Composite Design Patterns

Dirk Riehle

Ubilab, Union Bank of Switzerland, Bahnhofstrasse 45, CH-8021 Zurich
Phone: +41-1-234-2702, fax: +41-1-236-4671
E-mail: Dirk.Riehle@ubs.com or riehle@acm.org
WWW: http://www.ubs.com/ubilab + staff/riehle

Abstract

Software design patterns are the core abstractions from successful recurring problem solutions in software design. *Composite design patterns* are the core abstractions from successful recurring frameworks. A composite design pattern is a pattern that is best described as the composition of further patterns the integration of which shows a synergy that makes the composition more than just the sum of its parts. This paper presents examples of composite patterns, discusses a role-based analysis and composition technique, and demonstrates that composite patterns extend the pattern idea from single problem solutions to object-oriented frameworks.

1 Introduction

A developer versed in software design patterns might explain the Model-View-Controller paradigm (MVC, [KP88, Ree96a]) for designing interactive software systems like this: "(a) MVC helps you design applications with graphical user interfaces. Its core are three collaborating objects: a Model object which represents an instance of a domain concept, a View object which realizes a specific user interface representation of the Model, and a Controller object which handles user input in order to work with the Model. (b) These objects interact according to the Observer, Strategy and Composite pattern: A View observes a Model—thus the View is an Observer of the Model which is its Subject. A View does not handle user input but leaves this to the Controller—thus the Controller is a Strategy for handling user input. Moreover, Views can have Subviews which represent smaller parts of the user interface and which can have Subviews themselves—thus the View is a Component in the Composite pattern and different Views can be either Leafs or Composites."

From MVC's overall point of view (described in (a) above), the use of each of the design patterns (described in (b) above) is of a tactical nature: every pattern is used to solve a specific problem at hand. Taken together and directed towards the goal of designing a reusable and flexible user interface architecture, the patterns achieve a synergy which constitutes a whole that is larger than just the sum of some patterns. *MVC is a composite design pattern.*

A composite pattern is first of all a pattern: It is a design theme that keeps recurring in specific contexts as a solution to a problem. It is a composite pattern, because it can best be explained as the composition of some other patterns. However, a composite pattern goes beyond a mere composition: It captures the synergy which arises from the integration of several patterns into an overall composition structure. A composition of some patterns turns into a composite pattern, if and only if (a) a relevant synergy between the pattern interactions arises, and (b) this synergy can be observed as a recurring design theme.

This paper discusses the notion of composite pattern. It introduces a notation based on roles to better describe and compose patterns. It presents an analysis and composition technique to better cope with the complexity of composite patterns. The notation and technique cover patterns based on object collaborations which constitute the majority of patterns known today. Eventually, the paper compares composite patterns with object-oriented frameworks and pattern languages.

Composite patterns and frameworks are orthogonal concepts. However, analyzing and understanding successful recurring frameworks as composite patterns helps us better capture the core abstraction behind such frameworks, much like atomic design patterns help us capture the solution to a recurring design problem. Not every composite pattern constitutes a framework, but behind every successful recurring framework there must be a core abstraction that can be captured as a pattern, usually a composite pattern. As such, composite patterns hold the same promise as atomic patterns, but on a much larger scale.

Section 2 examines the notion of composite pattern more closely, presents a first example, and lists further ones. Section 3 introduces role diagrams and composition constraints as a means for describing patterns. This prepares the way to section 4, which presents an elaborate example of a composite pattern. Doing so, it introduces an analysis and composition technique based on the concepts of prototypical pattern application and role relationship matrix. In section 5, the definition of composite pattern is compared with frameworks and pattern languages. Section 6 discusses related work and section 7 presents some final conclusions.

2 Composite Patterns

This section examines and defines the notion of composite design pattern and sets up a proper terminology. An example further illustrates the idea of composite pattern. This prepares the way to the analysis and composition technique of the following sections.

Published in *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '97). ACM Press, 1997. Page 218-228.

2.1 Definition

A *composite design pattern* is first of all a design pattern: It is the abstraction from a concrete recurring solution that solves a problem in a certain context [GOF95, POSA96, RZ96]. A pattern is a *composite* pattern, if it can be best explained as the composition of further atomic or composite patterns. An atomic pattern is a pattern that, on a given level of abstraction, cannot sensibly be described as the composition of further patterns.

In a composite pattern, the constituting patterns integrate with each other to achieve a synergy that gives the composite pattern its own identity beyond being just the sum of some patterns. This distinguishes a composite pattern from an arbitrary pattern composition, which may be a suitable solution for a specific design problem, but which does not recur as a pattern of its own.

Usually, a pattern can be described as the abstraction from a recurring form that consists of several elements which interact with each other and their context in specific ways [RZ96]. A composite pattern's synergy emerges from the interaction of different elements from different patterns. This interaction can take on many shapes, depending on the kind of pattern. Generally speaking, each pattern element interacts with further elements from other patterns. If these non-trivial element interactions keep recurring, they constitute the synergy that motivates the composite pattern.

2.2 Active Bridge

As a first example, consider the Active Bridge pattern [Rie97a]. This pattern represents a recurring type of frameworks which is used to connect an application to underlying event-driven resources like window-system widgets or inter-process communication channels. This pattern has been used in ET++ [WG95], Geo (our current project), and the newly redesigned VisualWorks window-system framework presented at OOPSLA '96 [Yel96].

Figure 1 presents the role diagram of the pattern. It shows the five object roles Application, Abstraction, Implementor, Resource and Factory. A role defines a view on an object within a given object collaboration. The use of roles will be discussed in more detail in the next section. For the Active Bridge pattern it suffices to know that each of the aforementioned roles maps directly on a single object in a pattern instantiation.

The Active Bridge pattern can best be understood as the composition of the five patterns Bridge, Proxy, Observer, Abstract Factory and Factory Method, all of which are described in [GOF95].

- At the heart of the Active Bridge pattern is the *Bridge* pattern which serves to decouple an Abstraction from a number of different Implementors. For instance, a Window abstraction can be implemented based on a lower-level Implementor interface.
- Each Implementor acts as a *Proxy* for an underlying resource like a window-system native window widget [Yel96]. The Proxy is implemented based on the resource and acts as an object-oriented placeholder for it.
- Communication flows in both directions. An application directly uses an Abstraction which uses an Implementor which uses a Resource. A Resource may cause events due to user interactions or incoming communication requests. It forwards them to the Implementor. The Implementor informs the Abstraction using the *Observer* pattern, thus becoming the Subject of the Abstraction which is its Observer.
- The configuration of such a subsystem requires some thought and is therefore delegated to an *Abstract Factory*. The Factory creates the Implementor instances and thereby ensures that the chosen implementations can coexist, meet the applications' needs, and work with the available resources.
- The creation operations of the Abstract Factory are implemented using *Factory Methods*. Factory Methods are the best choice because the Factory is chosen once at startup time and there is no need to reconfigure it.

Each of the five patterns defines a set of roles for its elements. Every object in an instance of the Active Bridge pattern plays several roles from the constituting patterns. Figure 2 groups the roles from the atomic patterns into composite roles. Composite

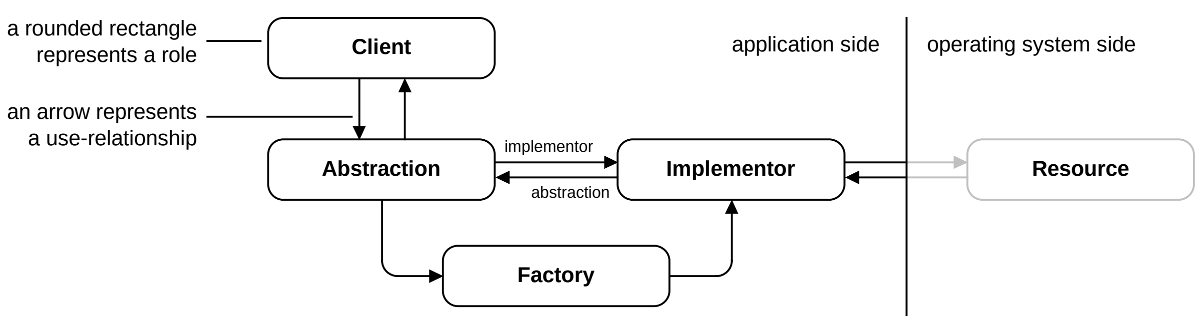

Figure 1: Role diagram of the Active Bridge pattern

$$
\begin{aligned}
\mathtt{Client}_{AB} &= \{\ \mathtt{Client}_{B}\ \} \\
\mathtt{Abstraction}_{AB} &= \{\ \mathtt{Abstraction}_{B},\ \mathtt{Client}_{P},\ \mathtt{Observer}_{O},\ \mathtt{Client}_{AF}\ \} \\
\mathtt{Implementor}_{AB} &= \{\ \mathtt{Implementor}_{B},\ \mathtt{Proxy}_{P},\ \mathtt{Subject}_{O},\ \mathtt{Product}_{AF},\ \mathtt{Product}_{FM}\ \} \\
\mathtt{Factory}_{AB} &= \{\ \mathtt{Factory}_{AF},\ \mathtt{Client}_{FM},\ \mathtt{Creator}_{FM}\ \} \\
\mathtt{Resource}_{AB} &= \{\ \mathtt{Subject}_{P}\ \}
\end{aligned}
$$

Figure 2: Composite roles of the Active Bridge pattern

roles are the roles of the composite pattern; they are a composition of some atomic roles. The index of a role name in figure 2 indicates the pattern in which it is defined. AB stands for Active Bridge, B for Bridge, P for Proxy, O for Observer, AF for Abstract Factory, and FM for Factory Method.

Active Bridge is a fairly simple composite pattern, because the patterns it consists of are fairly simple. The constituting patterns' roles can be grouped easily to form the composite roles.

It is easy now to prove that Active Bridge is a composite pattern according to the definition of section 2.1: Active Bridge can be observed to recur; it is described in [WG95, Yel96] and has been used in many other frameworks. As has just been demonstrated, it can be explained well as a composition of some patterns. Finally, the composite roles defined in figure 2 are a visible expression of the synergy achieved by the composite pattern. This synergy stems from integrating different roles from different constituting patterns in a composite role played by a single object at runtime.

2.3 Further examples

The Active Bridge pattern is described in some more depth in [Rie97a], where further composite patterns can be found (Bureaucracy, Model-View-Controller and Role Object). The Bureaucracy pattern will be discussed in section 4. The Role pattern serves to adapt a core abstraction to different contexts by means of role objects, as it is ubiquitously needed in many complex domains. Further composite patterns can be abstracted from recurring frameworks for such topics as product specification, resource allocation or order management.

Another concrete example of a such a framework-level composite pattern is the user-defined product specification pattern, the abstraction from product specification frameworks. A product specification framework lets users dynamically define banking or insurance products (like loans, financial instruments, or insurance contracts). Because the number of different products is large, constantly changes and must be adapted (to some extent) for every customer, a naive approach like modeling each product as a class of its own is doomed to fail.

Instead, product specification frameworks repeatedly compose Type Object [JW97], Composite [GOF95], and Property [Rie97a] to define the structure of self-describing products. The dynamics of the resulting object structure can be described best using the Interpreter pattern (viewing products as attribute grammars). While the composite pattern behind this framework type has not yet been worked out in detail, an excellent pattern-based description of it is presented by Johnson and Oakes in [JO97].

Framework-level composite patterns hold a high promise of explaining recurring frameworks in such a way that experience can be more easily communicated and transferred from one domain to related domains and reused across programming language boundaries. A high leverage can be expected from uncovering, describing and communicating these composite business patterns.

3 Pattern Description

If all composite patterns were as simple as the Active Bridge pattern, this paper would probably end here. However, many composite patterns are much more complex than the introductory pattern. To cope with this complexity, we need proper concepts, techniques and tools to support human intuition and experience while defining a pattern.

This section presents a new notation for describing patterns, the role diagram notation. Role diagrams are based on Reenskaug's role models [Ree96a], but extend them with the notion of composition constraint [Rie96, Rie97a]. The next section discusses an analysis and composition technique for composite patterns based on role diagrams.

Choosing role diagrams as the primary means for describing patterns ignores class inheritance based patterns for which further techniques have to be developed. This is justified, because role diagrams can be used more effectively than class diagrams when dealing with patterns based on object collaborations.

3.1 Role diagrams

Almost all pattern descriptions today use class diagrams to describe patterns [CS95, VCK96, MRB97]. The reason is that classes have long been the primary means for modeling object-oriented software systems. In addition, they are explicitly represented in today's object-oriented programming languages. However, class diagrams often mix the actual problem solution in terms of the distribution of responsibilities between objects with efficient ways of implementing it. Role diagrams in turn are better suited for describing object collaboration based patterns than class diagrams, because they better focus on the actual problem solution as a set of collaborating objects.

A *role diagram* describes how some collaborating objects, each one playing one or more *roles*, achieve a common goal. A *role* represents the view some objects in the collaboration hold on another object playing that role [Rie96, KO96]; it captures the responsibilities of an object with respect to achieving the purpose defined by the role diagram. An object can play several roles at once, and the same role can be played by several objects. An object collaboration usually serves several purposes—it can be viewed as a set of overlapping role diagrams. Thus, role diagrams can be composed easily, which makes them attractive for describing composite patterns.

Figure 3 shows the role diagram of the Composite pattern. The Composite pattern is defined in [GOF95]; it serves to describe and implement hierarchical object structures (trees). The notion of "composite patterns" as discussed in this paper does not have anything to do with "the Composite pattern" which is written with a capital "C." Composite patterns define a category of patterns, while the Composite pattern is a concrete pattern.

The Composite pattern as understood here defines the roles Node, NodeClient, Parent, Child, Root and RootClient. This is a reinterpretation of the Gang-of-four definition, extended with the notion of root objects. Figure 3 shows three pair-wise interactions: a NodeClient makes use of a Node, a Parent has several Child objects, and a RootClient makes use of the Root object. These interactions are linked together by *composition constraints:* An object playing the Root role also always plays the Parent role which also always plays the Node role. On an implementation level, this type of composition constraints frequently maps well on class inheritance.

Role diagrams are more abstract than class diagrams. Role diagrams can be mapped on several class diagrams. The use of class inheritance often introduces important implementation twists that help handle the instantiated pattern more easily. [Rie96] discusses the resulting levels of abstraction, and shows how the general role diagram of the Observer pattern depicted in figure 6 maps on at least three different class diagrams that make it more concrete.

3.2 Role relationships

Figure 3 does not fully show how the roles can be mapped on objects. The mapping of roles on objects may be subject to composition constraints. A *composition constraint* is a binary relationship between roles which may take on one of three different values: Two roles *may be played* by the same object, or they *must be played* by the same object, or they *must not be played* by the same object.

The set of composition constraints can be expressed as a role relationship matrix which relates every role with every other role. Figure 4 shows the role relationship matrix of the Composite pattern. It uses the three aforementioned different values for a matrix entry (A, B) which defines a relationship between the two roles. These three values are:

- An object playing role A also always plays role B in the same collaboration. Thus role A implies role B. This is depicted by a black rectangle for the matrix entry (A, B).
- An object playing role A never plays role B in the same collaboration. Thus, role A prohibits role B. This is depicted by a white rectangle for the matrix entry (A, B).
- Two roles A and B arbitrarily mix and match ("don't care"). Thus, role A may or may not go with Role B; nothing can be said. This is depicted by a gray rectangle for the matrix entry (A, B).

In the Composite pattern, two examples of "A implies B" are the pairs (Parent, Node) and (Child, Node) which specify that an object playing the Parent or Child role also always plays the Node role. The relationship is not symmetric: One cannot conclude that an object playing the Node role also plays the Parent or Child role. An example of "A prohibits B" is the pair (Child, Root) which specifies that a Child object never plays the Root role.

The role relationship matrix depicted in figure 4 is a visual representation of a propositional calculus formula. Part of the description of an object's behavior within a collaboration is the set of roles it may play, and the constraints which define valid sets can be described well and in simple terms using propositional logic.

The meaning of the matrix entries is as follows: The "implies" relationship between two roles corresponds to a logical implication between two roles "($A \Rightarrow B$)," the "prohibits" relationship between two roles corresponds to "$\neg(A \wedge B)$" (due to its symmetry) and the "don't care" relationship corresponds to "true." The

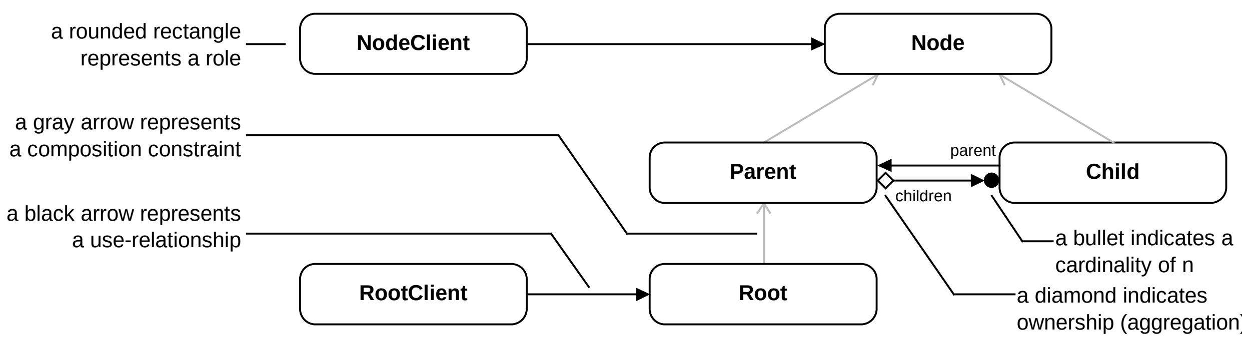

Figure 3: Role diagram of the Composite pattern

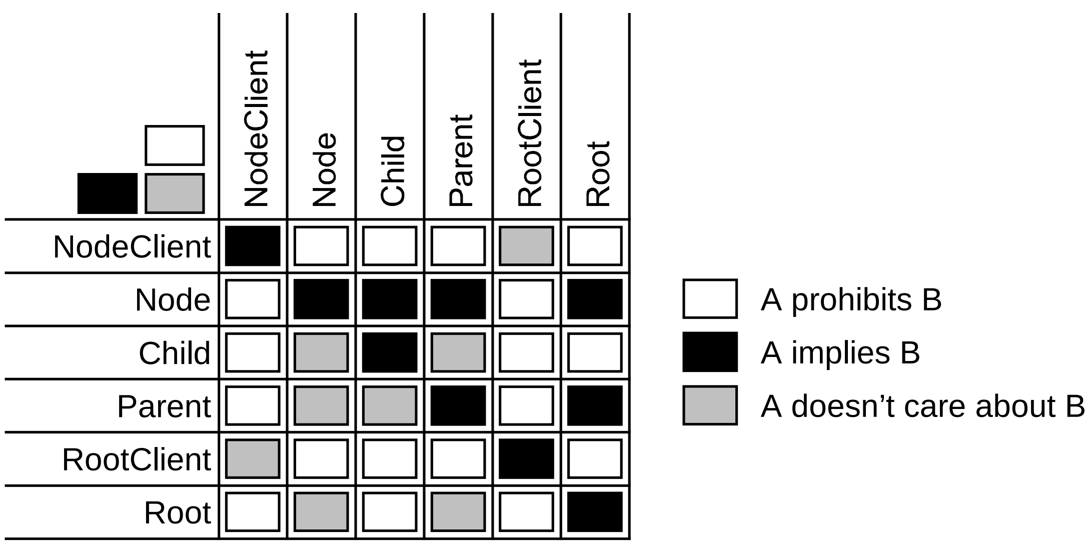

Figure 4: Role relationship matrix of the Composite pattern

overall matrix represents a conjunction of its entries. An object's role set is valid within a collaboration if the binding of the variables makes the formula evaluate to true.

3.3 Further examples

Figures 5 to 7 show three further patterns needed for this paper.

- The *Mediator pattern* (figure 5) serves to decouple, manage and integrate several Colleague objects by means of a coordinating Mediator.
- The *Observer pattern* (figure 6) serves to decouple Observer objects from a Subject object while maintaining state dependencies. The maintenance is achieved by using events for inter-object communication.
- The *Chain of Responsibility pattern* (figure 7) serves to define an object chain along which requests are passed until they are handled. Thus, by configuring the chain, the receiver of a request can be defined dynamically.

These examples are taken from [Rie97a] where more atomic and composite patterns are described using role diagrams. Some of these descriptions are simpler than the original from [GOF95], and some are more complex. In particular, descriptions of patterns with a recursive structure like Composite or Chain of Responsibility are more complex, because satisfying the boundary conditions increases the number of roles and composition constraints.

4 Analysis and Composition

The previous section has introduced role diagrams as a means of describing patterns. This section shows how complex object structures can be analyzed and how composite patterns can be derived and defined. The key concepts are the set of prototypical pattern applications and the role relationship matrix. I use the Bureaucracy pattern [Rie97b] as an example of a complex composite pattern.

4.1 Prototypical pattern application

Patterns grow from experience. Thus, every effort to devise a new pattern should be based on previously known pattern instantiations, that is concrete designs in which the patterns have been applied (albeit implicitly). The documentation of existing systems is a good starting point. However, concrete designs and implementations often vary greatly, even if a common core indicates a potential composite pattern.

In a Bureaucracy pattern instantiation, the objects form a hierarchy, with each parent-node object being a manager to its child-node objects. This management and integration task conforms to the Mediator pattern. Child objects, like subordinates in a bureaucratic hierarchy, inform their parent about state changes, for example when they have finished a task they had to fulfill. This is done according to the Observer pattern. Moreover, if an external client requests a task from an object in the hierarchy which exceeds this object's context information, the object forwards the request up the hierarchy until it can be satisfied by a node that has enough context information to do so. This interaction is handled according to the Chain of Responsibility pattern. Finally, the hierarchical structure itself is defined using the Composite pattern. All four patterns, Composite, Mediator, Observer and Chain of Responsibility can be found in [GOF95].

A first step to abstract from concrete designs is to devise a *set of prototypical pattern applications,* that is a set of object collaboration structures in which all relevant roles and role interactions are present. These applications can then serve as the primary object of

Figure 5: Role diagram of the Mediator pattern

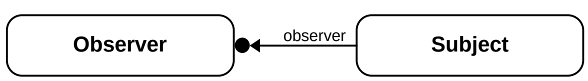

Figure 6: Role diagram of the Observer pattern

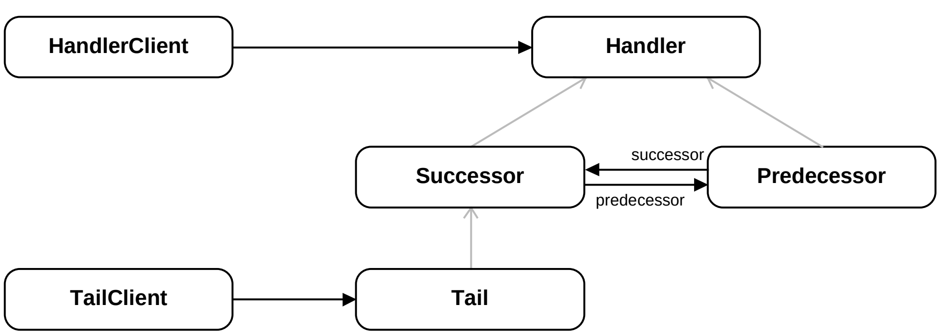

Figure 7: Role diagram of the Chain of Responsibility pattern

study. Frequently, a single prototypical pattern application is sufficient to capture all relevant configurations. If not, further applications must be devised until all relevant configurations are covered. This forms the set of prototypical pattern applications.

Figure 8 shows a prototypical pattern application of the Bureaucracy pattern, as it can be found in graphical editors like HotDraw [Joh92], ET++Draw [WG95], and Sane [RZ95]. It shows a hierarchy of visual objects which represent the elements of a drawing. They are displayed in an editor window, ready for being manipulated by the user. The visual objects communicate with each other in order to keep the hierarchy in a consistent state as well as to perform user initiated actions. This section's analysis of the collaboration structure will show that its driving force is a recurring design theme which can be captured as the Bureaucracy pattern.

4.2 Involved patterns

An analysis of the prototypical pattern application identifies four patterns: Composite, Mediator, Chain of Responsibility, and Observer.

- The *Composite pattern* is used to define the hierarchy. Every visual object in the hierarchy plays the role of Node, some play the role of Parent (aFigure, aGroup), some play the role of Child (aCircle, aGroup, aRectangle, anArrow, aTriangle), and one plays the role of Root (aFigure). A user plays the role of NodeClient and anApplication plays the role of RootClient.
- The *Mediator pattern* is used to let a Parent in the hierarchy manage its Child objects. The roles of Mediator in the Mediator pattern maps on Parent and the role of Colleague maps on Child in this pattern. For example, aGroup plays the role of a Mediator for aRectangle, anArrow, and aTriangle which it coordinates to behave like a group.
- The *Chain of Responsibility* pattern is used to handle client requests. Every Node is a Handler which can receive client requests. Child objects are Predecessors which forward requests up the hierarchy if they cannot handle them. If aCircle is manipulated, for example moved, it sends an invalidate drawing region request to its Successor. Thus, aCircle is both a Handler and a Predecessor, and aFigure is both aNode and a Successor as well as a Tail for the Chain.
- The *Observer pattern* is used to keep up with changes that are not explicitly forwarded up the hierarchy. Every Parent in the hierarchy is an Observer of its Child objects which are its Subjects. If aCircle is not constrained by its Parent aFigure to ask first before letting a user change its label, it can do so on its own. Nevertheless, further objects, either inside or outside the hierarchy, might have to change accordingly, so aCircle informs its Observers about the change.

In short, the Composite pattern defines the hierarchical structure, the Mediator pattern shows how each hierarchy node manages its subordinate nodes, the Chain of Responsibility pattern shows how client requests are forwarded up the hierarchy, and the Observer pattern shows how nodes observe their subordinate nodes in order to readjust the hierarchy in case of unanticipated changes.

Working together, these patterns achieve a synergy that goes beyond their individual purposes: Their integration helps to design hierarchical structures which can maintain their inner stability (invariants) themselves while still allowing clients to interact with every hierarchy level. Client requests may cause a complex control flow inside the hierarchy which it uses to readjust itself.

4.3 Role relationship matrix

Figure 9 shows how the different roles of the constituting patterns are assigned to objects. Every role was taken and assigned to those objects which play that role. This defines the set of all roles an object may play in a collaboration.

Now the role relationship matrix can be derived. As defined, a role relationship matrix specifies how the roles objects play in an

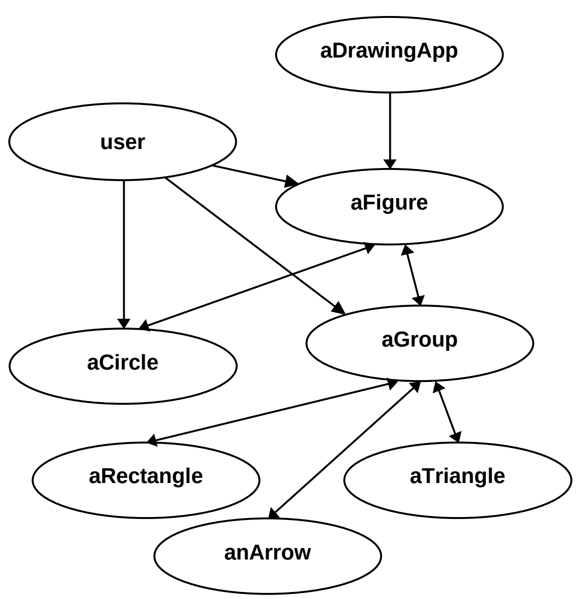

Figure 8: A prototypical pattern application of the Bureaucracy pattern

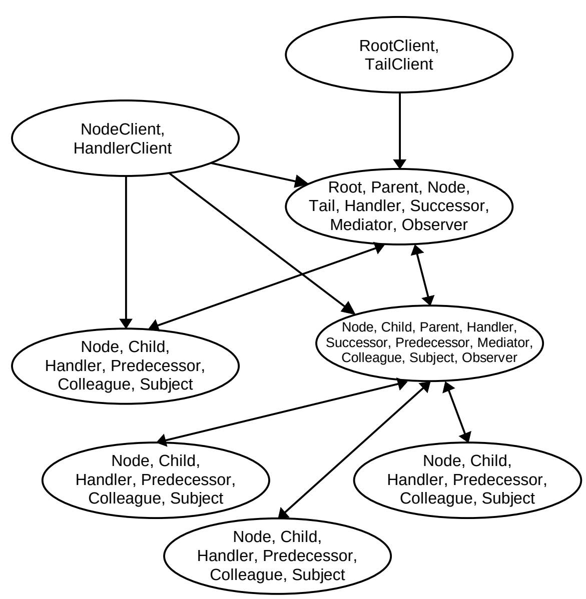

Figure 9: The roles objects in the prototypical pattern application play

object collaboration relate to each other. Its purpose here is to describe the pattern interaction synergies and help uncover hidden composite roles. Figure 10 shows the preliminary non-consolidated role relationship matrix of the Bureaucracy pattern, as derived from figure 9.

If it were not for the "implies" and "prohibits" relationship between roles, there would be no composite patterns. If it were possible to arbitrarily map roles on objects, anything would be possible but nothing could be said about pattern interaction synergies. However, it is exactly the set of *composition constraints,* which represents the synergy the composition achieves and which turns it from an arbitrary composition into a composite pattern.

The role relationship matrix is represented visually rather than as a large formula, because the visual presentation is more accessible to human perception and lets us much easier recognize composite roles and interaction synergies, as discussed now.

4.4 Pattern derivation

Analysis of the role relationship matrix reveals that several columns (and rows, the matrix is symmetric with respect to this) are equivalent. Thus, grouping equivalent columns lets us partition the overall role set into equivalence sets, each one representing a composite role. Why? Because if an object plays a role it always also plays any other role from its equivalence set so that in an instantiation of the composite pattern the roles are always played together. Thus, they constitute the pattern's composite roles.

Figure 11 shows the composite roles of the Bureaucracy pattern. It is based on the roles DirectorClient, Director, Manager, Subordinate, ClerkClient and Clerk, each of which represents an equivalence set of roles from the composed patterns. The index B stands for Bureaucracy, C for Composite, M for Mediator, CoR for Chain of Responsibility, and O for Observer.

Figure 12 shows a consolidated role relationship matrix which is solely based on composite roles. The redundancy of the equivalence sets has been eliminated. This leaves us with a matrix in which only the composition constraints between the composite roles are present (for example, every Director is a Manager, which in turns is always a Clerk—in accordance with the textbook definition of bureaucracy [Web47]).

Figure 13 shows the role diagram of the Bureaucracy pattern, as it conforms to the role relationship matrix of figure 12. The structure is isomorphic to the Composite pattern's structure of figure 3 and 4, but the dynamics are more elaborate. This role diagram is the result of this section's work.

This section has presented an analysis, composition and derivation technique for composite patterns based on object collaborations. The set of prototypical pattern applications serves to abstract from concrete designs. The role relationship matrix serves to analyze the interaction of different roles from the involved

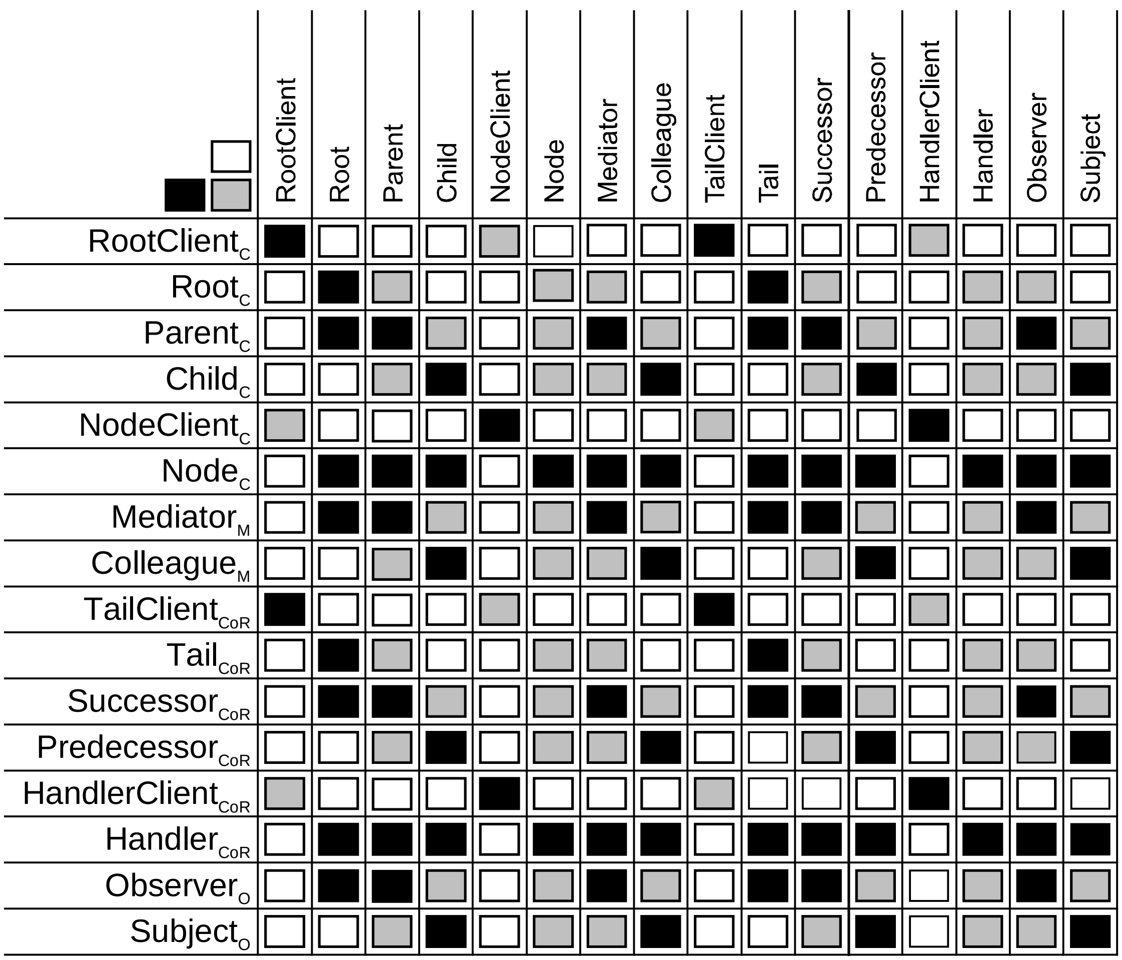

Figure 10: Role relationship matrix of the Bureaucracy pattern before consolidation

patterns. Starting with the role relationship matrix, role equivalent sets are defined which stand for composite roles. Based on composite roles, the preliminary role relationship matrix can be consolidated and the final role diagram can be defined.

The process demonstrated in this section is an after-the-fact rationalization [PC86]. The creative process of working out the pattern did not proceed in the linear fashion as implied by the steps taken in this section.

5 Comparison

This section discusses the relationship between composite patterns, frameworks and pattern languages. It shows that recurring frameworks can be abstracted into patterns and demonstrates that composite patterns are different from pattern languages.

5.1 Frameworks

A framework is a set of classes which model and solve a specific domain problem. Usually, this set of classes contains some abstract classes which define the design of the framework and the interaction of their instances, and some concrete classes which provide implementations for the abstract classes [JF88, GOF95, Lew95]. As already pointed out in [GOF95], patterns are abstractions from concrete designs and therefore are to be seen on a different level. In this paper I have claimed and illustrated that composite design patterns are the abstractions from concrete recurring frameworks. This point will now be clarified further.

Both patterns and frameworks can be described using class or role diagrams [Rie96, Rie97a, Ree96a]. Both frameworks and pattern instances can be understood well as solving a particular problem. A framework, which keeps recurring and which solves a specific problem, can be abstracted into a pattern. Thus, the abstraction

```
DirectorClient_B = { RootClient_C, TailClient_CoR }
Director_B       = { Root_C, Tail_CoR }
Manager_B        = { Parent_C, Mediator_M, Successor_CoR, Observer_O }
Subordinate_B    = { Child_C, Colleague_M, Predecessor_CoR, Subject_O }
ClerkClient_B    = { NodeClient_C, HandlerClient_CoR }
Clerk_B          = { Node_C, Handler_CoR }
```

Figure 11: Definition of the composite roles of the Bureaucracy pattern

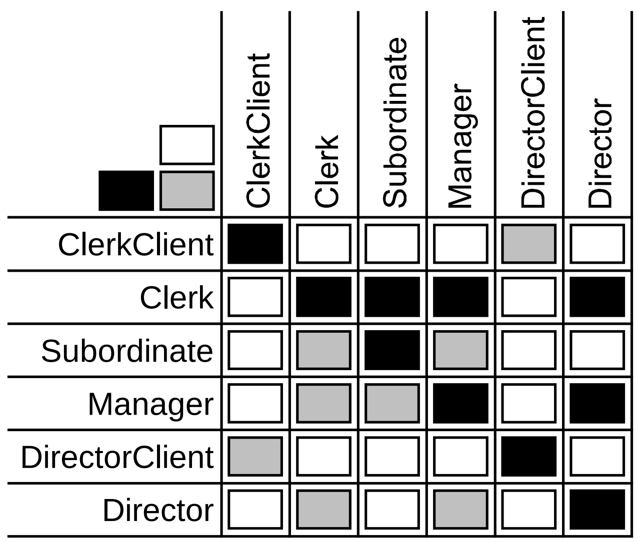

Figure 12: Role relationship matrix of the Bureaucracy pattern

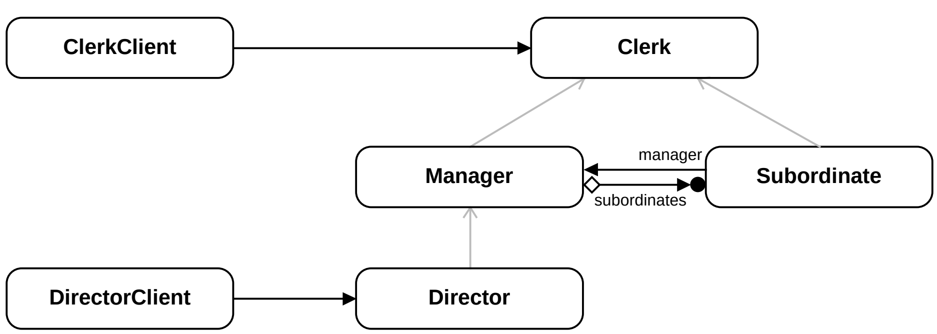

Figure 13: Role diagram of the Bureaucracy pattern

from a set of similar recurring frameworks is a pattern, and the concrete frameworks are its instances.

A framework, in which several patterns have been applied and in which these pattern instances interact in always the same way, can be abstracted into a composite pattern. The definition of composite pattern is pragmatic: Describing a composite pattern as the composition of several patterns helps to explain it better and makes clear that the way the pattern instances interact in a given framework keeps recurring as a pattern of its own.

5.2 Pattern languages

A pattern language is a collection of patterns which refer to each other in such a way that users can use the patterns to build software systems in a similar way as they use natural language to create sentences, paragraphs and books [Ale79, Cop97]. The power of a pattern language emerges from being "generative" in the sense that it helps users with going back and forth between patterns, describing the pattern relationships and interactions. Alexander has given up pattern languages [Gab94] because it did not meet his hopes and expectations, but the software and organizational patterns community discusses them as a possible means for effectively supporting software development activities.

A pattern language offers more freedom of choice than a composite pattern. When using a pattern language, users actually traverse paths in the language's pattern graph, each time choosing a path specific to the overall problem that is to be solved. With changing problems, different paths are chosen, and different overall solutions emerge. A composite pattern in turn has already defined the one single solution space for a problem as a set of patterns that interact in always the same recurring way. Thus, the implementation space of a composite pattern is more restricted than the one of a pattern language.

From using a pattern language, composite patterns might emerge. If a user of a pattern language notices that in similar problem contexts he or she is actually traversing and applying a pattern language using the same path over and over again, then there must be some hidden theme, a recurring hidden agenda behind that traversal path. This might be the beginning of the uncovering of a new composite pattern.

6 Related Work

First of all, the work of Reenskaug on the OOram software development methodology has to be discussed [Ree96a]. The OOram methodology defines the concepts *role model* and *role model synthesis* which are similar to concepts presented in this paper. A role model serves the same purpose as a role diagram: it describes an archetypal object collaboration in terms of the roles objects play in the collaboration. Role model synthesis is the process of composing several role models to yield a synthesized role model. Thus, role model synthesis is equivalent to composing role diagrams and the notion of synthesized role corresponds to the notion of composite role as used in this paper.

However, there are several aspects in which the work presented in this paper is different or goes beyond his work. On the concrete role modeling level, it introduces the notions of composition constraint and role relationship matrix to address issues of describing role interactions. Reenskaug suggests to directly synthesize role models instead of preserving the individual roles and annotating them with composition constraints [Ree96b]. Using composition constraints helps to keep independent roles separate while preserving important information about their relationship.

On page 260pp of [Ree96a], Reenskaug shows that design patterns can be described using role models. He presents a sequence of patterns for the MVC pattern and shows how to construct software tools from them. Effectively, he documents a framework using patterns described as role models, in a similar vain as Johnson has already done using class diagrams [Joh92]. This is a different issue than what this paper tries to achieve: To understand recurring frameworks as composite patterns, that is not to document them (although this is an important issue), but to abstract from them to make them reusable across notation, language and domain boundaries.

Moreover, this paper presents means for analyzing object collaboration structures with respect to involved patterns. Such analytical means are necessary to cope with the increasing complexity as we turn to more challenging patterns which promise increased leverage.

The importance of *behavioral compositions* has been acknowledged for some time. The work of Helm et al. [HHG90] used the notion of *contract* to describe the formal semantics of a behavioral composition, that is a collaboration of some objects. Similar to role diagrams and role models, the focus is on the collaboration of some objects rather than on single objects. Two important operations on contracts are discussed, refinement and inclusion, that is specialization and composition. With some enhancements, contracts could probably be used to formally describe design patterns, and the inclusion operation could be used to define composite patterns.

The implementation of collaboration-based composite pattern instances is non-trivial, in particular if an object may dynamically acquire and lose roles. In the simple static case, it might be appropriate to use multiple inheritance to derive a class which offers several role protocols each of which is represented by an abstract superclass. A more elaborate approach is presented by Aksit et al. [ABV92] who introduce *composition filters* which can be used to dynamically attach and control views (roles) on an object. A composition filter is used to control the dispatch of incoming operation calls to an appropriate target. Making the method dispatch an explicit target of configuration at runtime helps to compose objects and define multiple views on the resulting object conglomerate. A related approach to control the dispatch of operation calls is to use an appropriate metalevel architecture, exemplified in [KAR+93, CM93, McA95].

The importance of describing patterns through the responsibilities assigned to their elements has not only been emphasized by [GOF95], but also by Buschmann et al. [POSA96] who made this explicit by using CRC cards [WWW90] for describing patterns. More issues of modeling with roles and implementing them are discussed in [WJS95, GSR96, KO96].

7 Conclusions

This paper defines the notion of composite pattern and illustrates it using one elaborate and further small examples. It demonstrates that composite patterns can be understood to be the abstraction from successful recurring object-oriented frameworks. To support this, an analysis and derivation technique is presented that helps pattern authors work out the essence of complex composite patterns.

The discussion is restricted to deal with patterns based on object collaborations which represent the majority of software design patterns known today. This restriction serves to introduce a more effective description, analysis and composition technique than is possible without.

In particular, the patterns are described using role diagrams, an extension of Reenskaug's role models. The most important aspect of this extension is the definition of composition constraints which specify the set of roles an object may, has to, or must not play. Composition constraints can be expressed visually as a role relationship matrix. Such a matrix supports the analysis of complex object structures as needed when defining the core of a potential composite pattern.

These concepts and techniques are to be seen as tools which pattern authors use to attack ever more complex patterns. First evaluations show that frameworks for such topics as user-defined product specifications, resource allocation, and order management share a common repeating design core. I believe that abstracting these cores into composite patterns will help software developers communicate more effectively about their frameworks and will make learning from each other easier.

Acknowledgements

I wish to thank Walter Bischofberger, Thomas Gross, Kai-Uwe Mätzel, Trygve Reenskaug, and Antonio Rito da Silva for discussing the topics of this paper as well as the paper itself with me.

References

ABV92 Mehmet Aksit, Lodewijk Bergmans and Sinan Vural. "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach." ECOOP '92, *Conference Proceedings*. Springer-Verlag, 1992.

Ale79 Christopher Alexander. *A Pattern Language.* Oxford University Press, 1979.

CM93 Shigeru Chiba and Takashi Masuda. "Designing and Extensible Distributed Language with a Metalevel Architecture." ECOOP '93, *Conference Proceedings.* LNCS 707. Springer-Verlag, 1993. Page 482-501.

Cop97 James O. Coplien. "Pattern Languages." *C++ Report* 9, 1 (January 1997): 15-21.

CS95 James O. Coplien and Douglas Schmidt (editors). *Pattern Languages of Program Design.* Addison-Wesley, 1995.

Gab94 Richard P. Gabriel. "The Failure of Pattern Languages." *Journal of Object-Oriented Programming* 5, 9 (February 1994): 84-88.

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

GSR96 Georg Gottlob, Michael Schrefl, and Brigitte Röck. "Extending Object-Oriented Systems with Roles." *ACM Transactions on Information Systems* 14, 3 (July 1996): 268-296.

HHG90 Richard Helm, Ian M. Holland and Dipayan Gangopadhyay. "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems." OOPSLA '90, *Conference Proceedings*. ACM Press. Page 169-180.

JF88 Ralph E. Johnson and Brian Foote. "Designing Reusable Classes." *Journal of Object-Oriented Programming* 1, 2 (June/July 1988): 22-35.

JO97 Ralph E. Johnson and Jeff Oakes. "The User-Defined Product Framework." *In preparation.*

Joh92 Ralph E. Johnson. "Documenting Frameworks using Patterns." OOPSLA '92, *ACM SIGPLAN Notices* 27, 10 (October 1992): 63-70.

JW97 Ralph E. Johnson and Bobby Woolf. "Type Object." *In MRB97.* Chapter 4.

KAR+93 Gregor Kiczales, J. Michael Ashley, Luis H. Rodriguez Jr., Amin Vahdat and Daniel G. Bobrow. "Metaobject Protocols: Why We Want Them and What Else They Can Do." *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993. Page 101-118.

KO96 Bent Bruun Kristensen and Kasper Osterbye. "Roles: Conceptual Abstraction Theory and Practical Language Issues." *Theory and Practice of Object System* 2, 3 (1996): 143-160.

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.

Lew95 Ted Lewis. *Object-Oriented Application Frameworks.* Manning, 1995.

McA95 Jeff McAffer. "Metalevel Programming with CodA." ECOOP '95, *Conference Proceedings.* Lecture Notes in Computer Science 952. Berlin, Heidelberg: Springer-Verlag. Page 190-214.

MRB97 Robert C. Martin, Dirk Riehle, and Frank Buschmann (editors). *Pattern Languages of Program Design 3.* Addison-Wesley, 1997.

PC86 David L. Parnas and Paul C. Clements. "A Rational Design Process: How and Why to Fake It." *IEEE Transactions on Software Engineering*, 12, 2 (February 1986): 251-257.

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

Ree96a Trygve Reenskaug, with Per Wold and Odd Arild Lehne. *Working with Objects.* Manning: 1996.

Ree96b Trygve Reenskaug. *Personal communication,* 1996.

Rie96 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.

Rie97a Dirk Riehle. *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose.* Ubilab Technical Report 97.1.1. Zürich, Switzerland: Union Bank of Switzerland, 1997.

Rie97b Dirk Riehle. “Bureaucracy.” *In MRB97*. Chapter 11.

RZ95 Dirk Riehle and Heinz Züllighoven. “A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor.” *In CS95.* Chapter 2, page 9-42.

RZ96 Dirk Riehle and Heinz Züllighoven. “Understanding and Using Patterns in Software Development.” *Theory and Practice of Object Systems* 2, 1 (1996). Page 3-13.

VCK96 John Vlissides, James O. Coplien and Norm Kerth. *Pattern Languages of Program Design 2.* Reading, MA: Addison-Wesley, 1996.

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

WG95 Andre Weinand and Erich Gamma. “ET++ A Portable Homogeneous Class Library and Application Framework.” *In Lew95*. Chapter 7, page 154-194.

WJS95 Roel Wieringa, Wiebrien de Jonge and Paul Spruit. “Using Dynamic Classes and Role Classes to Model Object Migration.” *Theory and Practice of Object Systems* 1, 1 (1995): 61-83.

WWW90 Rebecca Wirfs-Brock, Brian Wilkerson und Lauren Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

Yel96 Phillip M. Yelland. “Creating Host Compliance in a Portable Framework.” OOPSLA ’96, *Conference Proceedings.* ACM Press, 1996. Page 18-29.