The Role Object Pattern

Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf

INTENT

Adapt an object to different client's needs through transparently attached role objects, each one representing a role the object has to play in that client's context. The object manages its role set dynamically. By representing roles as individual objects, different contexts are kept separate and system configuration is simplified.

ALSO KNOWN AS

MOTIVATION

An object-oriented system is typically based on a set of key abstractions. Each key abstraction is modeled by a corresponding class in terms of abstract state and behavior. This usually works fine for the design of smaller applications. However, once we want to scale up the system into an integrated suite of applications, we have to deal with different clients that need context-specific views on our key abstractions.

Suppose we are developing software support for the bank's investment department. One of the key abstractions to be expressed is the concept of **customer**. Thus, our design model will include a Customer class. The class interface provides operations to manage properties like the customer's name, address, savings and deposit accounts.

Let's assume that the bank's loan department also needs software support. It seems our class design is inadequate to deal with a customer acting as borrower. Obviously, we must provide further implementation state and operations to manage the customer's loan accounts, credits, and securities.

Integrating several context-specific views in the same class will most likely lead to key abstractions with bloated interfaces. Such interfaces are difficult to understand and hard to maintain. Unanticipated changes cannot be handled gracefully and will trigger lots of recompilation. Changes to a client-specific part of the class interface are likely to affect clients in other subsystems or applications as well.

A simple solution might be to extend the Customer class by adding new Borrower and Investor subclasses which capture the borrower-specific and investor-specific aspects respectively. From an object identity point of view, subclassing implies that two objects of different subclasses are not identical. Thus, a customer acting both as investor and as borrower is represented by two different objects with distinct identities. Identity can only be simulated by an additional mechanism. If two objects are meant to be identical, their inherited attributes must constantly be checked for consistency. However, we will inevitably run into problems in case of polymorphic searches, for example when we want to make up the list of all customers in the system. The same Customer object will appear repeatedly unless we take care of eliminating "duplicates".

The Role Object pattern suggests to model context-specific views of an object as separate **role objects** which are dynamically attached to and removed from the **core object.** We call the resulting composite object structure, consisting of the core and its role objects, a subject. A subject often plays several roles and the same role is likely to be played by different subjects. As an example consider two different customers playing the role of borrower and investor, respectively. Both roles could as well be played by a single Customer object.

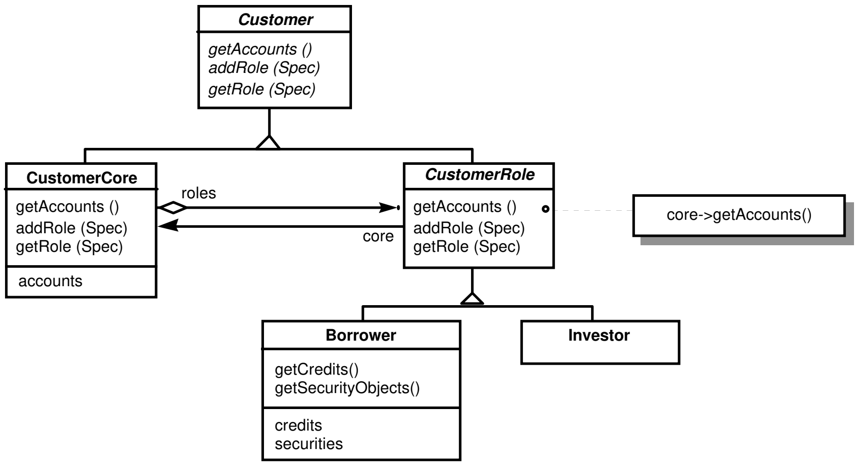

Fig. 1: Customer hierarchy in a banking environment

A key abstraction such as Customer is defined as an abstract superclass. It serves as a pure interface which does not define any implementation state. The Customer class specifies operations to handle a customer's address and accounts, and defines a minimal protocol for managing roles. The CustomerCore subclass implements the Customer interface.

The common superclass for customer-specific roles is provided by CustomerRole, which also supports the Customer interface. The CustomerRole class is abstract and not meant to be instantiated. Concrete subclasses of CustomerRole, for example Borrower or Investor, define and implement the interface for specific roles. It is only these subclasses which are instantiated at runtime. The Borrower class defines the context-specific view of Customer objects as needed by the loan department. It defines additional operations to manage the customer's credits and securities. Similarly, the Investor class adds operations specific to the investment department's view of customers.

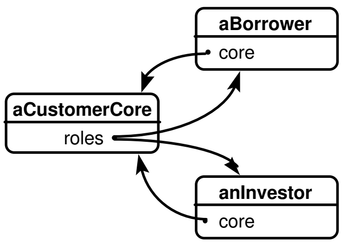

Figure 2: An object diagram of the Role Object pattern

A client like the loan application may either work with objects of the CustomerCore class, using the interface class Customer, or with objects of concrete CustomerRole subclasses. Suppose the loan application knows a particular Customer instance through its Customer interface. The loan application may want to check whether the Customer object plays the role of Borrower. To this end it calls hasRole() with a suitable role specification. For the purpose of our example, let's assume we can name roles with a simple string. If the Customer object can play the role named "Borrower," the loan application will ask it to return a reference to the corresponding object. The loan application may now use this reference to call Borrower-specific operations.

APPLICABILITY

Use the Role Object pattern, if

- you want to handle a key abstraction in different contexts and you do not want to put the resulting context-specific interfaces into the same class interface.
- you want to handle the available roles dynamically so that they can be attached and removed on demand, that is at runtime, rather than fixing them statically at compile-time.
- you want to treat the extensions transparently and need to preserve the logical object identity of the resulting object conglomerate.

- you want to keep role/client pairs independent from each other so that changes to a role do not affect clients that are not interested in that role.

Don't use this pattern

- if your potential roles have strong interdependencies.

There are several design variations on using roles. Fowler presents a guide on these variations and shows when to use which pattern [Fowler97].

STRUCTURE

The following picture shows the structure diagram of the Role Object pattern.

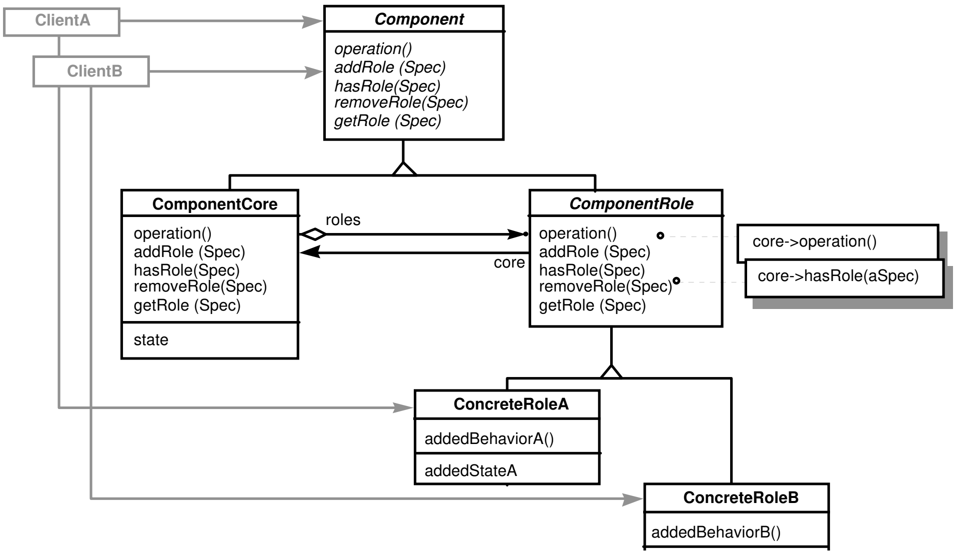

Figure 3: Structure diagram of the Role Object pattern

PARTICIPANTS

- *Component (Customer)*
  - models a particular key abstraction by defining its interface;
  - specifies the protocol for adding, removing, testing and querying for role objects. A Client supplies a specification for a ConcreteRole subclass. In the simplest case, it is identified by a string.
- *ComponentCore (CustomerCore)*
  - implements the Component interface including the role management protocol;
  - creates ConcreteRole instances;
  - manages its role objects.
- *ComponentRole (CustomerRole)*
  - stores a reference to the decorated ComponentCore;
  - implements the Component interface by forwarding requests to its core attribute.
- *ConcreteRole (Investor, Borrower)*
  - models and implements a context-specific extension of the Component interface;
  - can be instantiated with a ComponentCore as argument.

COLLABORATIONS

Core and role objects collaborate as follows:

- ComponentRole forwards requests to its ComponentCore object.

- ComponentCore instantiates and manages ConcreteRoles.

A client interacts with roles and core objects in the following ways:

- A client can extend core objects with roles. To this end, he describes the desired roles with specification objects.
- Whenever the client wants to work on a core object in a role specific way, he asks the core object for this role. If the core object is currently playing the requested role, it is returned to the client.
- If the core objects does not play the requested role, an error is thrown. A core object never creates role objects on its own.

CONSEQUENCES

The Role Object pattern has the following advantages and consequences:

- *The key abstraction can be defined concisely.* The Component interface is well-focussed on the essential state and behavior of the modeled key abstraction and is not bloated by context-specific role interfaces.
- *Roles can be evolved easily and independently of each other.* Extending the Component interface is very easy because you will not have to change the ComponentCore class. A ConcreteRole class lets you add new roles and role implementations while preserving the key abstraction itself.
- *Roles objects can be added and removed dynamically.* A role object can be added and removed at runtime, simply by attaching it to and detaching it from the core object. Thus, only those objects that are needed in a given situation are actually created.
- *Applications get better decoupled.* By explicitly separating the Component interface from its roles, the coupling of applications based on different role extensions is decreased. An application (ClientA) using the Component interface and some context-specific ConcreteRole classes does not need to know the ConcreteRole classes used in other applications (ClientB).
- *Combinatorial explosion of classes through multiple inheritance is avoided.* The pattern avoids the combinatorial explosion of classes as it would result from using multiple inheritance to compose the different roles in a single class.

The Role Object pattern has the following disadvantages and liabilities:

- *Clients are likely to get more complex.* Working with an object through one of its ConcreteRole interfaces implies slight coding overhead compared to using the interface provided by the Component interface itself. A client has to check whether the object plays the role in question. If it does, the client needs to query for the role. If it does not, the client is responsible for extending the core object in its specific use-context provided that the core object actually can play the role.
- *Maintainig constraints between roles becomes difficult.* Since a subject consists of several objects which are mutually dependent, maintaining constraints and preserving the overall subject consistency might become difficult. In the implementation section we discuss several of the arising issues.
- *Constraints on roles cannot be enforced by the type system.* You might want to exclude certain roles from being attached to the same core object in combination. Or, certain roles may depend on the existence of others. With the Role Object pattern, you can’t rely on the type system to enforce the constraints for you. You will have to use runtime checks instead.
- *Maintaining object identity gets more complex.* The core object and its role instances form a conceptual unit which should have a conceptual identity of its own. While technical object identity can be handled directly in any given programming language (checking objects for technical identity is carried out by comparing object references), checking for conceptual identity requires additional operations in the Component interface. This can be implemented by comparing core object references.

IMPLEMENTATION

An implementation of the Role Object pattern must address two critical issues: transparently extending key abstractions with roles, and dynamically managing these roles. For transparent extension, we use the Decorator pattern [Gamma+95]. For creating and managing roles, we apply the Product Trader pattern [Bäumer+97]. Thus, the Role Object pattern combines two well-known patterns thereby adding new semantics.

- *Providing interface conformance.* Since we want role objects to be used transparently wherever the core object can be used, they must support a common interface. Note that from a modeling point of view, a role class is considered a specialization of its core (i.e. an Investor *is-a* Customer). The Decorator pattern tells us how to do this. First, we factor out a common interface for all objects that can have roles added to them dynamically. This interface is provided by the Component class in the structure diagram and corresponds to the Component class in the Decorator pattern. For all context-specific roles that might extend the Component´s functionality, we introduce the abstract superclass ComponentRole which corresponds to the Decorator class in [Gamma95]. ComponentRole implements the Component's interface by forwarding operation invocations to the core object. Thus, roles transparently wrap the core. ConcreteRole classes must inherit from ComponentRole; they correspond to the ConcreteDecorator classes in [Gamma+95].
- *Hiding the role object creation process.* Role instances are used to decorate a core object at run-time. A key issue is how a ConcreteRole instance is actually created and attached to the core object. Notice that ConcreteRoles are not meant to be created by clients. Rather, the role creation process should be initiated by ComponentCore, thereby avoiding that role objects may exist of their own (i.e., independently of a core object). This also prevents clients from knowing how to instantiate role objects.
- *Decoupling role classes from the core.* Creating and managing roles should be done in a generic fashion. Otherwise, it becomes hard if not impossible to extend a ComponentCore with new and unforeseen roles without changing its implementation. Therefore, both the creation and the management process must be independent of concrete role classes; the ComponentCore code must not statically reference any of them.

  This can be achieved by using specification objects. To request a role the clients passes a specification object to the core. The simplest solution is to use the type name as the specification (see example section). The core returns the role object wich matches the specification. More elaborate specification mechanisms are discussed in [Riehle95, Evans+97].

  The same specification objects can be used for creation.To achieve this, the Product Trader pattern can be used [Bäumer+97]. The role object trader maintains a container of specification objects with associated creator objects, for example class objects, prototypes or exemplars. When a client wants to add a new role, it passes a specification object to the core. The core delegates the creation to the role object trader.
- *Choosing appropriate specification objects.* In many cases it will be sufficient to use the type name as a specification object. But sometimes it is worth to use more complex specifications:

  Suppose you have modeled Person as a core object class. Some persons may be employees, so there is a role type Employee. Because there are different kinds of employees, you will want to make Employee an interface class and model concrete roles as subclasses of Employee, for example Salesman, Developer and Manager. When a client needs information about the salary of a person, it will request the role "Employee" from the core. This cannot be done using the type name, because the concrete role objects will have "Salesman" etc. as type names. In such situations you can use Type Objects [Johnson+97] as specifications. The core can then retrieve the requested role object by evaluating sub/super type relations.
- *Managing role objects.* To let a core object manage its roles, the Component interface declares a role management protocol which includes operations for adding, removing, testing, and querying role objects. To support the role management protocol, the core object maintains a dictionary that maps role specifications to concrete role instances. Whenever a role object is attached to the core, the new role object is registered in the role dictionary together with its role specification.

  Please note that a core object manages its role objects through references of type ComponentRole, thereby excluding ComponentCore instances from acting as roles! Because the core owns its roles, it must take care of them. In particular, it must delete them when it gets deleted itself.
- *Maintaining consistent core and role object state.* Changes to the core object or to role objects may require updates to further role objects. As an example, consider changing the name of a Person which is also a Borrower. Whenever the Person receives a new name, a flag must be raised in the Borrower role to indicate that the Borrower had a name change. The flag indicates to the system that a name change must be reported to a national institute which collects and passes on information on the creditworthiness of borrowers. Notification is mandatory for the bank. There are several possible solutions to ensure these constraints, all of which come with a price; usually the dependencies are hard-coded. We discuss a more elaborate solution in the next bullet item.
- *Maintaing role attribute constraints by using Property and Observer.* If state integration becomes complex due to many interdependencies, the implementation state of the core object (or parts thereof) might be represented using a property list ([Riehle97], also called Variable State [Beck96]). A property list is a list of

key/value pairs which represent attribute names and values, respectively. It is usually implemented as a dictionary, which maps the attribute name on the attribute value (object).

A role object may then register interest in a particular attribute defined as part of the core object's state and be notified if a state changing operation is called on it. To enable this, every role object that changes an attribute must inform the core object about this so that it can notify dependent role objects about the change.

Property lists are usually perceived as being a bad thing, because they break encapsulation by exposing implementation state. Changes in attribute names may require all role classes depending on them to change accordingly. Moreover, bad code might cause unwanted side-effects. However, when carefully handled, these problems can be avoided. Perhaps the best known example of extensive use of this pattern are decorated nodes in abstract syntax trees, the key abstraction in many compilers and software development environments.

- *Maintaining conceptual identity.* The Role Object pattern lets you to manage the core and its roles as a single conceptual object with integrated state. Therefore, it should be possible for clients to find out whether two distinct technical objects are actually part of the same logical object, that is if they are conceptually identical. Different role objects share a single conceptual object identity whenever they have the same core.

  For example, consider a client application which works directly with the core object through the Component interface, and refers to its single role instance using a concrete role interface. From a technical point of view, the two references do not point to the same object because they reference two technically distinct objects. Thus, to find out that the two references are actually denote the same conceptual object, the client must use special identity comparison operations provided by the Component interface. Usually, this will be implemented as a direct comparison of two core object references.

- *Maintaining constraints among roles.* Between roles themselves (not just their state), there might be a number of constraints. A common case is that a role B requires a role A to be already played by the object. For example, if Customer and Borrower are both roles of Person, then the existence of a Customer role object is a precondition for allowing a Person to play the Borrower role. These are role-level constraints. The capability of an object to play a particular role B is restricted to the case that the object already plays a role A. Without that role A, role B must not be played. These constraints emerge from the application domain.

  Generally speaking, role B might not just depend on role A but also on role A being in a particular state. For the most complex cases, you won't be able to avoid using a constraint solving system. Fortunately, in practice, the situation almost never scales up to full complexity, and more pragmatic solutions are sufficient. Typical cases can be solved by using a two-phase commit protocol, that is to ask all roles first before finally executing a request, for example before removing a role object.

- *Maintaining constraints among roles by recursively applying the Role Object pattern.* Many problem cases of role-level constraints can be solved by applying the Role Object pattern recursively. If role A is a precondition for a number of Roles B, C, D, etc. then role A can be understood as a key abstraction for these roles. Borrower and Investor can be viewed as roles of Customer, and Customer and Guarantor can be viewed as roles of Person. Since being a Guarantor does not require being a Customer, it need not be a modeled as a role of Customer. The following figure shows a recursive application of the Role Object pattern.

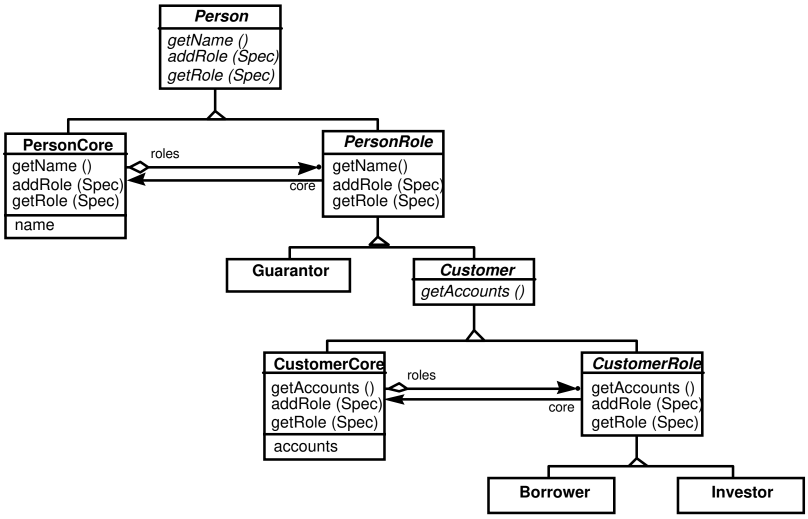

Figure 5: The Role Object pattern applied recursively

At runtime, this leads to a chain of role and core objects. The following figure depicts the situation:

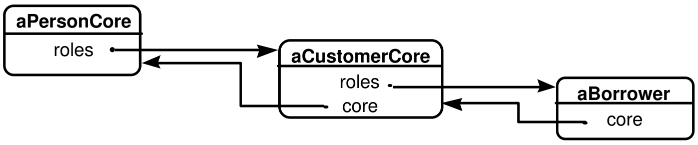

Figure 6: Dynamic object diagram with roles being attached to roles.

The role level constraints are simply enforced by role objects for Borrower or Investor not coming into existence unless the Customer role already exists. Thus, modeling Customer as another key abstraction for further roles serves to ensure a vital role-playing constraint on the more general Person key abstraction.

SAMPLE CODE

The following C++ code shows how to implement the customer example discussed in the motivation section. We'll assume there is a Component class called `Customer`.

```
class CustomerRole;
class Customer {
public:

// customer specific operations
   virtual list<Account *> getAccounts() = 0;

// Role management
   virtual CustomerRole * getRole(String aSpec) = 0;
   virtual CustomerRole * addRole(String aSpec) = 0;
   virtual CustomerRole * removeRole(String aSpec) = 0;
   virtual CustomerRole * hasRole(String aSpec) = 0;
};
```

The implementation of class `CustomerCore` might look like this:

```
class CustomerCore : public Customer {
public:
   CustomerRole * getRole(String aSpec)
   {
      return roles[aSpec];
   };

   CustomerRole * addRole(String aSpec)
   {
      CustomerRole * role = NULL;
      if ((role = getRole(aSpec)) == NULL)
      {
         if(role = CustomerRole :: createFor(aSpec, this)) roles[Spec] = role;
```

```
      }
      return role;
   };

   list<Account *> getAccounts() { ... };

private:
   map<String, CustomerRole *> roles;
};
```

The specification of a role is implemented as string which equals the class name of the ConcreteRole class. The mapping between the role's specification and the role object itself is implemented by a dictionary.

Next, we define a subclass of `Customer` called `CustomerRole` which we will subclass to obtain different concrete roles. `CustomerRole` decorates the `CustomerCore` referenced by the `core` instance variable. For each operation in `Customer`'s interface, `CustomerRole` forwards the request to `core`. Notice that the core instance variable is typed as `CustomerCore`, thereby ensuring that customer roles are not used as core objects. The mapping between a role's specification and an appropriate creator object which is able to instantiate the specified role is implemented as a lookup table. For a detailed discussion of how to implement and manage creator objects, see [Bäumer+97].

```
class CustomerRole : public Customer {
public:

   list<Account *> getAccounts() { return core->getAccounts() };

   CustomerRole * addRole(String aSpec) { return core->addRole(aSpec); };

   static CustomerRole * createFor(String aSpec, CustomerCore * aCore)
   {
      CustomerRole * newRole = NULL;
      if (newRole = lookup(aSpec)->create()) newRole->core = aCore;
      return newRole;
   };

private:
   CustomerCore * core;
};
```

Subclasses of `CustomerRole` define specific roles. For example the class `Borrower` adds operations to handle securities and credit accounts. Subclasses should not override the inherited role management operations.

```
class Borrower : public CustomerRole {
public:
   list<Security *> getSecurities() { return securities; };

private:
   list<Security *> securities;
};
```

Notice that clients must down-cast the role reference returned by the core component before they can invoke role-specific operations on the role instance:

```
Customer * aCoustomer = Database :: load("Tom Jones");
Borrower * aBorrower = NULL;

if (aBorrower = dynamic_cast<Borrower *> aCustomer->getRole("Borrower"))
{
   // access securities
   list<Security *> securities = aBorrower->getSecurities();

};
```

KNOWN USES

The GEBOS series of object-oriented banking projects makes extensive use of this pattern [Bäumer+97a]. It provides software support for a number of banking business sections including the teller, loan, and investment department as well as self-service and account management. The GEBOS system is based on a common business domain layer modeling the bank's core concepts. Concrete workplace applications extend these core concepts using the Role Object pattern.

The Tools and Materials framework described in [Riehle+95, Riehle+96] explores the role modeling design space of by protocol copy and paste, multiple inheritance, decorators and wrappers to achieve the effects of the Role Object pattern. These variations have more concisely been described by Fowler [Fowler+97].

The Geo system currently under development at Ubilab, the information technology research laboratory of Union Bank of Switzerland is using the Role Object pattern as an implementation variation of roles as first-class entities of programming.

Kristensen and Østerbye report on using the Decorator pattern to introduce roles into programming languages [Kristensen+96]. However, they do not address issues of creating and managing role objects in detail.

We have used a domain-specific example, Person and its roles, for illustration purposes. This example is so common that it actually represents a pattern of its own. Since a Person abstraction is needed in many contexts, there are also any number of different roles which it may play. Schoenfeld discusses several examples, for example Person and its roles in document centered business processes [Schoenfeld96]. We picked Person and its roles in the context of banking businesses with customers. Yet another example is Person and its roles in a bureaucratic hierarchy and the related payroll management problems.

A totally unrelated use of the Role Object pattern is the decoration of nodes in abstract syntax trees (AST's). AST's are the primary abstraction in most software development environments. They are viewed and used from many different tools, for example syntax-directed editors, symbol browsers, cross referencers, compilation support, dependency analysis and change impact tools. Every tool needs to annotate the AST nodes with its specific information yet is usually interested only in small aspects of the whole tree. The Role Object pattern works well to provide these tools with tool-specific interfaces on the nodes. Mitsui et al. discuss the use of the pattern in the context of a C++ programming environment [Mitsui+93], for the specific use just discussed as well as for more general purposes.

RELATED PATTERNS

The Extension Object pattern [Gamma97] addresses the same issue: A component is extended by means of extension objects in such a way that they satisfies context-specific requirements. The pattern, however, does not show how Component and ComponentRole objects can be treated transparently, which we consider a key aspect of applying the Role Object pattern. Moreover, the Extension Object pattern only touches the issue of extension object (role object) creation and management. We view the integration of the Decorator pattern with the Product Trader pattern to be a key part of the Role Object pattern.

The Extension Object pattern has been used for role modeling purposes by Zhao and Foster [Zhao+97] and Schoenfeld [Schoenfeld96]. Zhao and Foster discuss role objects as extension objects, that is they do not transparently wrap the core object. Their key example is the notion of Point (and its roles) as it is a key in transport software systems. Schoenfeld chose the same primary example as we did, Person and its roles, but also uses the Extension Objects pattern rather than transparently wrapping the core by a Decorator.

The Post pattern in [Fowler96] describes an interesting variant of this pattern. Similar to the Extension Object pattern, it describes the responsibilities of a core object in the context of a particular application. However, a Post object exists independently of the core and can live on without being assigned to a core.

ACKNOWLEDGMENTS

Ari Schoenfeld.

REFERENCES

Bäumer+97 Dirk Bäumer and Dirk Riehle. "Product Trader." In [Martin+97]. Chapter 3.

Bäumer+97a Dirk Bäumer, Guido Gryczan, Rolf Knoll, Carola Lilienthal, Dirk Riehle, and Heinz Züllighoven. "Framework Development for Large Systems." *Communications of the ACM* 40, 10 (October 1997).

Beck96 Kent Beck. *Smalltalk Best Practice Patterns.* Prentice-Hall, 1996.

Coplien+95 James O. Coplien and Douglas C. Schmidt (editors). *Pattern Languages of Program Design.* Addison-Wesley, 1995.

Evans+97 Eric Evans and Martin Fowler. "Specification Patterns." Submitted to PLoP '97.

Fowler96 Martin Fowler. Analysis Patterns. Addison-Wesley, 1996.

Fowler97 Martin Fowler. ”Role Patterns.” Submitted to PLoP ’97.

Gamma+95 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

Gamma97 Erich Gamma. ”Extension Object.” In [Martin+97]. Chapter 6.

Johnson+97 Ralph Johnson and Bobby Woolf. ”Type Object.” In [Martin+97]. Chapter 4.

Kristensen+96 Bent Bruun Kristensen and Kasper Østerbye. ”Roles: Conceptual Abstraction Theory and Practical Language Issues.” *Theory and Practice of Object System* 2, 3 (1996): 143-160.

Martin+97 Robert C. Martin, Dirk Riehle, and Frank Buschmann (editors). *Pattern Languages of Program Design 3.* Addison-Wesley, 1997.

Mitsui+93 Kin'ichi Mitsui, Hiroaki Nakamura, Theodore C. Law, and Shahram Javey. ”Design of an Integrated and Extensible C++ Programming Environment.” *Object Technology for Advanced Software* (ISOTAS-93, LNCS-742). Edited by Shojiro Nishio and Akinori Yonezawa. New York: Springer-Verlag, 1993. Page 95-109.

Riehle+95a Dirk Riehle. ”How and Why to Encapsulate Class Trees.” In Proceedings of the 1995 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95). ACM Press, 1995. Page 251-264.

Riehle+95b Dirk Riehle and Heinz Züllighoven. ”A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor.” In [Coplien+95]. Chapter 2.

Riehle+95 Dirk Riehle and Martin Schnyder. *Design and Implementation of a Smalltalk Framework for the Tools and Materials Metaphor.* Ubilab Technical Report 95.7.1, 1995.

Riehle97 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.

Schoenfeld96 Ari Schoenfeld. Domain Specific Patterns: *Conversions, Persons and Roles, and Documents and Roles*. PLoP' 96, Conference Proceedings. Washington University Department of Computer Science, Technical Report WUCS-97-07, 1997.

Zhao+95 Liping Zhao and Ted Foster. ”A Pattern Language of Transport Systems (Point and Route).” In [Martin+97]. Chapter 23.

Dirk Bäumer works for TakeFive Software AG, Eidmattstr. 51, CH-8032 Zurich, Switzerland. He welcomes e-mail at baeumer@takefive.ch.

Dirk Riehle works at Ubilab, the information technology research laboratory of UBS. He can be reached at UBS, Bahnhofstrasse 45, CH-8021 Zurich. He welcomes e-mail at Dirk.Riehle@ubs.com or riehle@acm.org.

Wolf Siberski works for RWG GmbH, Germany. He can be reached at RWG GmbH, Räpplenstraße 17, 70191 Stuttgart, Germany. He welcomes e-mail at Wolf_Siberski@rwg.e-mail.com.

Martina Wulf works at Union Bank of Switzerland, Bahnhofstrasse 45, CH-8021 Zurich. She can be reached at Martina.Wulf@ubs.ch.