DESIGN OF A COMMUNICATION FRAMEWORK FOR INTEROPERABLE INFORMATION SYSTEMS

W. Hasselbring[*]
Department of Computer Science, Software Technology
University of Dortmund
D-44221 Dortmund, Germany
email: hasselbring@acm.org

ABSTRACT

Frameworks are class hierarchies plus models of interactions which can be turned into complete applications through various kinds of specialization. Design patterns often guide the construction and documentation of frameworks. The run-time architecture of a framework is characterized by an inversion of control: event handler objects of the application are invoked via the framework's reactive dispatching mechanism.

This paper reports the development process of a software architecture that has been designed for accomplishing the transfer of operation specifications among interoperable information systems within a larger project, such that

- the communication framework does not need to know the structure and different types of operation specifications to be transferred and
- the individual information systems do not need to know the communication platform (in our case CORBA).

Some design patterns guided the construction of the resulting object-oriented framework to achieve a flexible software architecture. The emphasis of this paper is the description of the way in which the communication framework has been designed.

INTRODUCTION

Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context (Gamma et al., 1995). The pattern community catalogs useful design fragments and the context that guides their use. They do not make special distinctions between architectural patterns and patterns for code. An organized collection of related patterns for a particular application domain can be called a *pattern system* or *pattern language*.

All aspects of software systems, their development and their deployment are suitable topics of individual patterns or comprehensive pattern languages. Patterns might be so specific as to name particular objects, their responsibilities, and interaction. A well-known pattern of this kind is, for example, the Observer pattern from (Gamma et al., 1995). It supports keeping co-operating components consistent, with help of a change propagation mechanism.

Object-oriented frameworks can be regarded as incomplete software architectures which can be turned into complete applications through various kinds of specialization (Pree, 1995; Fayad and Schmidt, 1997). Design patterns guide the construction and documentation of frameworks, but they may also be *discovered* in existing object-oriented frameworks, e.g., in frameworks for graphical user interfaces, communication middleware, databases, etc.

The communication framework presented in this paper has been developed as part of a larger project in which heterogeneous information systems had to interoperate. An important goal for the project organization was to decouple the subsystem components as far as possible in a simple way such that the individual subgroups within the project team are able to work independently while agreeing on small interface specifications.

BACKGROUND

The larger project in which the communication framework has been developed aimed at integrating two components of a hospital information systems: a system for controlling the results of therapies in angiopathy and an administration system for charging the treatment. The therapy control system has been implemented in cooperation of the University of Dortmund and the University Hospital Wuppertal (Ullrich et al., 1996) with the object-oriented database system O_2 (Bancilhon et al., 1992). The patient data management system has been implemented with the relational database system Oracle (Bronzite, 1989).

This section gives a brief overview of the federated system architecture in this project which has been designed according to the specific requirements of integrating replicated information among heterogeneous information systems within hospitals (Hasselbring, 1997). Below, we present an extended schema architecture and the associated algorithms that restore the integrity of replicated informa-

[*]New address from August 1998 onwards: Department of Information Management and Computer Science, Tilburg University, 5000 LE Tilburg, The Netherlands.

tion when changes occur.

A *database system* (DBS) consists of a database management system and one or more databases that it manages. A federated DBS is an integration of autonomous database systems (Sheth and Larson, 1990). In a federated DBS, both global applications and local applications are supported. The local applications remain autonomous, but must restrict their autonomy to some extent to participate in the federation. Global applications can access multiple local DBSs through the federation layer. The federation layer can also control global integrity constraints such as data value dependencies across multiple component DBSs.

To achieve a division of labor between system components, *agents* connected to the component DBSs are introduced as *active* DBSs (Widom and Ceri, 1996) to serve as mediators between component DBSs and federation layer. Figure 1 displays the general system architecture illustrating the division of labor between kernel and agents. The local database management systems of the component DBSs consider the active agents as local applications.

This approach allows a 'separation of concerns' between the federation kernel and the component agents. The responsibility for monitoring and announcing changes in component DBSs is delegated from the kernel of the federation layer to the agents for the individual component DBSs. This way, the kernel of the federation layer sees the component DBSs as active DBSs. An active DBS is an extended conventional DBS which has the capability to monitor predefined situations (situations of interest) and to react with defined actions (Widom and Ceri, 1996). 'Separation of concerns' is an important principle for software engineering (Ghezzi et al., 1991).

Figure 2 refines this architectural view, where the communication service deploys an Object Request Broker according to the CORBA architecture (Mowbray and Zahavi, 1995). In Figure 2, only one active agent is shown. At the bottom layer in the federation kernel, the meta data of the kernel (schema dependencies, etc.) are stored in the object-oriented database system O_2 (Bancilhon et al., 1992).

For federated DBSs, the traditional three-level schema architecture (Date, 1995) is extended to support the dimensions of distribution, heterogeneity, and autonomy. The generally accepted reference architecture for schemas in federated DBSs is presented in (Sheth and Larson, 1990). It is obvious that this reference schema architecture has been designed primarily to support global access to the component DBSs, only secondarily to support integrity control. Therefore, we extended the reference schema architecture of (Sheth and Larson, 1990) with import, export and import/export distinction for *public* schemas to adequately support the algorithms for changing replicated information (Hasselbring, 1997).

Figure 3 illustrates an example scenario for changing replicas with our schema architecture that defines the dependencies among replicated data. To explain Figure 3: A *local* schema is the conceptual schema of a component DBS which is expressed in the (native) data model of that component DBS. A *component* schema is a local schema transformed into the (common) data model of the federation layer. An *export* or *import* schema is derived from a component schema and defines an interface to the local data that is made available to the federation. A *federated* schema is the result of the integration of multiple export or import schemas, and thus provides a uniform interface for global applications and the specification of the dependencies among replicated data. An *external* schema is a specific view on a federated schema or on a local schema which serves as a specific interface for applications (local or global). In Figure 3, no external schemas are shown.

Specifying an import schema in our architecture is a subscription to change notifications for the corresponding data items. Export schemas specify data to be exported to other systems. Import and export schemas are called *public* schemas (Hasselbring, 1997). The schema architecture is the basis for algorithms that restore the integrity of replicated information when changes occur.

We can only present an overview of our architecture in this paper. For a more detailed description refer to (Hasselbring, 1997). A crucial design decision is the architecture of the communication framework which is discussed in the next section.

THE COMMUNICATION FRAMEWORK

The communication framework encapsulates CORBA services to send and receive operations which restore the integrity of replicated information when changes occur in a component DBS. The transferred data objects contain specifications of the operations[1] to be transferred between the federation layer and the (active) database agents. The dependencies specified in the schema architecture determine the destinations for the operation objects.

CORBA is the 'Common Object Request Broker Architecture' of the Object Management Group to standardize interoperability among heterogeneous hardware and software systems (Mowbray and Zahavi, 1995). Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA defines the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA also defines interoperability by specifying how ORBs from different vendors can interoperate.

The ORB is the middleware that establishes the client-server relationships between objects. Clients can transparently invoke a method on a server object, which can be on

[1]For simplicity, the term *operation* is often used synonymously for *operation specification* when presenting the design.

the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can accept the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. We deployed the Chorus Cool CORBA implementation (Jacquemot et al., 1995) in the presented project.

A basic question arises: How to transfer operation specifications by the communication framework, such that the communication framework does not need to know their internal structure and the operation processing components do not need to know the communication platform (in our implementation CORBA)? The goal is a flexible software architecture that allows a flexible team organization!

The first idea for designing the communication framework was based on the design pattern *Abstract Factory*. The basic idea of this design pattern is that users of a 'factory' get an abstract interface for creating families of related objects without specifying their concrete classes (Gamma et al., 1995, pages 87ff). Figure 4 displays the static structure for this pattern as a class diagram in the UML notation (Fowler and Scott, 1997). The communication framework would be the client which only needs to know the abstract classes `OpFactory`, `InsertOp`, `UpdateOp` and possibly additional operation classes. The methods in the concrete subclasses of `OpFactory` create the corresponding concrete operation specifications (indicated by the dotted arrows).

A more detailed explanation of the model in Figure 4 is given as follows. Rectangles are the UML symbols for classes. Inheritance for specialization and generalization is shown in UML as a solid-line path from the subclass to the superclass, with a hollow triangle at the end of the path where it meets the superclass (Fowler and Scott, 1997). The C++ keyword `virtual` (Stroustrup, 1991) is used to specify abstract methods.

With the *Abstract Factory* pattern it has been achieved that the communication framework does not need to know the concrete classes. However, the number of different products (here operation specifications) in the product family is encoded within the model and the program code. In the case of the requirement for additional operation types, it becomes necessary to modify the communication framework as a client of the factory; thus, yielding a somewhat inflexible design (Hasselbring and Ziesche, 1997).

This situation led us to search for a solution in which the communication framework becomes decoupled from changes with respect to the structure *and* the number of different operation specifications. Our next idea was the design pattern *Prototype Factory*. The basic idea of this design pattern is that the different classes of operations and their handlers are represented through 'prototypical' instances that are able to 'clone' themselves (Gamma et al., 1995, pages 117ff).

Figure 5 displays the class structure for the prototypes of operations and their handlers. Handlers process received operation specifications. Figure 6 illustrates the architecture of the communication framework which only needs to know the abstract classes `Operation` and `Handler`, not their concrete subclasses.

A more detailed explanation of the model in Figure 6 is given as follows. In the UML, multiplicities for associations are specified through numerical ranges at the association links. The default multiplicity is 1. If the multiplicity specification comprises a single star, then it denotes the unlimited non-negative integer range (zero or more). Hollow diamonds indicate part-of relations (aggregation). The applied design patterns *Singleton* and *Prototype Factory* are indicated through comment boxes that are attached to the corresponding classes via dashed lines.

The classes `Sender` and `Receiver` manage the transfer of operations. They inherit some general methods for using the Object Request Broker from the abstract class `Communication Service` (see Figure 6). The return values of the methods for sending and receiving operations indicate the success or failure of a transfer.

It could be possible that the agent for a component DBS is unable to accept a change operation due to various reasons (out of memory, unavailability of the local database management system, network error, violation of local integrity constraints, etc.). These situations are reported to the federation layer via the Boolean return values of the corresponding handler calls. If an error occurs, the federation layer keeps the failed operation and, depending on the error type, manages the resulting inconsistency (Getta and Maciaszek, 1995).

The communication framework uses two abstract classes for which a user specifies concrete subclasses:

`Operation`: for each type of operation a concrete class is defined through inheritance from the abstract class `Operation` which specifies a uniform interface for all operation types (see Figure 5). Each concrete subclass specifies the specific structure for the specifications of one type of operation to be exchanged via object instances of this class. The communication framework itself is independent of this specific structure.

The `Clone` method is needed to obtain copies of the prototype objects. The attribute `OpType` identifies the type of the prototype objects.

The methods `ToSeq` and `FromSeq` (Figure 5) are needed to serialize the object structures into sequences and vice versa. We use sequence structures from the C++ Standard Template Library (STL) for this purpose (Robson, 1997). These sequences are then transferred via the CORBA ORB. This way, the CORBA

IDL remains independent of the concrete object structures.

`Handler`: to receive and process operations of a specific type, it is necessary to provide corresponding operation handlers to process the operations in an appropriate way (see Figure 5). On receipt of an operation object, the communication framework uses copies of *prototype* objects for operation/handler pairs, which are managed by the class Pool (see Figure 6). The handler is responsible for processing the associated operation; thus, realizing the corresponding application logic.

The presented mechanism, which makes the communication framework independent of the concrete operation classes, has been achieved through guiding the design by the pattern *Prototype Factory*.

Another design pattern used in Figure 6 is called *Singleton* (Gamma et al., 1995, pages 127ff). Each CORBA object (a C++ program) contains exactly one C++ object instance of the class `Receiver`, because each database agent is accessed as a CORBA object. However, several `Sender` objects may exist within a CORBA object.

It turns out that the developed communication framework is an object-oriented framework with *inversion of control* (Fayad and Schmidt, 1997): the framework calls the application which uses the framework. The handlers that represent the application logic for processing received operations are called by the communication framework. This is different to the reuse in procedural languages such as C, where the application calls functions/procedures which are provided by a library.

THE C++ IMPLEMENTATION

Due to space limitations, only an outline of the C++ implementation can be presented in this paper.

To achieve the requirement that each CORBA object (a C++ program) contains exactly one C++ object instance of the class `Receiver` (design pattern *Singleton*), the constructor of this class is declared as `private`. This prevents a direct instantiation with the `new` operator. Instead, `Receiver` offers the `static` method `Instance` which creates a new `static` object instance when first called. For every following call the method `Instance` just returns a reference to the singleton object instance.

The `Pool` that is associated to the `Receiver` object manages `Operation/Handler` pairs which are identified by the `Operation`'s attribute `OpType`. A client first registers the `Operation` prototypes to be transferred together with the associated `Handler` objects by means of the `Pool`'s method `RegisterOperation`:

```
void RegisterOperation
 (OpType, pair <Operation*, Handler*>);
```

The `pair` structure is provided by the C++ Standard Template Library (STL) (Robson, 1997). These pairs are retrieved by means of the `Pool`'s method `GetOperation`:

```
pair <Operation*, Handler*>
  GetOperation (OpType);
```

Figure 7 illustrates the dynamic behavior on receipt of an operation by means of a sequence diagram (Fowler and Scott, 1997).

SUMMARY AND FUTURE WORK

Design patterns can be viewed as abstract descriptions of simple frameworks that facilitate reuse of software architectures (Gamma et al., 1995). The present paper discusses how design patterns guide the construction and documentation of a framework. With the presented architecture, the operation specifications can be transferred through the communication framework in a way that

- the communication framework does not need to know the structure and different types of operation specifications to be transferred and
- the operation processing components do not need to know the communication platform.

This way, it was feasible decoupling the system components in a flexible way such that the individual subgroups within the project team were able to work independently while agreeing on small interface specifications.

A design pattern describes a family of solutions to a recurring problem. Patterns form larger wholes like pattern languages, systems or handbooks when woven together so as to provide guidance for solving complex problem sets. Patterns express the understanding gained from practice in software design and construction. Writing them is a good way to deepen, structure, and pass on the system skills we build up and that are called *experience*. The documentation of the employed design patterns is an important concern in explaining the framework to (potential) users (Johnson, 1992). In addition, the description of the design patterns in (Gamma et al., 1995) provides guidelines for the implementation together with some example code.

For the discipline of software engineering, modifiability and extensibility (for maintenance) are important quality properties that should be achieved in system's design (Ghezzi et al., 1991). As one result, the communication framework can be re-used for other systems with similar communication requirements, in particular exchange of information among interoperable information systems! Hereby, the receiving information systems define their application logic within application-specific handlers which are called by the communication framework on receipt of the corresponding information.

In the current implementation, the transfer of operations is synchronous. To increase the potential parallelism, the transfer of operations could be executed in separate threads of execution. With appropriate synchronization, even the execution of the event handlers could be parallelized. These extensions could be guided by the *Reactor* pattern (Schmidt, 1995) which dispatches handlers automatically when events occur from multiple sources.

At the current stage, classes inheriting from the abstract class `Operation` are required to provide the concrete serialization methods `ToSeq` and `FromSeq` for conversion to/from STL sequences. In a future version of the framework, we plan providing a *Serializable* interface for the class `Operation` similar to the *Serializable* interface of Java Beans (Englander, 1997). With the Java Beans *Serializable* interface, all subclasses of a class that implements this interface will also be serializable without the requirement of concrete serialization methods in the subclasses.

ACKNOWLEDGEMENTS

The author would like to thank Klaus Alfert, Andreas Dinsch, Mario Ellebrecht, Xi Gao, Sven Gerding, Betül Ilikli, Djamel Kheldoun, Patrick Koehne, Mischa Lohweber, Ulf Radmacher, Karl-Heinz Schulte, Dilber Yavuz, and Peter Ziesche for the cooperation in the project. Peter Ziesche implemented the prototype of the communication framework.

REFERENCES

Bancilhon, F., Delobel, C., and Kanellakis, P. (1992). *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufman.

Bronzite, M. (1989). *Introduction to Oracle*. McGraw-Hill, London.

Date, C. (1995). *An introduction to database systems*. Addison-Wesley, 6th edition.

Englander, R. (1997). *Developing Java Beans*. O'Reilly, Sebastopol, CA.

Fayad, M. and Schmidt, D. (1997). Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38.

Fowler, M. and Scott, K. (1997). *UML Distilled: Applying the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley, Reading, MA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA.

Getta, J. and Maciaszek, L. (1995). Management of inconsistent information in federated systems. In Papazoglou, M., editor, *Proc. 14th International Conference on Object-Oriented and Entity-Relationship Modeling (OOER'95)*, volume 1021 of *Lecture Notes in Computer Science*, pages 412–423, Gold Coast, Australia. Springer-Verlag.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (1991). *Fundamentals of Software Engineering*. Prentice-Hall, Englewood Cliffs, NJ.

Hasselbring, W. (1997). Federated integration of replicated information within hospitals. *International Journal on Digital Libraries*, 1(3):192–208.

Hasselbring, W. and Ziesche, P. (1997). The use of design patterns in the development of a federated hospital information system with CORBA. In *Proc. 'Verteilte Objekte in Organisationen' (Mobis '97)*, volume 4, pages 21–25, Bamberg. Rundbrief Informationssystem-Architekturen. (in German).

Jacquemot, C., Jensen, P. S., and Carrez, S. (1995). CHORUS/COOL: CHORUS object oriented technology. In *Object-Based Parallel and Distributed Computation (OBPDC '95)*, volume 1107 of *Lecture Notes in Computer Science*, pages 187–204. Springer-Verlag.

Johnson, R. (1992). Documenting frameworks using patterns. In *Proc. OOPSLA '92*, pages 63–76, Vancouver, BC.

Mowbray, T. and Zahavi, R. (1995). *The Essential CORBA: Systems Integration Using Distributed Objects*. Wiley, New York.

Pree, W. (1995). *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, Wokingham, England.

Robson, R. (1997). *Using the STL: The C++ Standard Template Library*. Springer-Verlag, New York.

Schmidt, D. (1995). Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74.

Sheth, A. and Larson, J. (1990). Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236.

Stroustrup, B. (1991). *The C++ Programming Language*. Addison-Wesley, Reading, MA, second edition.

Ullrich, I., Hasselbring, W., Jahnke, T., Röser, A., and Christmann, A. (1996). An object-oriented system for therapy control in angiopathy. In *Abstracts of the 41. GMDS-Jahrestagung*, Bonn, Germany. (in German).

Widom, J. and Ceri, S., editors (1996). *Active Database Systems – Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, San Francisco.

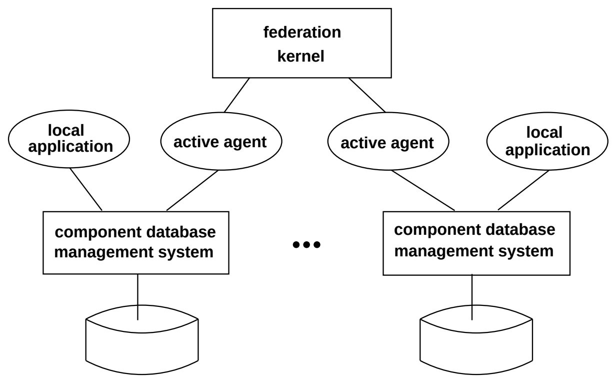

Fig. 1: The general system architecture with active agents as mediators between component DBSs and federation kernel. Global applications are not displayed in this figure, but they could be connected to the federation kernel.

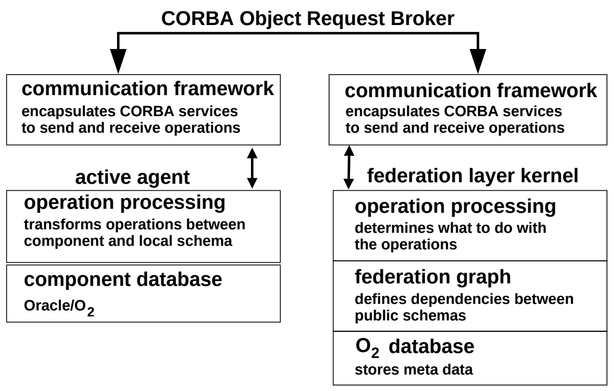
CORBA Object Request Broker
communication framework
encapsulates CORBA services to send and receive operations
communication framework
encapsulates CORBA services to send and receive operations
active agent
federation layer kernel
operation processing
transforms operations between component and local schema
operation processing
determines what to do with the operations
component database
Oracle/O2
federation graph
defines dependencies between public schemas
O2 database
stores meta data

Fig. 2: The layers in the CORBA-based architecture of our prototype implementation.

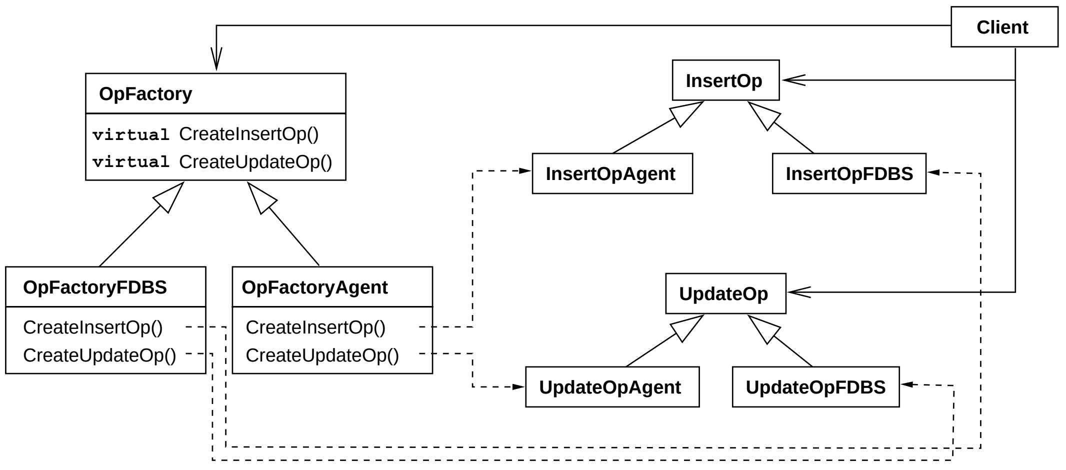

Fig. 4: The design pattern *Abstract Factory* in UML notation. The dashed arrows indicate the relations from concrete factory methods to concrete product classes (this annotation is not UML notation).

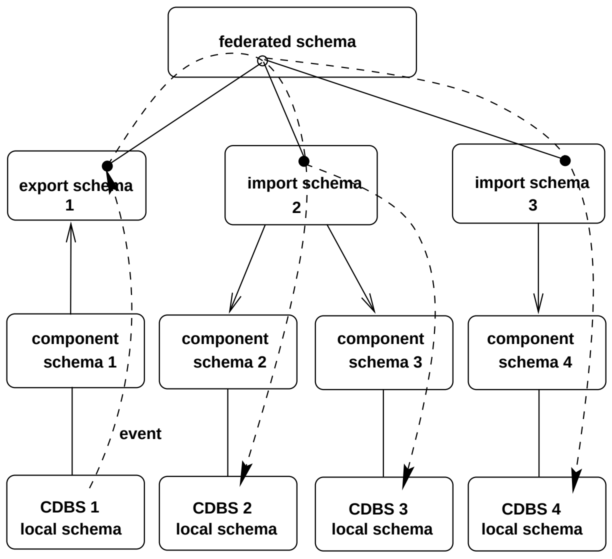

Fig. 3: An example scenario for changing replicas. The edges illustrate the data dependencies among the schemas. The dashed lines illustrate the data flow.

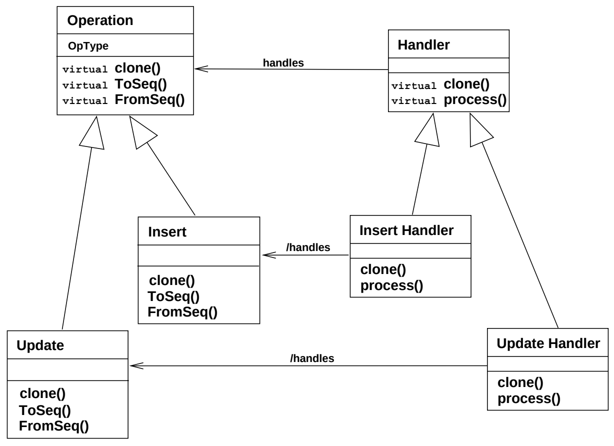

Fig. 5: Operations and their handlers in UML notation. The symbol '/' at the lower 'handles' associations indicates their inheritance relationship to the corresponding upper association.

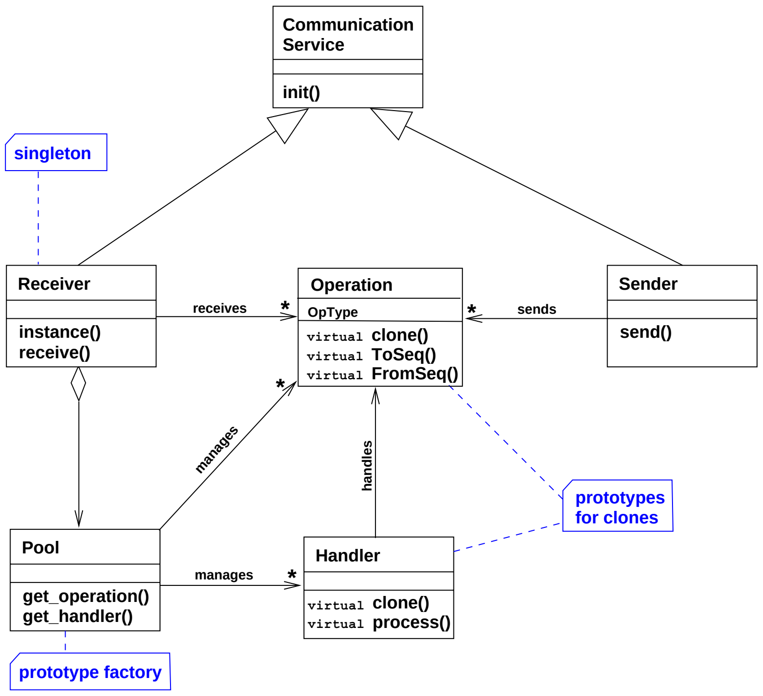

Fig. 6: The general architecture of the communication framework in UML notation.

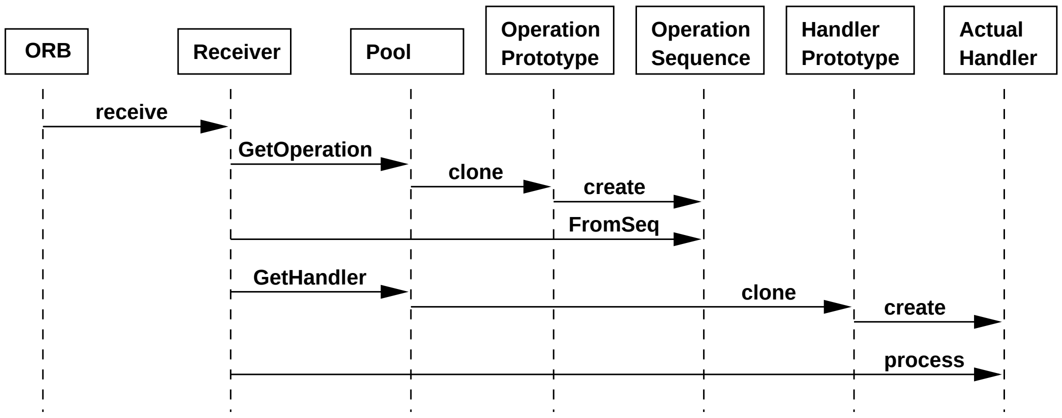

Fig. 7: A sequence diagram to illustrate the dynamic behavior on receipt of an operation. Within a sequence diagram, an object is shown as a box at the top of a dashed vertical line which is called the object's *lifetime* during some interaction with other objects. The sequential order of messages is represented by the vertical order of arrows between the lifelines of objects as instances of a class.