A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose

Dirk Riehle

Ubilab Technical Report 97-1-1

Copyright 1997 Dirk Riehle and Ubilab, Union Bank of Switzerland

Abstract

This technical report presents 25 design patterns using the role diagram notation. The role diagram notation describes a pattern using the concept of “role” as the central modeling construct. The purpose of this report is to describe a set of common reusable patterns in a form that makes them an easy target for composition. The ultimate goal is to explain frameworks as sets of instantiated patterns which integrate with each other to serve a common goal, that is to achieve the framework’s purpose.

Dirk Riehle (Dirk.Riehle@ubs.com)

October 1996, Zürich, Switzerland

Table of Contents

1 Introduction

This technical report presents 25 design patterns using the role diagram notation. The role diagram notation describes a pattern using the concept of “role” as the central modeling construct [KO96, Ree95, Rie96a]. I took 15 of the 23 patterns of the design pattern catalog [GOF95], sometimes varied or extended them, and added 10 more which I will need for further work.

I present the patterns using role diagrams, because role diagrams can be more easily composed than class diagrams. They are more abstract though, and therefore only complement the original class diagram based description. I hope that pattern descriptions based on role diagrams let us explain frameworks more easily as a composition of applied patterns than class diagram descriptions let us do.

Most of the pattern descriptions fit on a single page and show the role diagram and the explanation of the roles. The short explanation just wraps up the pattern. My focus is on presenting the role diagram and the accompanying role relationship matrix. I assume familiarity with the presented patterns. The main focus of each pattern description is to show the roles, their collaboration, and their composition constraints.

I think that this concise description form reveals the essence of the patterns with respect to roles and object collaborations. Some of the role diagrams may seem rather elaborate, in particular those of the structural patterns. It is well possible to use simpler and thus more convenient diagrams [Rie96b, Rie97]. I did not present those, however, because the fully elaborated role diagram version of a pattern should be defined before shortcuts are introduced.

The role diagram notation looks like the notation from the design pattern catalog, based on roles rather than classes. Role diagrams draw directly on Reenskaug’s work [Ree96]. However, his notation does not support the definition of composition constraints as I believe that I need them, for example, to define the Decorator or Composite pattern. For this reason, I chose to define a notation of my own which will both serve as a research vehicle and be subject to future extensions itself.

I would like to thank my colleague Kai-Uwe Mätzel for discussing some of the patterns with me as well as for reading and commenting on this report. I further would like to thank Trygve Reenskaug for e-mail and personal discussions on the topic of this report.

2 Roles and Role Diagrams

In any non-trivial design, objects collaborate. In any well-thought out design, the different collaborations an object is involved in can be told apart as different areas of concern [Ree96]. Any such area of concern can be described by a role diagram; it shows how the collaboration fulfills its specific purpose. Role diagrams can describe any role-based design and thus can describe any role-based pattern.

A role diagram describes the roles objects play in a collaboration. A role defines the abstract state and behavior of an object in the collaboration. It can be expressed formally as a role protocol, for example using any adequate type or interface notation. The actual definition of the role is based on what the other roles in the collaboration require from it in order to achieve the joint purpose. You can specify the collaboration's behavior using whatever specification notation or tool you prefer and which seems suitable to you [HHG92, SW96].

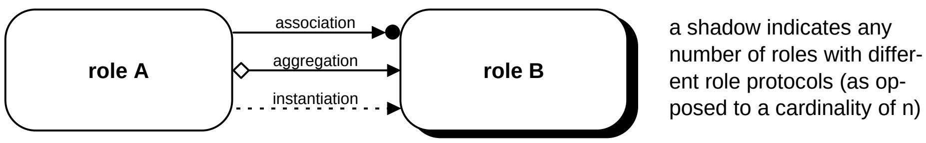

Figure 2-1: Roles and relationships

Objects play roles. Thus, a single object can play several roles, and several objects can play the same role. Usually, the roles an object can play are statically defined and implemented by the object's class. Most patterns in this report consist of a single collaboration only, except for the structural patterns.

I describe the structural patterns as well as any other composite pattern by annotating the involved role diagrams with composition constraints. Composition constraints capture the formalizable part of the synergy which arises from the composition of the participating patterns. Composition constraints are fairly simple binary role relationships: Role A may imply role B ($A \Rightarrow B$), two roles A and B may prohibit each other ($\neg(A \wedge B)$), or nothing can be said (anything goes).

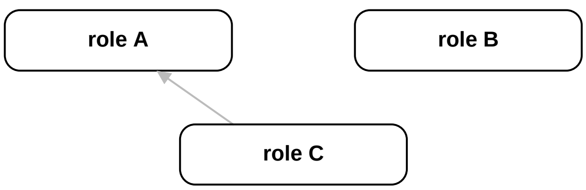

Figure 2-2: Implication between roles in role diagrams

Role A implies role B, if an object playing role A always also plays role B. If a role A prohibits role B, an object playing role A never plays role B within the same collaboration; this is symmetric. If nothing can be said, any combination may occur. Implication is usually implemented by inheritance on the class level.

The annotations in the role diagram only show implication relationships. The full set of composition constraints is therefore specified using an additional notational tool, the role relationship matrix. A matrix relates all roles and defines their relationship to be one of the three possible values defined above. It shows the constraints *within a single application*, not within several overlapping applications.

The composition constraints in a role relationship matrix are not symmetric. It is well possible that a role A implies a role B, but that the Role B does not necessarily implies role A. A role relationship matrix looks like this:

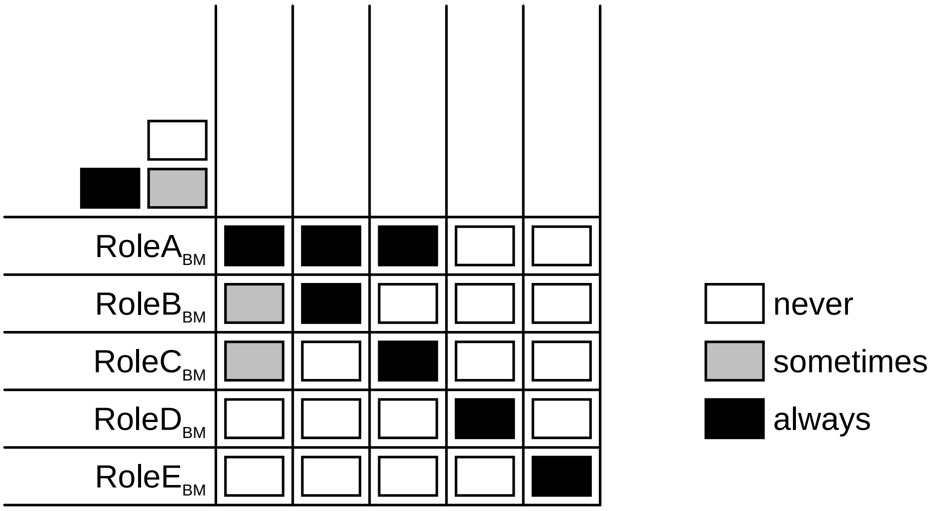

Figure 2-3: Exemplary role relationship matrix

3 Object Creation

Object creation deals with creating one or more related objects using the canonical object creation pattern depicted in figure 3-1. In this pattern, a Client asks a Creator to create an object that can play a given Product role. Object creation patterns are needed when a Client has to be decoupled from the concrete class implementing a Product. The different patterns vary in the degree of indirection and configurability.

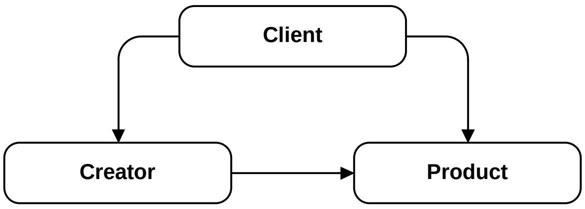

Figure 3-1: Role diagram of the canonical Object Creation pattern

The pattern's role relationship matrix is depicted in figure 3-2.

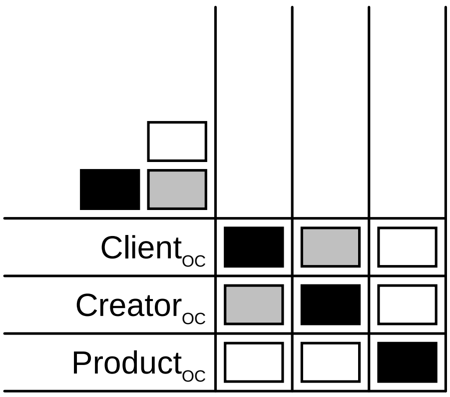

Figure 3-2: Role relationship matrix of the Object Creation pattern

I know of three fundamental object creation patterns which match this situation and make it more concrete, with different trade-offs respectively:

- *Factory Method.* The Creator hard codes the class to be instantiated. Factory Method is a convenient pattern and can be applied easily, but it is not very flexible.
- *Prototype.* The Creator and Product role are played by the same object, though in different pattern runtime instantiations. A prototype is frequently used to configure a more advanced object creation pattern instantiation at runtime.

- *Product Trader*. The Creator uses a (possibly very elaborate) specification to lookup the class for the Product which it instantiates then.

In addition, further patterns build on these three fundamental object creation patterns:

- *Builder*. A Builder creates an elaborate object structure with the Product being the root object to that structure. Different fundamental object creation patterns can be used.
- *Factory*. A Factory groups related object creation operations and implements them using any of the fundamental patterns. The choice of the fundamental pattern depends on the required degree of configurability.

This chapter sketches these patterns.

3.1 Factory Method

The Factory Method pattern lets a Client create a Product by calling a creation operation of a Creator. Client and Creator are frequently played by the same object. The pattern's role diagram is depicted in figure 3-3.

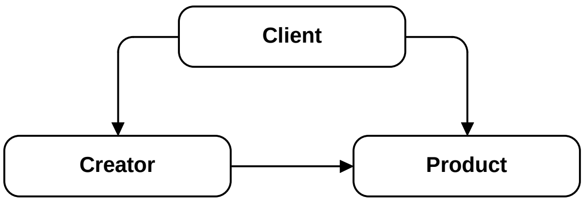

Figure 3-3: Role diagram of the Factory Method pattern

A Factory Method hard-codes the Product class in the Creator class so that its flexibility is limited, in particular, if Creator and Client are played by the same object. Nevertheless, the hook is sufficient to introduce a more elaborate object creation pattern in a subclass (the Factory Method implementation in a subclass may branch out to whatever other pattern seems appropriate).

The role relationship matrix is straightforward:

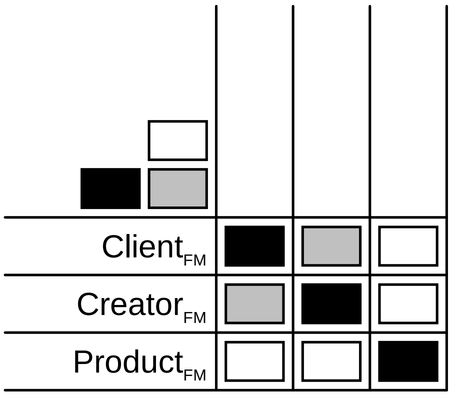

Figure 3-4: Role relationship matrix of the Factory Method pattern

For a more elaborate documentation see [GOF95].

3.2 Prototype

The Prototype pattern lets a Client create a Product by cloning a Prototype. The Client gets configured with a Prototype which represents the Product of choice.

The role diagram of the Prototype pattern shows only two roles, Client and Prototype, and is depicted in figure 3-5.

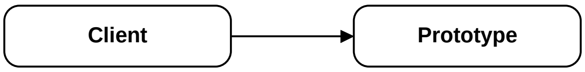

Figure 3-5: Role diagram of the Prototype pattern

Again, the Product class is hard-coded in the Creator class, that is the Prototype. Since Client and Prototype are different objects, some flexibility is achieved: A Client can be parameterized with different Prototypes if the Product class is to be varied.

The role relationship matrix is straightforward:

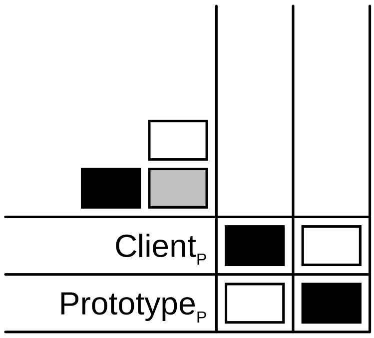

Figure 3-6: Role relationship matrix of the Prototype pattern

For a more elaborate documentation see [GOF95].

3.3 Product Trader

The Product Trader pattern lets a Client create objects which match a specific Product role protocol and fulfill an additional Specification. Figure 3-7 depicts its role diagram. The triple of CreatorManager, SimpleCreator and Specification can be abstracted into a single ProductTrader role as described in the Object Creation pattern.

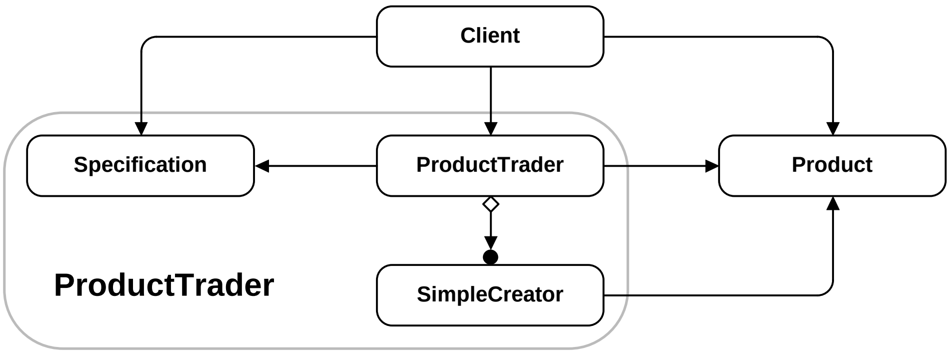

Figure 3-7: Role diagram of the Product Trader pattern

The Product Trader pattern is the most flexible object creation pattern among the fundamental creation patterns. The lookup of a specific Product class is fully configurable and is solely the responsibility of the Creator(-Manager). A Client stays independent of configuration issues, its only responsibility is to provide an adequate specification.

The role relationship matrix is defined as depicted in figure 3-8.

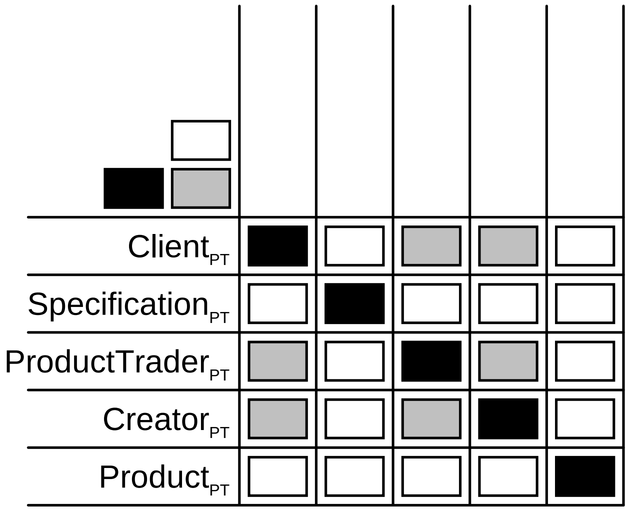

Figure 3-8: Role relationship matrix of the Product Trader pattern

For a more elaborate documentation see [BR97, Rie96a].

3.4 Builder

The Builder pattern lets a Client create a complex Product object by delegating the creation process to a Director which uses a Builder to create the Product. The Director controls and directs the Builder in its creation efforts. Factoring the creation process into two different roles, one for directing, one for creating, delivers a degree of flexibility that is high enough to deal with possibly very complex Product object structures.

The role diagram of the Builder pattern is depicted in figure 3-9.

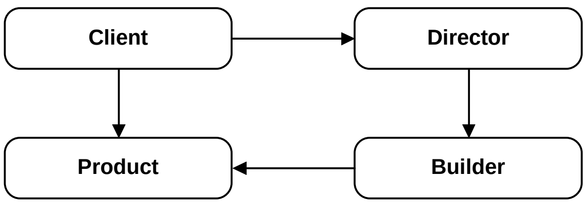

Figure 3-9: Role diagram of the Builder pattern

The role relationship matrix is straightforward:

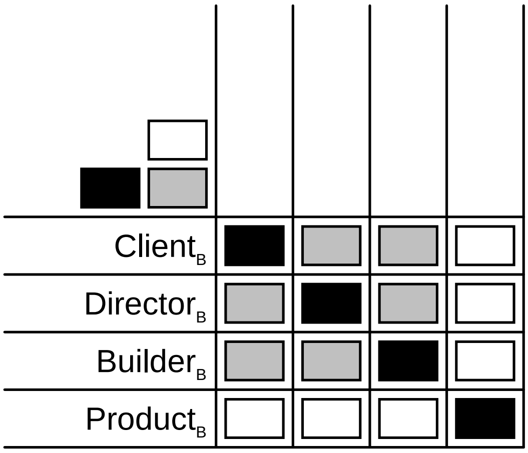

Figure 3-10: Role relationship matrix of the Builder pattern

The role relationship matrix indicates that the same object can play the Client, Director and Builder role. This is acceptable, as long as the roles are kept separate. On an implementation level, the different implementations of a Director and a Builder could be merged, for example, using mixins.

For a more elaborate documentation see [GOF95].

3.5 Factory

A Factory groups related object creation operations together in order to provide a convenient interface for Clients to instantiate Products. A Factory provides an indirection which decouples clients from the concrete Product class they wish to instantiate. Moreover, by grouping related creation operations together, a Factory maintains a specific configuration of interrelated concrete Product classes.

My understanding is that the Factory pattern predominantly serves configuration purposes. Its creation operations can be implemented by any of the three fundamental object creation patterns.

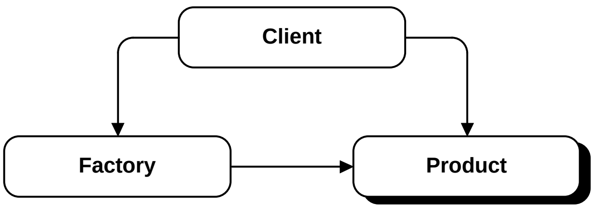

Figure 3-11: Role diagram of the Factory pattern

Strictly speaking, this is a composite pattern, because it puts together two or more different patterns: The Factory groups the different creation operations (as defined by the Abstract Factory pattern in [GOF95]), and it implements any of the operation procedures using whatever object creation pattern suits best.

It doesn’t seem to make much sense to me, however, to represent the grouping of creation operations as a pattern of its own. The role relationship matrix becomes simple then:

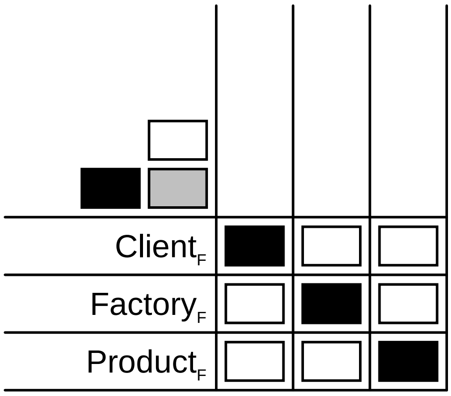

Figure 3-12: Role relationship matrix of the Factory pattern

For a more elaborate documentation see [GOF95].

4 State Management

State management deals with factoring, distributing, and integrating an object's state space so that its invariants are preserved. This section discusses two patterns:

- The *State* pattern shows how to partition a large and unwieldy state space so that it can be factored into different implementations. A Component manages its state by delegating state related behavior to State objects the implementations of which can vary but which all can play the State role.
- The *Property* pattern lets Clients extend a Component's state space. It is used when the state space cannot be defined in advance, and controlling and interpreting it must be done by Clients.

4.1 State

The State pattern shows how to factor a Component's state space into several distinct substates, each one represented by a State object. This way, a large and unwieldy state space and its accompanying implementation can be managed as smaller and more easily manageable chunks of state and behavior.

It is not a necessary precondition that the different substates represent distinct subspaces; it helps though. The state space can then be modeled as the conjunction of those subspaces, using, for example, state charts [HLN+90].

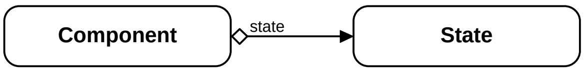

Figure 4-1: Role diagram of the State pattern

The role relationship matrix is straightforward:

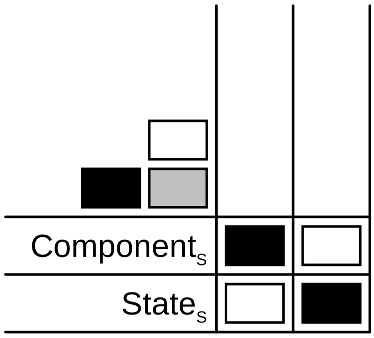

Figure 4-2: Role relationship matrix of the State pattern

For a more elaborate documentation see [GOF95] and [DA97].

4.2 Property

The Property pattern lets you dynamically extend a Component's state space. A Component can receive any number of Property objects at runtime each one serving as a handle for a new Component attribute. The pattern is used when you cannot determine an Component's state space in advance or you want to extend it at runtime.

The downside of the pattern is that it is hard to ensure any invariants on the Component's state space and that specialized Property dependent behavior requires the application of additional patterns like Strategy (see, for example, the Role pattern).

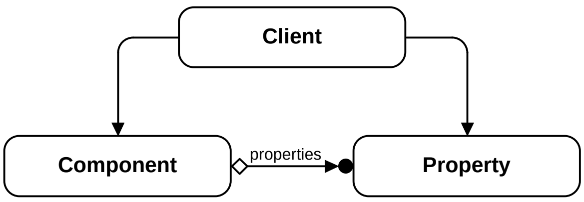

Figure 4-3: Role diagram of the Property pattern

There are two main interactions:

- *Component* and *Property*. A Component owns any number of Property objects. It manages them, that is it receives them from a client, returns them upon request and deletes them if it is deleted itself.
- *Client* with *Component* and *Property*. A Client creates Property objects and extends a Component with them. It interprets the Property objects as state space extensions of the Component. It handles the Component accordingly.

The role relationship matrix is straightforward:

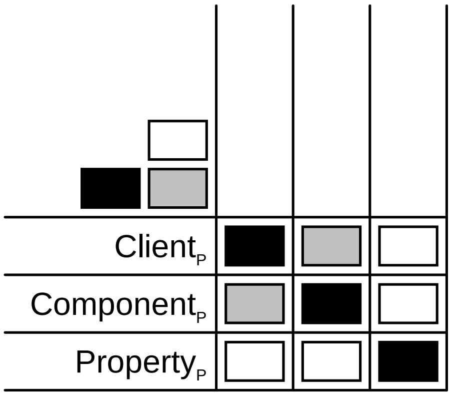

Figure 4-4: Role relationship matrix of the Property pattern

I don’t know of a more elaborate documentation, but a similarly short description has been presented in [Bec96] under the name “Variable State.”

5 Behavior Management

Behavior management deals with changing and extending a Component's behavior. Usually, behavior is attached by some means of delegation. This section sketches the following two patterns:

- The *Strategy pattern* lets you dynamically change implementations of operations of a Component by means of Strategy objects. A Component delegates the execution of an operation to a Strategy which encapsulates a specific implementation of that operation. The Strategy may call back on the Component as its execution context.
- The *Visitor pattern* lets you extend a set of Elements with a new operation all at once. The Visitor implements the new operation for each Element and relies on it to dispatch calls to that operation back to the Visitor.

5.1 Strategy

The Strategy pattern lets a Client dynamically configure a Component with Strategy objects, each one implementing a specific part of the behavior of the Component. The Component delegates the execution of a specific task or a certain computation to the Strategy. Thereby you can vary the Component's behavior at runtime, adapting it to different needs arising in varying contexts.

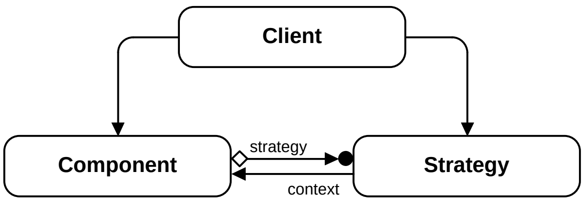

Figure 5-1: Role diagram of the Strategy pattern

The role relationship matrix is straightforward. The pattern lets Client pass itself as the Strategy to the Component, so that the same object can play both Client and Strategy roles:

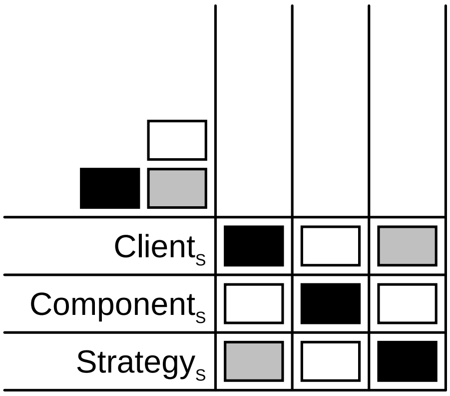

Figure 5-2: Role relationship matrix of the Strategy pattern

For a more elaborate documentation see [GOF95].

5.2 Visitor

The Visitor pattern lets you dynamically extend a set of related Elements with a new operation common to all Elements. The Visitor provides the implementation of a new operation for each Element that is to be enhanced. The Element dispatches an invocation of the new operation to the Visitor based on the current Element's type.

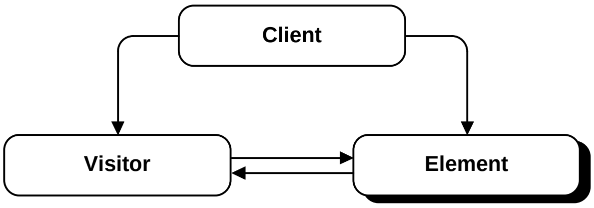

Figure 5-3: Role diagram of the Visitor pattern

The Visitor pattern is intrusive, and Elements are aware of the Visitor. A possible alternative has been discussed by Seiter et al. as so called context objects [SPL96]. However, this approach requires changing the underlying implementation language.

The role relationship matrix is straightforward:

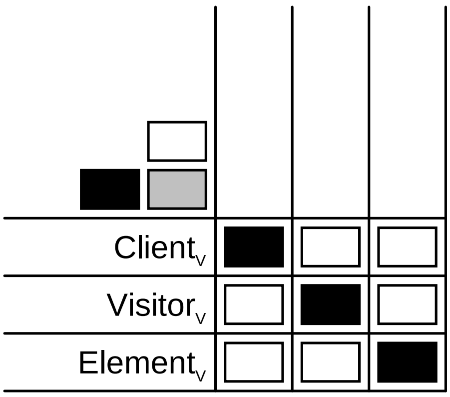

Figure 5-4: Role relationship matrix of the Visitor pattern

For a more elaborate documentation see [GOF95].

6 Dependency Management

Dependency management deals with maintaining state dependencies between different objects. State changes of one object must be reflected in state changes of further dependent objects. The update semantics and the update implementation mechanism vary greatly between different patterns. A general role diagram, however, can be described. It is shown in figure 6-1 and corresponds to the Observer pattern.

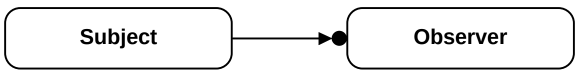

Figure 6-1: Role diagram of the Dependency Management pattern

Figure 6-1 is not fully equivalent to the Observer pattern, because it doesn't say anything about the actual implementation of the link between Subject and Observer. Different patterns elaborate with a different degree of complexity and flexibility on this pattern.

The role relationship matrix is simple, then:

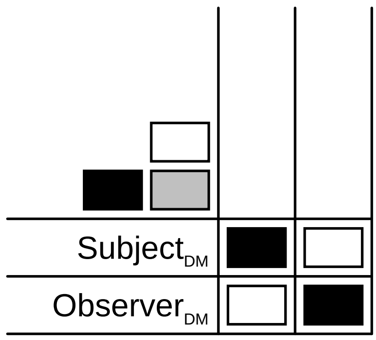

Figure 6-2: Role relationship matrix of the Dependency Management pattern

This chapter lists the following three patterns (and avoids too many variations):

- *Observer.* The Observer pattern defines the two roles Observer and Subject and implements the communication link as a set of references maintained by the Subject or a third party object. Updates are synchronous and usually unordered.
- *Event Notification.* The Subject makes its state space explicit by StateChange objects which Observers use as hooks to link in with EventPort objects.
- *Notification Service.* A Subject delegates the notification of Observers about Events to a NotificationService. The service handles the queuing and delivering of events and manages Observers independently of Subjects.

Whether updates are synchronous/asynchronous, ordered/unordered, and how possible races are handled or how explicitly an event is modeled, is in principle orthogonal to the pattern's structure. However, the more complicated patterns tend to handle the more complex situations of asynchronous notifications better.

6.1 Observer

The Observer pattern defines a general update mechanism between a Subject object and several Observer objects. It helps you maintain state dependencies without intertwining role protocols.

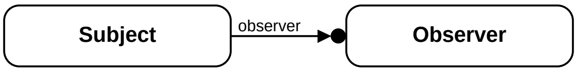

Figure 6-3: Role diagram of the Observer pattern

A Subject either directly maintains a set of Observers or delegates this to a third object which resides in the background (as Smalltalk or ET++ show, this background object is usually not a Notification Service but a technically motivated service used to keep the memory footprint small).

The role relationship matrix depicted below is straightforward. Please note that due to its focus on the constraints within a *single* collaboration, the Subject and Observer cannot be played by the same object (every time I was tempted to make an object observe itself I soon found myself revising the design). The same object, of course, can play both the Subject and Observer roles in different collaborations.

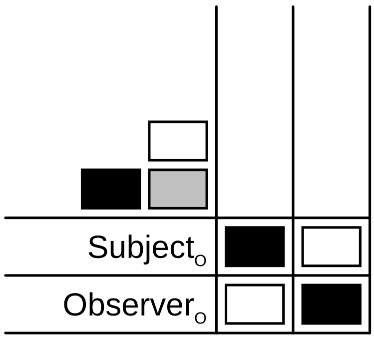

Figure 6-4: Role relationship matrix of the Observer pattern

For a more elaborate documentation see [GOF95].

6.2 Event Notification

The Event Notification pattern lets you maintain a state dependency between a Subject and an Observer via a Connector represented by an EventPort and a StateChange object on each end. StateChange objects make the abstract state space of a Subject explicit and let Observers hook in via EventPorts in a deliberate and selective fashion. Thus, the notification dispatch is taken from the Observer's shoulders and realized by a Connector.

There may be any number of Connectors between a Subject and Observer, at maximum one for every type of state change of the Subject, though. The frequency of calls and the weight of the Connectors determine their actual number.

Figure 6-5 depicts the role diagram.

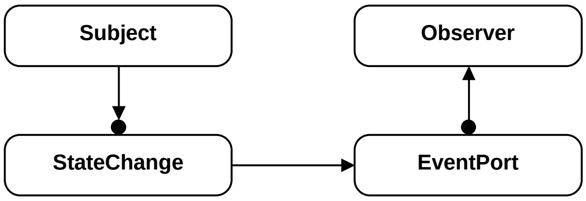

Figure 6-5: Role diagram of the Event Notification pattern

The pattern as described is composed from two patterns: The Dependency Management and the Connector pattern:

- The *Dependency Management* pattern defines the Subject and Observer roles and their semantics, that is how and why Subjects issue Events to inform Observers about state changes.
- The *Connector* pattern defines how communication protocols are negotiated and implemented which are decoupled from the clients using them. In this case, StateChange and EventPort establish a communication protocol and decouple Observer and Subject from it.

Figure 6-6 shows the role relationship matrix of the Event Notification pattern.

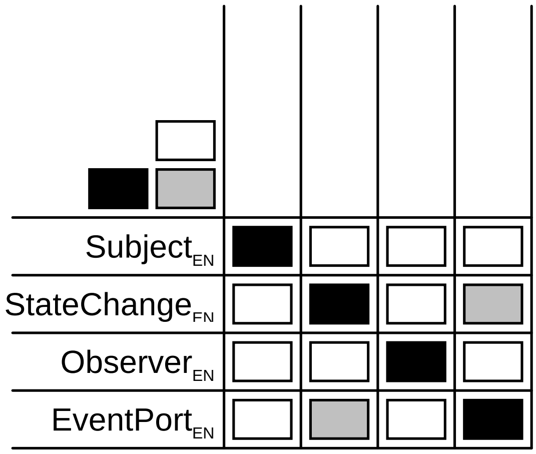

Figure 6-6: Role relationship matrix of the Event Notification pattern

For a more elaborate documentation see [Rie96b].

6.3 Notification Service

The Notification Service lets you deliver events asynchronously and manage Observer/Subject dependency relationships independently of the Subject. It is geared towards distributed systems with asynchronous event notifications. The NotificationService decouples Subjects from Observers, queues Events and delivers them in predefined ways.

Figure 6-7 depicts the pattern's role diagram.

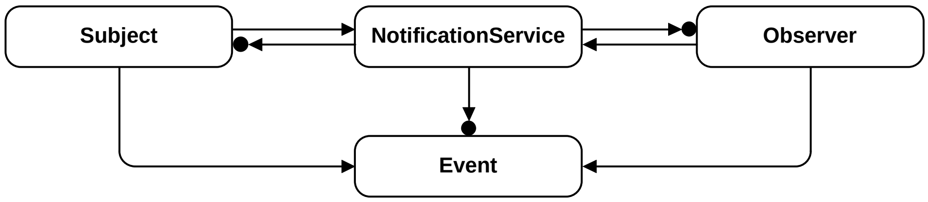

Figure 6-7: Role diagram of the Notification Service pattern

Right now, I don’t know whether I should break up the NotificationService role into several distinct roles, one for interest registration and one for accepting events.

Figure 6-8 shows the role relationship matrix.

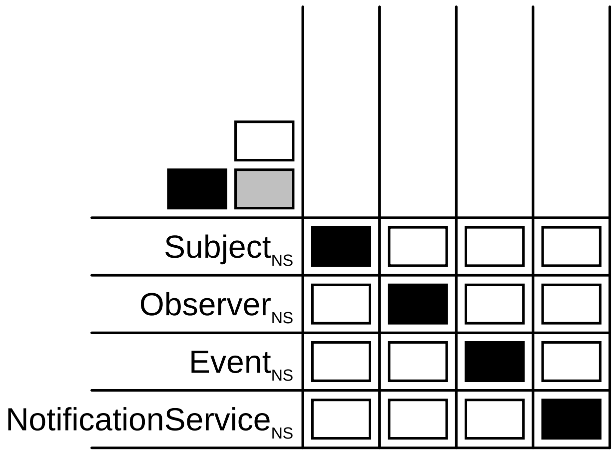

Figure 6-8: Role relationship matrix of the Notification Service pattern

For a more elaborate discussion and definition see [OMG96].

7 Context Adaptation

Context adaptation lets you adapt a Component to varying use contexts through Context objects. Each Context object customizes the Component's appearance and behavior for that specific context. All state relevant to the Component is managed in the Component, and all state relevant for the Context adaptation is managed by the Context object. Context adaptation patterns serve well for implementing role-based designs in programming languages which don't support the concept of role as a first class citizen.

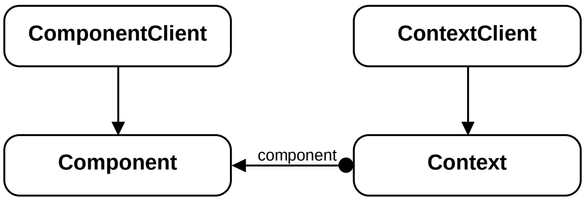

Figure 7-1: Role diagram of the Context Object pattern

The Context Object pattern is the canonical representation of the context adaptation problem; it makes the assumption that using multiple inheritance on the class level will not work for the addressed context of application.

The role relationship matrix is depicted in figure 7-2:

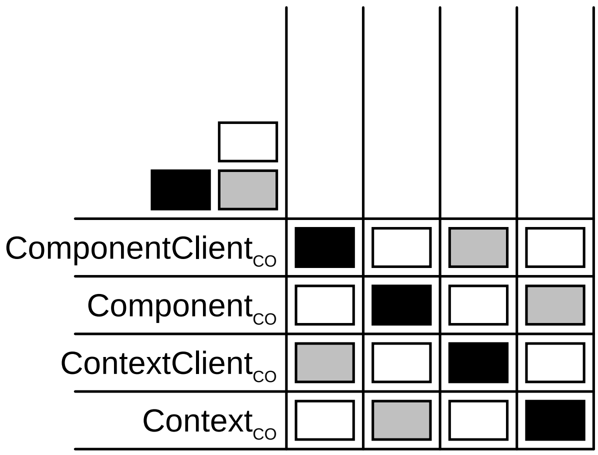

Figure 7-2: Role relationship matrix of the Context Object pattern

This chapter shortly describes the following patterns:

- The *Adapter* pattern lets you adapt an Adaptee by means of a single Adapter for use in a specific predefined context. This is the most straightforward context adaptation pattern and discussed only because it was the first one document [GOF95] (that is, it makes for a good introduction).

- The *Decorator* pattern lets you extend a Component with Decorators that wrap it in a transparent way. This lets a Client switch between awareness and unawareness of the context adaptation being done by the Decorator.
- The *Facet* pattern lets you define a Subject that manages its Facets. When asked by a Client for a specific Facet, it returns it. Facet and Subject interact in order to maintain the whole Component.
- The *Role* pattern lets you extend a Component according to the Context Objects pattern. It is a composite pattern which integrates the canonical Context Object pattern with the Strategy and Property pattern. This composition of pattern lets you manage the whole Component and its Roles as a single logical object with integrated state.

The Model-View-Controller pattern could have been listed here, because a large part of its purpose is to adapt a Model to the context of presentation to and interaction with users. I have put it into the Composite Patterns section, however, because it is a more general pattern.

7.1 Adapter

The Adapter pattern lets you adapt an Adaptee by an Adapter to a Client's needs. The Adapter offers the interface needed by the Client and implements it based on the Adaptee, thereby reusing it.

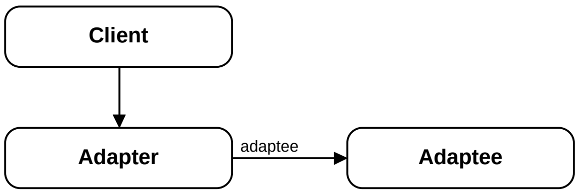

Figure 7-3: Role diagram of the Adapter pattern

The Adapter pattern is fairly simple and straightforward. In it, there is usually no explicit AdapteeClient so that it has been omitted from the role diagram and the role relationship matrix depicted in figure 7-4.

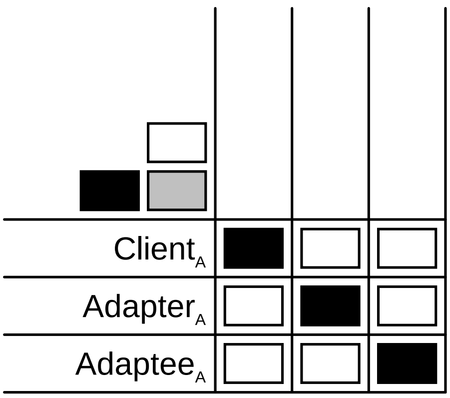

Figure 7-4: Role relationship matrix of the Adapter pattern

For a more elaborate documentation see [GOF95].

7.2 Decorator

The Decorator patterns shows how to extend a Component's interface so that a DecoratorClient can make use both of the Component and Decorator role of an object at once.

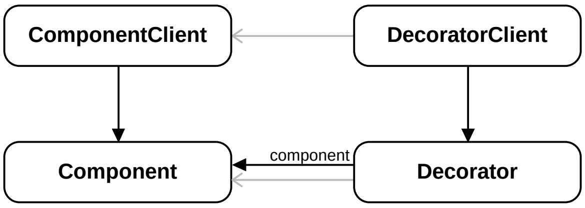

Figure 7-5: Role diagram of the Decorator pattern

There are three main interactions:

- *Component* and *Decorator*. A Decorator is partially implemented using the wrapped Component.
- *Component* and *ComponentClient.* A ComponentClient interacts with a Component to do some work.
- *Decorator* and *DecoratorClient.* A DecoratorClient interacts with both the Component and Decorator role of an object to do some work.

There are two composition constraints:

- An object playing the Decorator role always also plays the Component role in the same collaboration;
- As a consequence, a DecoratorClient also always plays the ComponentClient role.

It is interesting to note that the Decorator/Component constraint leads to the DecoratorClient/ComponentClient constraint. Thus, we have an isomorphism between two relationships.

The role relationship matrix reflects this:

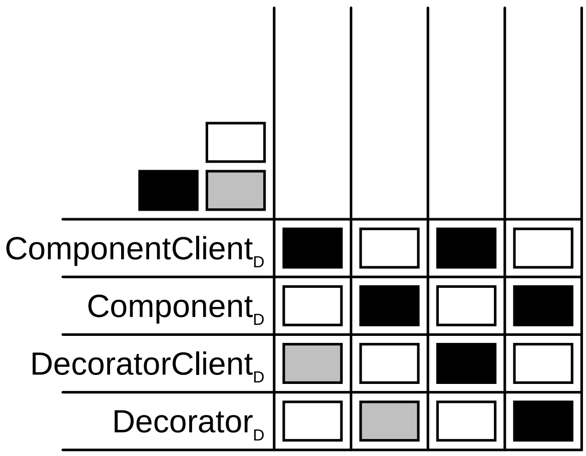

Figure 7-6: Role relationship matrix of the Decorator pattern

For a more elaborate documentation see [GOF95].

7.3 Facet

The Facet pattern shows how to extend a Subject with Facets so that a FacetClient can work with the Facet only; it does not have to know the extended Subject. The Subject manages its Facets itself and provides them upon request.

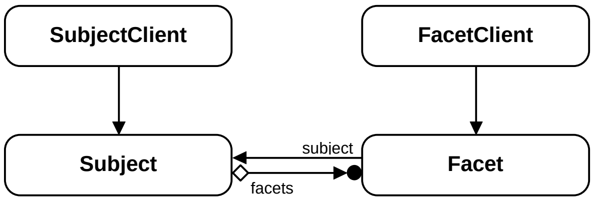

Figure 7-7: Role diagram of the Facet pattern

There are three main collaborations:

- *Subject* and *Facet.* A Subject manages its Facets.
- *Subject* and *SubjectClient.* A SubjectClient interacts with a Subject to do some work.
- *Facet* and *FacetClient.* A FacetClient interacts with a Facet to do some work.

The difference with the Decorator pattern is that there is no composition constraint between Subject and Facet, so that both roles are kept separate and a client has to explicitly switch between them.

The role relationship matrix is fairly simple:

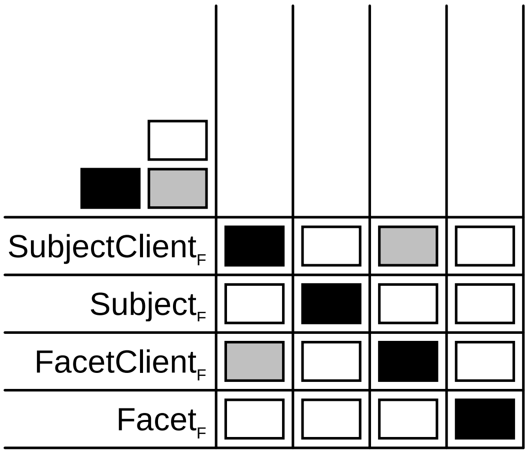

Figure 7-8: Role relationship matrix of the Facet pattern

For a more elaborate documentation see [Gam97].

7.4 Role

The Role pattern lets you design a Component which can be extend at runtime with new Context objects, called Roles. A Component is extended with Roles according to the Context Object pattern. The Component corresponds to a Decorator's Component, and a Role corresponds to a Context Object. State integration of a role-playing Component is achieved by applying Property and Strategy. Property is used to define the Component's state space, and Strategy is used to provide the Property dependent behavior.

In the Role pattern, any of the two patterns Facet or Decorator can fill out the place of the Context Object pattern. If Decorator is chosen, a composition constraint between ComponentClient and RoleClient emerges, otherwise it is left open.

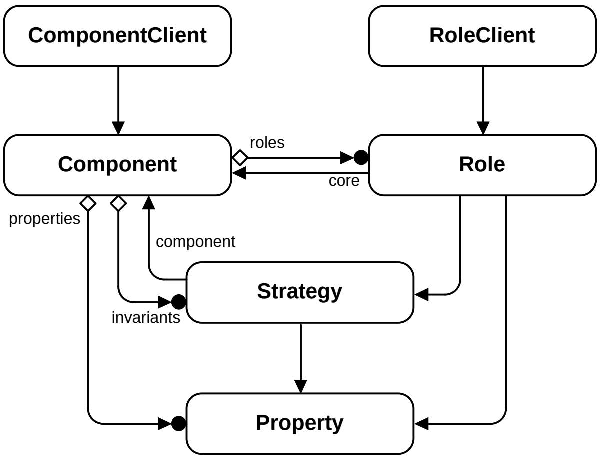

Figure 7-9: Role diagram of the Role pattern

The three composed patterns address the following issues, respectively:

- The *Context Object pattern* lends the Role pattern its structure. A Component can be extended by any number of Roles. It both owns and manages them. Role objects have no conceptual identity of their own. If the Component is deleted, the Roles die too.
- The *Property pattern* lets a Role extend a Component's state space. A Role defines the Properties relevant for its specific context and sets them to the Component. The Component subsequently owns and manages them. However, the Properties can only be properly interpreted by the Role which created them.
- The *Strategy pattern* lets a Role parameterize a Component with the code specific to the state space extension carried out by applying the Property pattern. The Role defines a Strategy for each Property, the dependencies of this Property on other Properties possibly defined by other Roles, and then sets the Strategy to the Component.

The pattern interactions are as follows:

- A Client calls a mutating operation on a Role object. The Role object carries out the state change using the Property objects of its Component.
- A Property is changed on which further Properties depend. This dependency is explicit in the Property dependencies managed by the Component. The Component straightens out the different dependencies and calls the Strategy registered for a specific Property change.
- A Strategy may change further Properties to maintain the Component's invariants. This triggers the control flow described in the previous bullet.

The two key roles are Component and Role, each of which plays several roles from the different patterns:

- A $Component_R$ object plays the Component roles from the Context Object, Property and Strategy patterns.
- A $Role_R$ object plays the Context Object role from the Context Object pattern, the Client role from the Property pattern, and the Client role from the Strategy pattern.

- Frequently, a $Role_R$ object also plays the Strategy role from the Strategy pattern.

The consolidated role relationship matrix of the Role Objects pattern exhibits six roles. These roles are composite roles which comprise the following roles from the constituting patterns:

```
ComponentClient_R = { ComponentClient_CO }
Component_R       = { Component_CO, Component_P, Component_S }
RoleClient_R      = { ContextClient_CO }
Role_R            = { Component_CO, Context_CO, Client_P, Client_S }
Property_R        = { Property_P }
Strategy_R        = { Strategy_S }
```

The resulting role relationship matrix is depicted in figure 7-10.

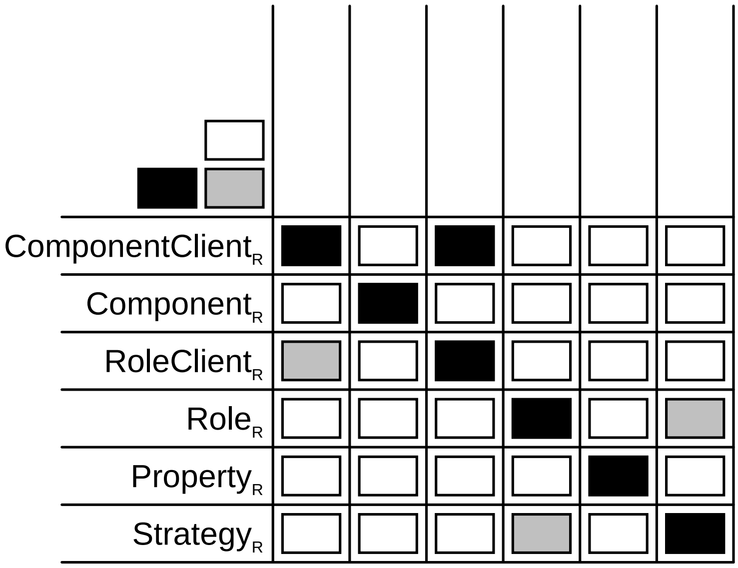

Figure 7-10: Role relationship matrix of the Role pattern

I don’t know of any documentation yet. The pattern was uncovered by discussions with Dirk Bäumer of RWG who uses a more complex variant of this pattern (it is subject to discussion, however, whether it really is a variant or something different).

8 Potpourri I—Atomic Patterns

This chapter lists some atomic patterns which haven't been grouped by purpose but which are referenced by other patterns in this report or which I need for further work.

This chapter sketches the following patterns:

- The *Bridge* pattern describes how you can vary the implementation of an Abstraction in terms of a more down-to-earth implementation by an Implementor. The Abstraction offers a high-level interface to a Client and is implemented based on the more low-level interface of the Implementor.
- The *Chain of Responsibility* pattern lets you build a chain of objects along which client requests are passed. Handling the requests by a chain lets you flexibly configure which object gets to handle which request with which priority. It defines the roles Handler, Predecessor, Successor and Tail.
- The *Composite* pattern lets you design a hierarchy of objects in a flexible way. It defines the roles Node, Parent, Child and Root. Hierarchies are ubiquitous in software design. They are used to manage complexity.
- The *Connector* pattern lets you vary the communication channel between two objects. The handling of communication issues is performed by two objects, the Sender and the Receiver, which negotiate the protocol.
- The *Iterator* pattern lets you traverse complex object structures without knowing about their internals (they stay encapsulated). It defines an Iterator which Clients use to retrieve Elements from an Aggregate.
- The *Mediator* pattern lets you decouple several Colleague objects through the use of a Mediator. By decoupling them through a Mediator, they become independent of each other and can be exchanged without dependencies causing the structure to break.

8.1 Bridge

A Bridge decouples an Abstraction from an Implementor. The Abstraction is implemented in terms of the Implementor; the Implementor can be chosen and changed at runtime, making the bridge easily configurable. Often, the Abstraction has a more elaborate interface than the Implementor. In the trivial case of identical interfaces, the Abstraction boils down to an interface to the Implementor.

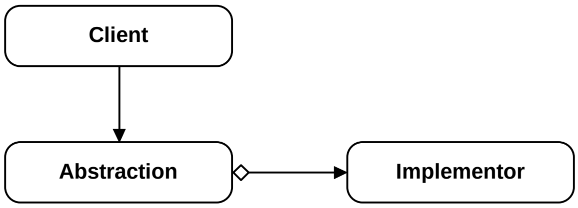

Figure 8-1: Role diagram of the Bridge pattern

There are two collaborations: The Abstraction is implemented in terms of the Implementor interface. At runtime, it gets configured with a specific Implementor implementation. The Client uses the Abstraction and does not make use of the Implementor.

Most applications I have seen chose a specific Implementor based on some general information, for example flags indicating a specific window system. An implementation, which lets the Client choose a specific implementation might use Class Semantic specifications [Rie96a] in the Product Trader style.

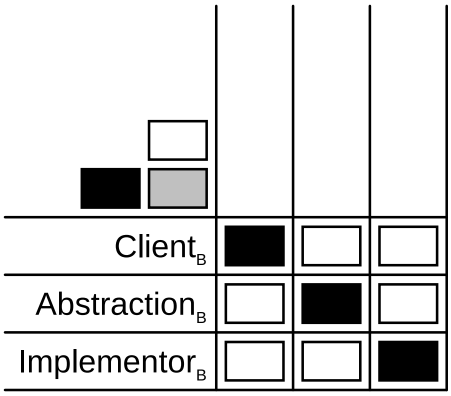

Figure 8-2: Role relationship matrix of the Bridge pattern

For a more elaborate documentation see [GOF95].

8.2 Chain of Responsibility

The Chain of Responsibility pattern defines a chain of objects, starting with a Head, ending with a Tail, each one capable of receiving requests from external Clients. Its lets you dynamically configure the receivers of these external requests. The ordering of objects in the chain defines which object gets to handle a request first. An object may handle a request or choose to pass it on to its Successor. A client can issue a request to every object in the chain, it does not have to submit the request to a head object (which is also the reason why there is no Head role).

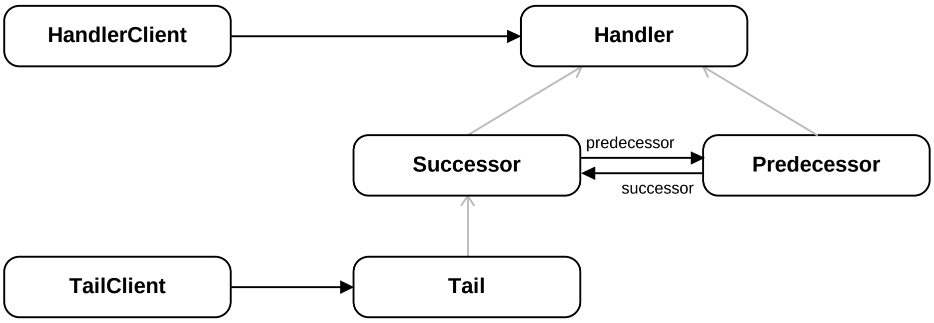

Figure 8-3: Role diagram of the Chain of Responsibility pattern

There are three levels of collaborations:

- *Handler and HandlerClient.* The Handler role defines how a HandlerClient can issue requests to a Handler object.
- *Successor and Predecessor.* A Handler object which has a Successor object can choose to forward a request to it, thus becoming its Predecessor object. This way, requests are passed along the chain.
- *Tail and TailClient.* A TailClient manages the whole chain by using the Tail object as a handle.

There are the following compositions constraints:

- A Successor and a Predecessor also always play the Handler role.
- A Tail object is always a Successor but never a Predecessor.

The role relationship is depicted in figure 8-4:

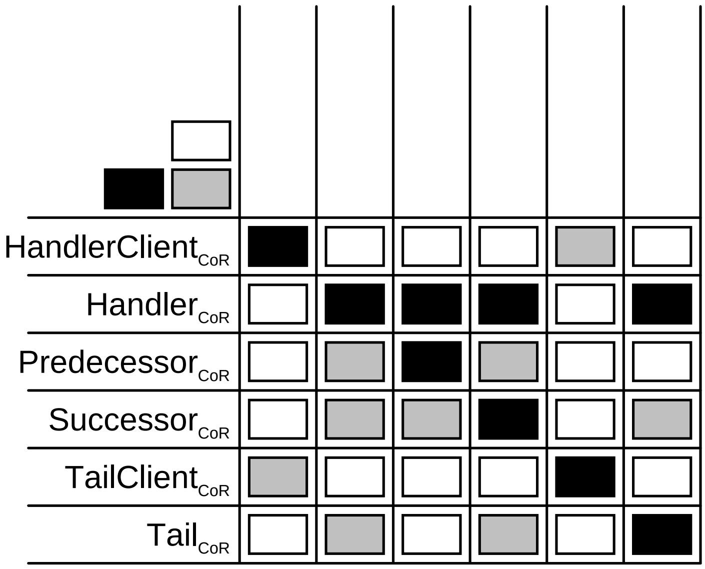

Figure 8-4: Role relationship matrix of the Chain of Responsibility pattern

For a more elaborate documentation see [GOF95].

8.3 Composite

The Composite pattern defines the roles in a tree, that is a hierarchical structure. It defines two roles, Parent and Child, each of which are Nodes, too. The additional Root role serves as a handle for the whole tree.

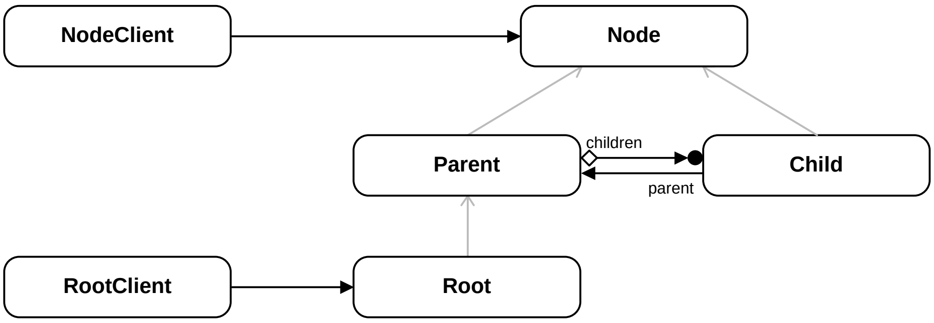

Figure 8-5: Role diagram of the Composite pattern

There are three levels of collaboration:

- *Node and NodeClient.* The Node role defines what a NodeClient can do with nodes from the tree. The Node role partially conforms to the Component participant in [GOF95].
- *Parent and Child.* A Parent object manages its Child objects so that the tree structure is ensured. The Child and Parent roles partially conform to the Component and Composite participants in [GOF95].
- *Root and RootClient.* A RootClient object handles the whole tree using its Root object. The Root role is not discussed in [GOF95].

There are some composition constraints:

- a Parent or a Child object always also plays the Node role
- a Root object always plays the Parent but never the Child role

The role relationship matrix defines the following composition constraints:

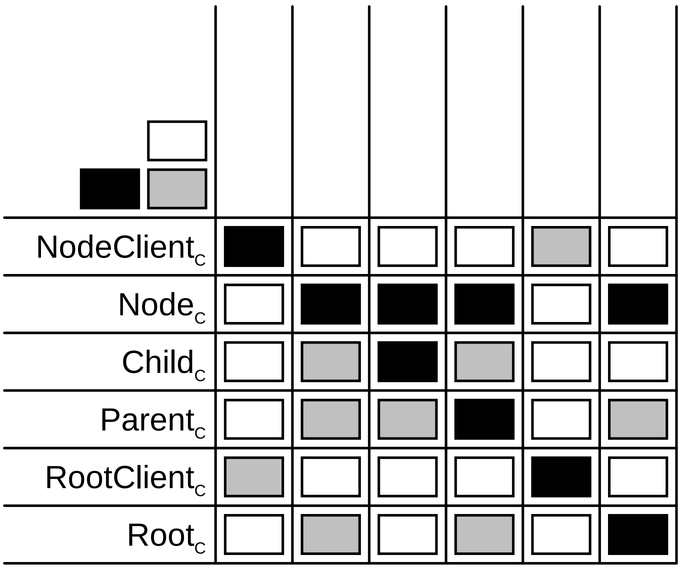

Figure 8-6: Role relationship matrix of the Composite pattern

For a more elaborate documentation see [GOF95].

8.4 Connector

A Connector encapsulates the communication between a Component and a Client by means of two objects, one of them implementing the Sender and one of them implementing the Receiver protocol. By hiding the communication mechanism and protocol from the Component and its Client, it can be configured easily and changed at runtime.

The role diagram is depicted in figure 8-7. In it, the pair of Sender and Receiver represents the Connector.

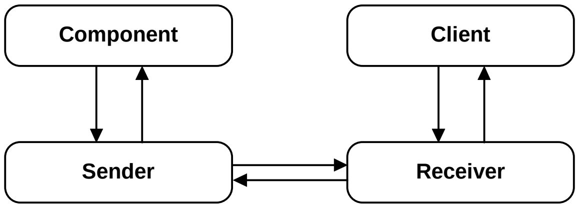

Figure 8-7: Role diagram of the Connector pattern

Known examples abound. Most distributed systems delegate the communication between a Client and a Service to some kind of Connector. The OSI/ISO communication model represents a recursive application of this pattern. CORBA event channels work according to the pattern. Moreover, they let developer chain Connectors. A known specialization of this pattern is the Event Notification pattern discussed in [Rie96b].

The role relationship matrix is straightforward.

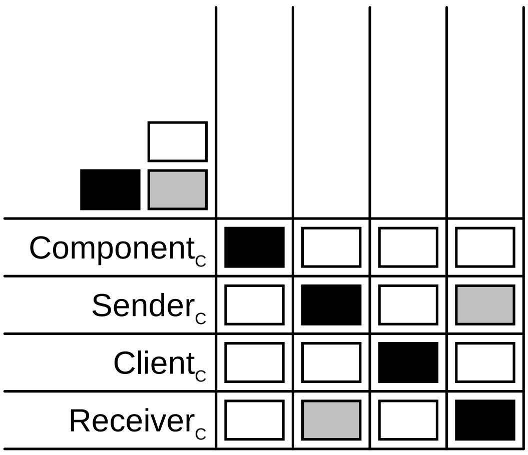

Figure 8-8: Role relationship matrix of the Connector pattern

I don’t know of any documentation of this pattern as such (though the recent emerging literature on software architecture uses the notion of connector as a basic building block).

8.5 Iterator

An Iterator is used by a Client to traverse an Aggregate in order to retrieve the Elements contained within.

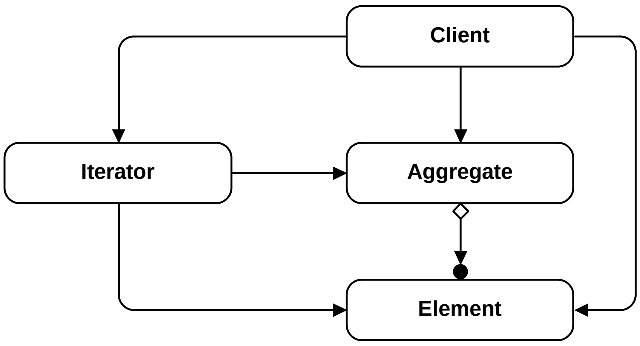

Figure 8-9: Role diagram of the Iterator pattern

The composition constraints are as follows:

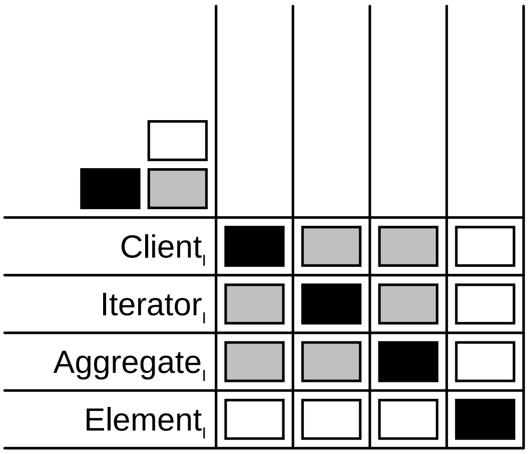

Figure 8-10: Role relationship matrix of the Iterator pattern

For a more elaborate documentation see [GOF95].

8.6 Mediator

A Mediator manages and integrates several related and interdependent Colleagues. Any communication and integration issues are handled by the Mediator, which thereby encapsulates all interdependencies as well as the communication protocol between the Colleagues.

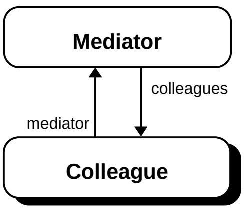

Figure 8-11: Role diagram of the Mediator pattern

The role relationship matrix is straightforward:

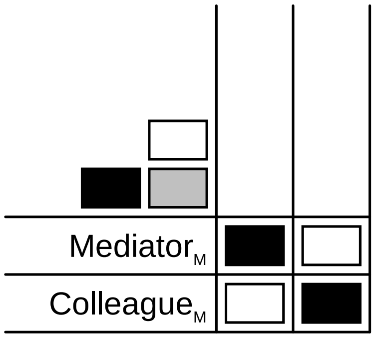

Figure 8-12: Role relationship matrix of the Mediator pattern

For a more elaborate documentation see [GOF95].

9 Potpourri II—Composite Patterns

Eventually, this chapter lists some composite patterns which didn't fit well into anyone of the other chapters.

This chapter lists the following patterns:

- The *Active Bridge* pattern lets you design a two-way bridge that facilitates communication between a Client and some underlying usually hostile API. It combines the traditional Bridge with the Proxy pattern (to encapsulate the API) and the Observer pattern (to facilitate events).
- The *Bureaucracy* pattern lets you design a self-contained hierarchy of objects that lets clients interact with it on any level but which maintains its inner invariants on any account. It uses the Composite, Mediator, Chain of Responsibility and Observer pattern.
- The *Model-View-Controller* pattern lets you design the user interface architecture of interactive software systems. A Model is adapted to a display context by a View and to a user interaction context by a Controller. It uses the Observer, Strategy, Composite and Context Object pattern, and often employs further patterns.

9.1 Active Bridge

An Active Bridge lets you decouple a Client from some external resource by means of a Bridge. Its Implementor is a Proxy to the API which wraps the resource. It communicates with its Abstraction by using the Observer pattern. Thus, control flows in two directions. The Client may trigger some action and the API might deliver some events to the Implementor. The proper configuration of the Bridge is ensured by a Factory implemented using a Factory method.

The pattern's role diagram is depicted in figure 9-1; it composes the Bridge, Proxy, Observer, Factory and Factory Method pattern. It is a frequently recurring framework when it comes to connecting to some external resources like window system or communication mechanisms like CORBA or Java RMI.

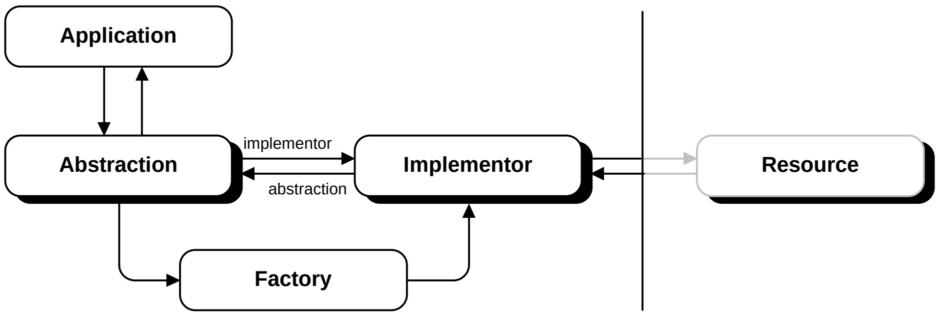

Figure 9-1: Role diagram of the Active Bridge

The roles from the constituent patterns map nicely on composite roles. The composite roles are defined as follows:

Application$_{AB}$ = { Client$_{B}$ }
Abstraction$_{AB}$ = { Abstraction$_{B}$, Client$_{P}$, Observer$_{O}$, Client$_{AF}$ }
Implementor$_{AB}$ = { Implementor$_{B}$, Proxy$_{P}$, Subject$_{O}$, Product$_{AF}$, Product$_{FM}$ }
Factory$_{AB}$ = { Factory$_{AF}$, Client$_{FM}$, Creator$_{FM}$ }
Resource$_{AB}$ = { Subject$_{P}$ }

This leads to the role relationship matrix of figure 9-2.

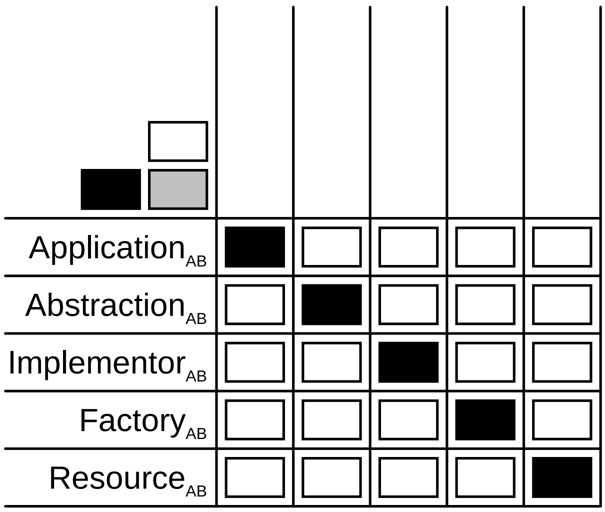

Figure 9-2: Role relationship matrix of the Active Bridge pattern

To my knowledge, this pattern has not been documented in detail. It can be found in ET++ [WG95], [Yel96] and our own frameworks.

9.2 Bureaucracy

The Bureaucracy pattern is a composite pattern which lets you build self-contained hierarchical structures that can interact with clients on every 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 [Web47] which seems to work well for software systems (at least).

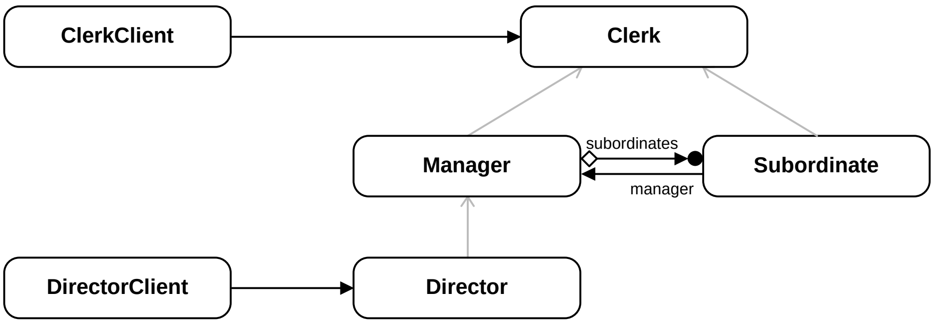

Figure 9-3: Role diagram of the Bureaucracy pattern

The Composite pattern lends the Bureaucracy pattern its structure, and the Mediator, Chain of Responsibility and Observer pattern define its dynamics.

- *Composite.* The Composite pattern defines the hierarchical structure of a Bureaucracy instantiation. At the root, the Director resides. On every level of the hierarchy, except at the bottom, Managers coordinate Subordinates. Each non-bottom level Subordinate is a Manager for its own Subordinates. Eventually, all objects are Clerks to serve their Clients (they should, at least).
- *Mediator.* A Manager is Mediator which receives information from its Subordinates and manages them according to that information. It’s goal is to fulfill Client requests passed up the hierarchy as well as to maintain the hierarchies inner stability and invariants.
- *Chain of Responsibility.* A Clerk which cannot handle a Client request to its satisfaction, passes the request up the hierarchy to its Manager. Thus, the Clerk plays the role of a Predecessor in the Chain of Responsibility pattern, and the Manager plays the role of the Successor.
- *Observer.* A Manager acts as the Observer of its Subordinates, that is its Subjects. State changes of a Subordinate might force the Manager to act in order to maintain the hierarchy with respect to invariants the Subordinate cannot and should not know about.

The Chain of Responsibility and the Observer pattern effectively define the communication protocol from the Subordinates to the Manager as required by the Mediator pattern. The Mediator pattern defines the traditional Manager roles and relationships. The Composite pattern finally lends structure to the hierarchy.

The composition of these patterns leads to the following composite roles:

$$
\begin{aligned}
\texttt{DirectorClient}_B &= \{\ \texttt{RootClient}_C,\ \texttt{TailClient}_{CoR}\ \} \\
\texttt{Director}_B &= \{\ \texttt{Root}_C,\ \texttt{Tail}_{CoR}\ \} \\
\texttt{Manager}_B &= \{\ \texttt{Parent}_C,\ \texttt{Mediator}_M,\ \texttt{Successor}_{CoR},\ \texttt{Observer}_O\ \} \\
\texttt{Subordinate}_B &= \{\ \texttt{Child}_C,\ \texttt{Colleague}_M,\ \texttt{Predecessor}_{CoR},\ \texttt{Subject}_O\ \} \\
\texttt{ClerkClient}_B &= \{\ \texttt{NodeClient}_C,\ \texttt{HandlerClient}_{CoR}\ \} \\
\texttt{Clerk}_B &= \{\ \texttt{Node}_C,\ \texttt{Handler}_{CoR}\ \}
\end{aligned}
$$

The role relationship matrix is depicted in figure 9-4.

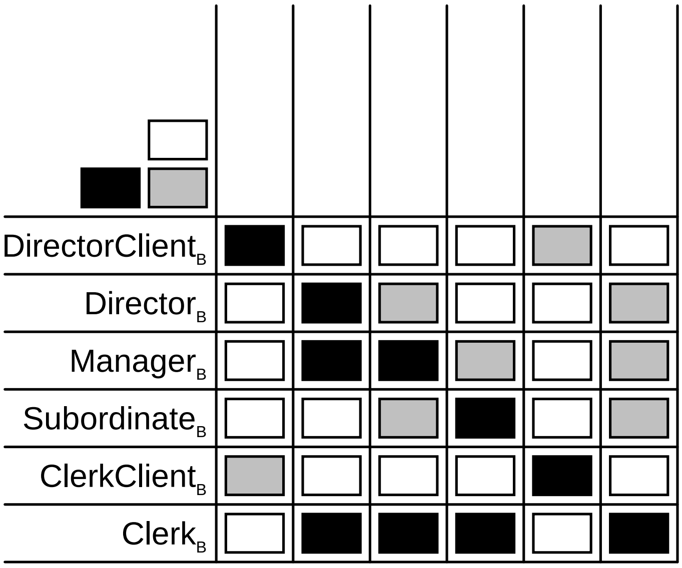

Figure 9-4: Role relationship matrix of the Bureaucracy pattern

The Bureaucracy pattern has been documented in [Rie97].

9.3 Model-View-Controller

The Model-View-Controller (MVC) pattern helps you design the user interface parts of interactive software systems [Ree96]. The representation of an application's Model is separated from its user interface presentation by means of a View. For every Model, there may be any number of Views presenting it to a user. A View always comes with a Controller. The Controller receives user input and takes appropriate action, for example by invoking mutating operation calls on the Model. A View observes the Model in order to react to changes to it—as they might occur, for example, due to manipulations through another View/Controller pair. Thereby, a Model's presentation by a View is always kept up-to-date.

Figure 9-5 shows the pattern's role diagram:

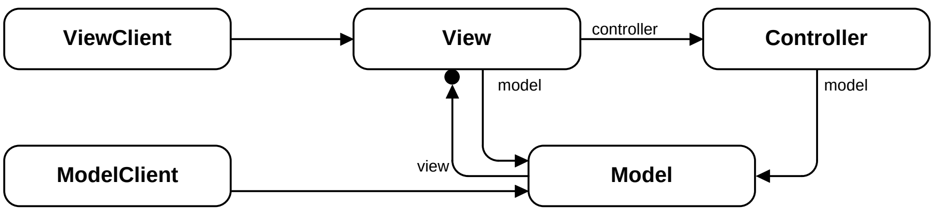

Figure 9-5: Role diagram of the Model-View-Controller pattern

The MVC pattern is composed from the following patterns:

- *Observer*. Both View and Controller observe the Model in order to react to changes by updating their state. The View reflects state changes of the Model in the user interface, and the Controller adjusts its input mode based on the Model's state.
- *Strategy*. The View delegates the interpretation and handling of user input events to the Controller. Thus, the View acts as the Context for the Controller which acts as a Strategy for evaluating user input (strictly speaking, this control flow is based on technical considerations of connecting to the underlying window system—in ObjectWorks, the user input was dispatched through a Controller hierarchy to the correct Controller).

- *Context Object*. Both View and Controller of MVC represent a specific Context adaptation of the Model as the Component in the Context Object pattern. The View adapts the Model to the context of user interface presentation, and the Controller adapts the Model to varying interaction modes and devices.

The MVC pattern is a very mature pattern which has seen many variations. The role diagram presented above comprises the core of the pattern, that is its essential structure. One of the most important extensions not covered by the role diagram is the use of the Composite pattern to structure the View object hierarchy. In addition, many other patterns are employed to serve tactical purposes, for example Factory Method or Null Object.

The role relationship matrix is straightforward, with one minor twist, explained below.

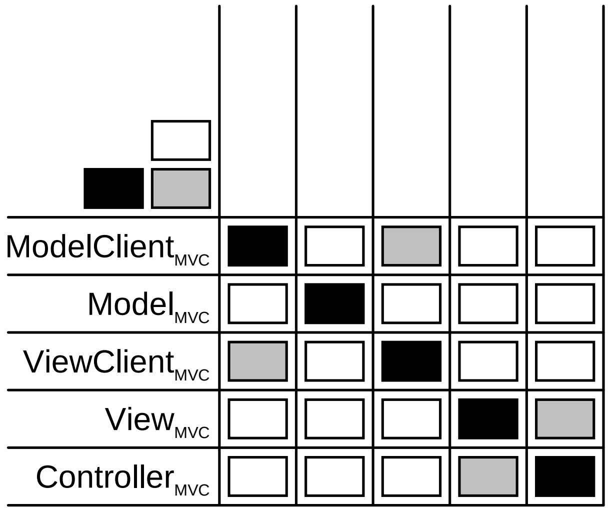

Figure 9-6: Role relationship matrix of the Model-View-Controller pattern

Many systems have omitted the explicit Controller object and merged it into the View. Thus, View and Controller have been marked in the role relationship matrix as roles possibly to be played by the same object.

The Model-View-Controller paradigm has been documented extensively, see for example [KP88, Ree96] and [POSA96].

References

Bec96 Kent Beck. *Smalltalk Best Practice Patterns.* Reading, Massachusetts: Addison-Wesley, 1996.

BR97 Dirk Bäumer and Dirk Riehle. “Product Trader.” In Pattern Languages of Program Design 3. Reading, MA: Addison-Wesley Longman, 1997.

DA97 Paul Dyson and Bruce Anderson. “State Patterns.” In Pattern Languages of Program Design 3. Reading, MA: Addison-Wesley Longman, 1997.

Gam97 Erich Gamma. “Facet.” In Pattern Languages of Program Design 3. Reading, MA: Addison-Wesley Longman, 1997.

GOF95 Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Design*. Reading, Massachusetts: Addison-Wesley, 1995.

HLN+90 David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring and Mark Trakhtenbrot. “STATEMATE: A Working Environment for the Development of Complex Reactive Systems.” *IEEE Transactions on Software Engineering* 16, 4 (April 1990): 403-414.

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

OMG96 OMG. *CORBA2, CORBAservices, CORBAfacilities*. Framingham, Massachusetts: Object Management Group, 1996.

POSA96 Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. *Pattern-Oriented Software Architecture.* England: Wiley & Sons, 1996.

Ree96 Trygve Reenskaug, with Per Wold and Odd Arild Lehne. *Working with Objects.* Greenwich, Manning: 1996

Rie96a Dirk Riehle. “Patterns for Encapsulating Class Trees.” *Pattern Languages of Program Design 2.* Edited by John M. Vlissides, James O. Coplien and Norman L. Kerth. Reading, Massachusetts: Addison-Wesley, 1996. Chapter 6, page 87-104.

Rie96b Dirk 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 *Proceedings of the Ubilab Conference ’96, Zürich.* Edited by Kai-Uwe Mätzel and Hans-Peter Frei. Konstanz, Universitätsverlag Konstanz, 1996. Page 137-152.

Rie97 Dirk Riehle. "Bureaucracy." In Pattern Languages of Program Design 3. Reading, MA: Addison-Wesley Longman, 1997.

SPL96 Linda M. Seiter, Jens Palsberg and Karl J. Lieberherr. “Evolution of Object Behavior using Context Relations.” SIGSOFT ’96, *Conference Proceedings.* ACM Press, 1996.

Web47 Max Weber. *The Theory of Social and Economic Organization*. New York: Oxford University Press, 1947.

WG95 Andre Weinand and Erich Gamma. “ET++ A Portable Homogeneous Class Library and Application Framework.” *Object-Oriented Application Frameworks.* Manning, 1995.

Yel96 Phillip M. Yelland. “Creating Host Compliance in a Portable Framework.” OOPSLA ’96, *Conference Proceedings.* ACM Press, 18-29.