {
         storeDevice(this);
       }
     };
    // get interaction form
    _store = (ActivatorIAF) iafsContext.interactionForm
                            (ActivatorIAF.class,"store");
    // attach the command object to the interaction form
    _store.attachActivateCommand(_storeCommand);
    ...
  }
  // callback operation for the command object
  public void storeDevice (Command cmd)
  {
    ...
  }
  // attributes
  // interaction form
  private ActivatorIAF _store;
  // command object
  private Command storeCommand;
}
```

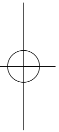


FIGURE 8.37

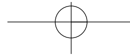
Using the command pattern for interaction forms.

Multiple registration of command objects

An IP often represents the same interaction form in different places of a tool's GUI. In our preceding example, device information entered in the `DeviceEditor` tool could be saved by clicking a button or by selecting a menu option or by typing a value in a popup menu. In this case, the IP can register the same command object with three interaction forms. Regardless of which of the three interaction forms activates the command object, all three cases would call the same operation of the IP to save device information.

One important benefit is that undo and redo mechanisms are prepared so that they can be implemented directly in the command object, where the relevant context for undoing or redoing the command can be stored. Previously executed command objects are stored in a list that maintains the command history. This means that we can implement unlimited sequences of undo and redo by traversing this list.

Zull-08.qxd 31/8/04 2:35 PM Page 236



RATIONALE

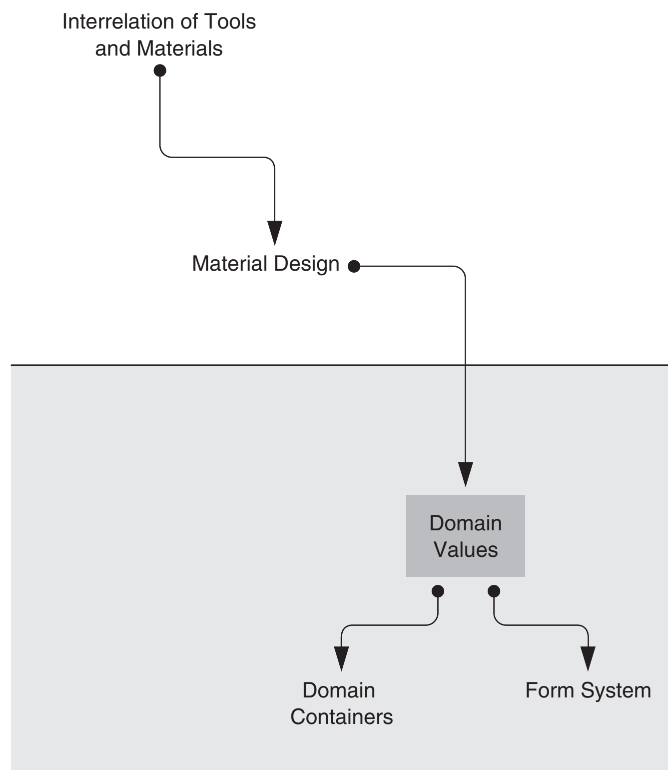
When you use interaction and presentation forms, you will have to solve the feedback problem that this pattern addresses.

WHAT NEXT

The design pattern *domain values* discusses how to extend the basic data types provided by object-oriented programming languages. Domain types are needed for decoupling interaction forms from the interaction part, and the interaction part from the functional part.

8.10 THE DOMAIN VALUES PATTERN

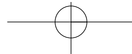
FIGURE 8.38 The domain values pattern.



INTENT

For application-oriented program design it is important to use the different value types relevant in an application domain. This pattern explains how to design and implement these domain values.

Zull-08.qxd 31/8/04 2:35 PM Page 237



PROBLEM

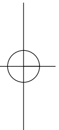
When using object-oriented programming languages to develop large systems, we quickly notice a major shortcoming. The standard data types available in the languages are not sufficient to implement the full bandwidth of basic values for an application domain. Even a value as simple as a money value cannot be elegantly mapped on a standard data type. The type `REAL` available in most languages has an arbitrary number of decimal places, while we need exactly two to represent a money value. Moreover, calculations on `REAL` numbers lead to rounding errors, which are unacceptable for money values in commercial applications. In addition, the arithmetic operations defined for standard data types are generally not suitable for arithmetic operations required in an application domain. For example, there are precise calculation rules for the conversion between currency values that differ from `REAL` arithmetic.

At the same time, we cannot simply use classes and objects to extend the set of primitive data types, because instances of classes have reference semantics, while data types have to obey value semantics (see Section 2.6.5). Therefore,

How can we extend the primitive data types of object-oriented programming languages to user-defined domain values based on value semantics?

RELATE TO

The conceptual pattern *material design* is the conceptual context for this pattern as domain values are a consequence of our application-oriented approach to modeling (see Figure 8.38).



SOLUTION

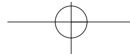
We implement domain values in an object-oriented language, similarly to other user-defined types, using classes. However, in the implementation we ensure that the instances of these classes behave like values.

Our new data types for domain values can be divided into two groups by general typing principles: *elementary domain values* and *composite domain values*. An elementary domain value (e.g., a time of day) represents exactly one “atomic” domain value, which is implemented internally as one or more encapsulated standard data types. At its interface, it offers only operations that are relevant for the domain to handle values of that type (e.g., adjust the time).

Composite domain values build on elementary and other composite domain values. They group different values into a new structured value (e.g., a period consisting of two time values). At their interfaces, there are operations that compose domain values from elements and access individual elements (constructors, selectors), in addition to the domain operations.

In addition, we can classify values based on their cardinality, that is, *finite domain values* and *infinite domain values*. Finite domain values correspond to an enumeration type, which means that they have a fixed number of values (e.g., days of the week). In contrast, infinite domain values cannot be limited (e.g., date), or they extend over a range that can be divided into an arbitrary number of parts (e.g., time).

Zull-08.qxd 31/8/04 2:35 PM Page 238



BACKGROUND: DOMAIN VALUES

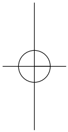
Programming languages normally offer a limited set of data types that follow value semantics. These standard data types, such as `INTEGER`, `REAL`, or `BOOLEAN`, are either related to mathematics or oriented to implementation aspects. An attempt to introduce domain value types to a programming language has not proven to be very helpful, as we know from some so-called fourth-generation languages. In contrast, functional programming languages, such as Miranda, are based on a totally different approach. They use powerful type constructors for user-defined types based on value semantics. Unfortunately, object-oriented programming languages have a serious shortcoming in this respect, as they offer only the class concept for user-defined types. It appears that language designers forgot the importance of domain-value data types for the application system to be developed.

RATIONALE

For systems based on the T&M approach, domain values are an essential concept for introducing domain-motivated values as the "atoms" of programming.

WHAT NEXT

The design patterns *domain-specific containers* and *form system* use domain values as an implementation concept.

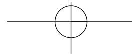


8.10.1 Construction Part: Domain Value Classes

We use classes to implement different domain values. They are arranged in a class hierarchy, where the top class is the `DomainValue` class. This superclass is an abstract class, specifying the following operations for its subclasses:

- Operations that test a passed parameter for a valid external representation of a domain value. Parameters can be strings or number values as a type of the representation (e.g., `InterestRate.IsValid("2,5")` or `InterestRate.IsValid(2.5)`).

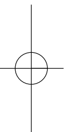
  For composite domain values, this test refers to the individual elements. These operations have to be implemented in the interface of a class or as class operations, depending on the programming language used. If classes are first-order objects, like in Smalltalk, and class methods are inherited, then these operations should be implemented as class methods. In contrast, if using C++, we would implement a prototype object. Clients would call this prototype object, which represents the class object, to do the test there. Java let's us use inner classes to statically embed factory classes in each domain value class, and to accommodate methods in these factory inner classes. Also, we could use the *Singleton* design pattern (Gamma et al.) to ensure that there is one and only one factory for each domain value type.
- Operations like `hasFiniteNumberOfValues()` or `getAllValues()` allow us to request the set of values valid for a domain value type, if this set is finite. These operations show us whether we are dealing with a finite or an infinite domain value. For example, this aspect is relevant when representing domain values by separate interaction forms. Alternatively, we could implement a special class, such as `Enumerable`.



- The operation `toString()` returns the value of a domain value in the form of a string. Most window systems accept only standard data types at their interfaces. Therefore, a domain value should be able to return its value in a string representation.
- Both C++ and Smalltalk let us overload operators, so that we could define relational operators (e.g., `<,<=,>,>=,==,and !=`) for domain values in the `DomainValue` class. Notice that `==` and `!=` are semantically required for all data types, as suggested by Hoare. In any event, we have to ensure that these operations have tests to make sure that two values can actually be compared. It can also prevent relational operators from executing in domain value classes with objects that cannot be compared (e.g., it wouldn't make much sense to use the greater-than operator on `Color`). Also, comparing with objects of the superclass is invalid.

Depending on the application domain, a class like `DomainValue` will apparently be the root of different inheritance structures of domain value classes (see Figure 8.39). Each of the domain value classes has to implement the operations specified in the superclass, `DomainValue`, for their specific values. To ensure that the value of a previously created domain value object cannot be changed, for example in C++, no public procedures that change the internal state should be offered. Instead, there should be one or more special constructors with corresponding parameters available.

To ensure that domain value objects are used according to value semantics, there are two different solution variants, more or less suitable, depending on the programming language: immutable and mutable domain value objects. The construction parts described in Section 8.10.2 to 8.10.5 explain these solutions.



DISCUSSION: USING DOMAIN VALUES

The T&M approach uses domain values as elements of materials. If you think of a material as a tree of objects, then the leaves of this tree are domain value objects that in turn, use standard data types for their implementation. Therefore, we have to build appropriate domain value classes for each special application domain, so that these domain value classes can be used as elementary components of materials.

Domain values for FP-IP interaction

When building tools on the basis of the FP-IP pattern, domain values are the only objects passed from the FP to the IP, in addition to standard data types. In any case, interaction forms are called with domain values. This explains another reason

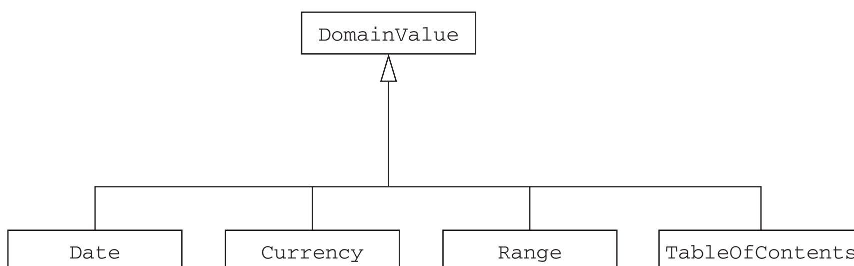
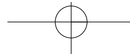


FIGURE 8.39 Domain value classes.

Zull-08.qxd 31/8/04 2:35 PM Page 240

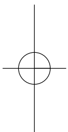


why domain values should be built by value semantics. Only the function and not the interaction should be able to directly manipulate a material (see Section 8.4). For this reason, an FP must not pass references to a material to the IP. If the FP passed domain values to the IP, then the IP would receive copies of the domain value objects, at least from the conceptual perspective. If a user changes a value, then the IP passes this value to the FP, and the FP will update the material to reflect this change.

Domain values for IAFs and PFs

To display domain values in the IP, we can use specialized interaction and presentation forms (see Section 8.8). These interaction forms for domain values are built to match a domain value class, and they can directly check user input for validity. This prevents that, for example, checking a date input for correct syntax is left to the FP, which then activates a feedback mechanism to alert the user. Accordingly, we let domain values do such checks that are not related to context. This means that domain values offer the means to let an interaction form test user inputs for validity. Domain values will then only have to be tested for consistency in the respective context. For example, if users have to enter a date of birth and a wedding date in forms, then the FP or the form itself could check whether the date of birth is older than the wedding date.

8.10.2 Construction Part: Immutable Domain Value Objects



This solution variant implements domain values as *invariable* objects so that changes to domain value objects are excluded. More specifically, we do not allow altering operations at the interfaces of domain value classes. This solution can be used in all object-oriented programming languages. It can be implemented both by the dynamic creation of objects (in Java or Smalltalk) and by the static creation of objects (in C++).

TRADE-OFFS

Drawbacks of solution

Unfortunately, this solution has the following drawbacks:

- New memory space has to be requested for each change to a domain value object.
- Programming with domain value objects from classes that do not offer altering operations is not elegant, because we have to create a new identifier for each change, and then assign the result of a change to that identifier.

Using flyweight *or* factory

The storage problem inherent in this solution can be solved by using the *flyweight* and *factory* design patterns. These design patterns ensure that we do not have to create a technical copy for each change. The *flyweight* pattern ensures that there will always be only one object for each individual value, while the *factory* pattern creates objects by means of appropriate operations. A factory stores all previously created domain value objects. When a new domain value is requested, then the factory verifies whether or not an object has been created for this domain value. If it finds such an object, it will return this object; otherwise, it will create a new one. This ensures that different components using the same domain value can share the same domain value object.

The *flyweight* pattern requires that all previously created domain values are stored by the factory, so that the factory obtains a reference to a domain value object, even if

an object is no longer used. This means that a garbage collector cannot remove a domain value object from the memory as long as this factory exists. This leads to an expanding system. Java lets you implement an elegant solution to this problem. You can use weak references to manage objects, as in a `WeakHashMap`. Objects in this hash map will be removed by the garbage collector when nothing in the system references them, except the hash map. The *flyweight factory* can use Java's `WeakHashMap` to store domain value objects and let the garbage collector remove them, as appropriate.

EXAMPLE

The EMS example

Figure 8.40 shows how we use several elementary domain values, such as `Date` and `EmployeePosition`, in our EMS example. `TableOfContents` can be thought of as a composite domain value that includes `ThingDescriptions`. Moreover, two device objects, that is, `Device: PC-1234` and `Device: PrinterMax`, share the same date from the *flyweight* pattern.

The flyweight pattern is suitable to manage domain value objects, in particular for domain values that have a finite and relatively small value range, such as days of the week or month. Domain value objects for these classes are often created when the application starts and deleted when the application is closed.

RATIONALE

Use this approach for domain value types with a limited number of values and a high number of occurences of these values in your programs. Check whether the programming language you use offers simpler means for "immutable" objects.

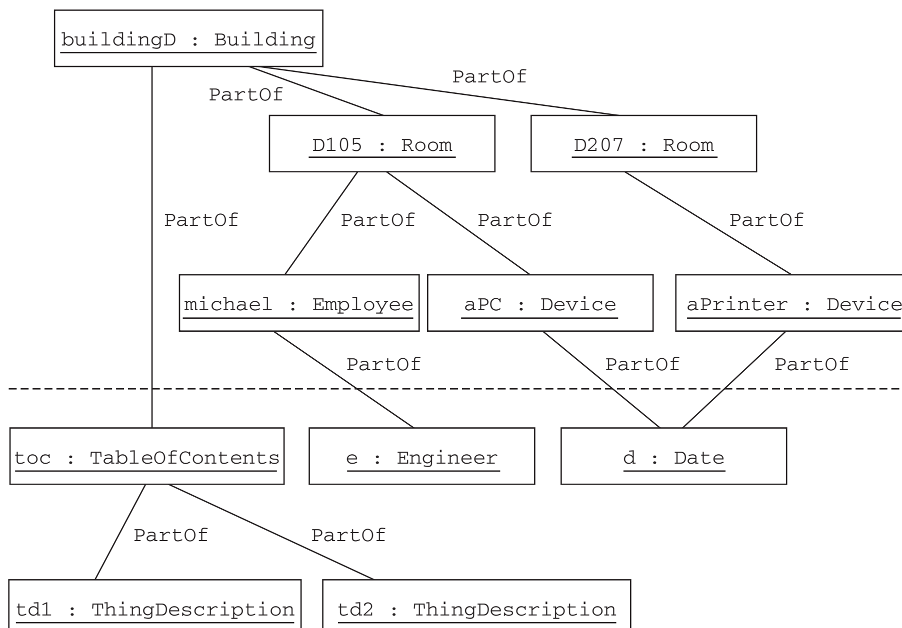
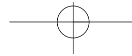


FIGURE 8.40
Using domain value objects in the EMS room plan.

Zull-08.qxd 31/8/04 2:35 PM Page 242



8.10.3 Construction Part: Mutable Domain Value Objects

To avoid problems inherent in the program-specific handling of immutable domain value objects, we can choose a solution variant that uses *mutable* domain value objects.

In this solution, the domain value classes provide all altering operations required. If the domain value objects created by these classes should be used exclusively by value semantics, then the application programmer has to either copy the domain value object before each altering operation, or the domain value classes have to use an internal copying mechanism.

Trade-Offs

The first of these two variants is basically workable. We can use the *flyweight* pattern to implement an internal performant copying mechanism. However, we have to prepare programming guidelines to ensure that the application programmers will really create a copy before each change to a domain value object.

Using the body/handle pattern

We can use the *body/handle* pattern to implement the copying mechanism for objects within our domain value classes. The body/handle pattern is extensively described in the literature; Gamma et al. refer to it as "copy-on-write" in connection with the *proxy* pattern. The body/handle pattern is based on the idea of building a second class tree, a so-called "shadow tree," in addition to the existing class tree, with both trees having an identical structure. The class tree for domain values is the body tree, and there is an additional isomorphic handle tree. Figure 8.41 shows an example.

Notice that this solution is well balanced; it let's you implement your storage management easily by counting the references in the body object, especially in C++. The major drawback of this solution is that handle objects must not be used by reference

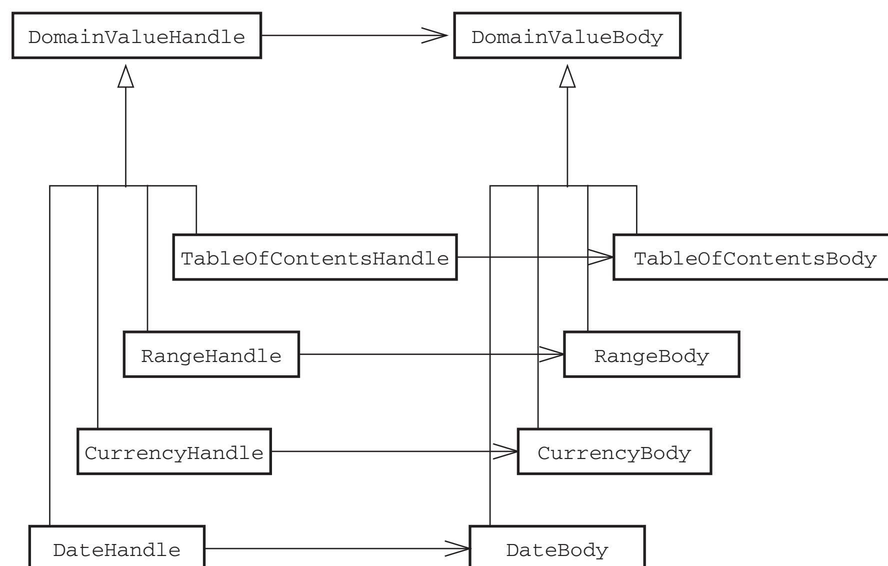
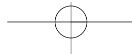


FIGURE 8.41 Domain value classes, using the *body/handle* pattern.

Zull-08.qxd 31/8/04 2:35 PM Page 243



"from the outside." In addition, this rule has to be established in programming guidelines. Another drawback is that parallel inheritance hierarchies have to be maintained.

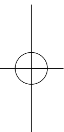
RATIONALE

Use this approach with a language like C++ or when you opt for a simple but potentially error-prone solution.

8.10.4 Construction Part: Implementing Domain Values as Streams

We often find simple values, similar to elementary units, used as domain values in many applications. This hides the fact that complex domain values can be large and extensive. One good example for such a domain value is a table of contents. When you build a table of contents for a selection from a database, you at least have to deal with its size and ask yourself whether it is necessary to keep the complete composite domain value in memory.

It is often a better idea to implement such a domain value as a stream. At the public interface, such streams often present themselves as usual domain values (with the difference of throwing additional exceptions). However, they are internally implemented so that they contain only the amount of data required to meet the interface. If more elements of a composite domain value are required, then these are generated or loaded "on demand."



RATIONALE

Use this approach with very complex and exceptionally large domain values.

8.10.5 Construction Part: Domain Value Types by Configuration

When analyzing an application domain, we often find a large number of similar domain value types, such as different types of amounts. These domain values often differ only in their value ranges and not in their operations. Such differences can basically be expressed by inheritance. For example, `CreditAmount` and `DebitAmount` would then be subclasses of `Amount`. However, subclassing can lead to a large number of domain value classes, making maintenance more difficult.

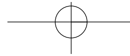
One construction approach uses dynamic configuration of domain value classes. More specifically, we develop only domain value classes for domain values with different operations. Then we express the difference in value ranges by configuration, thus saving the differences in configuration files or databases.

When creating a new domain value object, we additionally define an identifier (e.g., `String`) for the configuration to be used. We would then develop only the class `Amount` and configurations for `CreditAmount` and `DebitAmount`. These configurations will specify that both the credit amount and the debit amount must be positive values.

RATIONALE

Use this approach when a considerable number of domain value types have the same interface and differ only in value ranges.

Zull-08.qxd 31/8/04 2:35 PM Page 244



8.11 THE DOMAIN CONTAINER PATTERN

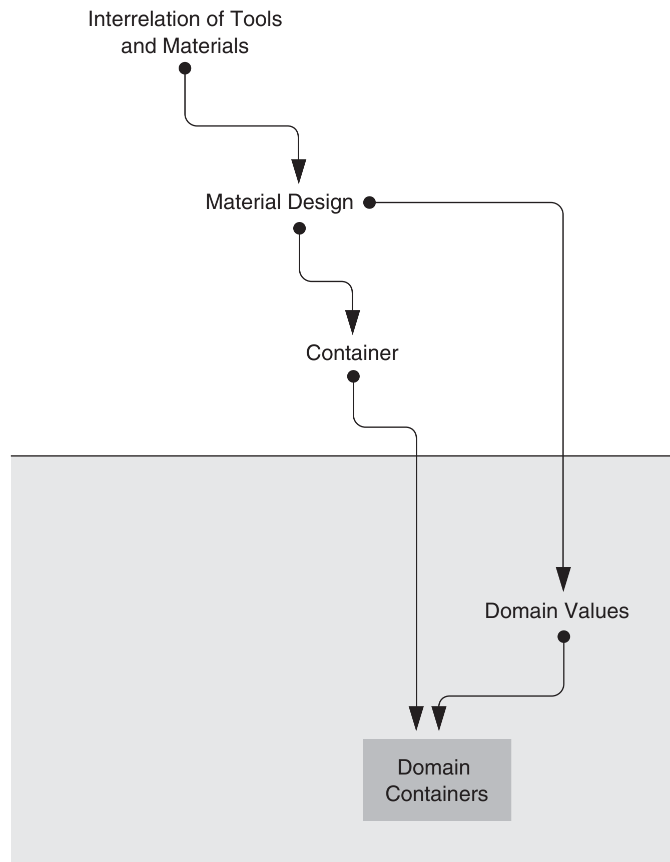


FIGURE 8.42 The domain container pattern.

INTENT

Design domain containers on the basis of the containers you have identified in the application domain and implement them using technical containers.

PROBLEM

Containers serve to hold objects. When we are modeling based on the T&M approach, we distinguish between two types of containers: domain and technical containers.

Domain containers are designed on the basis of existing containers (e.g., folders) that we identified when we analyzed the application domain (see Sections 3.5.16 and 7.7). In contrast, *technical* containers correspond to traditional data structures, such as lists, arrays, or complex technical structures like `WeakArray` in Smalltalk. They are normally used to implement domain containers.

We think that it is no longer necessary to build our own technical containers, since all common object-oriented programming languages include class libraries with powerful and standardized technical containers, so that we shouldn't expect major

Zu11-08.qxd 31/8/04 2:35 PM Page 245

interface changes. Therefore,

We want to model domain containers as a special type of materials that provide a place to store other materials and define an order principle for these materials.

Relate To

The conceptual pattern *container* explains the domain-related context for this pattern. The design pattern *domain value* explains a useful implementation concept used here (see Figure 8.42).

Background: Domain Container

The object-oriented methodology discussion has not taken domain containers into account. Most programming manuals and class libraries model and implement exclusively technical containers.

In general, there is no clear view of what domain containers are and how they can be designed. In contrast, we often find that collections and orders of materials are very important in daily work. However, when we implement existing container concepts in our application system model, we often map more than the container functionality, that is, we also try to utilize the potential of interactive software.

Solution

We build a domain container as an independent domain-specific object, which has the character of a material, but shows some specific interactions.

The elementary interactions of domain containers include storing and editing a set of materials, and accessing elements of this set. Usually a domain container provides a table of contents, which it updates internally.

Containers are materials that can contain materials. This means that we can add other containers to a container, structuring the contents of a container.

Like any other material, domain containers have an identity and normally user-defined names. They are active in the sense that they have their own "navigational model," that is, a specific way that we can navigate from one element to another. Internally, they manage their elements themselves. These interactions can be generally modeled for domain containers.

Example

The EMS example

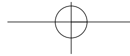
In the context of our EMS system, we could think of the room plan as a container for rooms. The room plan would be consistent when all the rooms it contains are consistent (i.e., no excessive occupancy by persons or devices). This consistency check can be elegantly implemented in the room plan. It will then be available for all tools that use room plans, so that we do not have to reimplement it as we add new tools.

Trade-Offs

Modeling real-world containers

We do not assume that we can directly transfer containers from an application domain to the software model. For example, we want to have large domain containers manage

Zull-08.qxd 31/8/04 2:35 PM Page 246



their own tables of contents. Domain containers can also encapsulate a persistence mechanism, if there is no independet persistency service.

When building domain containers, we often observe that there are certain interactions not found in containers existing in the application domain due to physical limitations. We usually will add such interactions, such as consistency checks, to domain containers.

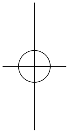
You can think of a domain container as a means to collect a set of materials. We often find that a domain activity or operation should be executed not only on single elements but on the entire set of materials. This is easy to model in containers. Though we use technical containers to implement domain containers, we will not derive our domain containers as a specialization of technical containers.

RATIONALE

For systems based on the T&M approach, which show characteristics of a workplace, domain containers are usually an essential element.

8.11.1 Construction Part: Using Technical Containers to Implement Domain Containers

In cases where we have access to a technical container library, we may consider a way to relate domain containers to technical containers. Should our domain containers inherit from or use technical containers?



In our approach, technical containers always serve to implement domain containers. Consequently, when implementing domain containers, we can use the full choice of variants and interactions offered by technical containers, regardless of the interfaces our domain containers should have.

BACKGROUND: INHERITANCE IN THE T&M APPROACH

It should be obvious from the discussion so far in this book that we should use an inheritance relationship in our domain modeling only provided that the two classes to be linked have a subclass-superclass relationship. A domain container should not be seen as a subtype of a technical container. In addition, its interface should primarily have operations motivated by the application domain, which means that they are not always compliant with the interface of a technical container. For this reason, we use a use relationship between domain and technical containers.

EXAMPLE

The EMS example

To better understand this idea, let's look at room plans as domain containers in our EMS example (see Figure 8.43). A domain container, `RoomMap`, uses a technical container to manage the set of rooms. The class `RoomMap` knows nothing about the concrete type of the technical container (`HashMap`) used. Instead, it works with the specific technical container exclusively over the abstract interface of a `Map`. In addition, the `RoomMap` can generate *iterators* if needed, and use them to traverse the set of rooms.

RATIONALE

Whenever a suitable class library for technical containers is available, you should use it to implement domain containers.

Zu11-08.qxd 31/8/04 2:35 PM Page 247

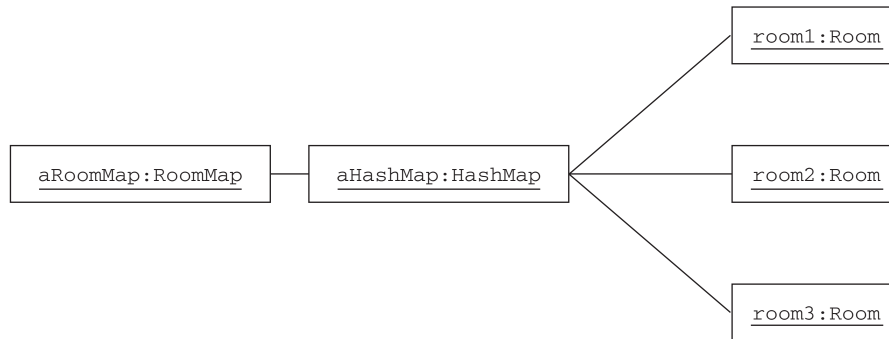


FIGURE 8.43
Domain and technical containers.

8.11.2 Construction Part: Loading Materials

If we do not want to cache all materials in the main memory, then we have to answer the question about what component should be responsible for loading a material on demand. This is the case especially when we want to encapsulate and provide persistent mass data in an object-oriented system. So far, we have implemented the following two strategies within our T&M approach.

Loading materials

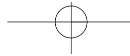
The easiest way is to let the using tools load materials into containers. This solution shows at a container's interface that this container holds an identifier for each material element instead of holding all materials. In this state, a container resembles a table of contents. With a container in this state, a tool cannot use the full interface. Though it can display material names and delete or add materials, there is no way yet to edit a material. If a tool wants to edit a material, then the tool will first get that material's identifier from the container and load the material itself by use of a persistence service (see Chapter 11.2). Subsequently, the tool replaces the material's identifier by the material itself. One major drawback of this construction approach is that the tool construction is more expensive, and there may be consistency problems.

Loading containers

Another solution lets containers load materials. Using the domain values (see Sections 2.6.5 and 8.10) identified in the table of contents, a container can use a persistence service (see Chapter 11.2) and request the material. If we use this construction approach, then the container must know the interface of the persistence service it wants to use. In addition, we have to define the depth in which the material is to load. Materials held in a container can consist of complex submaterial structures, requiring a relatively large amount of storage and long loading times. Notice that the entire material has to be loaded, since it is normally not known during the loading process what parts of a material a user wants to edit.

Yet another solution uses proxy objects, where the loading process is hidden from the containers and the tables of contents. Each pointer to a material object in a container references a proxy object. This proxy object can be thought of as a placeholder, and it behaves like a smart reference. When this proxy object is called, then it loads the material object into the memory. This material object can manage more references to other proxy objects, so that materials are loaded only on demand, even deeper material levels. In this solution, the proxy objects either have to know the interface of the persistence service they want to use, or we implement a kind of generic loader to request materials.

Zull-08.qxd 31/8/04 2:35 PM Page 248

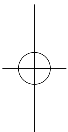


Trade-Offs

Loading materials from containers

The following points are important when loading materials from containers:

- The container or proxy object must know the interface of the persistence service they want to use.
- The container or proxy object must know the location from which to load. This can turn into an expensive task if we cannot assume that all materials are maintained in the same persistent location. This problem becomes even more serious when materials can be maintained in different persistent services or media over the system's runtime. In this case, the knowledge about a specific persistence service cannot be coded into the container or proxy object. This information has to be provided when a container loads or a proxy object is created.
- In cases where materials are not loaded completely, we have to deal with concurrent access by several users in a special way. For example, if an object is to be loaded from different sources, then we have to identify users who may modify the object. This problem is directly related to the question how exceptions should be handled. Since loading is now taking place in a material, there is no easy way to send feedback to users, such as when a load request fails.
- In cases where containers can load their elements themselves, we obviously have to anchor some technological knowledge in the containers. This means that containers lose an important material property, that is, their independence of underlying technologies. These issues will be further discussed in connection with different types of database systems (see Chapter 11).
- Depending on the persistence service used, a container or proxy object may have to handle transactions explicitly, some operations may require the transaction context as a parameter.



Rationale

Whenever you have to store complex or numerous materials in a persistence medium, you should consider this construction approach.

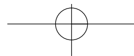
8.11.3 Construction Part: Tables of Contents for Containers

When working with large collections in an application domain, we often find that we use tables of contents to access these collections. From the domain perspective, we could argue that tables of contents are independent components in the construction phase. From the technical view, we can justify the separation of a table of contents from its collection.

Most containers hold large quantities of complex materials. For efficiency reasons, we often have to avoid that all materials managed in a container are loaded in the main memory. For the user to understand this point, we normally use the concept of an independent table of contents. A table of contents provides users with an overview of the materials (container elements) available in a container. The users see the table of contents as an object, avoiding the wrong impression that they can access elements in a container directly. In addition, the users understand that there is some delay involved in retrieving elements from a container.

A table of contents gives users structured access to the contents of a container. Each container can create a table of contents, including a defined set of information

Zull-08.qxd 31/8/04 2:35 PM Page 249



about the container's elements. To list container elements, we need only a small part of the information about a container element. We normally display the name and one additional attribute, such as a domain value (see Sections 2.6.5 and 8.10). When a user selects a container element for editing purposes from the table of contents, then the complete element is loaded or instantiated at this time.

EXAMPLE

The EMS example

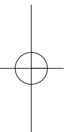
In our EMS example, if we think of the room plan as an example for a domain container, then its table of contents would include the room names. The room plan would return an entire room only when requested to do so.

RATIONALE

As soon as you have numerous materials in one or more domain containers, the question of a table of contents arises and you should consider this approach.

8.11.4 Construction Part: Implementing Tables of Contents as Materials

If we implement a container's table of contents as an independent material, we can treat it like any other material, for example, representing it by an icon on the desktop. The user can see that the table of contents exists independently of the container. The usage model shows clearly that the table of contents does not necessarily show the current state of a container. In summary, the table of contents shows the state of a container that prevailed when the table of contents was created, and the user can explicitly update it.



RATIONALE

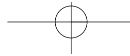
If you want to represent a table of contents as a material of its own, for example, represented as an icon on the desktop, you should use this approach.

8.11.5 Construction Part: Implementing Tables of Contents as Domain Values

If we implement tables of contents as domain values, it will be clear both from the domain and the technical views that a table of contents reflects the state of a container that prevailed when it output its table of contents. In this case, being a domain value, the table of contents cannot automatically update itself to changes in its container. Such tables of contents can be handled more easily in tools. In contrast to materials, domain values can be easily treated by GUI elements, as we can easily create a separate GUI element for a list of tables of contents. This list would show all tables of contents and directly supply identifiers for selected entries.

For tables of contents that become very big (e.g., tables of contents for persistent containers), we could implement them as domain value streams (see Section 8.10.4). The example of persistent containers shows clearly that the table of contents can fetch as many elements from its container as are currently requested by the user. However, implementing tables of contents as domain values means that we cannot arrange them on the desktop as we can any other material. If we need this functionality, we can embed a domain value, such as "table of contents," in a material by the same name.

Zull-08.qxd 31/8/04 2:35 PM Page 250



RATIONALE

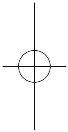
If you always handle a table of contents by means of a tool, then this approach is recommended.

8.11.6 Construction Part: Coping with Changes to Containers

Users can display the table of contents of a container and concurrently operate a tool on a container element. A user could change a material so that such a change could influence the table of contents.

A simple solution to this problem can be derived from the metaphors for materials and containers: To be able to change a material, this material has to be removed from its container. The container's table of contents marks this material accordingly to reflect that it has been removed. Removing materials means that the container is being changed by the use of a tool. This tool informs all other tools operating on this container over the work environment (see Section 8.15). All tools that receive such an announcement can decide whether or not they have to update their container representation. If a tool changes a material removed from a container, then all tools interested in this particular material will be informed. Each change will become effective, and an announcement to this effect will be sent once the tool has placed the material back into the container.

EXAMPLE



The EMS example

Returning to our EMS example, let's assume that a clerk is fetched from the registry. The name of this employee has changed due to marriage. When the user finishes editing, this employee is returned to the registry, and the registry updates the table of contents. When the editing user removes the employee from the registry, then all tools (e.g., other finders) currently using the registry are informed. When the employee's data is corrected, the employee editor receives an announcement, if it currently displays this employee, and if it registered for changes to this material. Once the employee is returned to the registry, all tools operating on the same employee will be informed and can then display the updated table of contents.

TRADE-OFFS

Implementing this reactive behavior can be expensive, especially if there are process borders between containers and tables of contents. In this case, tables of contents should have a way to handle exceptional situations, such as network failures.

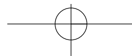
BACKGROUND: MATERIALS AND REACTION MECHANISMS

In the constructions based on the T&M approach, reaction or feedback mechanisms are used only between active components, that is, between tools, tool components, automatons, and the work environment. In addition, access to persistence and distribution services is limited to these components. Materials are treated as components that, in turn, are limited to their domain functionality, without interactive or reactive characteristics.

Materials usually are not reactive

The reason for this construction principle is found in the metaphor for materials. Materials are located in a place, where they can be accessed. Materials do not address their contexts or the technology used by their own initiative when they have been changed. Moreover, it is also meaningful for the technical implementation to keep materials clear of any knowledge about work contexts. A material should not know whether or not its change is relevant for other components. If it knew, then the material

Zull-08.qxd 31/8/04 2:35 PM Page 251



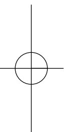
would have to decide whether or not a change to it is relevant for tools that directly operate on it or for other tools that operate on the underlying containers.

When generalizing a reaction mechanism, all materials in a use relationship to a changed material will eventually be informed. This would lead to an unmanageable cascade of announcements that make the entire construction too complex. We generalized a mechanism that is relevant for interactive applications only. Even when we merely divide an application into client and server systems, the pure reaction mechnism is no longer useful. For this reason, we generally discard this solution for materials held in containers.

Discussion: Complex Materials and Changes to Containers

The situation gets more difficult if we have complex composite materials with parts containing domain crossreferences, but where the elements still can be manipulated individually by tools. In such a case, the container is actually the only simple means to ensure domain consistency of the composite material. On the other hand, changes to a material component should have an immediate effect on other parts, even though these parts are currently not used by a tool. In such a case, we cannot go the simple way of using a reaction mechanism. Still, we should bear in mind that we have already expanded the container concept. A container manages the consistency of materials that are modified either in this container or outside. But rather than including the table of contents in our announcements, we let the individual material components coordinate themselves via a container that will then update the table of contents.

When containers are expanded to a coordinating context for complex materials, they often take the character of an automaton. In such a case, we check whether or not the desired functionality can be provided by a "real" automaton (see Section 8.13) or a service (see Section 8.14).



A combination of container, service, and automaton has proven useful, particularly for jointly used materials. More specifically, a container resides in a separate service and can never migrate to user workplaces. These containers are always accessed over an appropriate service, where the service may reside on a server "in front of" the container. Especially in workplace systems, we often find either technical proxies or independent automatons that access a service. Since the service is the only thing that accesses a container, it can easily ensure that container's consistency. In addition, the service can send messages as soon as a container has changed. This means that tools at other workplaces can respond elegantly to changes effected to a jointly used container.

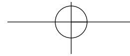
Example

The EMS example

The object chart represented in Figure 8.44 shows how a `Finder` tool edits a registry. The `Finder` starts a `SnifferAutomaton` to find room plans in the registry that match a specific search term in their names. The `SnifferAutomaton` requests matching room plans from the `RegistrarService` and receives a table of contents with all matching room plans. Note that the `RegistrarService` obtained these room plans from the `RegistryContainer`. The `Finder` displays the table of contents.

The user can select a room plan from the displayed list and place it on his or her desktop. This action causes the `RegistrarProxy` to be called, which forwards the request to the `RegistrarService`. The `RegistrarService` requests the room plan from the `RegistryContainer`, which marks it internally as logically removed and returns it (as a technical copy) to the finder via the `RegistrarProxy`. Finally, the `Finder` places the room plan on the desktop.

Zull-08.qxd 31/8/04 2:35 PM Page 252



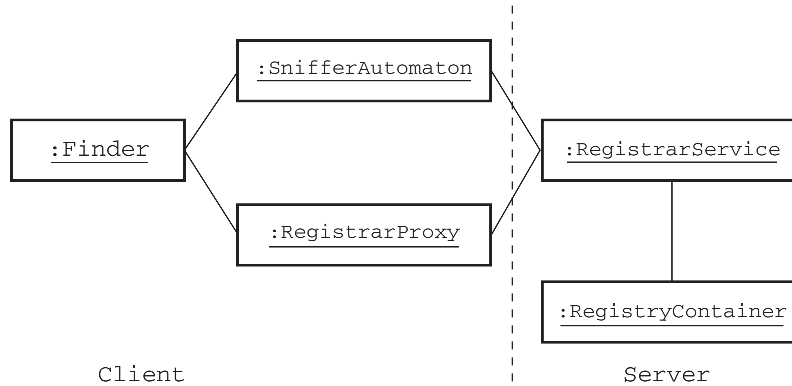


FIGURE 8.44 Object chart of an automaton for domain.

When the room plan is (logically) removed from the `RegistryContainer`, the `RegistrarService` sends a message to all tools registered for that type of message. These tools can now update their tables of contents.

RATIONALE

When you have complex or compound materials, which are changed by many users, then you should be aware of the potential pitfalls and solutions described in this approach.

8.12 THE FORM SYSTEM PATTERN

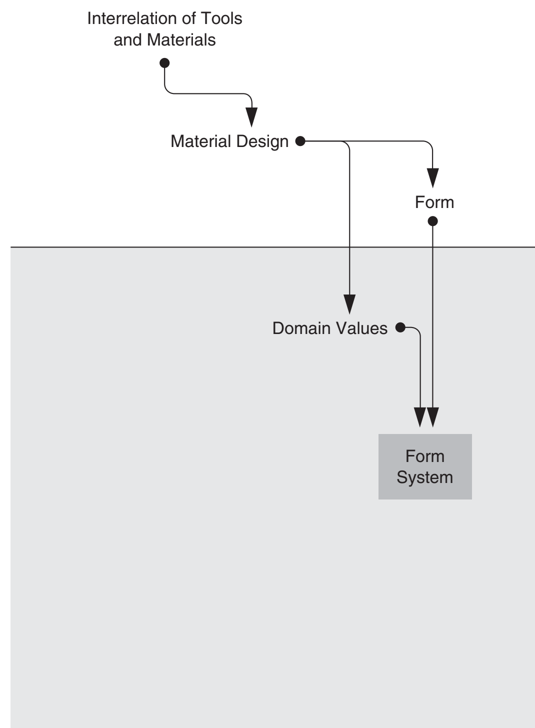
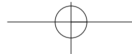


FIGURE 8.45 The *Form System* pattern.

Zull-08.qxd 31/8/04 2:35 PM Page 253



INTENT

When you have identified the frequent use of forms in an application domain, it is useful to transfer this concept into a system of software forms with appropriate tools.

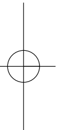
PROBLEM

Converting the conceptual pattern of forms into a construction appears to be easy at first sight. However, we often find that a large number of forms has to be implemented in a similar way. Implementing generic operations for forms is relatively expensive and does not offer tangible benefits, compared to simple "data containers." In addition, building tools to operate on such generic forms is normally just as expensive. Therefore,

How can we support the use of forms inexpensively to become part of an application?

RELATE TO

The conceptual pattern *forms* outlines the background of what forms are and how they relate to materials. The design pattern *domain values* provides the essential elements for handling domain-motivated data in the form system (see Figure 8.45).



SOLUTION

We build a form system, consisting of classes for forms, form templates, form elements, element lists, and form fields, where we base the form fields on existing domain value classes. We add generic form tools for defining and editing form templates and instances of forms.

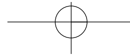
Forms are composed of form elements. The simplest structure of a form element is the *form* field, which contains a single domain value. These parts are sufficient to implement a simple form: the appropriate form fields are selected from a predefined set and parameterized by fitting domain value types.

If we build a form from individual form fields, then complex forms will soon become unclear and hard to handle. To avoid this problem, we structure form elements based on the *composite* pattern of Gamma et al. To utilize the full potential of the *composite* pattern, we treat the form itself as a subclass of a form element within the inheritance hierarchy, so that a form can, in turn, have subforms as its elements. It also allows us to reuse forms when building other forms. Figure 8.46 shows such a forms hierarchy.

If we want to use a specific form, we have to have already define a the elements that this form should contain. For normal use, we want to prevent changes to the structure of a concrete form. For this reason, we specify operations that change the structure at the interface of the class `Form`. But then, how should we initially define a form?

We can solve this problem by separating the usage of a specific form from its form definition. Consequently, we describe the form's structure in a form template,

Zull-08.qxd 31/8/04 2:35 PM Page 254



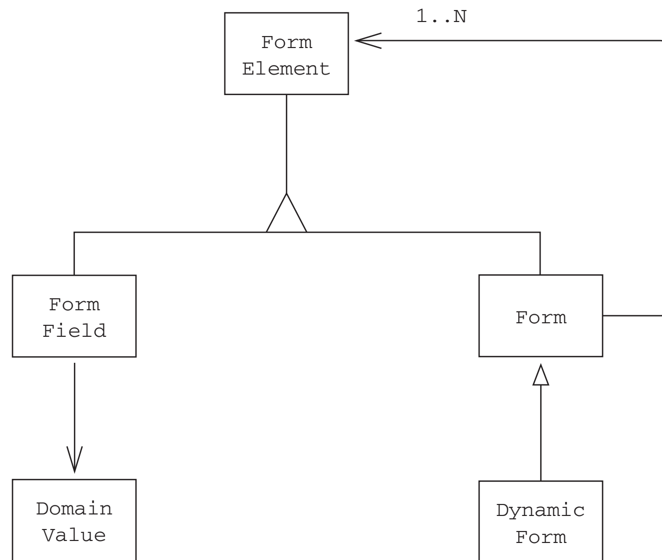


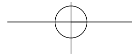
FIGURE 8.46 A forms hierarchy.

which will then be used to create similar types of forms. Let's look at the interface of the form class:

```
class Form
{
  public int elementCount ();
  public FormElement elementAt (int index);
  public boolean hasElement (String name);
  public FormElement elementByName (String name)
  public FormElement[] elementsByType (Class type)
  public boolean hasField (String name);
  public FormField fieldByName (String name);
  public boolean isValid ();
  public String[] validationHints ();
  public void setGeneralNote (String note);
  public String generalNote ();
}
```

Once we have defined a form, we cannot change its structure. A form is defined by a template called `DynamicForm`. A form can be defined only once. Changing the structure of an instantiated form would conflict with the requirement we just saw, namely that we want to prevent arbitrary changes to the structure of a form.

Zull-08.qxd 31/8/04 2:35 PM Page 255

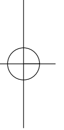


EXAMPLE

The EMS example

Let's see these ideas in the context of our EMS example. When developing the device manager, we initially represent employees as employee forms. The corresponding class for an employee is inherited from the `DynamicForm` class and defines its structure once in the constructor, that is, by composing an appropriate form template.

```
public class Employee extends DynamicForm
{
  public Employee()
  {
    addElement( new FormField(StringDV.Factory.instance(), "ini-
                tials", "Initials"));
    addElement( new FormField(StringDV.Factory.instance(), "name",
                "Name"));
    addElement( new FormField(StringDV.Factory.instance(), "first-
                name", "First Name"));
    addElement( new FormField(StringDV.Factory.instance(),
                "street", "street"));
    addElement( new FormField(StringDV.Factory.instance(), "zip",
                "ZIP"));
    addElement( new FormField(StringDV.Factory.instance(), "email",
                "E-Mail"));
    addElement( new FormField(StringDV.Factory.instance(), "tele-
                phone", "Telephone"));
    addElement( new FormField(EmployeePositionDV.Factory.
                instance(), "position", "Position"));
  };
}
```



That's all our sample class contains. The single fields in an employee form can generically be accessed from operations of the superclass. The forms editor uses the `FormEditable` aspect to work on the form superclass. Figure 8.47 shows the relation between generic components and the employee form used in our example. Note that the gray classes are generic parts, such as from a framework.

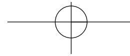
TRADE-OFFS

Using forms can mean a significant reduction of our development cost. We can quickly implement a basic functionality and save tedious implementations of recurring functionalities.

Unfortunately, using the forms concept can also entail some problems. The forms concept may entice developers to model materials as forms, when these materials should have various domain operations. Inexperienced developers especially will often tend to design data collections instead of materials.

Another risk is seen in a reduction of the type safety in typed programming languages due to generic forms. In many cases, this can mean that the compiler can no

Zull-08.qxd 31/8/04 2:35 PM Page 256



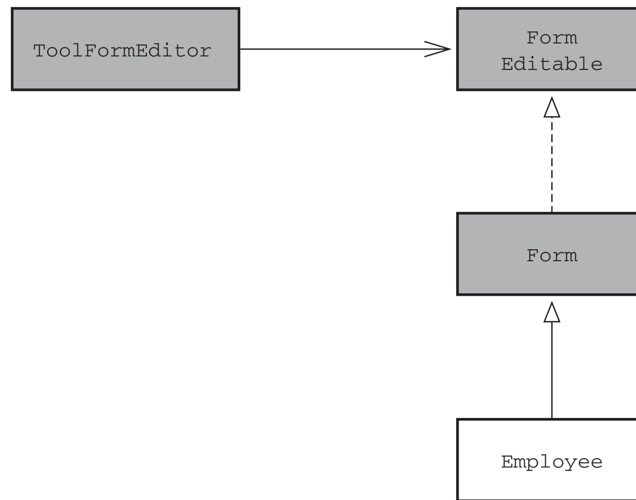


FIGURE. 8.47
An employee form used in the EMS example.

longer check whether assignments of form objects to identifiers are meaningful. In particular, there is no static way to ensure that fields accessed in a form actually exist under the specified names, and that they have the expected (domain value) type. This problem is particularly serious when forms are changed.

Changing field names and types represents a particular risk, because it may mean that the compiler cannot detect inconsistencies caused by such changes. We can reduce this problem if we make field contents accessible only via access operations. Then only these operations will be used whenever a known field needs to be accessed. Generic queries for field values will then be left exclusively to generic tools.

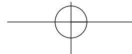
Another problem arises in connection with the persistence of forms. If a user loads an older version of a form, then this form may have some fields missing that were added in later versions. The only way to prevent this problem is by explicit programming. Once a form has been loaded, an extended form has to generate the new fields, that are missing in the older version and fill them with default values.

Discussion: Developing Forms

In the course of the development process, we may find that a material once identified as a form has actually more domain interactions than initially expected. In this case, we need to develop it from a generic form into a more specialized material.

Our general development strategy supports “migration.” This means that we can implement additional interactions in the form class, without needing to replace the generic implementation of the superclass. For example, if we want to add the total price to a purchase order form, we could implement an appropriate operation in the purchase order form class. The purchase order form class could then request information such as number of units and cost per unit from the generic part of the form, then do a calculation, and finally return a result. From the domain view, the purchase order form no longer uses the generic form fields, for example, to fill them in or read them; it now also has a specialized interaction.

Zull-08.qxd 31/8/04 2:35 PM Page 257



When developing a "normal" material from a form, this does not necessarily have to replace the existing form. The "normal" material can output its data like a form and update itself from a completed form.

Layout of forms

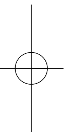
Basically, materials know nothing about their representation on the screen or on a printed hard copy. For this reason, we have to store layout information in another place. This layout information may then be read by a tool or automaton and used to process forms. This allows us to strictly separate the form from its screen representation.

For example, we could specify layout information in configuration files, independently of our program code. This allows us to define a layout for forms independently of the forms. Even programming novices could then create and adapt the layout of forms.

Consistency conditions

We saw how consistency can be checked independently of a context in connection with domain values (see Section 8.10). We may now use this consistency check for forms and ensure that each field of a form contains a valid value. Intra-form consistency checks are initially left to each single form. A form then has to provide probing operations indicating that it was properly filled in or whether there are any problems.

We also saw that it is important to allow inconsistent forms to be saved. This is a fundamental requirement for expert work with forms. Users can put an incomplete or inconsistent form aside and continue editing it later. However, some special tools or automatons may refuse further processing of a form if they find that consistency conditions are violated.



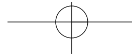
SOLUTION: TOOLS FOR FORMS

A framework can provide the generic tools we need to create generic forms. A generic forms editor can easily be used to fill in and represent fields in a form. Such a tool actually needs only to support the generic form interface. It does not have to know specific forms, such as a purchase order form.

We often find in practice that a generic tool is not sufficient, that is, it has to be adapted to a specific task. If such requirements concern only the user-interface of a tool, then it appears useful to split the tool into an FP and an IP, as discussed under the *separating FP and IP* pattern (see Section 8.7). All we have to do then is to specialize the IP, including the GUI resources it uses, by subclassing. This means that we use the basic functionality of the generic IP, while specializing the layout or other features.

However, we must frequently expand the functionality of a tool and adapt it to the set of a user's task. This is normally necessary when a generic form develops gradually into a specialized material. In that case, we have to adapt a generic tool to the specialized form to be able to utilize the new interactions with the material. Note, however, that this is not a general rule. Once a form has been expanded by a more specialized interaction, this may be of interest for only a single tool. At this point, remember our discussion of the *aspects* design pattern (see Section 8.3), which represents relations between tools and materials. We can add a new aspect to a generic form to make it suitable for manipulation by a specialized tool.

The important thing to remember is that we should continue taking advantage of the generic parts of a form and the relating tools. When a form grows to become a special material, then this doesn't mean that we have to give up our generic use of this form. This also applies to tools, where we could use the generic FP of a forms editor to easily fill in fields in a form and send the appropriate events to the IP.



RATIONALE

Consider building a form system when you have to support an office-like work context with many different paper forms as the relevant objects of work.

8.13 THE AUTOMATONS IN TECHNICALLY EMBEDDED SYSTEMS PATTERN

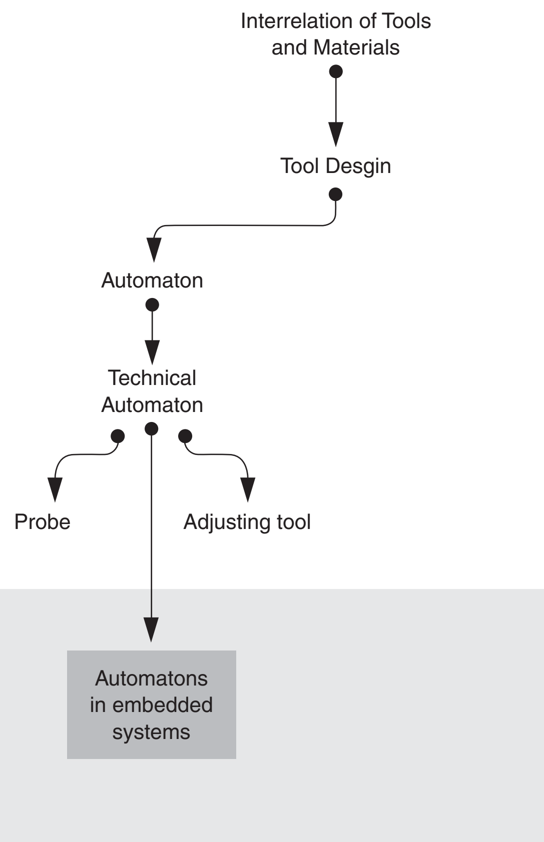


FIGURE 8.48 Automatons in technically embedded systems.

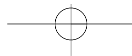
INTENT

Technically embedded systems can be interpreted as automatons sending autonomous events. These automatons can be probed and adjusted. This pattern shows how to implement these concepts.

PROBLEM

Technically embedded systems often use automatons to model the system's external components. This allows us to embed an autonomous technical device as an additional source of events in our application system.

Zull-08.qxd 31/8/04 2:35 PM Page 259



We can call altering operations on a technical automaton to set the technical device. These settings change the automaton's state. In addition, there are external events that cause a change of the automaton's state. This means that, in addition to an event channel for the window system, we also have to consider an additional event source in interactive applications. Therefore,

How can we implement an embedded application system based on the conceptual pattern for a technical automaton?

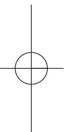
RELATE TO

The conceptual patterns *technical automaton*, *adjusting tool*, and *probe* introduce the concepts that this design pattern explains on a construction level (see Figure 8.48).

SOLUTION: ADJUSTING TOOLS

We connect adjusting tools and technical automatons over asynchronous communication within a multiprocess space. Then we connect probes to the automatons.

The conceptual pattern *technical automaton* has introduced a way of interpreting technical systems within the T&M approach. Technical automatons model the real-world devices and machines. Within the software model, we use special software tools, so-called adjusting tools, to let users operate and set these automatons. Such adjusting tools are normally tailored to their automatons. To achieve a flexible way to model the communication between technical automatons and their adjusting tools, we often use probes to represent parts of an automaton's state. In addition, probes are helpful in modeling the control flow, because they are connected to the tool over an observer mechanism.



Probes are normally connected to a technical automaton and fed with measurement readings by the automaton. Clients can register for events with the probe. They will then be informed about readings in the intervals they chose.

The FP of a tool or the IP of an automaton (if you decide to add a user interface to an automaton) can request a probe from the automaton and register with the automaton to probe it for domain values. Together with the registration, an FP or IP can specify a preferred method to obtain information about changes (e.g., by sampling rate or event). Figure 8.49 shows an IP that uses an automaton and one of two available probes.

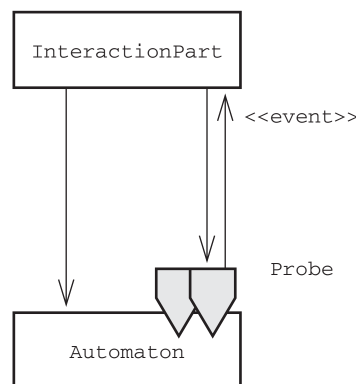
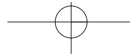


FIGURE 8.49
An IP using an automaton and one of two probes.

Zull-08.qxd 31/8/04 2:35 PM Page 260



Like other tools, adjusting tools are embedded in an environment. The adjusting tool itself will usually consist of an IP and an FP, and it uses an automaton. The IP represents the automaton state and supports requests for changes. The IP does not have its own memory, so it must rely on an FP to obtain the current domain state. Figure 8.50 shows the basic architecture of an adjusting tool.

The FP informs the IP about the domain state and coordinates the connection to the automaton. Though each automaton has its own state, we always need an FP to supply the necessary domain interpretation for the values of an automaton. If a tool is used to control several automatons, then the FP assumes the domain coordination and combination of technical event sources.

An FP buffers the probing results from an automaton, and it can interpret different values and calculate new values for the IP. Moreover, the FP can combine domain states and the settings of several automatons in one single tool.

SOLUTION: ADJUSTING TOOLS AND AUTOMATONS IN A MULTIPROCESSING SPACE

We want to implement adjusting tools and automatons so that the automatons operate as independent components on separate computers. To this end, we have to connect processes. We call this environment a multiprocess space. Let's look at the terminology before we continue our discussion.

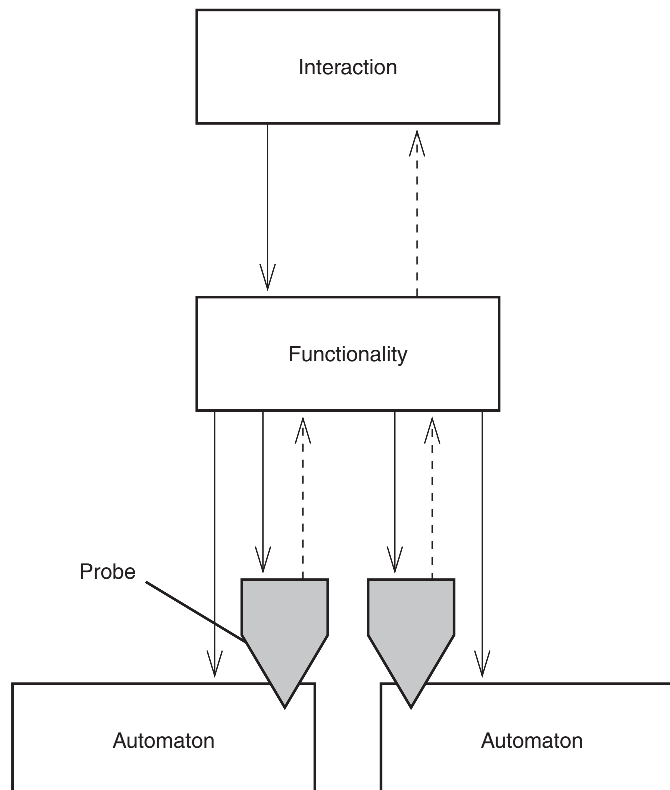
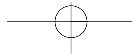


FIGURE 8.50 The basic architecture of an adjusting tool.

Zull-08.qxd 31/8/04 2:35 PM Page 261



A *multiprocess space* is the domain and technical space spanned by an application system when that system's tools and automatons run in different processes connected over a communication medium. We assume that the distribution of components over this space is explicit, both in the design model and in the implementation model.

The separation of components in a multiprocess space requires a connection concept beyond the approaches described so far. In this respect, we speak of asynchronous coupling.

***Asynchronous coupling* means the connection of two objects over an asynchronous communication medium in a multiprocess space. We distinguish three variants:**

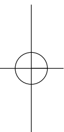
- **operation call without expecting a direct result;**
- **operation call and expecting a (direct) result; and**
- **abstract communication via events only.**

An asynchronous communication medium always has to expect connection errors to occur. For this reason, we need connection information on the metalevel.

Example

Sequence diagram for interprocess communication

We use the three asynchronous coupling variants to distribute adjusting tools and technical automatons. The sequence diagram in Figure 8.51 shows as an example of how tools and automatons can interact in different processes. Notice that we added two elements to this chart. First, the objects of a process are grouped at the top of the chart, so that several communicating processes can be represented in the same chart. Second, we introduced a broker object to better explain the interprocess communication.

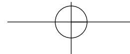


Let's look at the processes and events in Figure 8.51:

- *Registering*: The FP uses the proxy to register with the remote automaton. Note that there is a time delay in this asynchronous process. When the automaton receives the registration request, it generates an observer proxy. If the FP cannot be registered for technical reasons (e.g., because the network connection broke down), then the client process is informed about the failure. This error message is sent to the proxy object after a timeout specified by the system. The proxy converts the error into a communication status event. If the FP can successfully register for the corresponding event, it can now respond.
- *Adjusting*: To adjust the automaton, the FP calls the appropriate operation on the proxy. The proxy creates a data packet and sends it over the communication system; then the control flow is returned to the FP.

  The data packet is sent to the automaton's `Entry` object with a time delay, where it is converted into an operation call on the automaton. The automaton sets the requested values, and the control flow is returned to the communication system via the `Entry` object. When several clients communicate with the automaton in this way, then all adjusting requests are serialized by the communication system. Each successful change leads to an announcement about an effected state change. The consecutive execution of all requested changes determines the final state that is announced to the clients. This

Zull-08.qxd 31/8/04 2:35 PM Page 262



ensures the fundamental construction principle that all interested tools have to be informed about changes to materials (or the automaton, in this example).

- *Announcing*: The automaton changes its state upon request, and because it represents an independent technical device, this situation occurs frequently. A probe is informed about each state change. The probe informs the observer proxy about each state change. Once the observer proxy has been fed with the current state information, it converts this information into parameters and passes them to the communication system. Subsequently, the control is returned to the probe.

  After a time delay, the communication system informs the automaton proxy that it has received data. The automaton proxy encapsulates the conversion of data packets arriving from the communication system, interprets the contents, and changes its internal state. Subsequently, the automaton proxy passes this state change to the responsible probes, which inform all observing local objects, such as a tool's FP. The FP of a tool can probe both the probe itself and the automaton, because the current model of the automaton state is present in both objects. This entire process of announcement and probing can be thought of as a synchronous communication at the tool process end, which guarantees all required consistency criteria. Subsequently, the control over the probe and the automaton proxy is returned to the communication system. Finally, the control is returned to the tool's user via the window system.

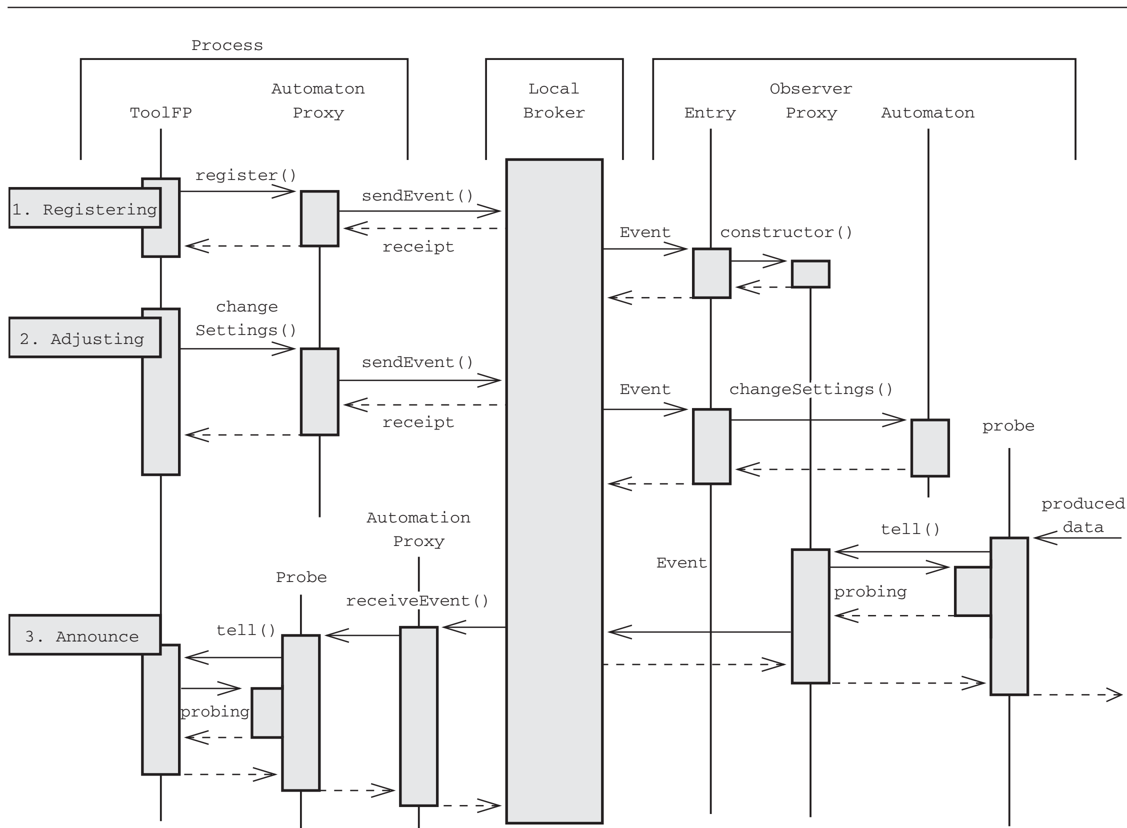
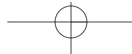


FIGURE 8.51 Asynchronous communication between a tool and an automaton.



The multiprocess architecture just described offers a nice way to handle reactive user inputs in one process and forward new automaton states concurrently. This architecture has been implemented in several projects, based on a special integration of events in the event queue of the window system.

To better understand this approach let's look at the flow of control among the three processes of the above example. Figure 8.52 shows how these three processes interact. The circled areas in this figure denote each of the three processes.

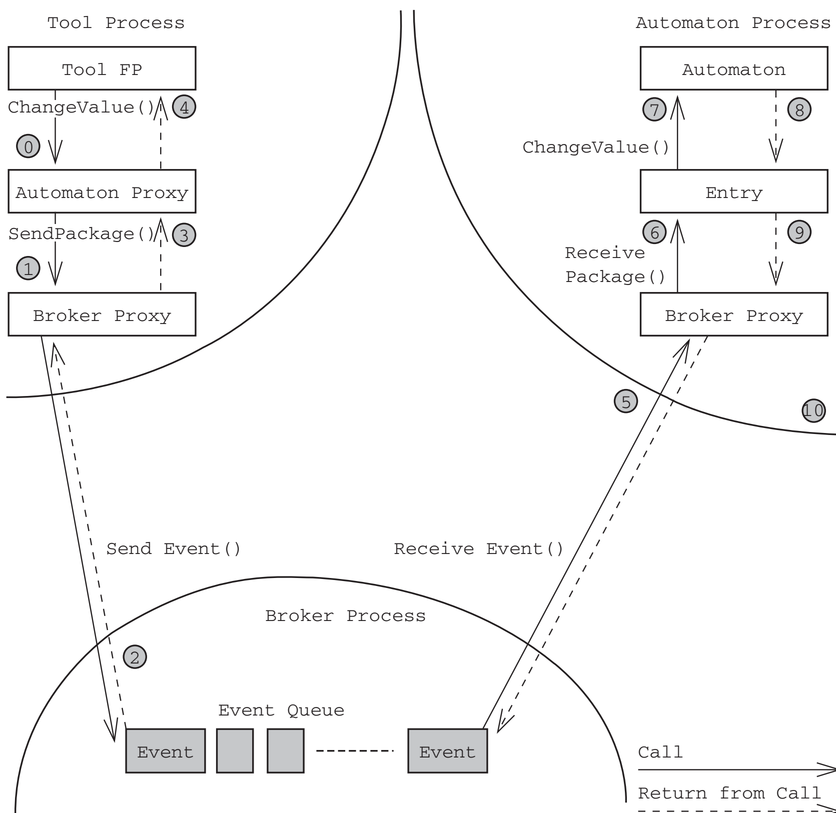
TRADE-OFFS

Usage model for distributed communication

The usage model for faultless and quick asynchronous communication in multiprocess spaces does not have to differ substantially from the synchronous model that we would use in a single-process system. Communication errors will be handled on the basis of known principles. Users are normally informed via a modal dialog when a request is unsuccessful, for example, because a remote component did not respond.

It has proven useful to symbolize the network state for remote components that are used frequently in a system. To this end, we could use entries in a status line or small icons that change their appearance, depending on the status.

There are similar options to represent time delays of communication processes. We could also use status line entries or icons to display the progress of communication



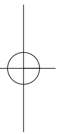
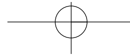


FIGURE 8.52 Flow of control and interaction between the three processes of Figure 8.51.

Zull-08.qxd 31/8/04 2:35 PM Page 264



processes for the user. On the other hand, we found in many projects that the use of a "sleeping mouse pointer" (e.g., represented by an hourglass) is less successful. The reason is that most users know this symbol from synchronous interprocess communication, expecting that they cannot continue working in this phase.

RATIONALE

Consider this pattern when you have to integrate autonomous technical components as part of an application system.

8.14 THE DOMAIN SERVICES PATTERN

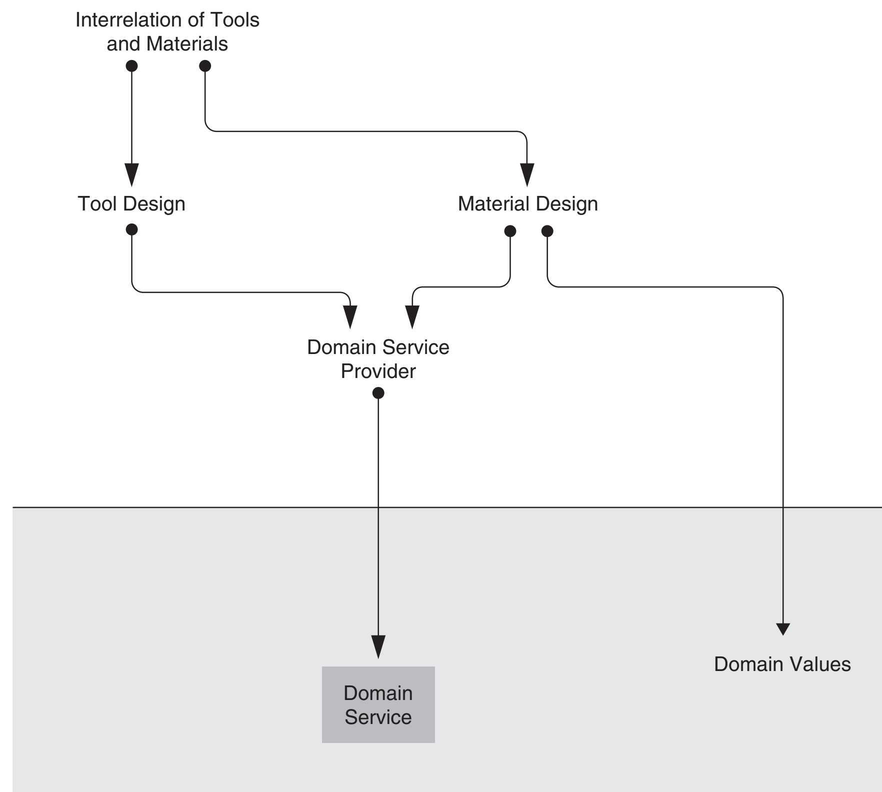


FIGURE 8.53 The *Domain services* pattern.

INTENT

Domain services offer the means to separate domain functionality, both from different front-ends and back-ends. They abstract from the various interaction models of frontend technologies.

PROBLEM

When looking at the equipment of different workplace types, we often find that many common domain features are implemented in several tools. If we subtract the interaction

Zu11-08.qxd 31/8/04 2:35 PM Page 265

parts of these tools with their different technologies, we can identify a domain functionality that is independent of different representations.

In this context, independent representation means for us independence both of the presentation and the interaction. For example, in Internet applications based on HTML, clients not only look different than clients in a fully fledged desktop application, because an HTML browser uses different GUI elements, they also behave differently. Although it is absolutely common for a desktop application to change a series of field contents immediately upon some user action without any noticeable delay, we normally cannot and will not implement this behavior for HTML clients.

This example leads to another issue. Many servers (HTML and other) implement domain functionality by directly accessing a database. This is another issue of independence that we could call back-end independence. In short,

We are looking for a concept to implement the same domain functionality, but use different interaction options, and design it for different workplace types and different technical front-ends (desktop, webtop, etc.). The solution we search for would also be useful for many issues relating to the distributed client-server architecture.

Relate To

The conceptual pattern *domain service provider* sketches the concept realized here. Domain values are useful for implementing the interface of a domain service (see Figure 8.53).

Solution

We combine functionality and knowledge about a service or about the use of a product, regardless of the specific user interface. We make this domain service available in the form of a bundled service package. This means that we encapsulate the materials it manages in this domain service.

A domain service is built internally so that it can handle materials. Domain services can also delegate a requested service to other services or automatons. References to materials within a domain service are never made public. This means that the interface either offers values, or it provides a copy of a material (the difference between a technical and domain copy of a material is explained in Section 10.1.1).

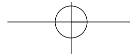
Requirement for domain services

A domain service is implemented so that it meets the following requirements:

- It provides services at its value-based interface.
- It accepts domain requests at its interface, that is, clients delegate standardized subtasks to the service.
- It encapsulates the materials it manages and the specific implementation of the domain interactions.
- It can be implemented to act as an equipment component and have different plug-in components to that end.
- It does not make any assumptions about the interaction model and the representation at the user interface.
- It basically supports multiuser and distributed systems.
- It should offer an appropriate state and session model.

Technically, a domain service can be implemented in a totally different way, but this shouldn't change its semantics. For example, if we implement a domain service as

Zull-08.qxd 31/8/04 2:35 PM Page 266



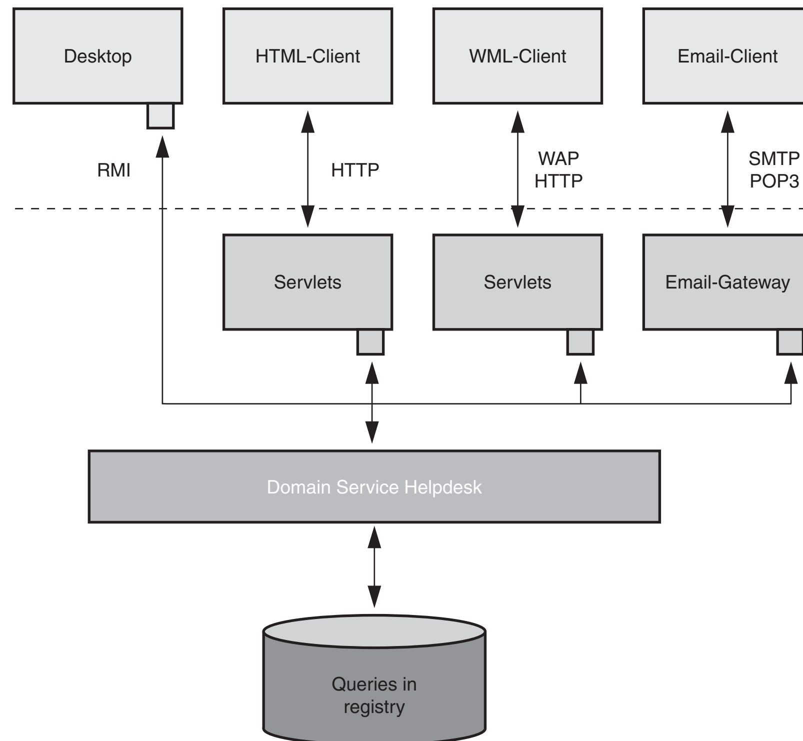


FIGURE 8.54
Domain Service Helpdesk.

an Enterprise Java Bean (SessionBean), we have to add a home interface for technical reasons. However, this does not change the domain use of that service.

Example

To better understand the domain service concept, we use the example of a helpdesk. Users can send their questions to providers and obtain answers from experts over this helpdesk. Figure 8.54 shows the domain service `Helpdesk`, which can be used by desktop tools and servlets or over a mail gateway. In this example, the requests sent by users are the materials of this application. The domain service manages these requests. To this end, it uses a registry. The dashed line separating the frontends from the domain service denotes a logical separation. The important thing for our client-server distribution is where we introduce an intermediate layer that connects clients to the domain service.

Let's look at the details of the example in Figure 8.54. The Web server appears useful for HTML-servlet clients. However, we could also opt for a two-layer solution, including a Web server and a separate server for the domain service.

For desktop tools, the FPs discussed in Section 8.7 appear to be suitable candidates to connect to the domain service. For simple tools, we could even do without separating IP and FP as different classes. The tool connects directly to the domain service. Since our tools should be highly interactive, we accommodate them on the client side. Practical experience has shown that it normally makes no sense to move a tool's FP to a server.

Zu11-08.qxd 31/8/04 2:35 PM Page 267

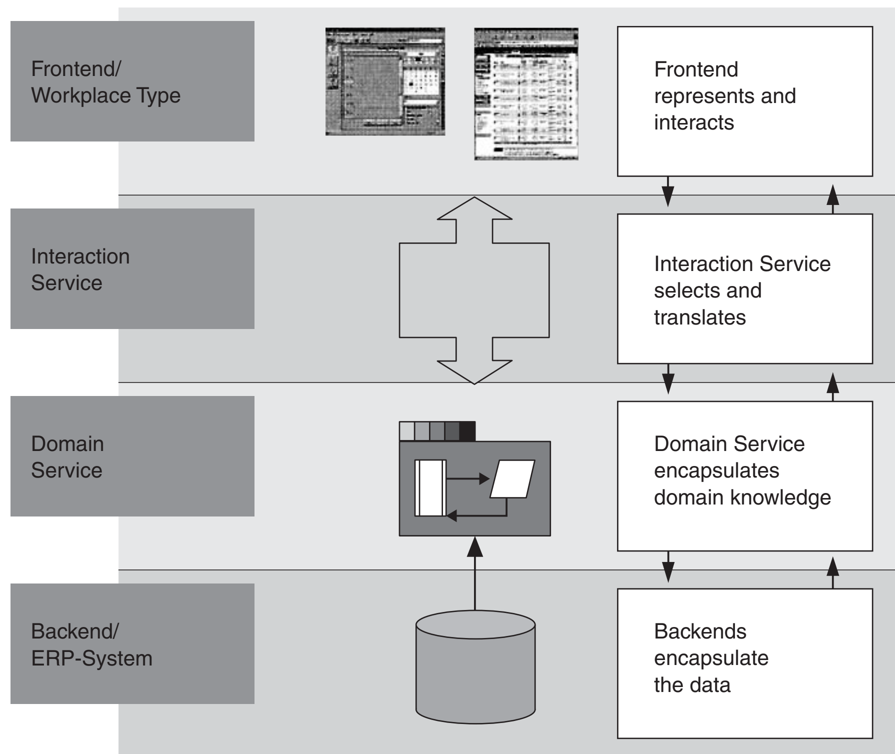


FIGURE 8.55 Logical layers of a service architecture.

Figure 8.55 shows the logical layers and their client-server separation. The domain service is usually located on a separate server and the user interface on the client. Then we need an intermediate layer that transforms the functionality of the service to the different interaction model of the frontends. We call it an interaction service, which can reside either on the client or the server.

TRADE-OFFS

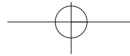
Operations of domain services should be atomic

The technical features of a domain service include the requirement that we treat its operations as being atomic, to the widest possible extent. This means that we should try to design all services to be stateless. Naturally, stateless services will have an impact on the interface. All implicit or explicit parameters required for a method have to be passed with each call. Parameters always have value semantics, that is, they do not include references to parameter objects. In general, you cannot use a domain service to directly access any of the materials it contains. To access a specific material, it is often useful to exchange unique identifiers across process boundaries.

Coping with state changes

Clients of many domain services need to be informed about state changes. For this reason, we generally allow clients to register with the interface of a domain service, so that they can receive announcements. This means that we specify an additional technical dependence, depending on the eventing mechanism used. Note at this point that the interaction model of some frontend types (e.g., HTML browsers) does not

Zull-08.qxd 31/8/04 2:35 PM Page 268



support such announcements. In this case, we have to provide expensive auxiliary constructions in the interaction service.

RATIONALE

Whenever you build a distributed application with different workplace types and technical front-ends, consider encapsulating the domain-related funtionality as domain services.

8.15 THE ENVIRONMENT PATTERN

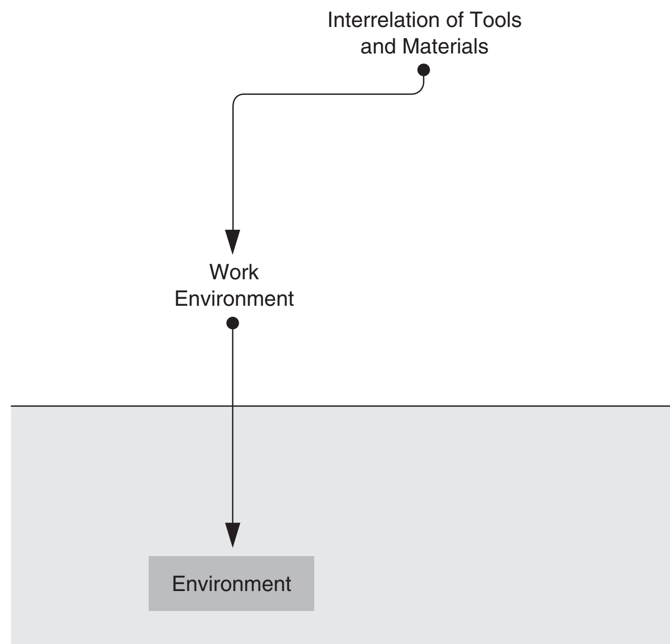


FIGURE 8.56 The *Environment* pattern.

PROBLEM

A work environment is a very important concept in the development of workplace systems for a specific application domain.

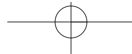
We use tools and materials within a limited space, such as on a desktop. In such an environment, we find a number of tools and materials. To manipulate a material by use of a tool, we bring both together and can always identify when we need which tools to work on which materials.

How can we implement an environment so that it is an abstraction of all the limited work spaces? How can an environment help to couple the appropriate tools and materials?

RELATE TO

The design pattern *environment* realizes the concepts outlined by the conceptual pattern *work environment* (see Figure 8.56).

Zu11-08.qxd 31/8/04 2:35 PM Page 269



BACKGROUND: ENVIRONMENT

An environment is a personalized space

The environment delimits the area that users move in to complete their tasks within an application domain. Since the environment can be different for each user, and each user can have their individual environment, different environments normally have to know their individual users, for example, by identifiers (user IDs). We can use these identities for building communication channels between environments.

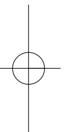
Tools and materials in an environment

Also, an environment manages tool and material types. It knows all tools its users can activate, and all the material types, for which its users can create instances. We know that tools operate on materials. An environment can determine which tools can manipulate which materials. In fact, an environment knows aspects, in addition to tool and material types. It can use these aspects to find out which tools and materials can be combined.

In addition to this management function, an environment knows which tools are active at a time, and which materials are currently manipulated by these tools. If several tools manipulate one single material, then they should always show the most recent material state. To this end, all tools have to be coordinated. The environment has the fundamental knowledge to coordinate tools.

While a tool manipulates a material, it is within the environment. It is not removed from the environment and passed to a tool for exclusive use. This corresponds to our notion of an environment, where we can use tools to manipulate materials, allowing several tools to operate on the same material within their environment. Also, it confirms our notion of spatial and logical dimensions discussed in Section 7.6.

If we say that a material must remain in its environment while it is manipulated by one or more tools, then we have to protect it so that the work effected on the material by a tool cannot be compromised. For example, we want to prevent a user from dragging a material to the trash can while another user is editing the same material.



Concepts of an environment

In summary, an environment knows the following domain elements:

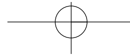
- user identification;
- workplace identification;
- existing tool types;
- existing material types;
- active tools; and
- materials currently manipulated by tools.

We have said in respect to tools construction that a tool object is used as a context representing the entire tool. If we split a tool into subtools, then the tool can be an observer of its subtools. This means that it represents the context of its subtools. Moving to the next higher level, we can think of the environment as the context of all the tools it contains. If the environment represents the context of its tools, then it has to implement the domain context of all of its tools.

SOLUTION

We use a class to build a environment. This class includes a so-called workspace, where we can arrange all objects needed in that environment. In addition to objects that an individual may require, the workspace also includes a number of permanently available containers, which are part of the basic equipment of an environment, for example, a template folder, a toolbox, and a trash bin to remove objects no longer used from the environment.

Zu11-08.qxd 31/8/04 2:35 PM Page 270



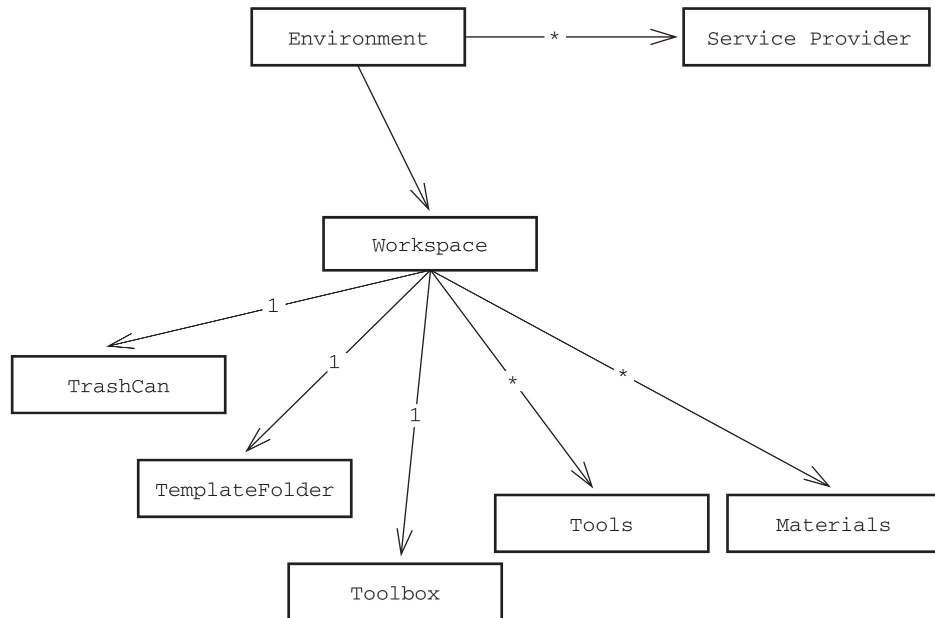


FIGURE 8.57

Environment and workspace.

EXAMPLE

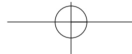
In the example in Figure 8.57 we sketch the standard equipment of an environment.

Features of an environment

The standard containers of an environment shown in Figure 8.57 can be implemented as domain containers. Based on the management functions of domain containers, we can implement a number of useful features.

- The toolbox can be used to identify tools generally available. An appropriate tool can be used to display the toolbox for the user and let users start tools from within the toolbox.
- A template folder could be used to hold available material types and return new instances of a material type upon demand. This expands the options we have from traditional manual template folders, where you normally find a finite set of templates. Once this reserve is exhausted, we have to refill it with new templates. In addition, materials held in template folders can age when the original is modified. We can solve both problems by using a software template folder. This folder is part of the environment, copying the original and returning copies upon request. Consequently, the template reserve will never be exhausted, and users always get the most recent version of the original.
- Material metainformation from within the environment show which tools can be used for which materials.
- As we have said, a tool cannot be started without a material. If no material is started when a tool is activated, then the system should use a new default material. Such a default material can be easily created for a tool from the template folder.
- When a tool modifies a material, then the tool can inform its environment to this effect. The environment can use the workspace to determine other tools that may manipulate the same material, and inform them about the first tool's modification to the material.

Zull-08.qxd 31/8/04 2:35 PM Page 271



TRADE-OFFS

As an alternative to using domain containers, we could implement toolboxes, template folders, and workspaces by technical containers (e.g., arrays or lists). In this case, however, we have to implement the domain functionality in the environment. Since an environment generally assumes a lot of tasks, it is recommended to use domain containers to keep the environment clear and easy to understand.

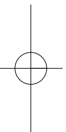
RATIONALE

Whenever you design and implement workplace systems you should consider using an environment. Even if you don't need a desktop interface, environments are useful for encapsulating and identifying a workplace in a distributed environment.

8.16 USING THE T&M DESIGN PATTERNS FOR THE JWAM FRAMEWORK

In this section we show the interplay among various T&M patterns. We have used these patterns in implementing the JWAM framework, which is a generic framework for constructing interactive applications in line with our approach. This section briefly describes these implementations and how they relate. Our description focuses on the relations and interplay between patterns.

The section begins with a description of the materials construction, followed by a description of the tools construction. Subsequently, we explain how we support domain value construction and domain container construction.



With its component model, the JWAM framework supports different options for user interface connection, including the concept of interaction forms (IAF) and presentation forms (PF). The forms concept supports the construction and use of forms. Similar to IAFs, forms are also components.

In a later section, we explain how domain services are built in JWAM, and finally how we implement a JWAM environment.

8.16.1 Materials Construction

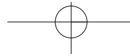
Materials construction is essentially supported by the `Thing` interface (see Figure 8.58) and the `ThingImpl` superclass. The level of `Thing` defines that each material has a name and a unique ID of the type `IdentificatorDV` across the entire system, where `IdentifierDV` is built as a domain value.

```
public interface Thing extends Serializable
{
public void rename (String newName);
public String name ();
public String[] toStrings ();
public void setID (IdentificatorDV id);
public IdentificatorDV id ();
public String typeDescription ();
public ThingDescriptionDV thingDescription ();
}
```

FIGURE 8.58

The Thing interface.

Zull-08.qxd 31/8/04 2:35 PM Page 272

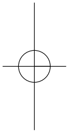



```
public class ThingDescriptionDV extends DomainValueImpl
{
public String name ();
public Class thingClass ();
public String iconName ();
public IdentificatorDV id ();

// Management of named values
public int entryCount ();
public String valueName (int index);
public DomainValue value (int index);
}
```

FIGURE 8.59

The Thing DescriptionDV interface.



In addition, each material can return a description in the form of the `ThingDescriptionDV` class (see Figure 8.59) about itself. This description is also a domain value. It includes information about material names, material IDs, the class a material belongs to, and so on.

Specific materials inherit from `ThingImpl` or fulfill at least the `Thing` interface. In addition, subclasses of `ThingDescriptionDV` can be defined for specific materials to provide additional information. One of the most frequent requirements in real-world applications is that the `ThingDescription` for a special material should carry additional information, but it should not define new operations. This means that no subclass should be derived from `ThingDescriptionDV`. It is normally sufficient to pass such additional information as a list of named values to `ThingDescriptionDV` when it is created. These value lists can then be output in tables.

Note that we derive not only materials but also all "things" from `Thing` or `ThingImpl`, respectively. In addition to materials, this includes tools, domain containers, automatons, and domain services.

8.16.2 Tools Construction

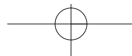
We use the `Tool` interface and the `ToolImpl` superclass for our tools construction. Specific tools inherit from `ToolImpl` or implement the `Tool` interface as a minimum requirement. Since tools are also "things," `Tool` inherits from `Thing` and `ToolImpl` inherits from `ThingImpl`. In addition, `ToolDescriptionDV` is a subclass of `ThingDescriptionDV`, which describes a tool in more detail.

Each tool can provide information about its functionality and its interaction. This information is represented by two interfaces, `Functionality` and `Interaction`.

Depending on whether or not we want to separate function and interaction when building a tool, a tool can be derived from `ToolFpIpImpl` (see Figure 8.60) or `ToolMonoImpl` (see Figure 8.61). `ToolFpIpImpl` represents a tool with its FP and IP separated, and `ToolMonoImpl` is the superclass for monolithic tools with their FP and IP not separated. In this case, the tool object returns itself when it is queried for its functionality or interaction.

At their interfaces, functionalities define events in the form of objects of the `Event` class. Objects that define events at their interface have to implement the

Zu11-08.qxd 31/8/04 2:35 PM Page 273



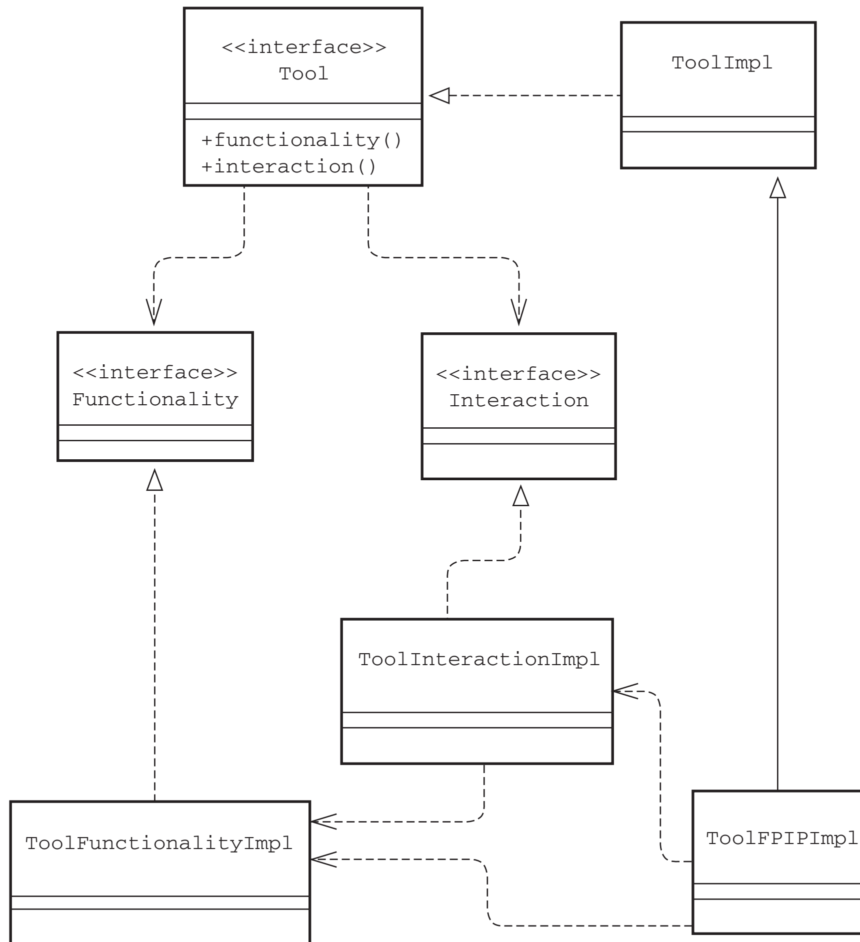
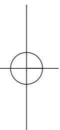


FIGURE 8.60
A tool with separate interaction and functionality.

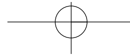


EventSubject interface. Therefore, the functionality interface inherits from EventSubject. The observers register directly with the events and concurrently pass objects, the classes of which implement the EventReaction interface. Since EventReaction has only one defined operation (update), we would normally work with anonymous inner classes, which are defined by an observer. In the JWAM framework, observers can be functionalities and interactions. Figure 8.62 shows the classes of the event mechanism.

A chain of responsibility existing between subtools and context tools is implemented by use of the RequestHandler interface and the Request class. Note that the tool objects are actually request handlers, which means that the Tool interface inherits from the RequestHandler interface. Each functionality knows its successor under the RequestHandler interface and can pass objects of the Request class to this request handler for further processing.

JWAM-based tools should not depend on a specific user interface. Therefore, rather than letting Interaction directly use the AWT or Swing windows and

Zull-08.qxd 31/8/04 2:35 PM Page 274



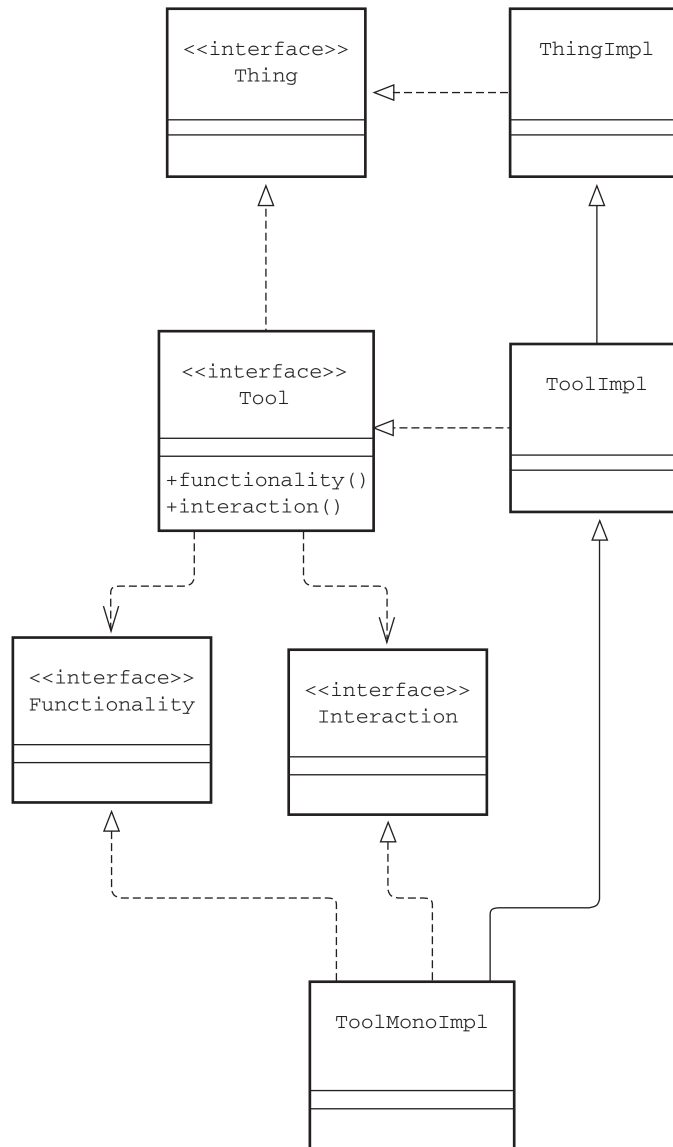
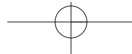


FIGURE 8.61
Monolithic tools.

widgets, it uses GUI contexts. A *GUI context* is an abstraction of windows and panels. A GUI context contains a set of user interface elements and may include embedded GUI contexts. Figure 8.63 shows an overview of the tools construction.

8.16.3 Domain Values

The JWAM framework uses classes to implement domain values. Domain value objects are always created by a related factory. Since there has to be a factory class for each


Zu11-08.qxd 31/8/04 2:35 PM Page 275

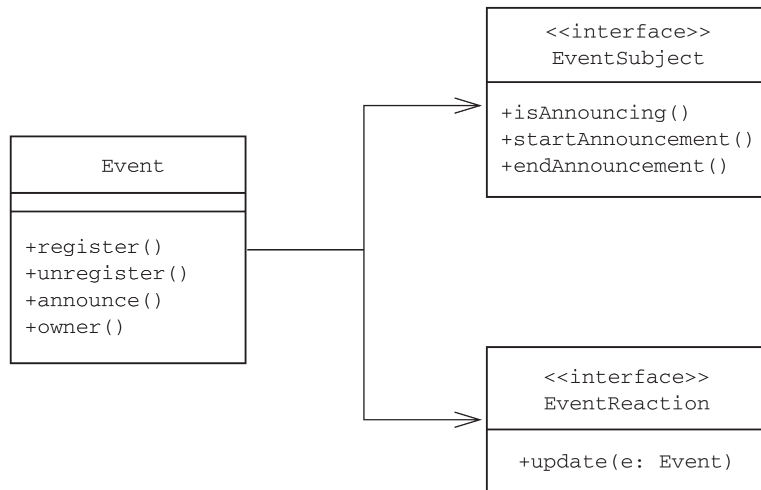


FIGURE 8.62 Example for an event pattern.

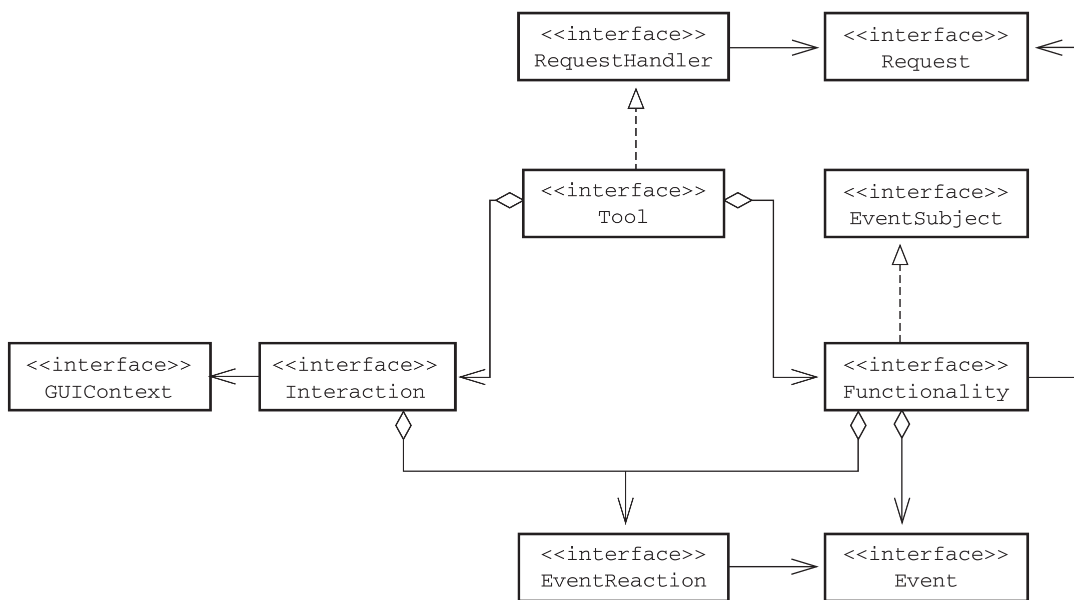


FIGURE 8.63 Tools construction overview.

domain value class, this relation is expressed in that the factory classes are inner classes of the domain value classes. There are two generic interfaces: `DomainValue` for domain values and `DomainValue.Factory` for factories. In addition, abstract implementations are made available: the class `DomainValueImpl` and the class `DomainValueImpl.Factory` (see Figure 8.64).