Bureaucracy

Dirk Riehle

The Bureaucracy pattern is a recurring design theme used to implement hierarchical object structures which allow interaction with every level of the hierarchy and maintain their inner consistency themselves. It is a composite pattern which is based on the Composite, Mediator, Chain of Responsibility and Observer pattern. Composite patterns require new presentation and modeling techniques since their complexity makes them more difficult to approach than non-composite patterns. In this paper, I use role diagrams to present the Bureaucracy pattern and to explore its design and implementation space. Role diagrams have proved to be very useful to get a grip on this complex pattern, and I believe they will work well for design patterns in general.

INTRODUCTION[1]

The Bureaucracy pattern is a composite pattern which helps developers build self contained hierarchical structures that can interact with clients on every hierarchy level but need no external control and maintain their inner consistency themselves. This pattern scales well to structure large parts of an application or a framework. It is based on the idea of modern bureaucracy [Weber47] which seems to work well for software systems (at least).

A composite pattern is first of all a *pattern*: It represents a design theme that keeps recurring in specific contexts. I call it a *composite* pattern, because it can best be explained as the composition of some other patterns. However, a composite pattern goes beyond a mere composition: It captures the synergy arising from the different *roles* an object plays in the overall composition structure. As such, composite patterns are more than just the sum of their constituting patterns.

The presentation of the Bureaucracy pattern is based on roles rather than “participants” as in the design pattern template of the Gang-of-four [Gamma+95]. The notion of role is crucial to understand the idea of composite patterns: An object in an instantiation of a composite pattern can play several roles and thus participate in different overlapping pattern instantiations. A composition of patterns turns into a composite pattern, if and only if (1) a relevant synergy between the roles an object plays arises, and (2) this synergy can be observed as a recurring design theme.

Section 2 presents the pattern without referring to the process and means of digging it out. However, there are many other composite patterns worthwhile being documented, so section 3 presents the concepts and techniques I used for eliciting the essence of the Bureaucracy pattern. This includes the role diagram notation for design patterns, the role relationship matrix as a means for analyzing prototypical pattern applications, and ways of interpreting the matrix. I hope that these concepts and techniques will prove to be helpful to fellow pattern writers.

[1] In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle and Frank Buschmann. Reading, MA: Addison-Wesley, 1997. Chapter 11.

THE BUREAUCRACY PATTERN

The general presentation form is based on the design pattern template of the Gang-of-four [Gamma+95], with some enhancements. In particular, the Structure and Collaboration sections are now based on roles rather than participants. This enhancement lets us discuss the pattern's structure and implementation issues on a broader scale.

The pattern requires a prior understanding of the Composite, Mediator, Chain of Responsibility and Observer pattern on which it is based (see [Gamma+95] for their documentation).

To avoid possible confusion, I would like to clarify the use of the word "composite." When written with a small "c" as in "a composite pattern," it denotes a pattern composed from further patterns. When written with a capital "C," it refers to the Composite pattern, either to the pattern itself or to its Composite participant as described in [Gamma+95].

Intent

Define a self-contained hierarchical structure which maintains its inner consistency, accepts interaction with clients on any level of the hierarchy, and scales for application design. This pattern combines the Observer, Chain of Responsibility, Mediator and Composite pattern to form a composite pattern.

Motivation

Suppose you are developing a small tool for managing the compilation shell scripts and other utilities of a much bigger project. This tool lets you work on a new version of the compilation support separate from the one currently in use. Such a tool will have a browser at its heart which lets you select and edit text files. It might also offer a shell window for syntax checking and testing scripts as well as compiling C utilities. Finally, the tool needs some configuration data of its own, for example, the working and the target directory.

The utility manager is a single application with an overall tool structure likely to follow the Composite pattern as depicted in figure 1. It is built from tools, with the utility manager tool at the top, a file browser, shell and version manager tool in the middle, and a lister and text editor tool at the bottom.

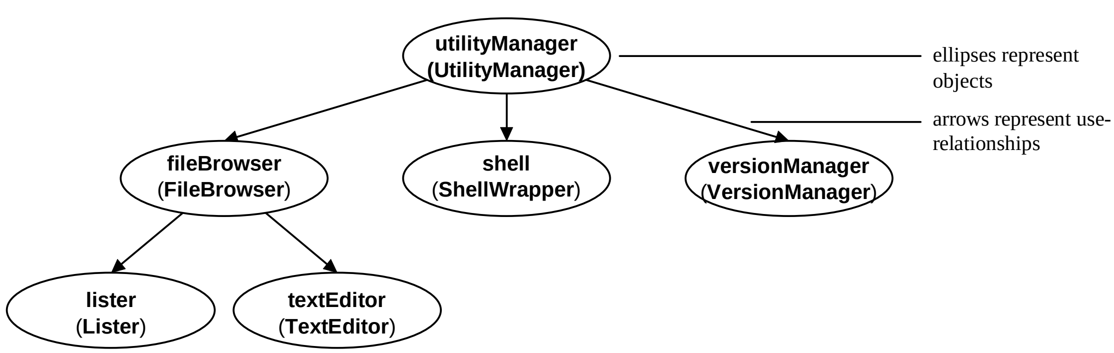

Figure 1: A hierarchy of tool objects forming a utility manager tool

A tool structure like this makes it challenging to correctly distribute functionality and responsibility among the participating tools. Without following a coherent pattern, spreading responsibilities over the hierarchy is likely to lead to a mess where it is unclear which tool is allowed to carry out which request at what point of time. Uncontrolled behavior and unmaintainable code are the consequences.

The Bureaucracy pattern shows how to coherently distribute responsibilities in such a hierarchy. Objects in a Bureaucracy hierarchy play two or three out of four roles. The four roles are the Clerk, Manager, Subordinate and Director roles. Every object plays the Clerk role. A bottom level object plays both the Clerk and Subordinate role. An intermediate level object plays the Clerk, the Manager (for its Subordinates) and the Subordinate role (for its Manager). A top level object, that is the hierarchy's root object, plays the Clerk, Manager and Director role.

An object playing the Manager role coordinates and manages objects playing the Subordinate role. The resulting responsibility assignments are defined by the Mediator pattern. The hierarchical structure is defined by the Composite pattern, with every object being a Clerk and also either a Manager or Subordinate or both. The hierarchies boundary conditions are the Manager which is a Director and therefore represents the root of the hierarchy, and the Subordinates which are no Managers and thus manage no further Subordinates.

The communication protocol between a Subordinate and its Manager is defined by the Chain of Responsibility and the Observer pattern. If a Subordinate is asked to carry out a request which it cannot fully handle itself, it forwards the request up the hierarchy to its Manager, expecting the Manager to possess more context knowledge to carry out the request. This process is defined by the Chain of Responsibility pattern. If a Subordinate fully handles a request, it informs its Manager about relevant state changes which might have occurred. The Manager interprets the state change and might initiate further action. This communication protocol is defined by the Observer pattern.

Figure 2 shows a class hierarchy frequently used for implementing the Bureaucracy pattern. The class hierarchy shows how the roles from the constituting patterns are assigned to class interfaces. The Composite pattern defines the Node, Child, Parent and Root roles, the Mediator pattern defines the Colleague and Mediator roles, the Chain of Responsibility pattern defines the Handler, Successor, Predecessor and Tail roles, and the Observer pattern defines the Subject and Observer roles (see section 3).

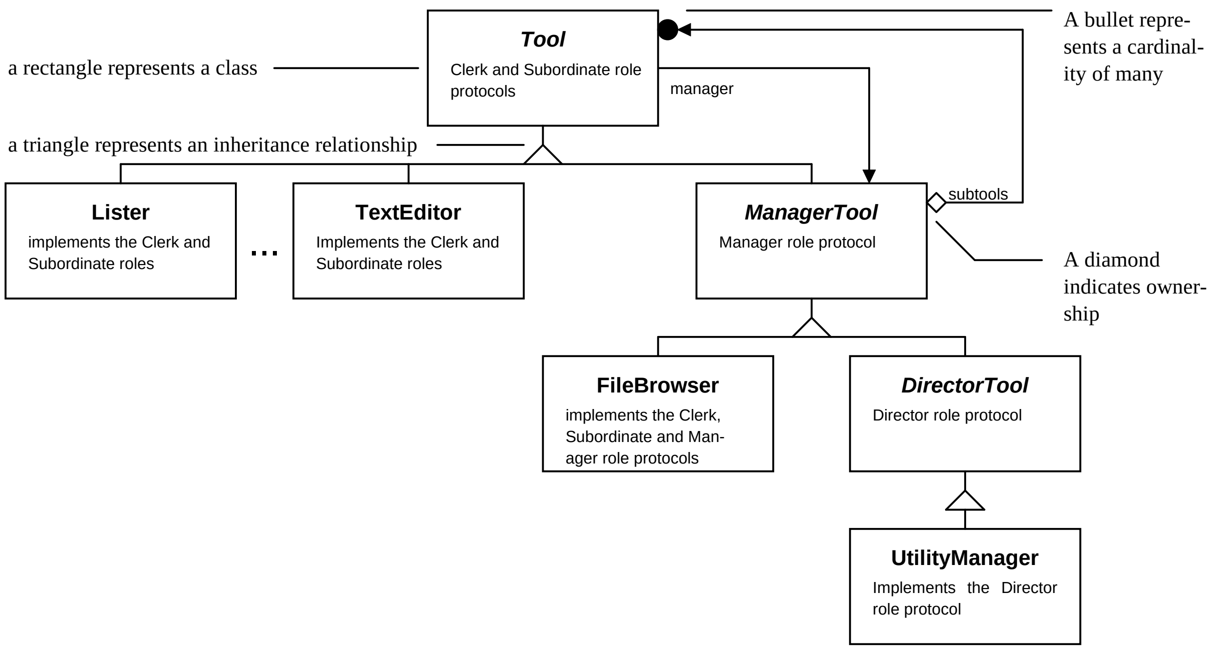

Figure 2: Class diagram for implementing the utility manager tool

Combining the Composite with the Mediator, Chain of Responsibility and Observer pattern helps you build complex object hierarchies while staying in control of how functionality and responsibility is distributed among the participating objects.

Applicability

Use the Bureaucracy pattern if you need to structure a large object-oriented body of state and behavior that can be expressed well in terms of a hierarchy, and

- clients are free to interact with any part of the hierarchy at any time, and
- the integrity of the overall hierarchy has to be maintained by the hierarchy itself.

Do not use the Bureaucracy pattern, if everything above seems to hold, but

- the hierarchy is not self sufficient, has to be controlled in detail by some external clients, and the requirements of this external control are hard to anticipate in advance.

If the hierarchy is subject to fine-grained manipulation by clients, it will have to go through inconsistent states until the client commits its interaction. This is not appropriate for the Bureaucracy pattern and raises the given counter indication.

Structure

Figure 3 shows the structure of the Bureaucracy pattern. It is presented as a role diagram, a notation which focuses on roles rather than on classes. A role diagram supports all relationships between roles that are also possible between objects, including association and aggregation. In addition, it defines the notion of composition constraint, which lets you specify whether two roles are always played together or not. A composition constraint is depicted by a gray arrow; it is a binary relationship between two roles. In figure 3, an object playing the Manager role also always plays the Clerk role, etc. It maps nicely on class inheritance on an implementation level.

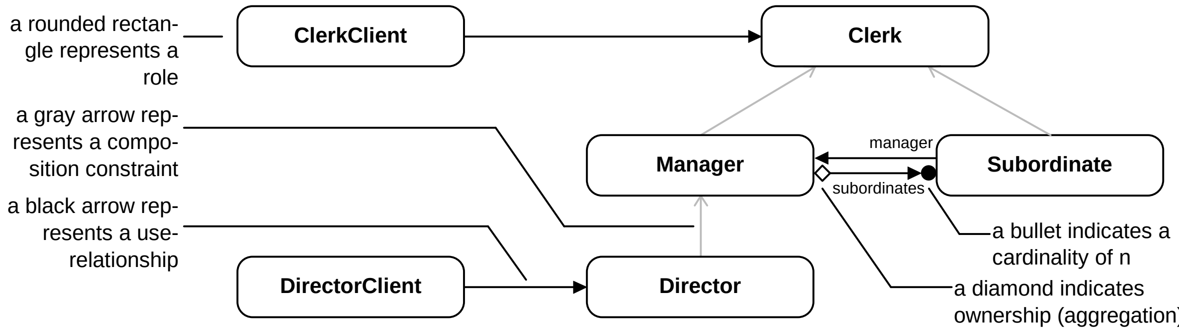

Figure 3: Role diagram of the Bureaucracy pattern

A role defines the responsibilities of an object within a collaboration of some objects. It is expressed as a role protocol. An object can participate in several collaborations and therefore can play several roles at once. A class implementing an object has to implement all those roles the object is playing in the different collaborations. The class interface will then be composed from the different role protocols.

A role can be composed from further roles. For example, the Mediator role from the Mediator pattern is an atomic role, while the Subordinate role from the Bureaucracy pattern is a composite role comprising the Child, Colleague, Predecessor and Subject roles from the Composite, Mediator, Chain of Responsibility and Observer pattern.

Figure 4 shows the most commonly found class diagram used for implementing the Bureaucracy pattern. The static structure of the class diagram is governed by the Composite pattern, with the Subordinate class representing the Component class defined in [Gamma+95], the Manager class representing the Composite class and the Director class representing the Root class (not present in [Gamma+95]).

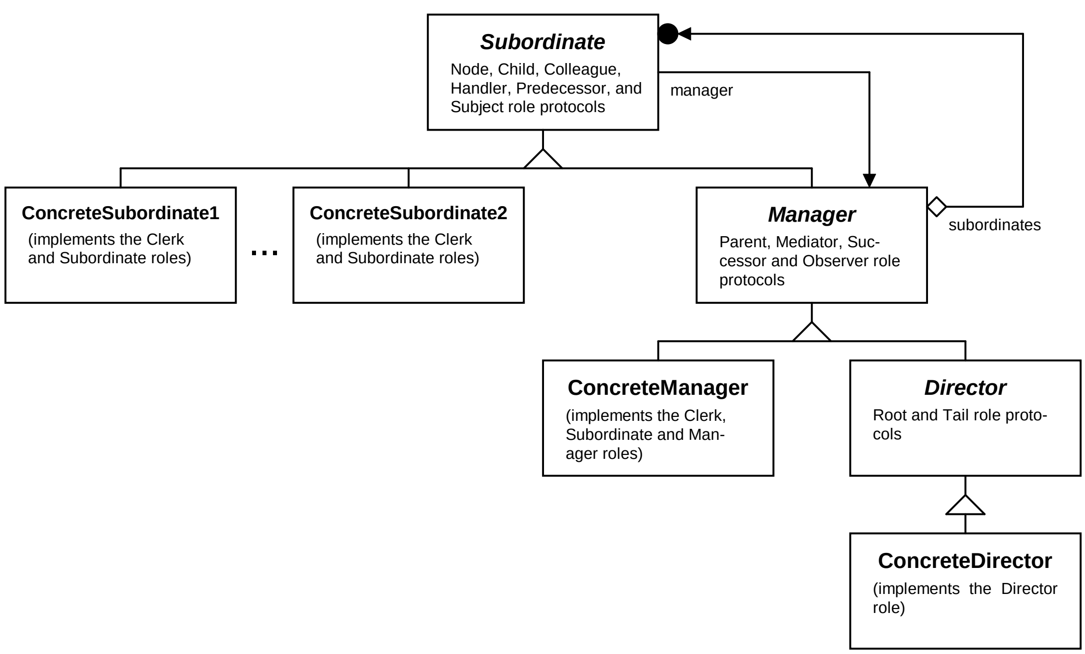

Figure 4: Most frequently found class diagram of the Bureaucracy pattern

The dynamic behavior of the pattern is governed by the Mediator, Chain of Responsibility and Observer pattern. The Manager acts as a Mediator to its Subordinates which act as Colleagues. The communication protocol from Subordinates to their Managers is described by the Chain of Responsibility and Observer pattern. Each pattern follows its own dynamics as described by the interaction diagrams in [Gamma+95]. The integration of the different roles from the constituting patterns in a single object delivers the synergy which drives the Bureaucracy pattern.

Figure 4 is not the only possible class diagram that can be used to implement the Bureaucracy pattern. In principle, every role protocol can be represented as an interface class of its own, and concrete classes of objects participating in a Bureaucracy hierarchy can selectively inherit from those interface classes according to the roles they play in the different collaborations.

Roles

The roles of the Bureaucracy pattern from figure 3 build on the roles from the constituent patterns.

- The *Manager* role combines the Parent, Mediator, Handler, Successor and Observer roles. A Manager coordinates objects playing the Subordinate role. It receives requests from them and observes their state changes.
  - Playing the *Mediator* role from the Mediator pattern, it manages, coordinates and delegates to its Subordinate objects. It does whatever it needs to do to keep the hierarchy in a consistent state.
  - Playing the *Successor* role from the Chain of Responsibility pattern, it receives requests from its Subordinate objects. It can either handle these requests or forward them to its own Manager.
  - Playing the *Observer* role from the Observer pattern, it receives state change notifications from its Subordinate objects. It can react to these notifications, for example by manipulating its Subordinate objects or by generating requests.
- The *Subordinate* role combines the Child, Colleague, Predecessor and Subject roles. A Subordinate is managed and coordinated with other Subordinates by a Manager. It receives requests from clients and decides whether to execute them or to forward them to its Manager. It notifies its Manager about state changes, for example when having finished executing a client request.
  - Playing the *Colleague* role from the Mediator pattern, it is subject to management by its Manager.
  - Playing the *Predecessor* role from the Chain of Responsibility pattern, it forwards client requests to its Manager.
  - Playing the *Subject* role from the Observer pattern, a Subordinate object is subject to observation by its Manager.
- The *Director* role holds the ultimate responsibility for the hierarchy and manages it according to its needs and purposes. An object playing the Director role also always plays the Manager. It extends the Manager role with the Root and Tail role.

  - Playing the *Root* role from the Composite pattern, it holds full responsibility for the whole object hierarchy which it manages according to its needs and purposes.
  - Playing the *Tail* role from the Chain of Responsibility pattern, it has the final decisive power how to handle client requests. It might decide to handle them, ignore them, or to raise an exception.
- The *Clerk* role defines the standard behavior of an object in the hierarchy as its responsibility with respect to external clients. It combines the *Node* and *Handler* roles from the Composite and Chain of Responsibility patterns. Every object in the hierarchy always plays the role of Clerk.

Collaborations

The single roles have the responsibilities of the participants as described by the constituting patterns Composite, Mediator, Chain of Responsibility and Observer. Of interest to a composite pattern, however, is the synergy which arises from the integration of several roles in a composite role as played by a single object.

Playing roles in different pattern instantiations means switching contexts during execution. An action in one context can cause an object to act in another context. The interface to each context, that is a pattern instance, is the object's role in that context.

- *Mediator and Successor (Manager).* If an object playing both the Mediator and Successor roles receives a request from a Subordinate object, it reacts to this request in terms of the whole object (sub-)hierarchy it stands for. If it chooses to handle the request, it might have to coordinate changes in subtrees different to the one the request emerged from.
- *Mediator and Observer (Manager).* If an object playing both the Mediator and Observer role receives a change notification from a Subordinate object, it has to react to the changes that happened in one of its subtrees. This might include coordinating all needed changes to other subtrees including the one from which the change notification emerged.
- *Observer and Handler (Manager).* If a Manager observing a Subordinate receives a state change notification from it, it might choose to create a request which it forwards to its own Manager. Thus, the interpretation of some Subordinate's state change causes a request to be created and forwarded up the hierarchy.
- *Handler and Subject (Subordinate).* An object might change its own and other objects state in reaction to an external request. This state change has to be announced and thus its Manager has to be notified.
- *Root and Tail (Director).* The Director of the whole tree has the final power to decide what will happen with a client request. If every subordinate object has given up on handling a request, the Director has the last word, either to handle the request, raise a system error, or to ignore the client's request altogether.

These role/role relationships to be integrated in a single object embody the synergy between the constituting patterns which turns the Bureaucracy pattern from an arbitrary composition of some patterns into a composite pattern.

Consequences

On the positive side, the Bureaucracy pattern

- defines a Composite hierarchy consisting of Managers and Subordinates. It can be assembled easily and is based on recursive decomposition. This hierarchy is self-stabilizing and follows inner rules of consistency which cannot be changed by clients (unless a Legislature protocol is introduced to the Director role).
- helps you build complex hierarchical structures which might define large parts of an application or framework. A Bureaucracy has its own rules which it enforces. This turns instantiations of the pattern into autonomous entities. They are either at the forefront of interaction with the user, or they interact in an agent-like style with other system components.
- makes it easy to introduce new Managers and Subordinates. Due to the predefined protocols, new objects can be embedded and reused easily. Viewed the other way round, a Bureaucracy instantiation can be assembled easily since the involved patterns define a simple plug-and-play structure.
- shows how you can distribute complex functionality over a large object structure in a coherent way. By applying the principle of divide-and-conquer, you can get a better grip at the system's complexity.

On the negative side, the Bureaucracy pattern

- restricts client interactions to a rather coarse-grained level. A client request is carried out according to the Bureaucracy's inner rules usually allowing for no client interference. Client requests might cause complex hierarchy internal control and data flow and might include hierarchy restructuring.
- might cause communication overhead, because Subordinates cannot communicate directly with each other. Instead, communication flow is always mediated by the Manager.

Implementation

Issues relevant to the implementation of the constituting patterns can be found in [Gamma+95]. Additional issues arise from the pattern interaction synergy. When implementing the Bureaucracy pattern, consider the following aspects:

- *Who's in charge?* The basic question to be answered when implementing a Bureaucracy is which responsibilities to assign to which level in the hierarchy. Generally speaking, the higher a Manager in the hierarchy, the more power and the more context knowledge it has to control the other objects.

  The ideal bureaucracy is built on the idea that the higher a clerk is positioned, the more knowledge and qualification he or she possesses [Weber47]. Here the analogy breaks down: There is no need to provide Managers high in the hierarchy with more specialized and elaborate implementations. In fact, objects high up in the hierarchy might be based on rather general implementations while objects further down the hierarchy do much of the detailed work.

- *Separation of Observer and Chain of Responsibility protocol.* You might consider reusing the Observer protocol to implement the Chain of Responsibility. However, this can make control flow hard to understand. Therefore, both protocols should be kept separate. It should be clear at any time whether an object is notifying its observer about a state change or whether it is forwarding a request it cannot handle alone.
- *Sharing of implementation state.* In the Bureaucracy pattern, the Parent of a Child, the Mediator of a Colleague, the Successor of a Handler and the Observer of a Subject is always the same object so that it can be handled by a single object reference.
- *Interaction on all levels versus interaction via Director object.* It depends on the type of interaction with clients whether these clients are allowed to interface with any object within the hierarchy or may communicate using the Director only.

  In graphical user interfaces, a user playing the role of a client can usually interact with any object in the hierarchy, thereby directly manipulating it. This is an important aspect of the Bureaucracy pattern, since this interaction is the reason for composing the four patterns: This type of interaction can bring the hierarchy into an inconsistent state at any level. The combination of Mediator, Chain of Responsibility and Observer is used to stabilize the hierarchy.

  In other situations, it is desirable to allow interaction only via the Director. In such situations, each Manager gets the chance to intervene and possibly interrupt the intended interaction of a client with its target object in the hierarchy. Furthermore, each Manager has the power to decide which way the interaction should go and thus which Subordinate a request is delegated to.
- *Short communication paths.* Some systems, for example HotDraw [Johnson92], allow Subordinates to observe other Subordinates in order to enable short communication paths. This defeats the purpose of the Mediator role played by the Subordinate's Manager and should be applied only if it is clear that the Manager will never have to interfere with this communication.
- *Initialization of the hierarchy.* Usually the hierarchy is built once. Sometimes it is desirable to exchange whole subtrees, as it is the case in user interfaces where some flags determine which variant of a user interface is to be shown and which is to be hidden. These subtrees can be created in advance and exchanged at runtime, for example by mapping and unmapping them.

  If clients frequently change the hierarchy, the Bureaucracy pattern might not be applicable. The more detailed a client's manipulation of the hierarchy is, the more likely it will conflict with the hierarchy's inner consistency rules. If a parent/child relationship is all you can define, there is no need to introduce the Observer and Chain of Responsibility pattern, and hence no Bureaucracy pattern.

There are some general issues related to implementing patterns based on role modeling. When implementing composite patterns, you have to face a number of difficult problems:

- *Class interface construction.* Role protocols can be defined as mixin-classes. Using multiple inheritance, a class implementing a role from the Bureaucracy pattern can simply inherit from all role protocols it needs to support. If multiple inheritance is not available, the role protocols might be maintained on paper and copied by hand into the class interfaces. In single inheritance systems, tool support for protocol mixin is likely to be very helpful [Reenskaug96, Riehle+96].

- *State integration.* If you use a single class to implement a Manager or a Subordinate, you can make sure that the abstract state defined in the different role protocols maps well on a single implementation state. In complex role modeling situations, however, you will probably use Decorators [Gamma+95] to adapt a core object to a specific collaboration. [Riehle95] shows how to lookup context adaptation objects for a given specification.

State integration of the abstract state defined by different interfaces in a single component is a difficult research problem, addressed by different researchers now. This includes the work of Reenskaug et al. [Reenskaug96], Harrison and Ossher [Harrison+93], Kiczales et al. [Kiczales+96], and the work of our group.

Sample Code

I illustrate a possible implementation of the class diagram presented in figure 2.

Class `Tool` comprises the role protocols of Child, Mediator, Handler and Observer as composed by the Subordinate role. This leads to the following interface:

```
class Tool
{
public:
    // Child role
  virtual ManagerTool* getManager();
  virtual void setManager(ManagerTool* newManager);
  virtual ManagerTool* asManagerTool();

    // Handler role
  virtual void forwardRequest(Request* request);

    // Subject role
  virtual void notify(Event* event);

protected:
  ManagerTool* manager;
};
```

A `Tool` object uses the Handler and Subject protocol to communicate with its Mediator when playing the Colleague role. Thus, there is no need to define an additional Colleague protocol which can be introduced on a derived class level only.

`ManagerTool` enhances the `Tool` interface with the role protocols of Manager, Successor and Observer:

```
class ManagerTool : public Tool
{
public:
    // Parent role
  virtual ManagerTool* asManagerTool();

    // Successor role
  virtual void handleRequest(Request* request);

    // Observer role
  virtual void update(Tool* tool, Event* event);
};
```

A Mediator role protocol can only be specified on the level of a concrete Manager like the `FileBrowser` class discussed below. Thus, there is no explicit Mediator role protocol in the `ManagerTool` class interface.

In general, the different role protocols are implemented according to the different patterns they are derived from. Of interest is the interaction between the different roles in a single object, so lets take a look now at the two scenarios from the Motivation section.

In the first scenario, the user double-clicks on an item in the list box of the lister. After catching and dispatching the select callback from the user interface, the following operation of `Lister` is called:

```
void Lister::selectItemRequested(int index) {
  DirItem* item = directory->at(index);
  Request* request = new SelectItemRequest(item, index);
  forwardRequest(request);
}
```

This operation interprets the user interaction as a request to select an item. By definition, the lister cannot fully handle this itself but forwards it to its Manager, the file browser. The browser dispatches the request to its implementation of `handleSelectItem`, an operation of its Mediator role protocol. It looks like this:

```
void FileBrowser::handleSelectItem(SelectItemRequest* request) {
  String itemName = request->item->getName();

  if (editor->hasDocChanged()) {
    // warning dialog
  }
  else
  if (!FileSystem::Instance()->isAvailable(itemName)) {
    forwardRequest(request);
  }
  else {
    selectItem(request);
  }
}
```

Once everything has been prepared properly, `selectItem` of `FileBrowser` is called. This might be done either from the superordinate utility manager tool or from the operation just listed. `selectItem` might look like this:

```
void FileBrowser::selectItem(SelectItemRequest* request) {
  lister->selectItem(request);
  String itemName = request->item->getName();
  Document* doc = FileSystem::Instance()->LoadDocument(itemName);
  editor->setDocument(doc);
}
```

This scenario has shown how a client request travels up the hierarchy and is handled based on the tools' responsibility assignments. Handling such a request often means coordinating subordinate tools like the lister and text editor.

The second scenario illustrates the interaction between Observer and Chain of Responsibility. Assume that the user has edited a file and just pushed the Save button. The activate callback of the push button is caught and dispatched to the following operation:

```
void TextEditor::saveRequested() {
    // transfer changes from working copy to original
  ...
    // notify Manager about save
```

```
  Event event = new OriginalUpdatedEvent;
  notify(&event);
}
```

The editor has been designed in such a way that it lets users manipulate a working copy of a document and upon Save transfers the changes to the original `Document` object originally passed in by the file browser. No client request has to be created since the editor fully handles the user interaction. However, an important state change of the editor (and the document) has occurred, and the superior tool has to be informed about this.

Thus, `notify(&event)` is dispatched to:

```
void FileBrowser::checkOriginalUpdated() {
  Document* doc = editor->getDocument();
  FileSystem::Instance->SaveDocument(doc);
  Request* request = new CheckDocumentRequest(doc);
  forwardRequest(request);
}
```

The state change notification of the text editor is interpreted and handled by the file browser. It creates and forwards a request to the utility manager to handle the situation of a new document version. The utility manager has to react to the changed set of utility and configuration files. If the document is a C file, it might startup a C compiler, or it might check in the file into the version control system. However, this is irrelevant to the file browser. It has simply been defined in such a way that it can handle a changed file to some extent, but eventually has to forward the request to a superior tool.

These two scenarios illustrate the interaction of Observer and Chain of Responsibility and their meaning as a communication mechanism between Subordinates and their Managers as required by the Mediator pattern. Both patterns are used to make tools easier reusable so that they can fit into a self stabilizing hierarchy.

The Observer pattern is used if a Subordinate has carried out a request to its own satisfaction. The Manager is notified upon request completion. It then interprets and reacts to the Subordinate’s state change according to its context, possibly creating a request. However, it faces a situation in which the original request has already been fully handled so that no detailed control is possible anymore.

The Chain of Responsibility pattern is used if components know that they can only partially handle a request. Thus, they are designed in such a way that they can do their part of the work but also allow Managers to apply their broader context knowledge. A request is passed up the hierarchy until a Manager fully handles it.

Observer and Chain of Responsibility interlock: Each one starts where the other one ends. A change notification using Observer might cause a Manger to create a request using Chain of Responsibility. Handling the request might then lead to a state change which causes another change notification, etc.

Known Uses

Many frameworks for the design of interactive applications use the Bureaucracy pattern, including ET++, InterViews, HotDraw, PowerPlant and the Tools and Materials Metaphor frameworks. I adjusted the class names in the following examples to increase readability.

ET++ [Weinand+94] provides a class EventHandler which defines the Chain of Responsibility protocol. The Observer protocol is already in place with the framework's root class Object. The Manager subclass of EventHandler defines the composite structure of Managers which can be embedded in Managers. In addition, each Manager acts as a Mediator to its subordinate Managers. Like many other frameworks, for example PowerPlant, ET++ defines a break between non-visual classes like Manager (and its subclasses like Application and Document) and VisualObjects directly accessible in the user interface. Visual objects are at the forefront of interaction with the user and thus far down the Bureaucracy hierarchy. The VisualObject hierarchy is also derived from EventHandler. Unlike Manager, it defines a CompositeVisualObject class to make the Composite pattern explicit.

PowerPlant [Trudeau96] offers an interesting implementation of the Bureaucracy pattern. It separates the different protocols into different superclasses: The Chain of Responsibility protocol is defined by the LCommander class and the Observer protocol is defined by the LListener and LBroadcaster classes. LCommander also defines a protocol for managing subcommmanders so that it represents an application of the Composite pattern. The Mediator protocol is always concrete and therefore not available as a mixin-class. This allows users to mix and match the different protocols; they become independent from a predefined single inheritance hierarchy. PowerPlant applies the Bureaucracy pattern in a similar fashion to ET++ by providing Application, DocApplication, Document, View and Pane classes the instances of which form a Bureaucracy.

The Tools and Materials Metaphor frameworks [Riehle+95b, Riehle+96] use the Bureaucracy pattern to structure the functional parts of tools. In [Riehle+95a], the FunctionalPart class represents the Subordinate role and the class CompositeFunctionalPart represents the Manager role. It implements the Bureaucracy pattern in a rather clean way as discussed in the Structure section.

The PLOTS pattern language of transport systems [ZF97] presents an application of the Bureaucracy pattern in the domain of transport systems. The class RouteComponent is the root class of a several layers deep Composite pattern application. The inner consistency of the hierarchy is maintained using the Chain of Responsibility, Observer and Mediator pattern.

CONCEPTS AND TECHNIQUES

This section discusses the concepts, notations and techniques which I used to dig out the Bureaucracy pattern. The first subsection presents the *role diagram* notation for design patterns and applies them to the Composite, Mediator, Chain of Responsibility and Observer pattern. The second subsection defines the notion of *prototypical pattern application* and introduces the *role relationship matrix*, an analytical means for eliciting the relationships between roles from overlapping pattern instantiations. I illustrate the issues using the Bureaucracy pattern as an example. The third subsection finally discusses the interpretation of the role relationship matrix and shows how it helps to derive the essence of the Bureaucracy pattern.

Role diagrams

Role diagrams are a means for describing collaborations of objects based on the roles the participating objects play in the collaboration. A role has a role protocol associated with it which represents the interface through which an object is accessed within the described collaboration. The notion of role diagram is based on Reenskaug's role models [Reenskaug96]; I enhanced it with the idea of composition constraint to indicate constraints between roles, for example if an object always plays a certain set of roles together.

Role diagrams are more abstract than class diagrams—they are less implementation oriented. Thus, role diagrams and class diagrams complement each other: Role diagrams focus on the essential collaboration and omit implementation details while class diagrams show how the collaboration can be implemented efficiently. I present a more detailed discussion of the resulting levels of abstraction for patterns and pattern composition in [Riehle96].

Figures 5 to 8 show the Composite, Mediator, Chain of Responsibility and Observer pattern using role diagrams.

Figure 5 shows the Composite pattern. Every object always plays the Node role. The Child role abstracts from the Component participant in [Gamma+95], and the Parent role abstracts from the Composite participant. Furthermore, I have introduced the Root role as an extension to the Composite pattern: It represents the root of the object hierarchy and thus offers a more elaborate protocol than the basic Parent role protocol. There are three composition constraints: An object playing the Root role always also plays the Parent role which also always plays the Node role. An object playing the Child role also always plays the Node role.

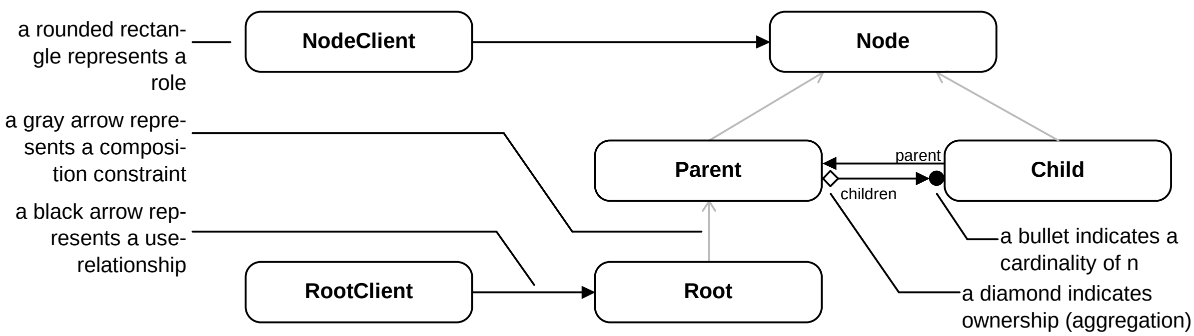

Figure 5: Role diagram of the Composite pattern

Figure 6 shows the Mediator pattern which corresponds to the original version.

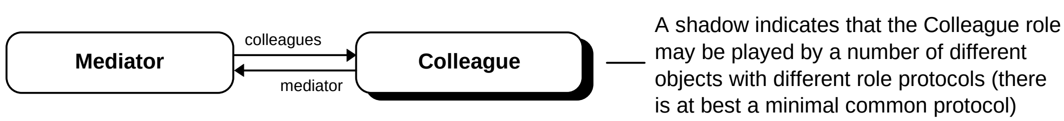

Figure 6: Role diagram of the Mediator pattern

Figure 7 shows the Chain of Responsibility pattern. The Handler participant in the original pattern has been split into the three roles Handler. Predecessor and Successor. They have distinct protocols. In addition, the Tail role has been introduced as a specialization of the Successor role. There are three composition constraints. An object playing the Tail role also always plays the Successor role which in turn also always plays the Handler role. An object playing the Predecessor role also always plays the Handler role.

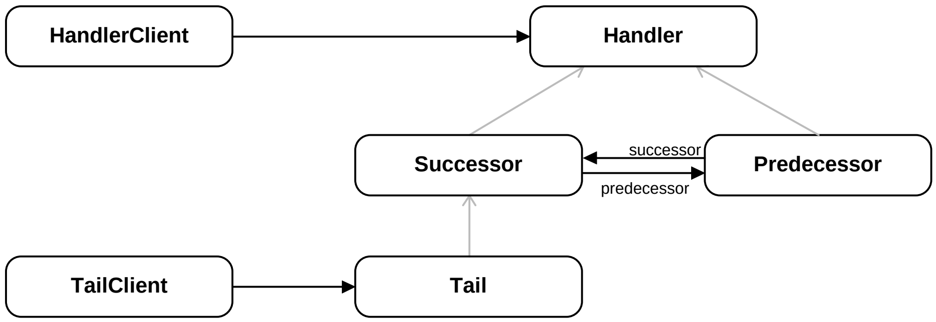

Figure 7: Role diagram of the Chain of Responsibility pattern

Finally, figure 8 presents the Observer pattern. It corresponds to the original pattern but omits the ConcreteSubject and ConcreteObserver subclasses. As demonstrated in [Riehle96] this role diagram is less implementation oriented than the original class diagram and thus allows for a wider class design and implementation space.

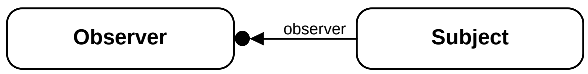

Figure 8: Role diagram of the Observer pattern

Role relationship matrix

Patterns stem from observation of existing systems, experience and reflection on it. We therefore need adequate concepts and techniques for digging out patterns. This includes means for describing patterns (previous subsection), means for analyzing existing systems (this subsection) and means for appropriately interpreting analysis results (next subsection). This subsection presents a simple analytical means, the role relationship matrix, which I used to determine the composite roles in the Bureaucracy pattern.

A first step is to abstract from the concrete pattern instantiations and devise a *prototypical pattern application* which exhibits all the properties considered important for the proposed pattern but omits unimportant implementation details. Figure 9 shows such a prototypical application of the Bureaucracy pattern, derived from the utility manager example. It uses its object structure, but only lists the roles the objects play in the overall collaboration.

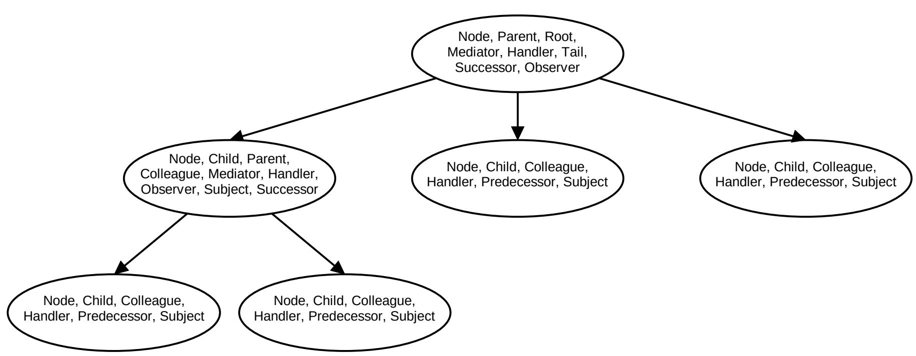

Figure 9: A prototypical application of the Bureaucracy pattern

Figure 10 shows the *role relationship matrix*, which displays the relationships between two roles as they appear in the prototypical pattern application. Thus, the matrix determines which roles may go with each other in a single object and which may not. This defines the design space of the Bureaucracy pattern.

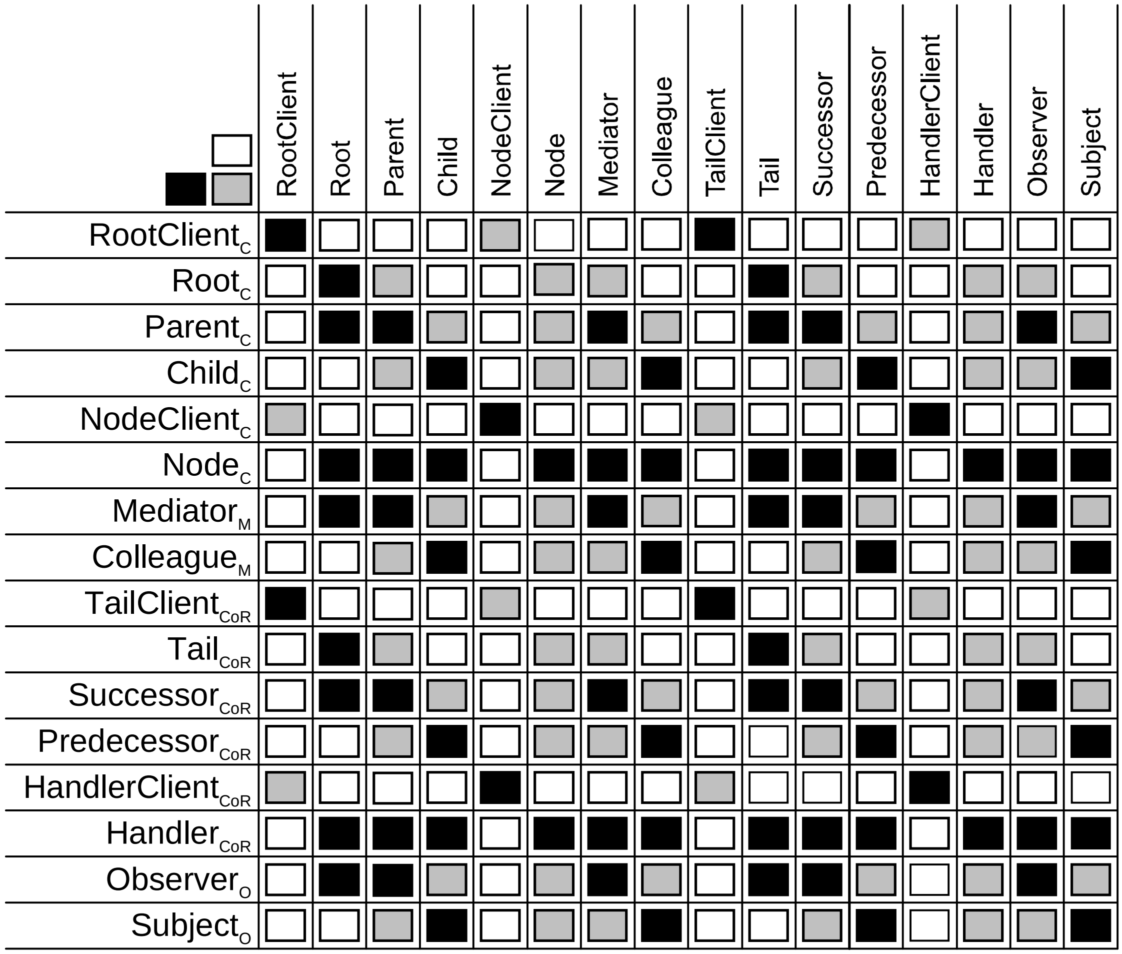

Figure 10: Role relationship matrix of the Bureaucracy pattern

In the role relationship matrix, each row and column stands for a role, and the matrix entry (A, B) for each intersection describes the relationship between these two roles. A white rectangle means "an object playing role A *never* plays role B," a gray rectangle means "*sometimes but not always* plays," and a black rectangle means "*always* plays." Role A are the roles from the top row, and role B are the roles from the left column.

Matrix interpretation

The matrix shows the role relationships as found in the prototypical pattern application and thus in every concrete pattern instantiation as well. It must be interpreted properly now.

Analysis of the matrix shows that certain rows and columns always are identical. Each resulting set of equivalent rows and columns can be interpreted as constituting a composite role which comprises all the roles listed in the set. The prototypical application and the role relationship matrix exhibit the following composite roles:

DirectorClient$_B$ = { RootClient$_C$, TailClient$_{CoR}$ }
Director$_B$ = { Root$_C$, Tail$_{CoR}$ }
Manager$_B$ = { Parent$_C$, Mediator$_M$, Successor$_{CoR}$, Observer$_O$ }

```
Subordinate_B    = { Child_C, Colleague_M, Predecessor_CoR, Subject_O }
ClerkClient_B    = { NodeClient_C, HandlerClient_CoR }
Clerk_B          = { Node_C, Handler_CoR }
```

Subscripts represent the name of the pattern in which the role is defined. Every composite role comprises the responsibilities of the constituting roles. For example, the Manager composite role comprises the Parent, Mediator, Successor and Observer roles, and its role protocol comprises the constituting roles' protocols.

Figure 11 presents the role relationship matrix after reducing the roles to composite roles. It shows how the new roles relate to each other: Every object playing one of the roles Subordinate, Manager and Director also always plays the Clerk role. In addition, every object playing the Director role also always plays the Manager role.

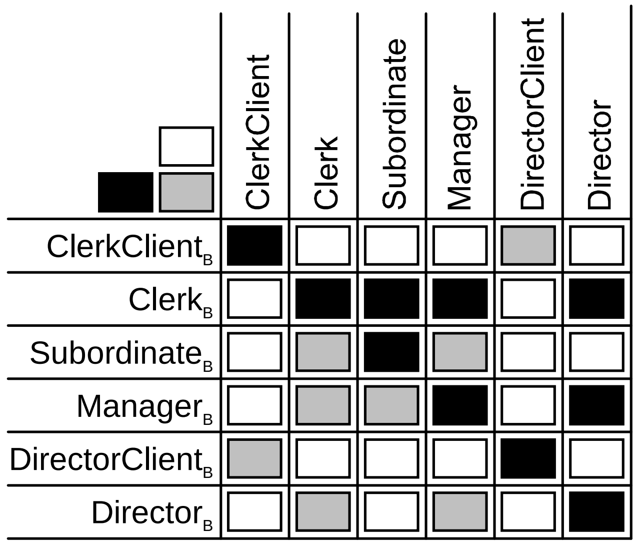

Figure 11: Role relationship matrix for the Bureaucracy pattern based on composite roles

The introduction of composite roles and the evaluation of their specialization relationships leads to figure 3, which represents the key role diagram of the Bureaucracy pattern.

Evaluation of approach

The concepts and techniques presented in this section represent some of the means I used to clarify the essence and intent of the Bureaucracy pattern. They emerged while grappling with the complexity of the pattern.

Can they be expected to work well for other patterns? I believe the concepts and techniques will be useful for analyzing and evaluating further composite patterns which can be described well by means of role diagrams. This applies mainly to behavioral patterns and probably will not work well with structural patterns.

A final conclusion, however, can only be drawn when enough composite patterns have been identified and analyzed using the presented techniques. I have applied the techniques to further composite patterns, and it seems to work well.

CONCLUSIONS

This paper has presented the Bureaucracy pattern, a pattern frequently found in interactive software systems and beyond. It describes how to structure large object hierarchies which allow interaction on every hierarchy level and have to maintain their consistency themselves. The Bureaucracy pattern helps developers to coherently and concisely assign functionality and responsibility to objects in the hierarchy. It represents a coherent pattern to control the dynamics of complex hierarchical structures.

I have described the Bureaucracy pattern as a composite pattern. The composition of the patterns is a non-trivial task, and the presentation of a composite pattern might become lengthy and arbitrarily structured. By presenting the Bureaucracy pattern as a composition of four patterns it is possible to leave out many implementation details that can be found in the constituent patterns' descriptions. The presentation focuses on the synergy arising from the composition and thereby demonstrates that the Bureaucracy pattern is more than just the sum of its constituent patterns, but a true pattern in its own right.

This paper uses role diagrams to describe patterns. The use of role diagrams was motivated by three reasons: Firstly, role diagrams seem to be an appropriate means for presenting patterns, composite or non-composite, simply because of their clear distinction between the different responsibilities assigned to objects. Secondly, the clear distinction between the different roles objects can play in a pattern instantiation allows the simplified composition of patterns to form larger wholes like the Bureaucracy. Thirdly, it makes pattern composition subject to more rigorous analysis and thereby helps to increase the confidence of having got at the heart of the pattern.

Role diagrams as applied in this paper have proved to be very useful. I believe that they will be useful for describing patterns in general. Since role diagrams seem to be promising in several respects, I will further investigate their use and application.

ACKNOWLEDGMENTS

I wish to thank Neil Harrison, Ralph Johnson, Trygve Reenskaug and Wolf Siberski for reading and commenting on an earlier version of the pattern. I further would like to thank the participants of the writer's workshop at EuroPLoP '96 whose comments helped me to improve the pattern. In addition, I especially thank the Swiss PTT Telecom which provided me with first-hand experience of a real-world bureaucracy [Telecom95]. Without this experience, I probably would never have written this paper.

REFERENCES

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Design*. Reading, MA: Addison-Wesley, 1995.

[Harrison+93] W. Harrison and H. Ossher. "Subject-Oriented Programming (A Critique of Pure Objects)." OOPSLA '93, *Conference Proceedings*. See also: *ACM SIGPLAN Notices* 28, 10 (October 1993): 411-428.

[Johnson92] R. E. Johnson. “Documenting Frameworks using Patterns.” OOPSLA ’92, *Conference Proceedings.* See also: *ACM SIGPLAN Notices* 27, 10 (October 1992): 63-70.

[Kiczales+96] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. “Aspect-Oriented Programming.” http://www.parc.xerox.com/spl/projects/aop/.

[Kristensen+96] B. B. Kristensen and K. Osterbye. “Roles: Conceptual Abstraction Theory and Practical Language Issues.” *Theory and Practice of Object Systems* 2, 3 (1996): 143-160.

[Telecom95] D. Riehle. “Experiences with the Swiss PTT Telecom.” http://swt-www.informatik.uni-hamburg.de/~riehle.

[Reenskaug96] T. Reenskaug, with P. Wold and O. A. Lehne. *Working with Objects.* Greenwich: Manning, 1996.

[Riehle95] D. Riehle. “How and Why to Encapsulate Class Trees.” OOPSLA ’95, *Conference Proceedings.* ACM Press, 1995, pp. 251-264.

[Riehle+95a] D. Riehle and M. Schnyder. *Design and Implementation of a Smalltalk Framework for the Tools and Materials Metaphor.* Ubilab Technical Report 95.7.1. Zürich, Switzerland: Union Bank of Switzerland, 1995.

[Riehle+95b] D. Riehle and H. Züllighoven. “A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor.” In J. O. Coplien and D. Schmidt (eds.), *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995, pp. 9-42.

[Riehle96] D. Riehle. “Describing and Composing Patterns Using Role Diagrams.” WOON ’96 (1st Int’l Conference on Object-Orientation in Russia), *Conference Proceedings.* St. Petersburg Electrotechnical University, 1996. Reprinted in K.-U. Mätzel and Hans-Peter Frei (eds.), *Proceedings of the Ubilab Conference ’96, Zürich.* Germany, Universitätsverlag Konstanz, 1996, pp. 137-152.

[Riehle+96] D. Riehle, B. Schäffer, and Martin Schnyder. “Design of a Smalltalk Framework for the Tools and Materials Metaphor.” *Informatik/Informatique* 3 (February 1996), 20-22.

[Trudeau96] J. Trudeau. *Metrowerks Powerplant Book.* Metrowerks, Inc., 1996.

[Weber47] M. Weber. *The Theory of Social and Economic Organization*. New York: Oxford University Press, 1947.

[Weinand+94] A. Weinand and E. Gamma. “ET++ — a Portable, Homogenous Class Library and Application Framework.” In W. R. Bischofberger and H.-P. Frei (eds.), *Computer Science Research at UBILAB.* Konstanz: Universitätsverlag Konstanz, 1994, pp. 66-92.

[Zhao+96] L. Zhao and T. Foster. “The PLOTS Pattern Language of Transport Systems: Point and Route.” This volume.

Dirk Riehle works at Ubilab, the information technology research laboratory of the Union Bank of Switzerland. He is interested in object-oriented modeling, frameworks, component and software architecture models as well as their application to the design and implementation of distributed systems. He can be reached at Union Bank of Switzerland, Bahnhofstrasse 45, CH-8021 Zürich. He welcomes e-mail at Dirk.Riehle@ubs.com or riehle@acm.org.