Pattern Density and Role Modeling of an Object Transport Service

Dirk Riehle. SKYVA International. 25 First Street, Cambridge, MA 02129, U.S.A.
E-mail: driehle@skyva.com or riehle@acm.org

Roger Brudermann. Winterthur Insurances. Konradstrasse 14/KS423, 8401 Winterthur, Switzerland.
E-Mail: roger.brudermann@winterthur.ch

Thomas Gross. Dept. Informatik, ETH Zürich, 8092 Zürich, Switzerland.

Kai-Uwe Mätzel. OTI International. Oberdorfstrasse 8, 8001 Zürich, Switerzland.
E-mail: kai-uwe_maetzel@oti.com or maetzel@acm.org

Abstract

Many object-oriented frameworks exhibit a *high density of design pattern applications.* Usually, there are more pattern instances in a framework than there are abstract classes. Yet, only little has been done so far to better cope with pattern density and the resulting *interaction and composition problems.* We illustrate the problem by describing the design of an object transport service, which provides functionality to copy objects across process boundaries. The service is used by higher-level services like object migration and remote request execution. It is designed and implemented as an object-oriented framework based on the composition of several interlocking design pattern applications. We argue that we need better ways of describing patterns and composing them than available today. We report on *describing patterns as role models.* This approach eases pattern composition and simplifies framework design. Our experience indicates that a pattern-based framework description using role models makes the design more comprehensible than a description that solely focuses on classes.

Categories: C.2.4, D.1.3, D.1.5, D.2.0, D.2.2, D.2.10, D.2.11, D.2.13.

General terms: design, documentation.

Keywords: framework, pattern, pattern density, pattern application, pattern composition, role, role model, role model composition, object transport, object migration, inter-process communication.

1 Introduction

Many object-oriented frameworks exhibit complex interactions between the involved objects. Objects use each other in different ways and for different purposes. Most of these interactions can be described as instances of specific design patterns [Gamma et al. 1995], and many frameworks show a high number of such pattern instantiations. It is usually higher than the number of employed design-level classes. This phenomenon has been called "a high pattern density."

Today, pattern-aware developers annotate classes with the pattern instances they are involved in. However, annotating classes is an informal practice, which is not precise enough when it comes to clearly defining how different pattern instantiations interact and form classes. To better cope with the complexity induced by high pattern density, we use role modeling [Reenskaug et al. 1996], which we add to existing class-based approaches. This paper illustrates the problem and motivates a role-modeling approach to pattern instantiation and composition.

As an example, we use the object transport service of the Geo system. The Geo system developed at Ubilab is a distributed system based on a service architecture. It uses object transport as its primitive of inter-process communication. The object transport service is designed and implemented as an object-oriented framework, the Geo-Transporter. Higher-level services like object migration and remote request execution build on it.

Originally scheduled for publication in 1998. Changes in 2000 only apply to author addresses, formatting, and copyright statement.

The Geo-Transporter is a process-local service implemented in Java [Arnold and Gosling 1996]. The framework is flexible in many respects. E.g., it may use different data transfer mechanisms or may employ various marshaling techniques. The need for flexibility is also the reason why we developed our own framework rather than using what Java or CORBA [Siegel 1996] offers.

Our experience with Geo and other systems suggests that the use of role models to describe patterns provides many benefits [Riehle 1996]. Patterns described as role models can be composed easily [Riehle 1997a], and framework designs can be better motivated.

2 Patterns as Role Models

A role represents the view an object holds on another object [Reenskaug 1996, Riehle 1996, Riehle 1997a]. It focuses on one particular aspect of an object, thereby separating the different contexts an object is involved in. An object may play several roles at once, and the same role may be played by different objects.

A role model describes a particular aspect of an object collaboration as a set of roles and their relationships. A role model has a particular purpose, e.g., to describe how objects notify each other about events or how objects may be serialized. Objects in a concrete collaboration achieve a role model's purpose by acting according to the definition of the roles they play. An object may participate in several role models at once, thereby integrating the different roles it plays in different contexts.

Role models achieve an excellent separation of concerns by modeling exactly one particular collaboration purpose while ignoring others. Role models are design fragments that can be composed much easier than classes.

An object collaboration pattern can be well illustrated using a role model. Many patterns translate easily into role-model based descriptions, if the concept of participant [Gamma et al. 1995] is replaced by the concept of role [Riehle 1997b].

Role models, e.g., derived from pattern applications, are composed by assigning roles to classes. A class defines one or more roles for its instances, which may play these roles at runtime. A class integrates several roles from different pattern applications through its definition and implementation.

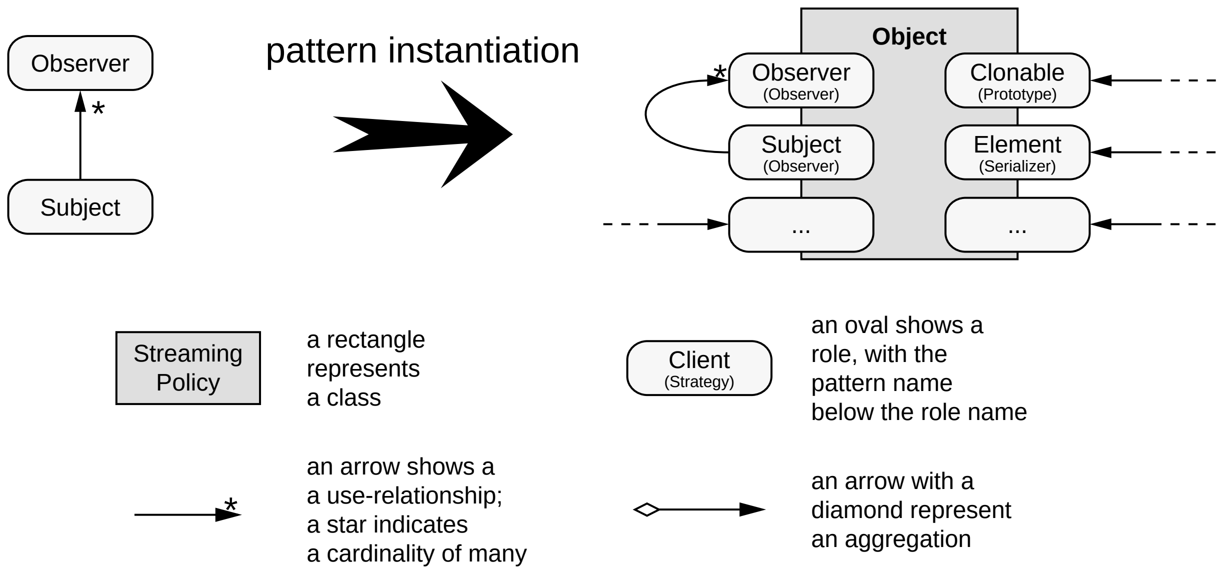
Observer
*
Subject
pattern instantiation
Object
Observer
(Observer)
Clonable
(Prototype)
Subject
(Observer)
Element
(Serializer)
...
...
Streaming Policy
a rectangle represents a class
Client
(Strategy)
an oval shows a role, with the pattern name below the role name
an arrow shows a a use-relationship; a star indicates a cardinality of many
an arrow with a diamond represent an aggregation

Figure 1: The Observer pattern as a role model and a typical instantiation.

Figure 1 illustrates the relationship between the Observer pattern, illustrated as a role model, its instantiation in a concrete design, and the assignment of roles to a class. In Smalltalk, e.g., class Object defines protocols both for the Observer and Subject roles, so that the two different roles of the Observer pattern are instantiated as two roles defined by the same class.

3 The Geo-Transporter Framework

In this section, we describe the Geo-Transporter framework using role-model based patterns. If not indicated otherwise, a pattern is defined in [Gamma et al. 1995] and described using role models in [Riehle 1997b]. Figure 2 shows the framework design.

The framework consists of three key design-level classes and many more implementation classes not shown in the figure.

The key classes are the ObjectTransportService class, which encapsulates the service as a subsystem and acts as a convenient access point, the OtsFactory class, which configures the service by instantiating appropriate implementations, and the Communicator class, which handles inter-process communication (IPC) issues. With the exception of the OtsClient class, all other classes shown in figure 2 stem from lower-level frameworks on which the Geo-Transporter builds.

At runtime, there is exactly one instance of the ObjectTransportService class, which maintains a list of Communicator objects, one for each remote process it is connected with. Based on the preferred IPC mechanism, different Communicator implementations are instantiated by the single instance of the OtsFactory class.

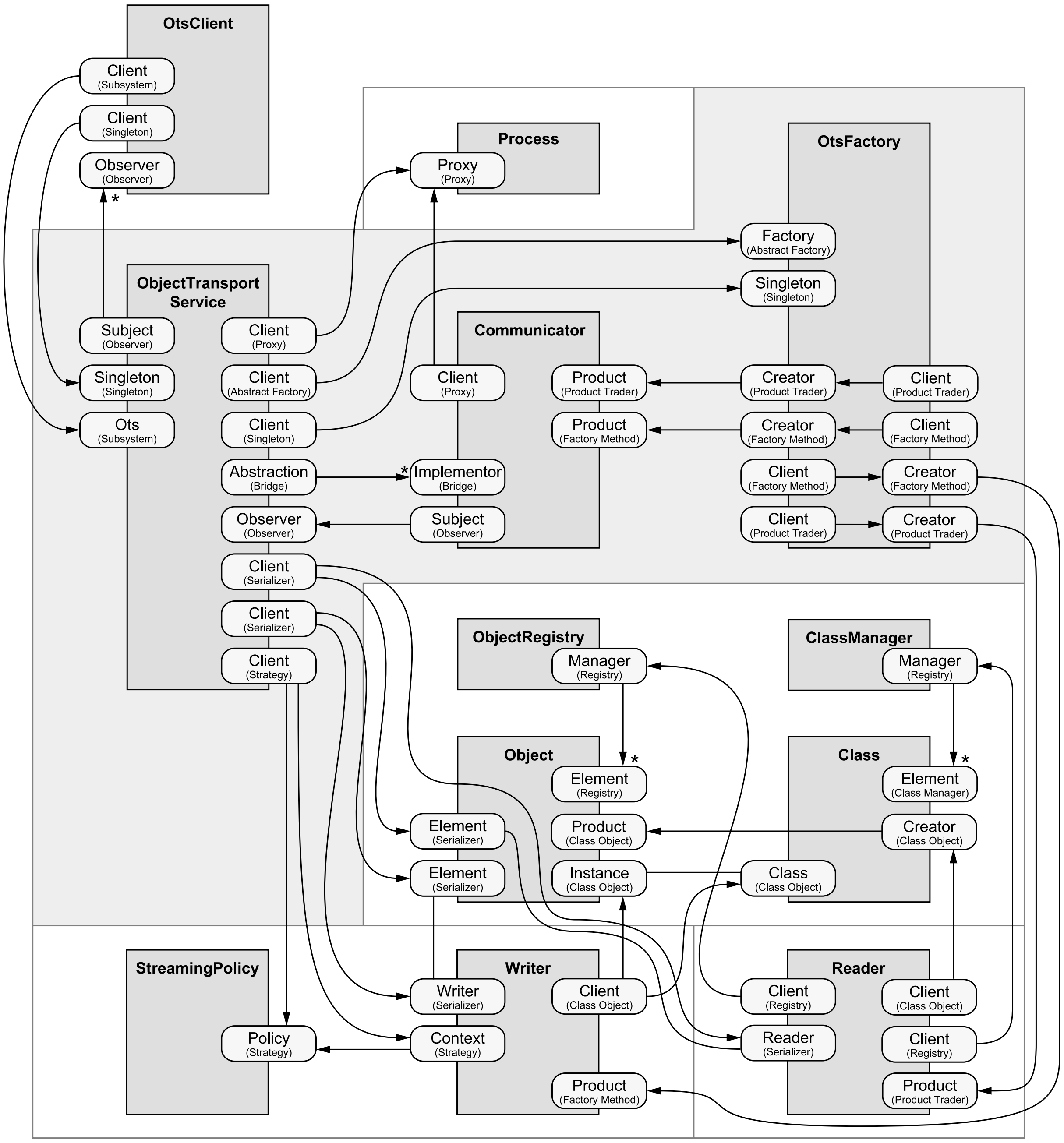

Figure 2: Design of the Geo-Transporter framework.

Clients interact with the service using the ObjectTransportService (OTS) class. They either tell the service to send some objects to another process, or they wait for it to notify them about new objects that have arrived from other processes.

- *Subsystem pattern.* The OTS acts as a subsystem interface by playing the Ots role. The OtsClient class defines the Client role for its instances.
- *Singleton pattern.* The OTS plays the Singleton role, because in every process there is exactly one instance of it. The OtsClient class defines the role Client.
- *Observer pattern.* The OTS plays the Subject role, because OtsClient instances, which play the Observer role, register with it to be notified when new objects have arrived.

These three pattern applications play a particularly important role in the design of the framework, because they determine how the framework is to be used by clients. Effectively, they are used to *integrate the framework* into a (yet unknown) use context.

The OTS interacts with a Communicator to transport objects between processes. When sending objects, the OTS converts them into data and tells the Communicator to send this data to a target process. The Communicator encapsulates the handling of the underlying IPC mechanism. When receiving objects, the Communicator forwards the data to the OTS, which turns them into objects and notifies clients.

- *Bridge pattern.* The OTS plays the Abstraction role by providing higher-level functionality like object graph serialization. It is implemented based on Communicator instances, which play the Implementor role for simple data buffer sending and receiving.
- *Observer pattern.* The OTS plays the Observer role of its Communicators, which play the Subject role. When a Communicator receives a data buffer, it notifies the OTS.

Remote processes are handled in an object-oriented way using a dedicated instance of class Process. It provides access to some initial remote references (like the local naming context and object registry).

- *Proxy pattern.* The OTS plays the Client role and a Process instance plays the Proxy role for some remote process, which plays the Subject role.

A single instance of OtsFactory configures the service by instantiating appropriate subclasses of Communicator as well as of Reader and Writer for serialization (see below).

- *Abstract Factory pattern.* The OTS plays the Client role, and delegates object creation to the OtsFactory, which plays the Factory role.
- *Singleton pattern.* The OTS plays the Client of the OtsFactory, which plays the Singleton role, because there is only one instance of the factory.

The factory implements its creation operations using the Factory Method and Product Trader [Bäumer and Riehle 1998] patterns. The factory provides a default Writer object for streaming, determines a proper Reader object for a given buffer protocol, and selects Communicator implementations given a protocol name.

The Geo-Transporter framework builds on a number of lower-level frameworks, which provide further services. The serialization framework is based on the Serializer pattern [Riehle et al. 1998] and provides Reader and Writer classes for object serialization. Serialization may be configured using streaming policies (Strategy pattern) that determine which part of an object graph is to be written.

The basic object management framework is based on the Class Object and Registry patterns (yet undocumented), which provide object and class management functionality.

4 Discussion

Figure 2 shows the design of the Geo-Transporter and its supporting frameworks. Summing up, it shows 12 classes and 19 pattern applications (and we already omitted pattern applications that do not contribute to the understanding of the Geo-Transporter). Figure 3 shows the same design without roles. All object collaboration information is collapsed into simple use relationships between classes.

The role-model based description of the framework, and in particular of the ObjectTransportService class, has illustrated the complexity of object collaboration. Section 3 discusses 8 patterns from which 8 roles are derived for the OTS class. More have been either hinted at (serialization) or omitted. Thus, the whole framework shows a high pattern density.

We draw the following conclusions:

- Classes alone do not present object collaboration well; role models help make the collaboration more explicit. Without roles, one cannot distinguish the different use contexts of objects and the collaboration purposes.
- Classes alone do not provide pattern application information well; role-model based pattern descriptions help make the pattern application explicit as role models connecting classes.

We therefore view the class diagram as a coarse-grained view of the same framework, with the roles being implicit in the interfaces. Role modeling allows the designer to “zoom in”, to see the roles, and to obtain a much more detailed view of how objects interact according to the different patterns. These details are essential information and should be provided—to simplify the design of the framework and to allow its comprehension by subsequent users.

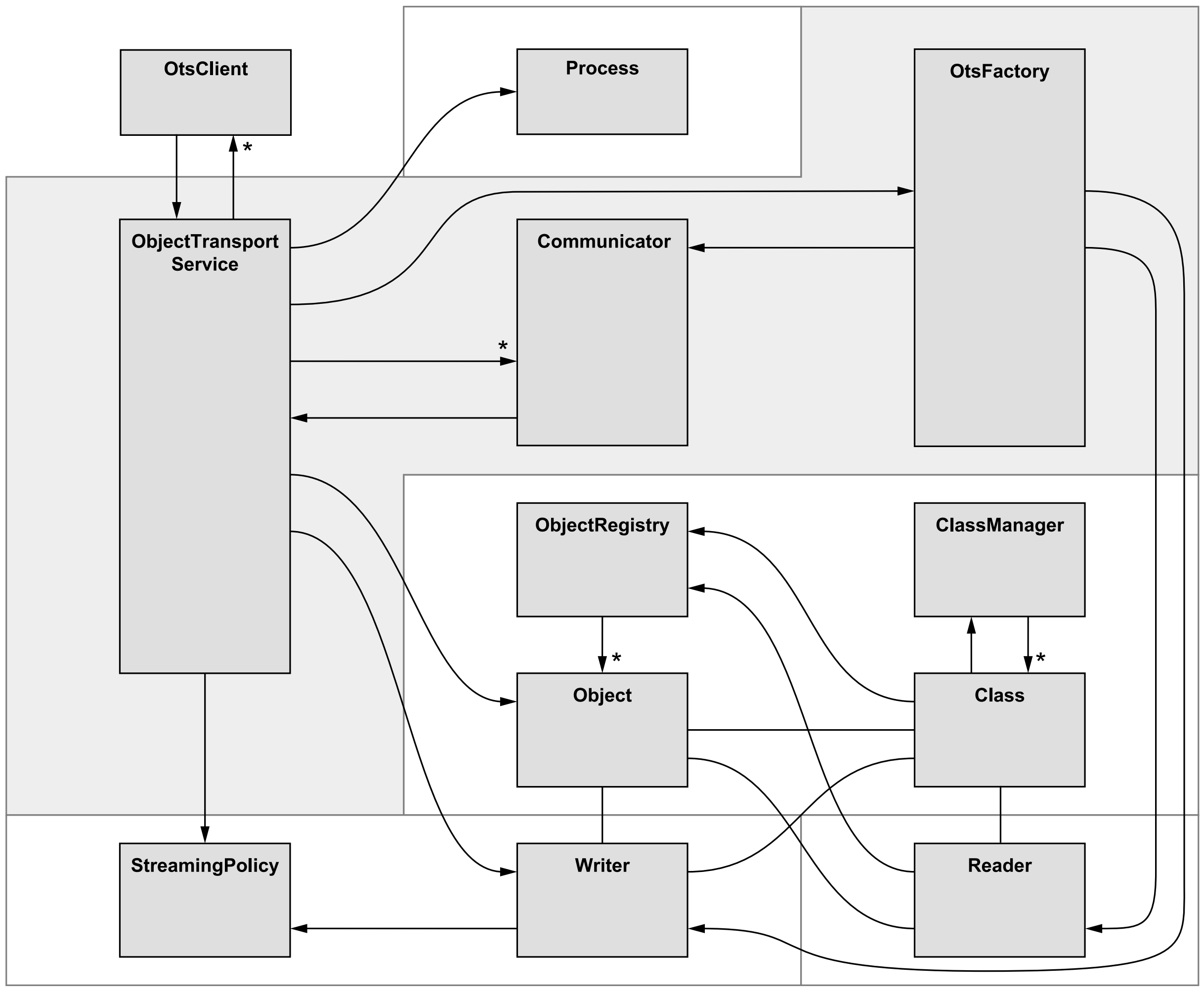

Figure 3: Class-based description of the abstract framework design.

5 Conclusions

Role modeling is important for the design and documentation of frameworks: Role models help in the description of design patterns, their application, and their composition. Role modeling makes a framework easier to design and comprehend than using only classes. The focus on classes tends to hide the interaction complexity in a single interface instead of breaking out the various roles that stem from the different object collaborations.

The design and implementation of a complex framework, e.g., the Geo-Transporter, experimentally validates the contribution of role modeling. We can succinctly describe the Geo-Transporter design using design patterns, and the role-based description of the involved patterns clarifies the dynamics of the framework in a concise way. Roles allow us to achieve a clear separation of concerns, and we strongly recommend their use in the design of object-oriented frameworks.

References

ARNOLD, K., and GOSLING, J. *The Java Programming Language.* Addison-Wesley, 1996.

BÄUMER, D., and RIEHLE, D. Product Trader. *Pattern Languages of Program Design 3.* Addison-Wesley, 1998, pp. 29-46.

GAMMA, E., HELM, R., JOHNSON, R., and VLISSIDES, J. *Design Patterns.* Addison-Wesley, 1995.

REENSKAUG, T., WOLD, P., and LEHNE, O. A. *Working with Objects.* Greenwich: Manning, 1996.

RIEHLE, D. Describing and composing patterns using role diagrams. *Proceedings of the 1st International Conference on Object-Oriented Technology in Russia* (WOON '96), 1996, pp. 169-178. Reprinted in: *Proceedings of the Ubilab ’96 Conference.* Konstanz, Germany: Universitätsverlag Konstanz, 1996, pp. 137-152. Available from www.ubs.com/ubilab.

RIEHLE, D. (a). Composite design patterns. *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA ’97). ACM Press, 1997, pp. 218-228.

RIEHLE, D. (b). *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. Available from www.ubs.com/ubilab.

RIEHLE, D., SIBERSKI, W., BÄUMER, D., MEGERT, D., and ZÜLLIGHOVEN, H. Serializer. *Pattern Languages of Program Design 3.* Addison-Wesley, 1998, pp. 293-312.

SIEGEL, J. *CORBA. Fundamentals and Programming.* Wiley, 1996.