# Describing and Composing Patterns Using Role Diagrams

Dirk Riehle

Ubilab, Union Bank of Switzerland
Bahnhofstrasse 45, CH-8021 Zurich

e-mail: Dirk.Riehle@ubs.com

*Design patterns are patterns of classes and objects that represent solutions to recurring design problems. They are usually described using class diagrams. Class diagrams, however, often intertwine the actual solution with efficient ways of implementing it. This paper uses role diagrams to describe and compose patterns. Role diagrams help designers focus on the collaborations and distribution of responsibilities between objects. Role diagrams also are a better starting point for composing patterns. This paper presents several examples and reports on first experiences with using role diagrams for composing patterns which have been promising.*

## 1 Introduction[1]

Object-oriented software design patterns are used to document and communicate software design experience. If directed at solving problems in design, patterns are often defined as solutions to recurring design problems that turn up in specific contexts. The work of Gamma et al. comprises 23 such patterns which consist of collaborating objects, classes and operations [Gam95]. Each of these patterns has been a long known solution to a problem. However, their explication as patterns has made them an easier approachable subject of specification and communication.

While the given definition of patterns as recurring problem solutions in contexts is quite general, the appeal and positive resonance patterns have garnered is mainly based on their focus on the interaction of the basic elements of object-oriented design and programming: The patterns are described through class diagrams for which the structural and dynamic relationships of the involved elements like classes, objects and operations are discussed.

However, class diagrams are solution oriented in that they actually prescribe design templates which can be reused by developers. Reusing means applying the pattern to solve a particular design problem when building a concrete system. Since class diagrams are the primary means of specifying patterns today, patterns tend often to be too implementation oriented thereby missing potential other solutions which could have been applicable as well.

This paper shows that class diagrams should be supplemented with role diagrams that specify the solution to the design problem more concisely and without implementation ballast. Class diagrams then are used as design templates which tailor the general solution of the role diagram to satisfy specific implementation constraints.

---

1 Published in *Proceedings of the Ubilab Conference '96, Zurich.* Edited by Kai-Uwe Mätzel and Hans-Peter Frei. Konstanz, Universitätsverlag Konstanz. Pages 137-152. Originally published in *Proceedings of WOON '96*, the 1st International Conference on Object-Orientation in Russia. Edited by A. Smolyani and A. Shestialtynov. St. Petersburg Electrotechnical University, 1996.

Role diagrams focus on the interaction of objects. They define the roles played by objects and thus the views objects hold on each other. Role diagrams are a much better means for modeling complex situations of interacting objects than class diagrams [Ree96]. Based on the different roles an object can play, its interface is partitioned into distinct protocols, each one associated with a role. Modeling with roles on a technical level can be seen as an extension to working with methods like CRC cards [Wir90] or Contracts [Hel90].

This leads to a three tier structure for describing a pattern: The first tier specifies the core of the pattern by means of a role diagram. The second tier consists of a set of class diagrams that serve as design templates for implementing the pattern. The last tier then are concrete systems which implement the pattern in various ways.

The benefits of using role diagrams for describing patterns reach farther. Class diagrams make attempts to compose them both on a pattern as well as a pattern instantiation level difficult. However, no relevant system has yet been built from a single pattern, nor will it ever be: It is always the composition of pattern instantiations which makes up a framework or a whole system. In role diagrams, the different responsibilities of an object are kept distinct, namely as the different roles an object can play. If these roles are expressed as distinct protocols, composition of pattern instantiations is eased: The different roles are simply merged into the class interfaces of the objects participating into the object structure.

Section 2 shows that class diagrams have shortcomings as the sole description mechanism for patterns. Section 3 shows that the trade-offs introduced in section 2 can be resolved by using role diagrams. Section 4 examines the resulting three tier structure of role diagrams, class diagrams and concrete systems. In section 5 pattern compositions are discussed. Finally, section 6 discusses related work, and section 7 presents some concluding remarks.

## 2 Patterns and Class Diagrams

Object-oriented software design patterns are patterns of interacting classes, objects and operations. Gamma et al. demonstrate how these basic elements work together to form larger wholes like the Composite, Observer or Decorator pattern [Gam95].

Most researchers use a presentation form for patterns that consists of several sections, each one devoted to a specific aspect of understanding and implementing the pattern. For example, Gamma et al. provide a Structure, Participants and Collaborations section (and other more). The Structure section presents a class diagram, the Participants section explains the purpose of the different classes in the diagram, and the Collaboration sections describes the interaction between the classes.

A typical example is the Observer pattern which is presented with a class diagram like the one in figure 2.1. Two abstract classes, Subject and Observer, can be identified, and some interactions, possible calls from the Subject class to the Observer class and vice versa. Rectangles depict classes, an arrow depicts a use-relationship, a line with a triangle depicts an inheritance relationship, and a half of a filled circle depicts a cardinality of 0..n.

The purpose of the Observer pattern is to implement a dependency relationship between classes on an abstract level. It lets Subject instances notify Observer instances about state changes without intimately intertwining the concrete subclasses of Subject and Observer. State changes of the “subject of observation” are signaled to the “observer” so that it can react to the changes immediately, for example by maintaining some possible dependencies and invariants.

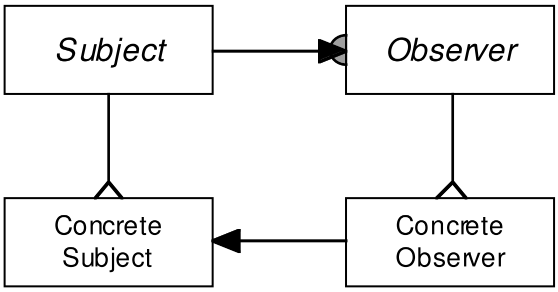



Figure 2.1: Class diagram of the Observer pattern.

While the structure diagram of figure 2.1 concisely presents a pattern, it is the abstraction of only one possible variant of concrete instantiations of this pattern. Frameworks like ET++ [Wei94] and Smalltalk [Gol89] do not offer two distinct Subject and Observer classes, but only one root class, Object, which offers both the protocols of the Subject and the Observer class. Figure 2.2 shows how a class diagram abstracting from ET++ and Smalltalk might present the pattern.

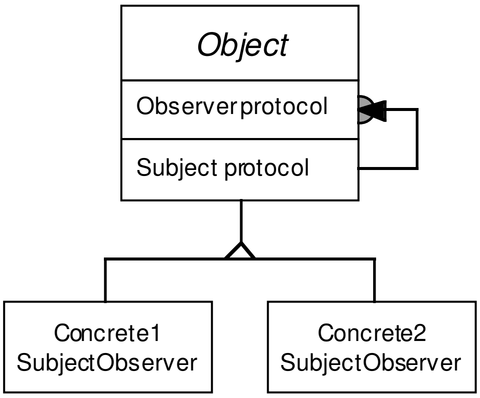



Figure 2.2: Structure of the Observer pattern if abstracted from ET++ or Smalltalk.

A third variant is illustrated in figure 2.3, which presents another class diagram that can be applied to implement the dependency relationship expressed by the Observer pattern. This time the concrete design structure does not resemble the one from figure 2.1 any more, but is different in light of different requirements for implementing it. Figure 2.3 has been elaborated as the Event Notification pattern, because its class structure diagram significantly differs from the Observer class diagram of figure 2.1. Conceptually, however, it is an Observer, since it deals with maintaining update dependencies [Rie96a].

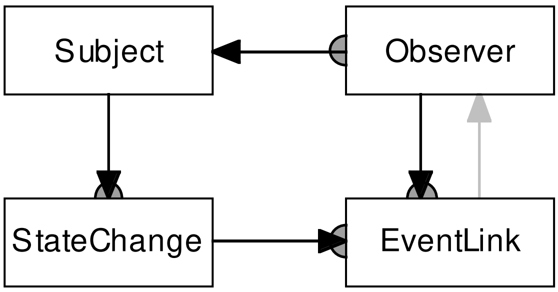



Figure 2.3: The class diagram from the Structure section of the Event Notification pattern.

All three presented class diagrams react to variations in implementation requirements. However, they all share a common vision, that is the actual problem solution of allowing anonymous notification of dependent objects about state changes of those objects which they depend on. It is this common solution which software developers use to solve their problems in concrete designs. These solutions can be captured more concisely by role diagrams, subsuming the different class diagrams as implementation oriented design templates that are used to implement the pattern efficiently.

## 3 Patterns and Role Diagrams

A role played by an object is a specific view a client object holds on the other object. It is the client which defines what constitutes the role [Kri96]. Roles can be formally expressed by some role type which specifies the interface of the role. An object can play several roles. It can therefore interface with client objects in different ways. Different clients can have different views on the same object.

Modeling a pattern as the abstraction from a set of interacting objects whose interaction is specified by the roles they play opens a new perspective on pattern description and, as discussed later, on pattern composition. In this paper, role diagrams are used for technical purposes and not for purposes of conceptually modeling an application domain. In light of this view, the notion of class looses its status as the abstraction from a phenomenon [Dah72] and becomes an implementation construct only [Ree96].

Figure 3.1 shows the Observer pattern using a role diagram. Roles are depicted as rounded rectangles. The possible relationships between roles are the same as for classes. Thus, the same notation is used. Kristensen and Osterbye discuss role aggregation and specialization in more detail [Kri96].

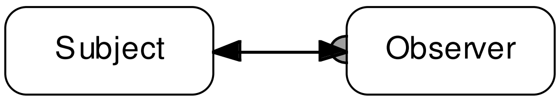



Figure 3.1: Role diagram of the Observer pattern.

The role diagram for the Observer pattern is fairly simple, and the major part of the discussion of such a pattern will be devoted to the behavioral specification of the single roles, for example using techniques from OOram [Ree96] or Contracts [Hel90]. Since roles can be formally expressed as types, every specification mechanism can be used that is adequate for the pattern's domain and intentions.

The power and utility of role modeling becomes apparent when considering how such a role diagram can be implemented. A role diagram does not prescribe a certain class design, but only sets up constraints in such a way that the actual core of the solution is maintained. Objects only have to be capable of playing certain roles but are not forced to adhere to a strict class inheritance hierarchy. This leaves a much wider design space to developers.

The role diagram of figure 3.1 can be implemented using the design template of figure 2.1 with two abstract superclasses that represent the Subject and Observer role protocols. It can also be implemented according to figure 2.2 in which the two roles are merged into a single superclass. Finally, figure 2.3 represents a third viable design template in which the two objects are fully decoupled using intermediate state change and event link objects.

Figure 3.2 presents a role diagram for the Composite pattern. Gamma et al. present the Composite pattern as an efficient means for representing and implementing tree structures. Rewriting the Composite pattern as proposed by this paper leads to the role diagram of figure 3.2 and several possible class diagrams, for example, the one in figure 3.3.

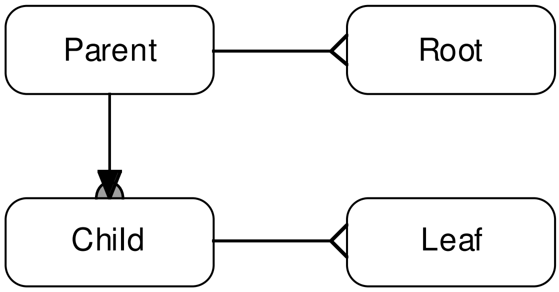



Figure 3.2: Role diagram of the Composite pattern.

The Composite role diagram identifies the roles Parent and Child, and the specializations Root of Parent and Leaf of Child. Concrete designs for this pattern are likely to follow the class diagram from the pattern in [Gam95], however, they don't have to.

The role diagram for the Composite pattern is similar to the default class diagram of the Composite pattern as depicted in figure 3.3. Gamma et al. classify the Composite pattern as a structural pattern, and role models are largely concerned with interaction so that there isn't much to abstract.

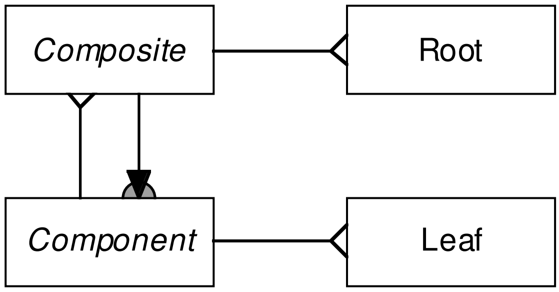



Figure 3.3: Class diagram of the Composite pattern.

In figure 3.3 the Component class provides the Child role protocol and the Composite class provides the Parent role protocol. The class diagram further adds the idea of making Composite a subclass of Component, thereby allowing convenient and efficient implementation of the pattern.

While structural patterns tend to have similar role and class diagrams, behavioral patterns seem to allow for a wider class design space. Behavioral patterns are more concerned with the behavior of the participating objects than with their structure.

Figure 3.4 presents the role diagram of the Chain of Responsibility pattern. The purpose of this pattern is to forward requests along a chain of objects until an object handles the request. Thereby it is left unspecified which object will eventually handle a request. This leads to enhanced flexibility.

For the role diagram of the Chain of Responsibility pattern, the two central roles Handler and Successor can be identified. The role Tail represents a boundary condition, just like the Leaf and Root role in the Composite pattern, used to satisfy the recursive nature of object structures following this pattern.

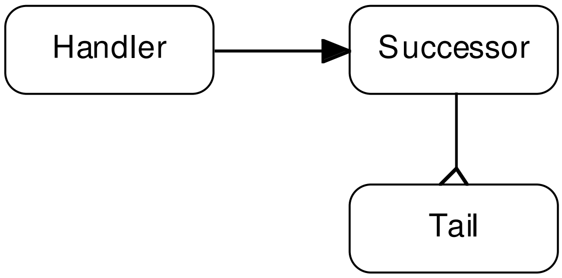



Figure 3.4: Role diagram of the Chain of Responsibility pattern.

Figure 3.5 and 3.6 show two different class diagrams that can be used as design templates for implementing the pattern.

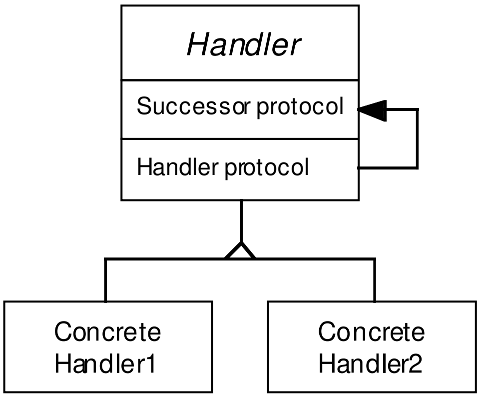



Figure 3.5: A class diagram for the Chain of Responsibility pattern.

Figure 3.5 shows the standard design template as taken from Gamma et al. [Gam95]. The Handler class comprises both role protocols, Handler and Successor. When applied, it will also comprise a problem specific protocol based on the application domain it is applied in. Figure 3.6 shows an alternative design template which separates the two protocols as two distinct classes.

It should be noted that other authors propose different role diagrams. In particular, Reenskaug et al. suggest that the Composite pattern consists of only two roles, Parent and Child, and that more elaborate role diagrams are derived by synthesis from this basic role diagram. It is further suggested that object chains like those described by the Chain of Responsibility pattern consist of at least three roles, Predecessor, Job and Successor. I consider it to be too early now to settle on specific role diagrams for patterns. This paper focuses on presenting the basic techniques and their application to patterns. The future will provide changes and revisions of the role diagrams.

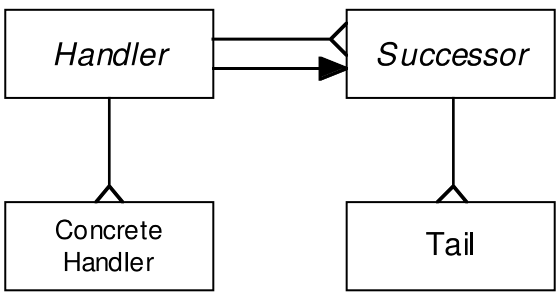



Figure 3.6: An alternative class diagram for the Chain of Responsibility pattern.

# 4 Levels of Abstraction

Section 2 demonstrated that class diagrams are a viable means for describing design templates for patterns. However, they are close to concrete designs and thereby fail to encompass the full possible design and implementation space of the pattern to be described. Section 3 then demonstrated that role diagrams are a more adequate means of capturing the essentials of a pattern at the expense of being less concrete. This section finally shows that role and class diagrams can work together and thus provide a more complete and concise way of specifying patterns than role or class diagrams alone.

Role and class diagrams aim at different levels of abstraction. While role diagrams explicitly aim at describing the distribution of responsibilities and the interaction between objects playing specific roles in an exemplary object composition, class diagrams are focusing more on how this responsibility and interaction can be assigned to classes as primary means of programming. Thus, role diagrams are specification oriented, while class diagrams are implementation oriented. Both is needed, since a good object-oriented design pattern should comprise a clear and unbiased problem solution as well as efficient ways of implementing it.

One day, roles might become primary means of programming. Today, however, classes are the primary means of object-oriented programming, and therefore have to be explicitly included into a pattern. This leads to figure 4.1 which illustrates the three levels of abstraction to be taken into account when describing a pattern.

Figure 4.1 is to be interpreted in the following way: A pattern specified by a role diagram can be made more concrete by a class template which in turn can be instantiated in numerous applications. Thus, a role pattern specifies and abstracts from a family of class patterns which in turn specify and abstract from a family of concrete implementations.

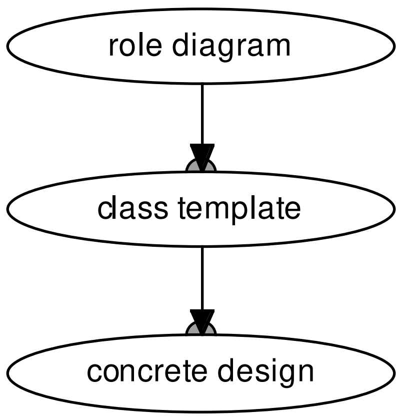


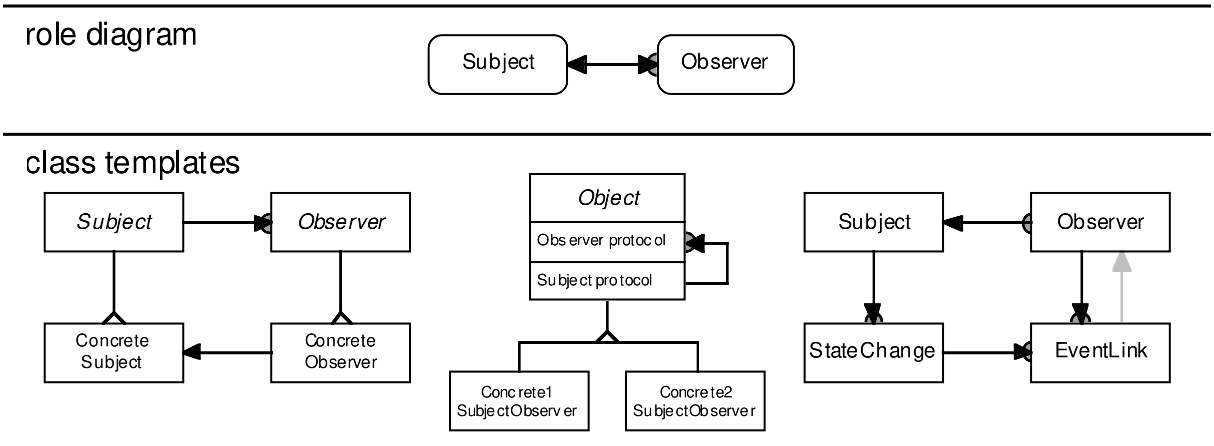

Figure 4.1: Levels of pattern abstractions.

Figure 4.2 illustrates the case of the Observer pattern. At the top, a single role diagram is depicted. In the middle, the three class diagrams from section 2 are depicted, and at the bottom a number of concrete systems are listed which applied the class diagrams in concrete designs.

A very concise three layer description of the Observer pattern might now like this:

### Intent

The Observer pattern lets “Observer” objects monitor “Subject” objects without statically coupling their interfaces and implementations. Observer objects may immediately synchronize and update their state according to changes of state in the Subject object. Thus,




concrete systems

- InterViews
- Unidraw
- Andrew Toolkit
- ET++
- Smalltalk
- THINK library
- Prism
- Tools and Materials Framework
- CORBA Event Channels

Figure 4.2: Levels of abstraction for the Observer pattern.

the maintenance of state dependencies between observing objects and their subjects of observation is accomplished.

**Role diagram**

Two roles can be identified:

- The Observer role is played by objects which depend on other objects that play the Subject role. Its protocol allows linking it to a Subject object and to receive state change notifications.
- The Subject role protocols offers operations to attach and detach Observer objects, as well as some facilities to specify their interest in specific events.

**Class templates**

At least three class templates are in use today:

- The roles are expressed through two separate abstract superclasses. The classes of objects determined to play one or both roles have to inherit from these superclasses. This requires almost always multiple inheritance, since objects very often have to play both roles.
- The two roles are merged into a single abstract class from which all classes of objects determined to play a role in the Observer pattern have to inherit. This is the simplest solution and also the most frequently found.
- Objects playing the Observer role link in via anonymous operation references to Subject objects. The linking is carried out through intermediate state change and event link objects. This variant requires operation references. In return, Observer and Subject objects are fully decoupled and no common superclasses are needed.

Each described variant corresponds to a variant in figure 4.2.

**Implementation issues**

Concrete systems have to address questions like how to specify interests in specific events, how to carry out efficient notification dispatch, how to efficiently manage the dependency relationship, how to distribute events, etc.

# 5 Pattern Composition

Role diagrams provide a concise specification of the roles and associated interfaces which objects in an object structure can play. As discussed, role diagrams can be mapped on class diagrams which serve as design templates from which concrete designs are derived. This approach can be used advantageously to compose patterns.

Composing patterns based on their role diagrams means deriving a new role diagram from the single patterns' role diagrams. In this new role diagram, several roles have been merged into a new composite role. Such a new composite role might but need not introduce a new role protocol. Essentially, each composite role in a pattern composition represents a set of composition constraints: An object playing the composite role has to be able to play all the constituting roles.

Among the possible pattern compositions certain compositions stick out. These compositions provide a synergy from the interaction of the roles united in a single object which keeps recurring as a design theme that is used to solve problems in specific contexts. These pattern compositions represent true new patterns, called composite patterns in this paper. They are to be distinguished from arbitrary pattern compositions since they should be described as patterns in their own right.

Composite patterns should not be confused with the Composite pattern as described in [Gam95]: “A composite pattern” is a composition of some patterns which has pattern status itself, while “the Composite pattern” is a concrete pattern used to represent hierarchical object structures. When talking about composite patterns, “composite” is written with a small “c.” When talking about the Composite pattern, it is written with a capital “C.” In addition, there is only one Composite pattern while there can be any number of composite patterns.

The first subsection discusses arbitrary pattern compositions and the second subsection discusses composite patterns. The third subsection finally provides a simple technique for specifying and working with composition constraints in composite patterns.

## 5.1 Pattern compositions

Pattern compositions can be found in large quantities. Gamma et al. [Gam95] and Zimmer [Zim95] discuss and show many relationships between known patterns. For example, an Abstract Factory often exists only once and therefore is usually implemented as a Singleton. Furthermore, an Abstract Factory is usually implemented using Factory Methods, however, it can be made more flexible by parameterizing it with Prototypes and replacing the factory methods with concrete methods cloning the prototypes.

While these pattern compositions seem to be useful, there is no real need to deal with them unless they are actually needed in software design as a solution to a concrete problem at hand. Then, these patterns can be applied. The pattern application is usually unproblematic, since the patterns do not interfere with each other. The objects make use of each other through their role protocols, but there is little general synergy between the different roles played by an object.

All in all, there seem to be no constraints which prohibit or foster certain combinations of patterns, and the application and composition of patterns always takes place in the concrete situation, that is on a pattern instantiation level.

## 5.2 Composite patterns

A composite pattern is first of all a pattern: It represents a design theme that keeps recurring in specific contexts. It is called a composite pattern, because it can best be explained as the composition of 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 constituent patterns.

Riehle presents a first pattern of this kind, called Bureaucracy [Rie96b]. It is composed from the Composite, Observer and Chain of Responsibility pattern. The Bureaucracy pattern is used to build hierarchical object structures which are capable of maintaining their inner consistency. The overall structure of the Bureaucracy pattern is determined by the Composite pattern while the control flow in the hierarchy is managed through the interlocking application of the Observer and Chain of Responsibility pattern.

The Composite pattern prescribes the hierarchical tree structure of a Bureaucracy application. The Observer pattern is used to make a superior component in the hierarchy monitor a subordinate component. This is used effectively to integrate components into the hierarchy which have not been designed with the particular intent of integrating them into a hierarchy. The Chain of Responsibility pattern is used to forward client requests issued to a component up the hierarchy until it is handled by some component with enough context information. This pattern can be found in frameworks like ET++ [Wei94], Unidraw [Vli90], and Tools and Materials Frameworks [Rie95b].

Figure 3.1, 3.2 and 3.4 show the role diagrams of the Observer, Composite and Chain of Responsibility pattern. Figure 5.1 shows the Bureaucracy pattern as the composition of these three patterns. It can be observed that composite roles like Manager comprise several roles, here the Parent, Successor and Observer role. The next subsection elaborates how the composition was carried out and how class templates can be derived from it.

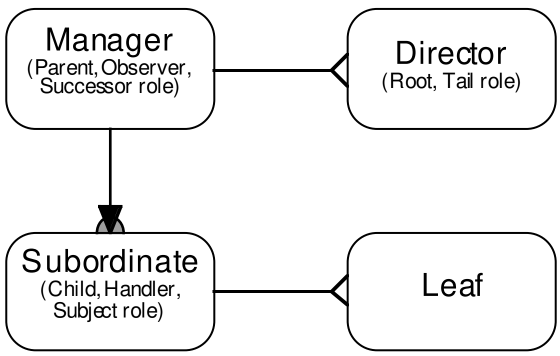



Figure 5.1: Role diagram of the Bureaucracy pattern.

Composite patterns capture the rationale or parts of the rationale behind successful recurring frameworks. For example, MVC [Kra88] can be understood to be a composite pattern consisting of (at least) the Observer, Composite and Separation of View from Document patterns. Today, there are several variants of MVC implemented in frameworks. Thus, composite patterns can be used to design and motivate frameworks, or help to restructure them to met their objectives better.

## 5.3 Composition constraints

As explained, a composite pattern can be described best as the composition of some other patterns. This composition is not arbitrary but follows some constraints which ensure that the proper semantics of the composite pattern are maintained. These constraints are called composition constraints.

A composition constraint is an element of the set of pairwise relationships between the roles objects can play. The set of possible roles is the union of all roles from the constituting patterns. Such a composition constraint might state something like: "An object playing role Tail must always be able to play role Root." Or: "An object playing role Child must sometimes but not always also be able of playing role Observer." Or: "An object playing role Parent will never have to play role Leaf."

Of the last three examples, only the first one was a real constraint in the sense of a logical implication, while the others are more adequately said to describe degrees of freedom. Nevertheless, they are subsumed under the notion of constraint here. Degrees of freedom are important for efficiently implementing pattern instances, see below.

Composition constraints are derived from the analysis of an archetypal object structure representing a typical instance of the pattern. Figure 5.2 depicts such an archetypal object structure for the Bureaucracy pattern. Each ellipse represents an object. It names all the roles the object has to play in the overall object structure.

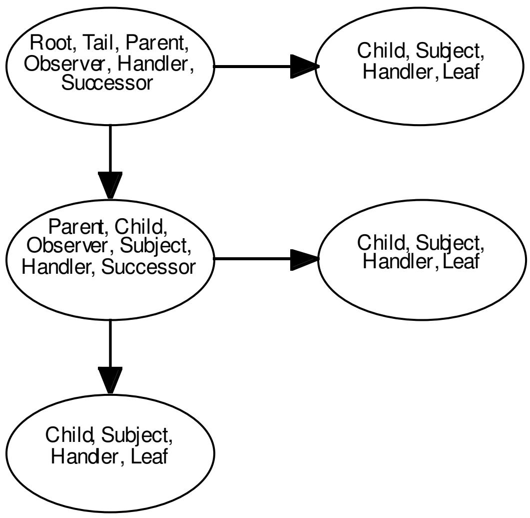



Figure 5.2: Archetypal object structure of the Bureaucracy pattern.

Three relationships between two roles A and B can be observed:

- Role A always occurs together with role B.
- Role A sometimes but not always occurs together with role B.
- Role A never occurs together with role B.

The composition constraints of a composite pattern can be captured by a role relationship matrix. This matrix presents the relationship between two roles A and B as the matrix entry at (A, B). Figure 5.3 shows the result of an analysis of the archetypal object structure of figure 5.2 according to the relationships defined above. In figure 5.3, a black rectangle depicts case 1, a gray rectangle depicts case 2, and a white rectangle depicts case 3. Thus, the black rectangle at (Observer, Successor) shows that an object playing role Observer also always plays role Successor, etc. Note that the relationships are not symmetric (while Root implies Parent, Parent does not imply Root).

Essentially, the role relationship matrix contains the “raw data” from the analysis of the archetypal object structure of figure 5.2. It then has to be interpreted appropriately. Below, a pragmatic interpretation is presented, which motivates and explains the Bureaucracy pattern. The role relationship matrix can also be interpreted by using more formal methods, for example by utilizing propositional calculus formulas on the role set as a means of capturing the composition constraints.

The role relationship matrix contains all composition constraints and therefore vital information: Several disjoint subsets of the overall set of roles can be observed the members of which have the same relationships with all other roles. This is depicted by identical rows and columns. For example, Observer, Successor and Parent have the same rows and columns.

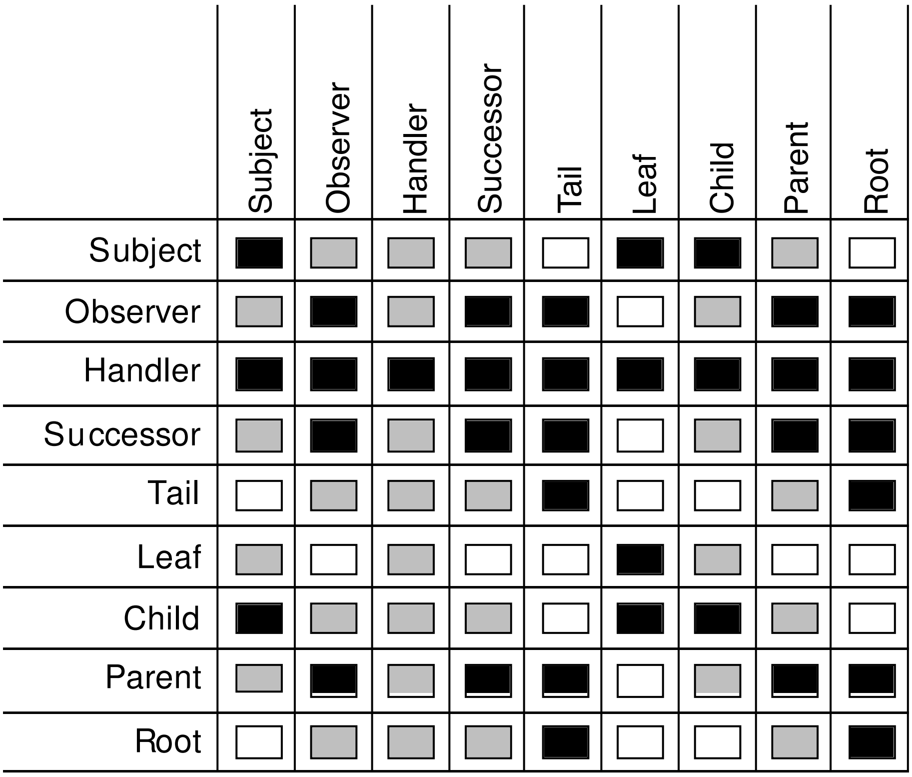



Figure 5.3: Role relationship matrix of the Bureaucracy pattern.

These disjoint subsets effectively provide an equivalence partitioning of the overall role set. Each equivalence set can be assigned to a new composite role. Figure 5.3 shows the equivalence sets Manager (roles Observer, Successor and Parent), Subordinate (roles Subject and Child), Director (roles Root and Tail), Leaf (role Leaf) and Handler (role Handler). From this, the four composite roles Manager, Subordinate, Director and Leaf of figure 5.1 can be derived. Role Handler has been omitted, because it is played by every object as can be seen in figure 5.3.

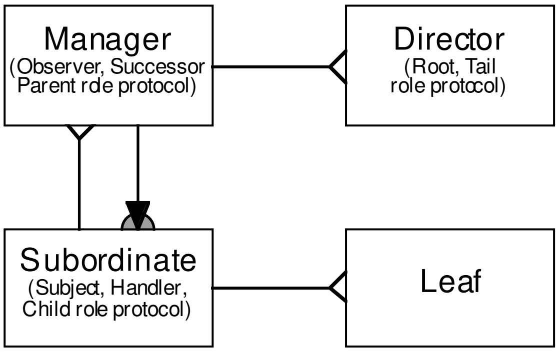



Figure 5.4: Class diagram for the Bureaucracy pattern.

Figure 5.4 shows the most frequently found design template for the Bureaucracy pattern used for its implementation. Note that a case 3 relationship (roles A and B never occur together) has been interpreted as ìdonít care.î This has been used to optimize and simplify the class structure, for example by making the Director class a subclass of Subordinate. A Director object will never play the Subordinate role so this doesn't harm the system.

Again, figure 5.4 is only one of a possible number of design templates. For example, if insisted upon, it might make sense to take the Director class out of the inheritance hierarchy and let it directly inherit from abstract protocol classes for the Observer, Successor, Parent, Root and Tail role protocol.

## 6 Related Work

This section discusses some of the work of Reenskaug et al., Kristensen and Osterbye, and Wieringa et al. In particular, Reenskaug et al. have inspired the work presented in this paper.

Reenskaug et al. present a full-fledged approach to modeling based on roles [Ree96]. The OOram method, formerly known as OOrass [Ree92], uses the notion of role model as its primary means of modeling. A role model is defined as a pattern of interacting roles abstracted from a pattern of interacting objects (Reenskaug's notion of pattern is a different one than used in this paper, see below). A role is defined to describe an object in the context of a role model. Roles and role models are treated as primary means of modeling, and classes are treated as implementation level constructs only. A class simply implements an object.

Next to raising the level of modeling from single roles to role models as patterns of interacting roles, OOram also provides means for composing role models, called role model synthesis. The result of a role model synthesis is a derived model that is composed from smaller role models, the base role models. The derived role model is said to inherit from the base role models. In a role model synthesis, a new role model is derived, in which every role is the composition of one or more roles from the base role models. The key issue is that a role in a derived role model might combine several roles to be treated as a single role. This way, composition constraints are introduced.

In addition to this, OOram provides further notations and techniques to make role modeling work for system design.

As mentioned, Reenskaug uses the notion of pattern in a different way than it is done in the world of (object-oriented software) design patterns and as is done here. Role models are the result of the analysis one (or more) domains, and object patterns are the archetypal object structures of one (or more) domain. The patterns community, however, explicitly introduced “the used twice rule”: A pattern is a pattern if and only if it has been used in more than one application [Vli95]. Thus, a pattern can never be the result of a single analysis, but only of a cross-project analysis.

OOram explicitly avoids introducing a specialization relationship between roles, while this paper relies on them (mainly to address boundary conditions in recursive structures). In addition, this paper has introduced the notion of role relationship matrix to make composition constraints in role diagram compositions explicit. While OOram doesn't lend itself to particular class diagrams for implementing role models, this paper explicitly proposed to make a role diagram denote a family of design templates that can be applied to implement the pattern.

Kristensen and Osterbye introduce roles in the context of conceptual modeling [Kri96]. Conceptual modeling is an approach to software development in which it is considered to be most important that the used notations directly support the human abstraction processes. As a consequence, an explicit notion of role should be provided both in software design and in the chosen programming language.

Kristensen and Osterbye focus on embedding the notion of role as a first class abstraction in design and programming. A role represents the perspective on some object as it emerges from a certain point of view. Thus, all properties offered by an object playing a certain role exist solely due to the client viewing the object according to that role. Roles are specializations of general concepts. Therefore, the usual abstraction processes apply. Firstly, objects can be classified that is objects can be seen as role instances. Secondly, roles can be aggregated, which combines several roles to form a new composed role. Finally, roles can be specialized or generalized, that is roles can be classified with respect to each other into hierarchies.

With roles as primary means of modeling, several issues arise. How are roles attached to objects? How is their state managed if it only exists in the context of a client viewing the object as a player of a certain role? How are state transitions managed within roles and when crossing role boundaries? How is identity preserved and managed? Kristensen and Osterbye report on an extension to BETA [Mad93] which addresses some of the questions. These issues have also been taken up in the work on “Subject-Oriented Programming,” see for example Harrison and Ossher [Har93].

The standard implementation for modeling with roles in the context of an object-oriented design notation which does not offer them as first class concepts is to use the Decorator pattern as described, for example, in [Gam95]. Riehle and Bäumer discuss some of the arising issues in the context of the Tools and Materials Metaphor [Rie95a]. In particular they provide automated mapping schemes for finding the right role/object adaptation for a given context.

Wieringa et al. review roles with a focus on object migration and identification [Wie95]. Role classes are classes the instances of which cannot exist on their own but always belong to another object. These role instances adapt the object to a certain context. This other object can be a regular object or a role instance. However, each chain of role instances has to end with a regular object that serves as the chain's anchor and its final player.

Wieringa et al. model the roles that objects of a certain class can play as subclasses of that class. This ensures that role classes always support the interface of the class for which it is defined as a role. Furthermore, the set of all role classes of a class is partitioned into sets of related role classes. An object can only migrate between roles within that restricted set. When doing so, it should be noted that the object maintains its original identifier and thus doesn't change identity. Wieringa et al. further discuss life cycle issues and state models of roles, and discuss the differences between static, dynamic and role classes.

## 7 Conclusions

This paper has presented role diagrams as a means of expressing the relationships of objects in a design pattern. Role diagrams allow designers to abstract from class diagrams which are found to be too implementation oriented. Role diagrams allow for a family of class templates that can be used to efficiently implement design patterns.

Furthermore, this paper has taken role diagrams as a starting point for composing patterns. The notion of composition constraint has been defined, and a simple technique for its notation, the role relationship matrix, has been presented. A distinction between arbitrary pattern compositions and composite patterns has been made.

Open questions and directions for future research abound. Are there certain kinds of patterns which always go together for certain problem situations, for example, a creational with a structural pattern, but can otherwise be varied freely? Are all patterns equally suited for pattern compositions or does their granularity influence their composability? For example, the fine-grained patterns from Pree [Pre95] might serve as glue patterns for the more coarse-grained patterns from Gamma et al. [Gam95].

This paper has brought up two new topics in the fast moving field of patterns and pattern techniques, role diagrams and pattern compositions. New or enhanced techniques might lead to revisions of the presented examples. However, pattern compositions hold the promise of more adequately specifying, documenting, motivating, constructing and discussing large designs like frameworks, and this paper has taken first steps on this new and highly interesting ground.

## Bibliography

[Cop95] Coplien, J., and Schmidt, D. (eds). *Pattern Languages of Program Design.* Reading, Massachusetts: Addison-Wesley, 1995.

[Dah72] Dahl, O.-J., and Hoare, C. A. R. “Hierarchical Program Structures.” *Structured Programming.* Edited by Ole-Johan Dahl, Edsger W. Dijkstra and C. A. R. Hoare. Academic Press, 1972.

[Gam95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Design.* Reading, Massachusetts: Addison-Wesley, 1995.

[Gol80] Goldberg, A., and Robson, D. *Smalltalk-80: The Language.* Reading, Massachusetts: Addison-Wesley, 1989.

[Har93] Harrison, W. and Ossher, H. “Subject-Oriented Programming (A Critique of Pure Objects).” OOPSLA ’93, *ACM SIGPLAN Notices* 28, 10 (October 1993): 411-428.

[Hel90] Helm, R., Holland, I., and Gangopadhyay, D. “Contracts: Specifying Behavioral Compositions in Object-Oriented Systems.” OOPSLA ’90, *SIGPLAN Notices* 25, 10 (October 1990): 169-180.

[Kra88] Krasner, G. and Pope, S. “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.

[Kri96] Kristensen, B., and Osterbye, K. “Roles: Conceptual Abstraction Theory and Practical Language Issues.” *Theory and Practice of Object Systems.* To appear.

[Mad93] Madsen, O., Möller-Pedersen, B., and Nygaard, K. *Object-Oriented Programming in the BETA Programming Language.* Reading, MA: Addison-Wesley, 1993.

[Pre95] Pree, W. *Design Patterns for Object-Oriented Software Development.* Reading, MA: Addison-Wesley, 1995.

[Ree92] Reenskaug, T., Andersen, E., Berre, A., Hurlen, A., Landmark, A., Lehne, O., Nordhagen, E., Nêss-Ulseth, E., Oftedal, G., Skaar, A., and Stenslet, P. “OORASS: seamless support for the creation and maintenance of object-oriented systems.” *Journal of Object-Oriented Programming* 5, 6 (October 1992): 27-41.

[Ree96] Reenskaug, T., with Wold, P., and Lehne, O. *Working with Objects.* Greenwich, Manning, 1996.

[Rie95a] Riehle, D. and Bäumer, D. “Subjectivity in Object-Oriented Systems.” Position Paper for OOPSLA ’95, *Workshop on Subjectivity in Object-Oriented Systems.* Available from the author.

[Rie95b] Riehle, D., and Züllighoven, H. “A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor.” In [Cop95]. 9-42.

[Rie96a] Riehle, D. “The Event Notification Pattern--Integrating Implicit Invocation with Object-Orientation.” *Theory and Practice of Object Systems* 2, 1 (1996). To appear.

[Rie96b] Riehle, D. “Bureaucracy-A Composite Pattern.” Accepted forEuroPLoP ’96.

[Rum91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-Oriented Modeling and Design.* London: Prentice-Hall, 1991.

[Vli90] Vlissides, J., and Linton, M. “Unidraw: A Framework for Building Domain-Specific Graphical Editors.” *ACM Transactions on Information Systems* 8, 3 (July 1990): 237-268.

[Vli95] Vlissides, J. “Reverse Architecture.” Position Paper *Dagstuhl Seminar 9508,* 1995. 7 pages.

[Wei94] Weinand, A., and Gamma, E. “ET++-A Portable, Homogenous Class Library and Application Framework.” *Computer Science Research at UBILAB.* Edited by Walter R. Bischofberger and Hans-Peter Frei. Konstanz: Universitätsverlag Konstanz, 1994. 66-92.

[Wie94] Wieringa, R., de Jonge, W., and Spruit, P. “Using Dynamic Classes and Role Classes to Model Object Migration.” *Theory and Practice of Object-Oriented Systems* 1, 1: 61-84.

[Wir90] Wirfs-Brock, R., Wilkerson. B., and Wiener, L. *Designing Object-Oriented Software.* Englewood Cliffs, New Jersey: Prentice-Hall, 1990.

[Zim95] Zimmer, W. “Relationships Between Design Patterns.” In [Cop95]. 345-364.